

## Top 30 Most Influential Papers In The World Of Big Data



These papers provide a breadth of information about Big Data (a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software) that is generally useful and interesting from a computer science perspective.

### Contents

1. Dremel: Interactive Analysis of WebScale Datasets
2. Large-scale Incremental Processing Using Distributed Transactions and Notifications
3. Availability in Globally Distributed Storage Systems
4. Scientific Data Management in the Coming Decade
5. What Next? A Dozen Information-Technology Research Goals
6. Volley: Automated Data Placement for Geo-Distributed Cloud Services
7. Dynamo: Amazon's Highly Available Key-value Store
8. Bigtable: A Distributed Storage System for Structured Data

9. The Collective: A Cache-Based System Management Architecture
10. Cloud Storage for Cloud Computing
11. Data-Intensive Supercomputing: The case for DISC
12. MapReduce Online
13. Frustratingly Easy Domain Adaptation
14. The Google File System
15. Cassandra - A Decentralized Structured Storage System
16. MapReduce: Simplified Data Processing on Large Clusters
17. NoSQL Databases
18. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications
19. Parallax: Virtual Disks for Virtual Machines
20. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems
21. Large-scale Incremental Processing Using Distributed Transactions and Notifications
22. Interpreting the Data: Parallel Analysis with Sawzall
23. Spanner: Google's Globally-Distributed Database
24. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems
25. What is Data Science?
26. The Dangers of Replication and a Solution
27. Data clustering: 50 years beyond K-means

28. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds

29. Lithium: Virtual Machine Storage for the Cloud

30. Bayesian Semi-supervised Learning with Graph Gaussian Processes

# Dremel: Interactive Analysis of Web-Scale Datasets

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer,  
Shiva Shivakumar, Matt Tolton, Theo Vassilakis  
Google, Inc.  
{melnik, andrey, jlong, gromer, shiva, mtolton, theov}@google.com

## ABSTRACT

Dremel is a scalable, interactive ad-hoc query system for analysis of read-only nested data. By combining multi-level execution trees and columnar data layout, it is capable of running aggregation queries over trillion-row tables in seconds. The system scales to thousands of CPUs and petabytes of data, and has thousands of users at Google. In this paper, we describe the architecture and implementation of Dremel, and explain how it complements MapReduce-based computing. We present a novel columnar storage representation for nested records and discuss experiments on few-thousand node instances of the system.

## 1. INTRODUCTION

Large-scale analytical data processing has become widespread in web companies and across industries, not least due to low-cost storage that enabled collecting vast amounts of business-critical data. Putting this data at the fingertips of analysts and engineers has grown increasingly important; interactive response times often make a qualitative difference in data exploration, monitoring, online customer support, rapid prototyping, debugging of data pipelines, and other tasks.

Performing interactive data analysis at scale demands a high degree of parallelism. For example, reading one terabyte of compressed data in one second using today's commodity disks would require tens of thousands of disks. Similarly, CPU-intensive queries may need to run on thousands of cores to complete within seconds. At Google, massively parallel computing is done using shared clusters of commodity machines [5]. A cluster typically hosts a multitude of distributed applications that share resources, have widely varying workloads, and run on machines with different hardware parameters. An individual worker in a distributed application may take much longer to execute a given task than others, or may never complete due to failures or preemption by the cluster management system. Hence, dealing with stragglers and failures is essential for achieving fast execution and fault tolerance [10].

The data used in web and scientific computing is often non-relational. Hence, a flexible data model is essential in these domains. Data structures used in programming languages, messages

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

*Proceedings of the VLDB Endowment*, Vol. 3, No. 1  
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

exchanged by distributed systems, structured documents, etc. lend themselves naturally to a *nested* representation. Normalizing and recombining such data at web scale is usually prohibitive. A nested data model underlies most of structured data processing at Google [21] and reportedly at other major web companies.

This paper describes a system called Dremel<sup>1</sup> that supports interactive analysis of very large datasets over shared clusters of commodity machines. Unlike traditional databases, it is capable of operating on *in situ* nested data. *In situ* refers to the ability to access data ‘in place’, e.g., in a distributed file system (like GFS [14]) or another storage layer (e.g., Bigtable [8]). Dremel can execute many queries over such data that would ordinarily require a sequence of MapReduce (MR [12]) jobs, but at a fraction of the execution time. Dremel is not intended as a replacement for MR and is often used in conjunction with it to analyze outputs of MR pipelines or rapidly prototype larger computations.

Dremel has been in production since 2006 and has thousands of users within Google. Multiple instances of Dremel are deployed in the company, ranging from tens to thousands of nodes. Examples of using the system include:

- Analysis of crawled web documents.
- Tracking install data for applications on Android Market.
- Crash reporting for Google products.
- OCR results from Google Books.
- Spam analysis.
- Debugging of map tiles on Google Maps.
- Tablet migrations in managed Bigtable instances.
- Results of tests run on Google’s distributed build system.
- Disk I/O statistics for hundreds of thousands of disks.
- Resource monitoring for jobs run in Google’s data centers.
- Symbols and dependencies in Google’s codebase.

Dremel builds on ideas from web search and parallel DBMSs. First, its architecture borrows the concept of a serving tree used in distributed search engines [11]. Just like a web search request, a query gets pushed down the tree and is rewritten at each step. The result of the query is assembled by aggregating the replies received from lower levels of the tree. Second, Dremel provides a high-level, SQL-like language to express ad hoc queries. In contrast to layers such as Pig [18] and Hive [16], it executes queries natively without translating them into MR jobs.

Lastly, and importantly, Dremel uses a column-striped storage representation, which enables it to read less data from secondary

<sup>1</sup>Dremel is a brand of power tools that primarily rely on their speed as opposed to torque. We use this name for an internal project only.

storage and reduce CPU cost due to cheaper compression. Column stores have been adopted for analyzing relational data [1] but to the best of our knowledge have not been extended to nested data models. The columnar storage format that we present is supported by many data processing tools at Google, including MR, Sawzall [20], and FlumeJava [7].

In this paper we make the following contributions:

- We describe a novel columnar storage format for nested data. We present algorithms for dissecting nested records into columns and reassembling them (Section 4).
- We outline Dremel’s query language and execution. Both are designed to operate efficiently on column-striped nested data and do not require restructuring of nested records (Section 5).
- We show how execution trees used in web search systems can be applied to database processing, and explain their benefits for answering aggregation queries efficiently (Section 6).
- We present experiments on trillion-record, multi-terabyte datasets, conducted on system instances running on 1000-4000 nodes (Section 7).

This paper is structured as follows. In Section 2, we explain how Dremel is used for data analysis in combination with other data management tools. Its data model is presented in Section 3. The main contributions listed above are covered in Sections 4-8. Related work is discussed in Section 9. Section 10 is the conclusion.

## 2. BACKGROUND

We start by walking through a scenario that illustrates how interactive query processing fits into a broader data management ecosystem. Suppose that Alice, an engineer at Google, comes up with a novel idea for extracting new kinds of signals from web pages. She runs an MR job that cranks through the input data and produces a dataset containing the new signals, stored in billions of records in the distributed file system. To analyze the results of her experiment, she launches Dremel and executes several interactive commands:

```
DEFINE TABLE t AS /path/to/data/*
SELECT TOP(signal1, 100), COUNT(*) FROM t
```

Her commands execute in seconds. She runs a few other queries to convince herself that her algorithm works. She finds an irregularity in signal1 and digs deeper by writing a FlumeJava [7] program that performs a more complex analytical computation over her output dataset. Once the issue is fixed, she sets up a pipeline which processes the incoming input data continuously. She formulates a few canned SQL queries that aggregate the results of her pipeline across various dimensions, and adds them to an interactive dashboard. Finally, she registers her new dataset in a catalog so other engineers can locate and query it quickly.

The above scenario requires interoperation between the query processor and other data management tools. The first ingredient for that is a *common storage layer*. The Google File System (GFS [14]) is one such distributed storage layer widely used in the company. GFS uses replication to preserve the data despite faulty hardware and achieve fast response times in presence of stragglers. A high-performance storage layer is critical for *in situ* data management. It allows accessing the data without a time-consuming loading phase, which is a major impedance to database usage in analytical data processing [13], where it is often possible to run dozens of MR analyses before a DBMS is able to load the data and execute a single query. As an added benefit, data in a file system can be conveniently manipulated using standard tools, e.g., to transfer to another cluster, change access privileges, or identify a subset of data for analysis based on file names.

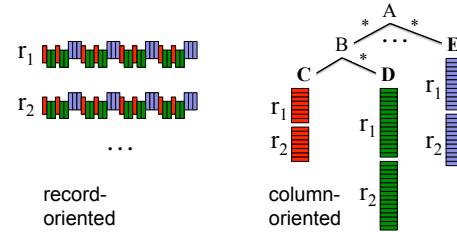


Figure 1: Record-wise vs. columnar representation of nested data

The second ingredient for building interoperable data management components is a *shared storage format*. Columnar storage proved successful for flat relational data but making it work for Google required adapting it to a nested data model. Figure 1 illustrates the main idea: all values of a nested field such as A.B.C are stored contiguously. Hence, A.B.C can be retrieved without reading A.E, A.B.D, etc. The challenge that we address is how to preserve all structural information and be able to reconstruct records from an arbitrary subset of fields. Next we discuss our data model, and then turn to algorithms and query processing.

## 3. DATA MODEL

In this section we present Dremel’s data model and introduce some terminology used later. The data model originated in the context of distributed systems (which explains its name, ‘Protocol Buffers’ [21]), is used widely at Google, and is available as an open source implementation. The data model is based on strongly-typed nested records. Its abstract syntax is given by:

$$\tau = \text{dom} \mid \langle A_1 : \tau[*|?], \dots, A_n : \tau[*|?] \rangle$$

where  $\tau$  is an atomic type or a record type. Atomic types in **dom** comprise integers, floating-point numbers, strings, etc. Records consist of one or multiple fields. Field  $i$  in a record has a name  $A_i$  and an optional multiplicity label. *Repeated* fields (\*) may occur multiple times in a record. They are interpreted as lists of values, i.e., the order of field occurrences in a record is significant. *Optional* fields (?) may be missing from the record. Otherwise, a field is *required*, i.e., must appear exactly once.

To illustrate, consider Figure 2. It depicts a schema that defines a record type **Document**, representing a web document. The schema definition uses the concrete syntax from [21]. A Document has a required integer **DocId** and optional **Links**, containing a list of **Forward** and **Backward** entries holding **DocIds** of other web pages. A document can have multiple **Names**, which are different URLs by which the document can be referenced. A **Name** contains a sequence of **Code** and (optional) **Country** pairs. Figure 2 also shows two sample records,  $r_1$  and  $r_2$ , conforming to the schema. The record structure is outlined using indentation. We will use these sample records to explain the algorithms in the next sections. The fields defined in the schema form a tree hierarchy. The full *path* of a nested field is denoted using the usual dotted notation, e.g., **Name.Language.Code**.

The nested data model backs a platform-neutral, extensible mechanism for serializing structured data at Google. Code generation tools produce bindings for programming languages such as C++ or Java. Cross-language interoperability is achieved using a standard binary on-the-wire representation of records, in which field values are laid out sequentially as they occur in the record. This way, a MR program written in Java can consume records from a data source exposed via a C++ library. Thus, if records are stored in a columnar representation, assembling them fast is important for interoperation with MR and other data processing tools.

DocId: 10	<b>r<sub>1</sub></b>
Links	
Forward: 20	
Forward: 40	
Forward: 60	
Name	
Language	
Code: 'en-us'	
Country: 'us'	
Language	
Code: 'en'	
Url: 'http://A'	
Name	
Url: 'http://B'	
Name	
Language	
Code: 'en-gb'	
Country: 'gb'	

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

DocId: 20	<b>r<sub>2</sub></b>
Links	
Backward: 10	
Backward: 30	
Forward: 80	
Name	
Url: 'http://C'	

Figure 2: Two sample nested records and their schema

## 4. NESTED COLUMNAR STORAGE

As illustrated in Figure 1, our goal is to store all values of a given field consecutively to improve retrieval efficiency. In this section, we address the following challenges: lossless representation of record structure in a columnar format (Section 4.1), fast encoding (Section 4.2), and efficient record assembly (Section 4.3).

### 4.1 Repetition and Definition Levels

Values alone do not convey the structure of a record. Given two values of a repeated field, we do not know at what ‘level’ the value repeated (e.g., whether these values are from two different records, or two repeated values in the same record). Likewise, given a missing optional field, we do not know which enclosing records were defined explicitly. We therefore introduce the concepts of repetition and definition levels, which are defined below. For reference, see Figure 3 which summarizes the repetition and definition levels for all atomic fields in our sample records.

**Repetition levels.** Consider field Code in Figure 2. It occurs three times in  $r_1$ . Occurrences ‘en-us’ and ‘en’ are inside the first Name, while ‘en-gb’ is in the third Name. To disambiguate these occurrences, we attach a repetition level to each value. It tells us *at what repeated field in the field’s path the value has repeated*. The field path Name.Language.Code contains two repeated fields, Name and Language. Hence, the repetition level of Code ranges between 0 and 2; level 0 denotes the start of a new record. Now suppose we are scanning record  $r_1$  top down. When we encounter ‘en-us’, we have not seen any repeated fields, i.e., the repetition level is 0. When we see ‘en’, field Language has repeated, so the repetition level is 2. Finally, when we encounter ‘en-gb’, Name has repeated most recently (Language occurred only once after Name), so the repetition level is 1. Thus, the repetition levels of Code values in  $r_1$  are 0, 2, 1.

Notice that the second Name in  $r_1$  does not contain any Code values. To determine that ‘en-gb’ occurs in the third Name and not in the second, we add a NULL value between ‘en’ and ‘en-gb’ (see Figure 3). Code is a required field in Language, so the fact that it is missing implies that Language is not defined. In general though, determining the level up to which nested records exist requires extra information.

**Definition levels.** Each value of a field with path  $p$ , esp. every NULL, has a definition level specifying *how many fields in p that could be undefined (because they are optional or repeated) are ac-*

DocId			Name.Url			Links.Forward			Links.Backward		
value	r	d	value	r	d	value	r	d	value	r	d
10	0	0	http://A	0	2	20	0	2	NULL	0	1
20	0	0	http://B	1	2	40	1	2	10	0	2
			NULL	1	1	60	1	2	30	1	2
			http://C	0	2	80	0	2			

Name.Language.Code			Name.Language.Country		
value	r	d	value	r	d
en-us	0	2	us	0	3
en	2	2	NULL	2	2
NULL	1	1	NULL	1	1
en-gb	1	2	gb	1	3
NULL	0	1	NULL	0	1

Figure 3: Column-striped representation of the sample data in Figure 2, showing repetition levels (r) and definition levels (d)

tually present in the record. To illustrate, observe that  $r_1$  has no Backward links. However, field Links is defined (at level 1). To preserve this information, we add a NULL value with definition level 1 to the Links.Backward column. Similarly, the missing occurrence of Name.Language.Country in  $r_2$  carries a definition level 1, while its missing occurrences in  $r_1$  have definition levels 2 (inside Name.Language) and 1 (inside Name), respectively.

We use integer definition levels as opposed to is-null bits so that the data for a leaf field (e.g., Name.Language.Country) contains the information about the occurrences of its parent fields; an example of how this information is used is given in Section 4.3.

The encoding outlined above preserves the record structure losslessly. We omit the proof for space reasons.

**Encoding.** Each column is stored as a set of blocks. Each block contains the repetition and definition levels (henceforth, simply called levels) and compressed field values. NULLs are not stored explicitly as they are determined by the definition levels: any definition level smaller than the number of repeated and optional fields in a field’s path denotes a NULL. Definition levels are not stored for values that are always defined. Similarly, repetition levels are stored only if required; for example, definition level 0 implies repetition level 0, so the latter can be omitted. In fact, in Figure 3, no levels are stored for DocId. Levels are packed as bit sequences. We only use as many bits as necessary; for example, if the maximum definition level is 3, we use 2 bits per definition level.

### 4.2 Splitting Records into Columns

Above we presented an encoding of the record structure in a columnar format. The next challenge we address is how to produce column stripes with repetition and definition levels efficiently.

The base algorithm for computing repetition and definition levels is given in Appendix A. The algorithm recurses into the record structure and computes the levels for each field value. As illustrated earlier, repetition and definition levels may need to be computed even if field values are missing. Many datasets used at Google are sparse; it is not uncommon to have a schema with thousands of fields, only a hundred of which are used in a given record. Hence, we try to process missing fields as cheaply as possible. To produce column stripes, we create a tree of *field writers*, whose structure matches the field hierarchy in the schema. The basic idea is to update field writers only when they have their own data, and not try to propagate parent state down the tree unless absolutely neces-

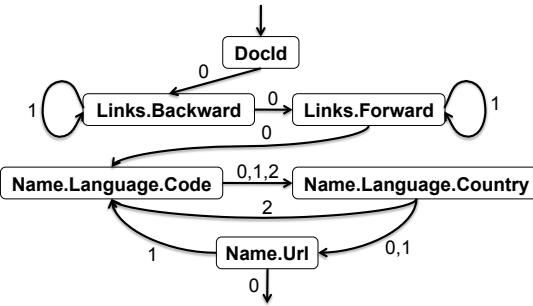


Figure 4: Complete record assembly automaton. Edges are labeled with repetition levels.

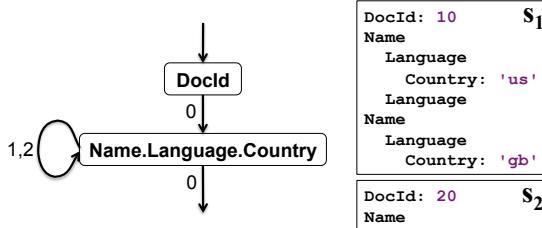


Figure 5: Automaton for assembling records from two fields, and the records it produces

sary. To do that, child writers inherit the levels from their parents. A child writer synchronizes to its parent’s levels whenever a new value is added.

### 4.3 Record Assembly

Assembling records from columnar data efficiently is critical for record-oriented data processing tools (e.g., MR). Given a subset of fields, our goal is to reconstruct the original records as if they contained just the selected fields, with all other fields stripped away. The key idea is this: we create a finite state machine (FSM) that reads the field values and levels for each field, and appends the values sequentially to the output records. An FSM state corresponds to a field reader for each selected field. State transitions are labeled with repetition levels. Once a reader fetches a value, we look at the next repetition level to decide what next reader to use. The FSM is traversed from the start to end state once for each record.

Figure 4 shows an FSM that reconstructs the complete records in our running example. The start state is DocId. Once a DocId value is read, the FSM transitions to Links.Backward. After all repeated Backward values have been drained, the FSM jumps to Links.Forward, etc. The details of the record assembly algorithm are in Appendix B.

To sketch how FSM transitions are constructed, let  $l$  be the next repetition level returned by the current field reader for field  $f$ . Starting at  $f$  in the schema tree, we find its ancestor that repeats at level  $l$  and select the first leaf field  $n$  inside that ancestor. This gives us an FSM transition  $(f, l) \rightarrow n$ . For example, let  $l = 1$  be the next repetition level read by  $f = \text{Name.Language.Country}$ . Its ancestor with repetition level 1 is Name, whose first leaf field is  $n = \text{Name.Url}$ . The details of the FSM construction algorithm are in Appendix C.

If only a subset of fields need to be retrieved, we construct a simpler FSM that is cheaper to execute. Figure 5 depicts an FSM for reading the fields DocId and Name.Language.Country. The figure shows the output records  $s_1$  and  $s_2$  produced by the automaton. Notice that our encoding and the assembly algorithm

```

SELECT DocId AS Id,
       COUNT(Name.Language.Code) WITHIN Name AS Cnt,
       Name.Url + ',' + Name.Language.Code AS Str
  FROM t
 WHERE REGEXP(Name.Url, '^http') AND DocId < 20;

```

<b>t<sub>1</sub></b>	<b>message QueryResult { required int64 Id; repeated group Name { optional uint64 Cnt; repeated group Language { optional string Str; }}}}</b>
<b>Name</b> Id: 10 Name Cnt: 2 Language Str: 'http://A,en-us' Str: 'http://A,en' Name Cnt: 0	

Figure 6: Sample query, its result, and output schema

preserve the enclosing structure of the field Country. This is important for applications that need to access, e.g., the Country appearing in the first Language of the second Name. In XPath, this would correspond to the ability to evaluate expressions like  $/Name[2]/Language[1]/Country$ .

## 5. QUERY LANGUAGE

Dremel’s query language is based on SQL and is designed to be efficiently implementable on columnar nested storage. Defining the language formally is out of scope of this paper; instead, we illustrate its flavor. Each SQL statement (and algebraic operators it translates to) takes as input one or multiple nested tables and their schemas and produces a nested table and its output schema. Figure 6 depicts a sample query that performs projection, selection, and within-record aggregation. The query is evaluated over the table  $t = \{r_1, r_2\}$  from Figure 2. The fields are referenced using path expressions. The query produces a nested result although no record constructors are present in the query.

To explain what the query does, consider the selection operation (the WHERE clause). Think of a nested record as a labeled tree, where each label corresponds to a field name. The selection operator prunes away the branches of the tree that do not satisfy the specified conditions. Thus, only those nested records are retained where Name.Url is defined and starts with http. Next, consider projection. Each scalar expression in the SELECT clause emits a value at the same level of nesting as the most-repeated input field used in that expression. So, the string concatenation expression emits Str values at the level of Name.Language.Code in the input schema. The COUNT expression illustrates within-record aggregation. The aggregation is done WITHIN each Name subrecord, and emits the number of occurrences of Name.Language.Code for each Name as a non-negative 64-bit integer (uint64).

The language supports nested subqueries, inter and intra-record aggregation, top-k, joins, user-defined functions, etc; some of these features are exemplified in the experimental section.

## 6. QUERY EXECUTION

We discuss the core ideas in the context of a read-only system, for simplicity. Many Dremel queries are one-pass aggregations; therefore, we focus on explaining those and use them for experiments in the next section. We defer the discussion of joins, indexing, updates, etc. to future work.

*Tree architecture.* Dremel uses a multi-level serving tree to execute queries (see Figure 7). A root server receives incoming queries, reads metadata from the tables, and routes the queries to the next level in the serving tree. The leaf servers communicate

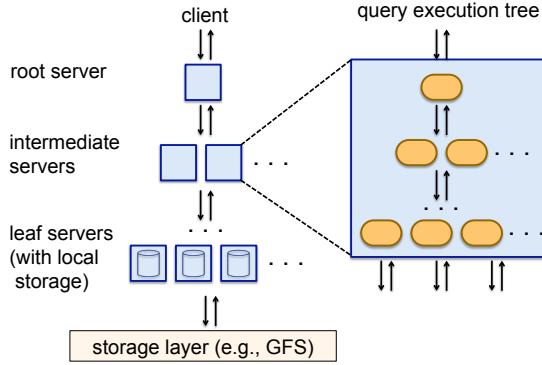


Figure 7: System architecture and execution inside a server node

with the storage layer or access the data on local disk. Consider a simple aggregation query below:

`SELECT A, COUNT(B) FROM T GROUP BY A`

When the root server receives the above query, it determines all *tablets*, i.e., horizontal partitions of the table, that comprise  $T$  and rewrites the query as follows:

`SELECT A, SUM(c) FROM ( $R_1^1$  UNION ALL ...  $R_n^1$ ) GROUP BY A`

Tables  $R_1^1, \dots, R_n^1$  are the results of queries sent to the nodes  $1, \dots, n$  at level 1 of the serving tree:

$R_i^1 = \text{SELECT A, COUNT(B) AS c FROM } T_i^1 \text{ GROUP BY A}$

$T_i^1$  is a disjoint partition of tablets in  $T$  processed by server  $i$  at level 1. Each serving level performs a similar rewriting. Ultimately, the queries reach the leaves, which scan the tablets in  $T$  in parallel. On the way up, intermediate servers perform a parallel aggregation of partial results. The execution model presented above is well-suited for aggregation queries returning small and medium-sized results, which are a very common class of interactive queries. Large aggregations and other classes of queries may need to rely on execution mechanisms known from parallel DBMSs and MR.

**Query dispatcher.** Dremel is a multi-user system, i.e., usually several queries are executed simultaneously. A query dispatcher schedules queries based on their priorities and balances the load. Its other important role is to provide *fault tolerance* when one server becomes much slower than others or a tablet replica becomes unreachable.

The amount of data processed in each query is often larger than the number of processing units available for execution, which we call *slots*. A slot corresponds to an execution thread on a leaf server. For example, a system of 3,000 leaf servers each using 8 threads has 24,000 slots. So, a table spanning 100,000 tablets can be processed by assigning about 5 tablets to each slot. During query execution, the query dispatcher computes a histogram of tablet processing times. If a tablet takes a disproportionately long time to process, it reschedules it on another server. Some tablets may need to be redispersed multiple times.

The leaf servers read stripes of nested data in columnar representation. The blocks in each stripe are prefetched asynchronously; the read-ahead cache typically achieves hit rates of 95%. Tablets are usually three-way replicated. When a leaf server cannot access one tablet replica, it falls over to another replica.

The query dispatcher honors a parameter that specifies the minimum percentage of tablets that must be scanned before returning a result. As we demonstrate shortly, setting such parameter to a lower value (e.g., 98% instead of 100%) can often speed up execu-

Table name	Number of records	Size (unrepl., compressed)	Number of fields	Data center	Repl. factor
T1	85 billion	87 TB	270	A	3×
T2	24 billion	13 TB	530	A	3×
T3	4 billion	70 TB	1200	A	3×
T4	1+ trillion	105 TB	50	B	3×
T5	1+ trillion	20 TB	30	B	2×

Figure 8: Datasets used in the experimental study

tion significantly, especially when using smaller replication factors.

Each server has an internal execution tree, as depicted on the right-hand side of Figure 7. The internal tree corresponds to a physical query execution plan, including evaluation of scalar expressions. Optimized, type-specific code is generated for most scalar functions. An execution plan for project-select-aggregate queries consists of a set of iterators that scan input columns in lockstep and emit results of aggregates and scalar functions annotated with the correct repetition and definition levels, bypassing record assembly entirely during query execution. For details, see Appendix D.

Some Dremel queries, such as top-k and count-distinct, return approximate results using known one-pass algorithms (e.g., [4]).

## 7. EXPERIMENTS

In this section we evaluate Dremel’s performance on several datasets used at Google, and examine the effectiveness of columnar storage for nested data. The properties of the datasets used in our study are summarized in Figure 8. In uncompressed, non-replicated form the datasets occupy about a petabyte of space. All tables are three-way replicated, except one two-way replicated table, and contain from 100K to 800K tablets of varying sizes. We start by examining the basic data access characteristics on a single machine, then show how columnar storage benefits MR execution, and finally focus on Dremel’s performance. The experiments were conducted on system instances running in two data centers next to many other applications, during regular business operation. Unless specified otherwise, execution times were averaged across five runs. Table and field names used below are anonymized.

**Local disk.** In the first experiment, we examine performance tradeoffs of columnar vs. record-oriented storage, scanning a 1GB fragment of table  $T_1$  containing about 300K rows (see Figure 9). The data is stored on a local disk and takes about 375MB in compressed columnar representation. The record-oriented format uses heavier compression yet yields about the same size on disk. The experiment was done on a dual-core Intel machine with a disk providing 70MB/s read bandwidth. All reported times are cold; OS cache was flushed prior to each scan.

The figure shows five graphs, illustrating the time it takes to read and uncompress the data, and assemble and parse the records, for a subset of the fields. Graphs (a)-(c) outline the results for columnar storage. Each data point in these graphs was obtained by averaging the measurements over 30 runs, in each of which a set of columns of a given cardinality was chosen at random. Graph (a) shows reading and decompression time. Graph (b) adds the time needed to assemble nested records from columns. Graph (c) shows how long it takes to parse the records into strongly typed C++ data structures.

Graphs (d)-(e) depict the time for accessing the data on record-oriented storage. Graph (d) shows reading and decompression time. A bulk of the time is spent in decompression; in fact, the compressed data can be read from the disk in about half the time. As

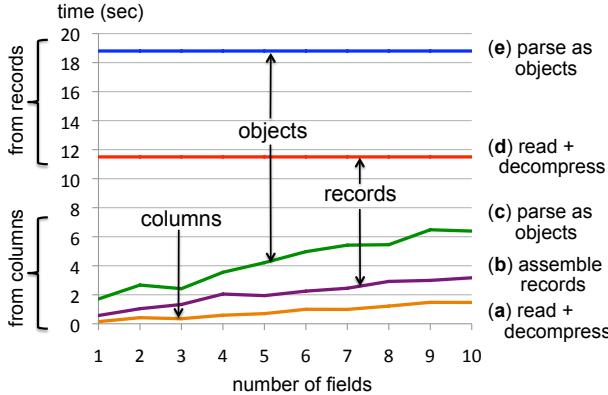


Figure 9: Performance breakdown when reading from a local disk (300K-record fragment of Table  $T_1$ )

Graph (e) indicates, parsing adds another 50% on top of reading and decompression time. These costs are paid for all fields, including the ones that are not needed.

The main takeaways of this experiment are the following: when few columns are read, the gains of columnar representation are of about an order of magnitude. Retrieval time for columnar nested data grows linearly with the number of fields. Record assembly and parsing are expensive, each potentially doubling the execution time. We observed similar trends on other datasets. A natural question to ask is where the top and bottom graphs cross, i.e., record-wise storage starts outperforming columnar storage. In our experience, the crossover point often lies at dozens of fields but it varies across datasets and depends on whether or not record assembly is required.

**MR and Dremel.** Next we illustrate a MR and Dremel execution on columnar vs. record-oriented data. We consider a case where a single field is accessed, i.e., the performance gains are most pronounced. Execution times for multiple columns can be extrapolated using the results of Figure 9. In this experiment, we count the average number of terms in a field `txtField` of table  $T_1$ . MR execution is done using the following Sawzall [20] program:

```
numRecs: table sum of int;
numWords: table sum of int;
emit numRecs <- 1;
emit numWords <- CountWords(input.txtField);
```

The number of records is stored in the variable `numRecs`. For each record, `numWords` is incremented by the number of terms in `input.txtField` returned by the `CountWords` function. After the program runs, the average term frequency can be computed as `numWords/numRecs`. In SQL, this computation is expressed as:

$Q_1: \text{SELECT SUM}(\text{CountWords}(\text{txtField})) / \text{COUNT}(\text{*}) \text{ FROM } T_1$

Figure 10 shows the execution times of two MR jobs and Dremel on a logarithmic scale. Both MR jobs are run on 3000 workers. Similarly, a 3000-node Dremel instance is used to execute Query  $Q_1$ . Dremel and MR-on-columns read about 0.5TB of compressed columnar data vs. 87TB read by MR-on-records. As the figure illustrates, MR gains an order of magnitude in efficiency by switching from record-oriented to columnar storage (from hours to minutes). Another order of magnitude is achieved by using Dremel (going from minutes to seconds).

**Serving tree topology.** In the next experiment, we show the impact of the serving tree depth on query execution times. We consider two GROUP BY queries on Table  $T_2$ , each executed using

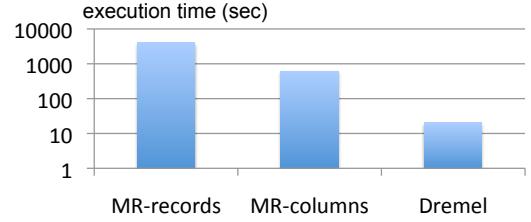


Figure 10: MR and Dremel execution on columnar vs. record-oriented storage (3000 nodes, 85 billion records)

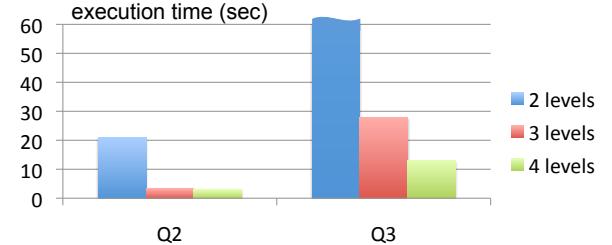


Figure 11: Execution time as a function of serving tree levels for two aggregation queries on  $T_2$

a single scan over the data. Table  $T_2$  contains 24 billion nested records. Each record has a repeated field item containing a numeric amount. The field item.amount repeats about 40 billion times in the dataset. The first query sums up the item amount by country:

$Q_2: \text{SELECT country, SUM(item.amount) FROM } T_2 \text{ GROUP BY country}$

It returns a few hundred records and reads roughly 60GB of compressed data from disk. The second query performs a GROUP BY on a text field domain with a selection condition. It reads about 180GB and produces around 1.1 million distinct domains:

$Q_3: \text{SELECT domain, SUM(item.amount) FROM } T_2 \text{ WHERE domain CONTAINS '.net' GROUP BY domain}$

Figure 11 shows the execution times for each query as a function of the server topology. In each topology, the number of leaf servers is kept constant at 2900 so that we can assume the same cumulative scan speed. In the 2-level topology (1:2900), a single root server communicates directly with the leaf servers. For 3 levels, we use a 1:100:2900 setup, i.e., an extra level of 100 intermediate servers. The 4-level topology is 1:10:100:2900.

Query  $Q_2$  runs in 3 seconds when 3 levels are used in the serving tree and does not benefit much from an extra level. In contrast, the execution time of  $Q_3$  is halved due to increased parallelism. At 2 levels,  $Q_3$  is off the chart, as the root server needs to aggregate near-sequentially the results received from thousands of nodes. This experiment illustrates how aggregations returning many groups benefit from multi-level serving trees.

**Per-tablet histograms.** To drill deeper into what happens during query execution consider Figure 12. The figure shows how fast tablets get processed by the leaf servers for a specific run of  $Q_2$  and  $Q_3$ . The time is measured starting at the point when a tablet got scheduled for execution in an available slot, i.e., excludes the time spent waiting in the job queue. This measurement methodology factors out the effects of other queries that are executing simultaneously. The area under each histogram corresponds to 100%. As the figure indicates, 99% of  $Q_2$  (or  $Q_3$ ) tablets are processed under one second (or two seconds).

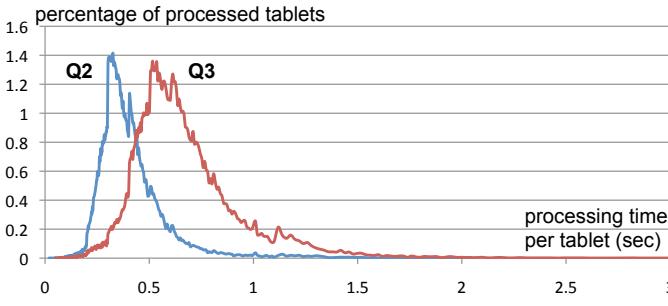


Figure 12: Histograms of processing times

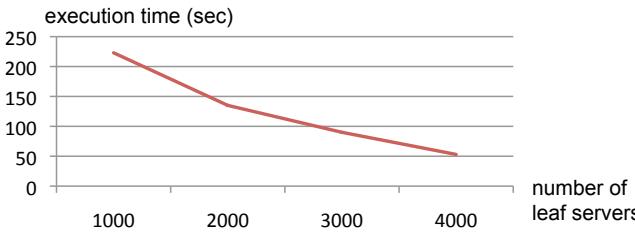


Figure 13: Scaling the system from 1000 to 4000 nodes using a top-k query  $Q_5$  on a trillion-row table  $T_4$

**Within-record aggregation.** As another experiment, we examine the performance of Query  $Q_4$  run on Table  $T_3$ . The query illustrates within-record aggregation: it counts all records where the sum of a.b.c.d values occurring in the record are larger than the sum of a.b.p.q.r values. The fields repeat at different levels of nesting. Due to column striping only 13GB (out of 70TB) are read from disk and the query completes in 15 seconds. Without support for nesting, running this query on  $T_3$  would be grossly expensive.

```
Q4 : SELECT COUNT(c1 > c2) FROM
      (SELECT SUM(a.b.c.d) WITHIN RECORD AS c1,
              SUM(a.b.p.q.r) WITHIN RECORD AS c2
       FROM T3)
```

**Scalability.** The following experiment illustrates the scalability of the system on a trillion-record table. Query  $Q_5$  shown below selects top-20 aid's and their number of occurrences in Table  $T_4$ . The query scans 4.2TB of compressed data.

```
Q5: SELECT TOP(aid, 20), COUNT(*) FROM T4
      WHERE bid = {value1} AND cid = {value2}
```

The query was executed using four configurations of the system, ranging from 1000 to 4000 nodes. The execution times are in Figure 13. In each run, the total expended CPU time is nearly identical, at about 300K seconds, whereas the user-perceived time decreases near-linearly with the growing size of the system. This result suggests that a larger system can be just as effective in terms of resource usage as a smaller one, yet allows faster execution.

**Stragglers.** Our last experiment shows the impact of stragglers. Query  $Q_6$  below is run on a trillion-row table  $T_5$ . In contrast to the other datasets,  $T_5$  is two-way replicated. Hence, the likelihood of stragglers slowing the execution is higher since there are fewer opportunities to reschedule the work.

```
Q6: SELECT COUNT(DISTINCT a) FROM T5
```

Query  $Q_6$  reads over 1TB of compressed data. The compres-

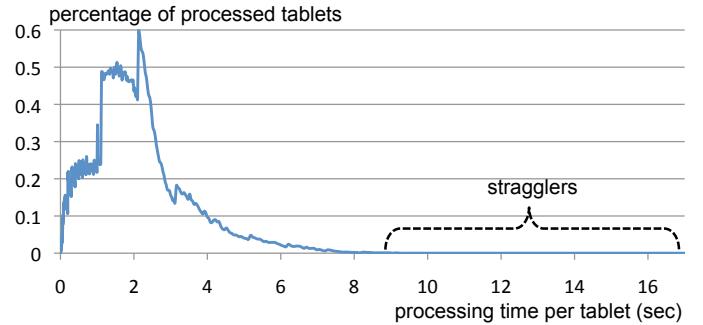


Figure 14: Query  $Q_6$  on  $T_5$  illustrating stragglers at 2× replication

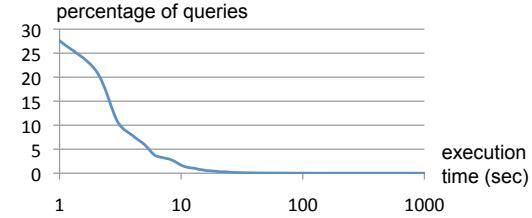


Figure 15: Query response time distribution in a monthly workload

sion ratio for the retrieved field is about 10. As indicated in Figure 14, the processing time for 99% of the tablets is below 5 seconds per tablet per slot. However, a small fraction of the tablets take a lot longer, slowing down the query response time from less than a minute to several minutes, when executed on a 2500 node system. The next section summarizes our experimental findings and the lessons we learned.

## 8. OBSERVATIONS

Dremel scans quadrillions of records per month. Figure 15 shows the query response time distribution in a typical monthly workload of one Dremel system, on a logarithmic scale. As the figure indicates, most queries are processed under 10 seconds, well within the interactive range. Some queries achieve a scan throughput close to 100 billion records per second on a shared cluster, and even higher on dedicated machines. The experimental data presented above suggests the following observations:

- Scan-based queries can be executed at interactive speeds on disk-resident datasets of up to a trillion records.
- Near-linear scalability in the number of columns and servers is achievable for systems containing thousands of nodes.
- MR can benefit from columnar storage just like a DBMS.
- Record assembly and parsing are expensive. Software layers (beyond the query processing layer) need to be optimized to directly consume column-oriented data.
- MR and query processing can be used in a complementary fashion; one layer's output can feed another's input.
- In a multi-user environment, a larger system can benefit from economies of scale while offering a qualitatively better user experience.
- If trading speed against accuracy is acceptable, a query can be terminated much earlier and yet see most of the data.
- The bulk of a web-scale dataset can be scanned fast. Getting to the last few percent within tight time bounds is hard.

Dremel’s codebase is dense; it comprises less than 100K lines of C++, Java, and Python code.

## 9. RELATED WORK

The MapReduce (MR) [12] framework was designed to address the challenges of large-scale computing in the context of long-running batch jobs. Like MR, Dremel provides fault tolerant execution, a flexible data model, and *in situ* data processing capabilities. The success of MR led to a wide range of third-party implementations (notably open-source Hadoop [15]), and a number of hybrid systems that combine parallel DBMSs with MR, offered by vendors like Aster, Cloudera, Greenplum, and Vertica. HadoopDB [3] is a research system in this hybrid category. Recent articles [13, 22] contrast MR and parallel DBMSs. Our work emphasizes the complementary nature of both paradigms.

Dremel is designed to operate at scale. Although it is conceivable that parallel DBMSs can be made to scale to thousands of nodes, we are not aware of any published work or industry reports that attempted that. Neither are we familiar with prior literature studying MR on columnar storage.

Our columnar representation of nested data builds on ideas that date back several decades: separation of structure from content and transposed representation. A recent review of work on column stores, incl. compression and query processing, can be found in [1]. Many commercial DBMSs support storage of nested data using XML (e.g., [19]). XML storage schemes attempt to separate the structure from the content but face more challenges due to the flexibility of the XML data model. One system that uses columnar XML representation is XMill [17]. XMill is a compression tool. It stores the structure for all fields combined and is not geared for selective retrieval of columns.

The data model used in Dremel is a variation of the complex value models and nested relational models discussed in [2]. Dremel’s query language builds on the ideas from [9], which introduced a language that avoids restructuring when accessing nested data. In contrast, restructuring is usually required in XQuery and object-oriented query languages, e.g., using nested for-loops and constructors. We are not aware of practical implementations of [9]. A recent SQL-like language operating on nested data is Pig [18]. Other systems for parallel data processing include Scope [6] and DryadLINQ [23], and are discussed in more detail in [7].

## 10. CONCLUSIONS

We presented Dremel, a distributed system for interactive analysis of large datasets. Dremel is a custom, scalable data management solution built from simpler components. It complements the MR paradigm. We discussed its performance on trillion-record, multi-terabyte datasets of real data. We outlined the key aspects of Dremel, including its storage format, query language, and execution. In the future, we plan to cover in more depth such areas as formal algebraic specification, joins, extensibility mechanisms, etc.

## 11. ACKNOWLEDGEMENTS

Dremel has benefited greatly from the input of many engineers and interns at Google, in particular Craig Chambers, Ori Gershoni, Rajeev Byrissetti, Leon Wong, Erik Hendriks, Erika Rice Scherpelz, Charlie Garrett, Idan Avraham, Rajesh Rao, Andy Kreling, Li Yin, Madhusudan Hosaagrahara, Dan Belov, Brian Bershad, Lawrence You, Rongrong Zhong, Meelap Shah, and Nathan Bales.

## 12. REFERENCES

- [1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-Oriented Database Systems. *VLDB*, 2(2), 2009.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
- [3] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *VLDB*, 2(1), 2009.
- [4] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *RANDOM*, pages 1–10, 2002.
- [5] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
- [6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *VLDB*, 1(2), 2008.
- [7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*, 2010.
- [8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.
- [9] L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. *SIGMOD Rec.*, 18(2), 1989.
- [10] G. Czajkowski. Sorting 1PB with MapReduce. Official Google Blog, Nov. 2008. At <http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html>.
- [11] J. Dean. Challenges in Building Large-Scale Information Retrieval Systems: Invited Talk. In *WSDM*, 2009.
- [12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
- [13] J. Dean and S. Ghemawat. MapReduce: a Flexible Data Processing Tool. *Commun. ACM*, 53(1), 2010.
- [14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.
- [15] Hadoop Apache Project. <http://hadoop.apache.org>.
- [16] Hive. <http://wiki.apache.org/hadoop/Hive>, 2009.
- [17] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *SIGMOD*, 2000.
- [18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-so-Foreign Language for Data Processing. In *SIGMOD*, 2008.
- [19] P. E. O’Neil, E. J. O’Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHS: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.
- [20] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4), 2005.
- [21] Protocol Buffers: Developer Guide. Available at <http://code.google.com/apis/protocolbuffers/docs/overview.html>.
- [22] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Commun. ACM*, 53(1), 2010.
- [23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

```

1 procedure DissectRecord(RecordDecoder decoder,
2   FieldWriter writer, int repetitionLevel):
3   Add current repetitionLevel and definition level to writer
4   seenFields = {} // empty set of integers
5   while decoder has more field values
6     FieldWriter chWriter =
7       child of writer for field read by decoder
8     int chRepetitionLevel = repetitionLevel
9     if set seenFields contains field ID of chWriter
10    chRepetitionLevel = tree depth of chWriter
11  else
12    Add field ID of chWriter to seenFields
13  end if
14  if chWriter corresponds to an atomic field
15    Write value of current field read by decoder
16    using chWriter at chRepetitionLevel
17  else
18    DissectRecord(new RecordDecoder for nested record
19      read by decoder, chWriter, chRepetitionLevel)
20  end if
21 end while
22 end procedure

```

Figure 16: Algorithm for dissecting a record into columns

## APPENDIX

### A. COLUMN-STRIPPING ALGORITHM

The algorithm for decomposing a record into columns is shown in Figure 16. Procedure `DissectRecord` is passed an instance of a `RecordDecoder`, which is used to traverse binary-encoded records. `FieldWriters` form a tree hierarchy isomorphic to that of the input schema. The root `FieldWriter` is passed to the algorithm for each new record, with `repetitionLevel` set to 0. The primary job of the `DissectRecord` procedure is to maintain the current `repetitionLevel`. The current `definitionLevel` is uniquely determined by the tree position of the current writer, as the sum of the number of optional and repeated fields in the field's path.

The while-loop of the algorithm (Line 5) iterates over all atomic and record-valued fields contained in a given record. The set `seenFields` tracks whether or not a field has been seen in the record. It is used to determine what field has repeated most recently. The child repetition level `chRepetitionLevel` is set to that of the most recently repeated field or else defaults to its parent's level (Lines 9-13). The procedure is invoked recursively on nested records (Line 18).

In Section 4.2 we sketched how `FieldWriters` accumulate levels and propagate them lazily to lower-level writers. This is done as follows: each non-leaf writer keeps a sequence of (repetition, definition) levels. Each writer also has a ‘version’ number associated with it. Simply stated, a writer version is incremented by one whenever a level is added. It is sufficient for children to remember the last parent’s version they synced. If a child writer ever gets its own (non-null) value, it synchronizes its state with the parent by fetching new levels, and only then adds the new data.

Because input data can have thousands of fields and millions of records, it is not feasible to store all levels in memory. Some levels may be temporarily stored in a file on disk. For a lossless encoding of empty (sub)records, non-atomic fields (such as `Name.Language` in Figure 2) may need to have column stripes of their own, containing only levels but no non-NULL values.

### B. RECORD ASSEMBLY ALGORITHM

In their on-the-wire representation, records are laid out as pairs of

```

1 Record AssembleRecord(FieldReaders[] readers):
2   record = create a new record
3   lastReader = select the root field reader in readers
4   reader = readers[0]
5   while reader has data
6     Fetch next value from reader
7     if current value is not NULL
8       MoveToLevel(tree level of reader, reader)
9       Append reader's value to record
10    else
11      MoveToLevel(full definition level of reader, reader)
12    end if
13    reader = reader that FSM transitions to
14      when reading next repetition level from reader
15    ReturnToLevel(tree level of reader)
16  end while
17  ReturnToLevel(0)
18  End all nested records
19  return record
20 end procedure
21
22 MoveToLevel(int newLevel, FieldReader nextReader):
23  End nested records up to the level of the lowest common ancestor
24  of lastReader and nextReader.
25  Start nested records from the level of the lowest common ancestor
26  up to newLevel.
27  Set lastReader to the one at newLevel.
28 end procedure
29
30 ReturnToLevel(int newLevel) {
31  End nested records up to newLevel.
32  Set lastReader to the one at newLevel.
33 end procedure

```

Figure 17: Algorithm for assembling a record from columns

a field identifier followed by a field value. Nested records can be thought of as having an ‘opening tag’ and a ‘closing tag’, similar to XML (actual binary encoding may differ, see [21] for details). In the following, writing opening tags is referred to as ‘starting’ the record, and writing closing tags is called ‘ending’ it.

`AssembleRecord` procedure takes as input a set of `FieldReaders` and (implicitly) the FSM with state transitions between the readers. Variable `reader` holds the current `FieldReader` in the main routine (Line 4). Variable `lastReader` holds the last reader whose value we appended to the record and is available to all three procedures shown in Figure 17. The main while-loop is at Line 5. We fetch the next value from the current reader. If the value is not NULL, which is determined by looking at its definition level, we synchronize the record being assembled to the record structure of the current reader in the method `MoveToLevel`, and append the field value to the record. Otherwise, we merely adjust the record structure without appending any value—which needs to be done if empty records are present. On Line 12, we use a ‘full definition level’. Recall that the definition level factors out required fields (only repeated and optional fields are counted). Full definition level takes all fields into account.

Procedure `MoveToLevel` transitions the record from the state of the `lastReader` to that of the `nextReader` (see Line 22). For example, suppose the `lastReader` corresponds to `Links.Backward` in Figure 2 and `nextReader` is `Name.Language.Code`. The method ends the nested record `Links` and starts new records `Name` and `Language`, in that order. Procedure `ReturnsToLevel` (Line 30) is a counterpart of `MoveToLevel` that only ends current records without starting any new ones.

```

1 procedure ConstructFSM(Field[] fields):
2 for each field in fields:
3   maxLevel = maximal repetition level of field
4   barrier = next field after field or final FSM state otherwise
5   barrierLevel = common repetition level of field and barrier
6   for each preField before field whose
7     repetition level is larger than barrierLevel:
8     backLevel = common repetition level of preField and field
9     Set transition (field, backLevel) -> preField
10  end for
11  for each level in [barrierLevel+1..maxLevel]
12    that lacks transition from field:
13    Copy transition's destination from that of level-1
14  end for
15  for each level in [0..barrierLevel]:
16    Set transition (field, level) -> barrier
17  end for
18 end for
19 end procedure

```

Figure 18: Algorithm to construct a record assembly automaton

## C. FSM CONSTRUCTION ALGORITHM

Figure 18 shows an algorithm for constructing a finite-state machine that performs record assembly. The algorithm takes as input the fields that should be populated in the records, in the order in which they appear in the schema. The algorithm uses a concept of a ‘common repetition level’ of two fields, which is the repetition level of their lowest common ancestor. For example, the common repetition level of `Links.Backward` and `Links.Forward` equals 1. The second concept is that of a ‘barrier’, which is the next field in the sequence after the current one. The intuition is that we try to process each field one by one until the barrier is hit and requires a jump to a previously seen field.

The algorithm consists of three steps. In Step 1 (Lines 6-10), we go through the common repetition levels backwards. These are guaranteed to be non-increasing. For each repetition level we encounter, we pick the left-most field in the sequence—that is the one we need to transition to when that repetition level is returned by a FieldReader. In Step 2, we fill the gaps (Lines 11-14). The gaps arise because not all repetition levels are present in the common repetition levels computed at Line 8. In Step 3 (Lines 15-17), we set transitions for all levels that are equal to or below the barrier level to jump to the barrier field. If a FieldReader produces such a level, we need to continue constructing the nested record and do not need to bounce off the barrier.

## D. SELECT-PROJECT-AGGREGATE EVALUATION ALGORITHM

Figure 19 shows the algorithm used for evaluating select-project-aggregate queries in Dremel. The algorithm addresses a general case when a query may reference repeated fields; a simpler optimized version is used for flat-relational queries, i.e., those referencing only required and optional fields. The algorithm has two implicit inputs: a set of FieldReaders, one for each field appearing in the query, and a set of scalar expressions, including aggregate expressions, present in the query. The repetition level of a scalar expression (used in Line 8) is determined as the maximum repetition level of the fields used in that expression.

In essence, the algorithm advances the readers in lockstep to the next set of values, and, if the selection conditions are met, emits the projected values. Selection and projection are controlled by two variables, `fetchLevel` and `selectLevel`. During execution, only

```

1 procedure Scan():
2   fetchLevel = 0
3   selectLevel = 0
4   while stopping conditions are not met:
5     Fetch()
6     if WHERE clause evaluates to true:
7       for each expression in SELECT clause:
8         if(repetition level of expression) >= selectLevel:
9           Emit value of expression
10    end if
11  end for
12  selectLevel = fetchLevel
13 else
14  selectLevel = min(selectLevel, fetchLevel)
15 end if
16 end while
17 end procedure
18
19 procedure Fetch():
20   nextLevel = 0
21   for each reader in field reader set:
22     if(next repetition level of reader) >= fetchLevel:
23       Advance reader to the next value
24     endif
25   nextLevel = max(nextLevel, next repetition level of reader)
26 end for
27 fetchLevel = nextLevel
28 end procedure

```

Figure 19: Algorithm for evaluating select-project-aggregate queries over columnar input, bypassing record assembly

readers whose next repetition level is no less than `fetchLevel` are advanced (see `Fetch` method at Line 19). In a similar vein, only expressions whose current repetition level is no less than `selectLevel` are emitted (Lines 7-10). The algorithm ensures that expressions at a higher-level of nesting, i.e., those having a smaller repetition level, get evaluated and emitted only once for each deeper nested expression.

# Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek

dpeng@google.com, fdabek@google.com

Google, Inc.

## Abstract

Updating an index of the web as documents are crawled requires continuously transforming a large repository of existing documents as new documents arrive. This task is one example of a class of data processing tasks that transform a large repository of data via small, independent mutations. These tasks lie in a gap between the capabilities of existing infrastructure. Databases do not meet the storage or throughput requirements of these tasks: Google’s indexing system stores tens of petabytes of data and processes billions of updates per day on thousands of machines. MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.

We have built Percolator, a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, we process the same number of documents per day, while reducing the average age of documents in Google search results by 50%.

## 1 Introduction

Consider the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. For example, if the same content is crawled under multiple URLs, only the URL with the highest PageRank [28] appears in the index. Each link is also inverted so that the anchor text from each outgoing link is attached to the page the link points to. Link inversion must work across duplicates: links to a duplicate of a page should be forwarded to the highest PageRank duplicate if necessary.

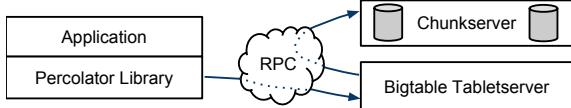
This is a bulk-processing task that can be expressed as a series of MapReduce [13] operations: one for clustering duplicates, one for link inversion, etc. It’s easy to maintain invariants since MapReduce limits the parallelism of the computation; all documents finish one processing step before starting the next. For example, when the indexing system is writing inverted links to the current highest-PageRank URL, we need not worry about its PageRank concurrently changing; a previous MapReduce step has already determined its PageRank.

Now, consider how to update that index after recrawling some small portion of the web. It’s not sufficient to run the MapReduces over just the new pages since, for example, there are links between the new pages and the rest of the web. The MapReduces must be run again over the entire repository, that is, over both the new pages and the old pages. Given enough computing resources, MapReduce’s scalability makes this approach feasible, and, in fact, Google’s web search index was produced in this way prior to the work described here. However, reprocessing the entire web discards the work done in earlier runs and makes latency proportional to the size of the repository, rather than the size of an update.

The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants. However, existing DBMSs can’t handle the sheer volume of data: Google’s indexing system stores tens of petabytes across thousands of machines [30]. Distributed storage systems like Bigtable [9] can scale to the size of our repository but don’t provide tools to help programmers maintain data invariants in the face of concurrent updates.

An ideal data processing system for the task of maintaining the web search index would be optimized for *incremental processing*; that is, it would allow us to maintain a very large repository of documents and update it efficiently as each new document was crawled. Given that the system will be processing many small updates concurrently, an ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

The remainder of this paper describes a particular incremental processing system: Percolator. Percolator provides the user with random access to a multi-PB repository. Random access allows us to process documents in-



**Figure 1:** Percolator and its dependencies

dividually, avoiding the global scans of the repository that MapReduce requires. To achieve high throughput, many threads on many machines need to transform the repository concurrently, so Percolator provides ACID-compliant transactions to make it easier for programmers to reason about the state of the repository; we currently implement snapshot isolation semantics [5].

In addition to reasoning about concurrency, programmers of an incremental system need to keep track of the state of the incremental computation. To assist them in this task, Percolator provides observers: pieces of code that are invoked by the system whenever a user-specified column changes. Percolator applications are structured as a series of observers; each observer completes a task and creates more work for “downstream” observers by writing to the table. An external process triggers the first observer in the chain by writing initial data into the table.

Percolator was built specifically for incremental processing and is not intended to supplant existing solutions for most data processing tasks. Computations where the result can’t be broken down into small updates (sorting a file, for example) are better handled by MapReduce. Also, the computation should have strong consistency requirements; otherwise, Bigtable is sufficient. Finally, the computation should be very large in some dimension (total data size, CPU required for transformation, etc.); smaller computations not suited to MapReduce or Bigtable can be handled by traditional DBMSs.

Within Google, the primary application of Percolator is preparing web pages for inclusion in the live web search index. By converting the indexing system to an incremental system, we are able to process individual documents as they are crawled. This reduced the average document processing latency by a factor of 100, and the average age of a document appearing in a search result dropped by nearly 50 percent (the age of a search result includes delays other than indexing such as the time between a document being changed and being crawled). The system has also been used to render pages into images; Percolator tracks the relationship between web pages and the resources they depend on, so pages can be reprocessed when any depended-upon resources change.

## 2 Design

Percolator provides two main abstractions for performing incremental processing at large scale: ACID transactions over a random-access repository and ob-

servers, a way to organize an incremental computation.

A Percolator system consists of three binaries that run on every machine in the cluster: a Percolator worker, a Bigtable [9] tablet server, and a GFS [20] chunkserver. All observers are linked into the Percolator worker, which scans the Bigtable for changed columns (“notifications”) and invokes the corresponding observers as a function call in the worker process. The observers perform transactions by sending read/write RPCs to Bigtable tablet servers, which in turn send read/write RPCs to GFS chunkservers. The system also depends on two small services: the timestamp oracle and the lightweight lock service. The timestamp oracle provides strictly increasing timestamps: a property required for correct operation of the snapshot isolation protocol. Workers use the lightweight lock service to make the search for dirty notifications more efficient.

From the programmer’s perspective, a Percolator repository consists of a small number of tables. Each table is a collection of “cells” indexed by row and column. Each cell contains a value: an uninterpreted array of bytes. (Internally, to support snapshot isolation, we represent each cell as a series of values indexed by timestamp.)

The design of Percolator was influenced by the requirement to run at massive scales and the lack of a requirement for extremely low latency. Relaxed latency requirements let us take, for example, a lazy approach to cleaning up locks left behind by transactions running on failed machines. This lazy, simple-to-implement approach potentially delays transaction commit by tens of seconds. This delay would not be acceptable in a DBMS running OLTP tasks, but it is tolerable in an incremental processing system building an index of the web. Percolator has no central location for transaction management; in particular, it lacks a global deadlock detector. This increases the latency of conflicting transactions but allows the system to scale to thousands of machines.

### 2.1 Bigtable overview

Percolator is built on top of the Bigtable distributed storage system. Bigtable presents a multi-dimensional sorted map to users: keys are (row, column, timestamp) tuples. Bigtable provides lookup and update operations on each row, and Bigtable row transactions enable atomic read-modify-write operations on individual rows. Bigtable handles petabytes of data and runs reliably on large numbers of (unreliable) machines.

A running Bigtable consists of a collection of tablet servers, each of which is responsible for serving several tablets (contiguous regions of the key space). A master coordinates the operation of tablet servers by, for example, directing them to load or unload tablets. A tablet is stored as a collection of read-only files in the Google

SSTable format. SSTables are stored in GFS; Bigtable relies on GFS to preserve data in the event of disk loss. Bigtable allows users to control the performance characteristics of the table by grouping a set of columns into a locality group. The columns in each locality group are stored in their own set of SSTables, which makes scanning them less expensive since the data in other columns need not be scanned.

The decision to build on Bigtable defined the overall shape of Percolator. Percolator maintains the gist of Bigtable’s interface: data is organized into Bigtable rows and columns, with Percolator metadata stored alongside in special columns (see Figure 5). Percolator’s API closely resembles Bigtable’s API: the Percolator library largely consists of Bigtable operations wrapped in Percolator-specific computation. The challenge, then, in implementing Percolator is providing the features that Bigtable does not: multirow transactions and the observer framework.

## 2.2 Transactions

Percolator provides cross-row, cross-table transactions with ACID snapshot-isolation semantics. Percolator users write their transaction code in an imperative language (currently C++) and mix calls to the Percolator API with their code. Figure 2 shows a simplified version of clustering documents by a hash of their contents. In this example, if Commit() returns false, the transaction has conflicted (in this case, because two URLs with the same content hash were processed simultaneously) and should be retried after a backoff. Calls to Get() and Commit() are blocking; parallelism is achieved by running many transactions simultaneously in a thread pool.

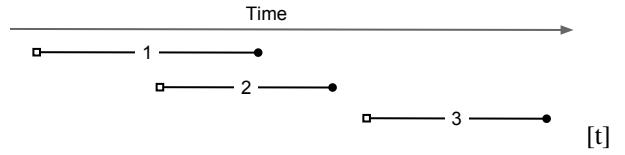
While it is possible to incrementally process data without the benefit of strong transactions, transactions make it more tractable for the user to reason about the state of the system and to avoid the introduction of errors into a long-lived repository. For example, in a transactional web-indexing system the programmer can make assumptions like: the hash of the contents of a document is always consistent with the table that indexes duplicates. Without transactions, an ill-timed crash could result in a permanent error: an entry in the document table that corresponds to no URL in the duplicates table. Transactions also make it easy to build index tables that are always up to date and consistent. Note that both of these examples require transactions that span rows, rather than the single-row transactions that Bigtable already provides.

Percolator stores multiple versions of each data item using Bigtable’s timestamp dimension. Multiple versions are required to provide snapshot isolation [5], which presents each transaction with the appearance of reading from a stable snapshot at some timestamp. Writes appear in a different, later, timestamp. Snapshot isolation pro-

```
bool UpdateDocument(Document doc) {
    Transaction t(&cluster);
    t.Set(doc.url(), "contents", "document", doc.contents());
    int hash = Hash(doc.contents());

    // dups table maps hash → canonical URL
    string canonical;
    if (!t.Get(hash, "canonical-url", "dups", &canonical)) {
        // No canonical yet; write myself in
        t.Set(hash, "canonical-url", "dups", doc.url());
    } // else this document already exists, ignore new copy
    return t.Commit();
}
```

**Figure 2:** Example usage of the Percolator API to perform basic checksum clustering and eliminate documents with the same content.



**Figure 3:** Transactions under snapshot isolation perform reads at a start timestamp (represented here by an open square) and writes at a commit timestamp (closed circle). In this example, transaction 2 would not see writes from transaction 1 since transaction 2’s start timestamp is before transaction 1’s commit timestamp. Transaction 3, however, will see writes from both 1 and 2. Transaction 1 and 2 are running concurrently: if they both write the same cell, at least one will abort.

tects against write-write conflicts: if transactions A and B, running concurrently, write to the same cell, at most one will commit. Snapshot isolation does not provide serializability; in particular, transactions running under snapshot isolation are subject to write skew [5]. The main advantage of snapshot isolation over a serializable protocol is more efficient reads. Because any timestamp represents a consistent snapshot, reading a cell requires only performing a Bigtable lookup at the given timestamp; acquiring locks is not necessary. Figure 3 illustrates the relationship between transactions under snapshot isolation.

Because it is built as a client library accessing Bigtable, rather than controlling access to storage itself, Percolator faces a different set of challenges implementing distributed transactions than traditional PDBMSs. Other parallel databases integrate locking into the system component that manages access to the disk: since each node already mediates access to data on the disk it can grant locks on requests and deny accesses that violate locking requirements.

By contrast, any node in Percolator can (and does) issue requests to directly modify state in Bigtable: there is no convenient place to intercept traffic and assign locks. As a result, Percolator must explicitly maintain locks. Locks must persist in the face of machine failure; if a lock could disappear between the two phases of com-

key	bal:data	bal:lock	bal:write
Bob	6: 5: \$10	6: 5:	6: data @ 5 5:
Joe	6: 5: \$2	6: 5:	6: data @ 5 5:

1. Initial state: Joe's account contains \$2 dollars, Bob's \$10.

	7:\$3	7: <b>I am primary</b>	7: 6: data @ 5
Bob	6: 5: \$10	6: 5:	5:
Joe	6: 5: \$2	6: 5:	6: data @ 5 5:

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

	7: \$3 6: 5: \$10	7: I am primary 6: 5:	7: 6: data @ 5 5:
Bob	7: <b>\$9</b> 6: 5: \$2	7: <b>primary @ Bob.bal</b> 6: 5:	7: 6: data @ 5 5:
Joe			

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

	8: 7: \$3 6: 5: \$10	8: 7: 6: 5:	8: <b>data @ 7</b> 7: 6: data @ 5 5:
Bob	7: \$9 6: 5: \$2	7: primary @ Bob.bal 6: 5:	7: 6: data @ 5 5:
Joe			

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value \$3.

	8: 7: \$3 6: 5: \$10	8: 7: 6: 5:	8: data @ 7 7: 6: data @ 5 5:
Bob	8: 7: \$9 6: 5: \$2	8: 7: 6: 5:	8: <b>data @ 7</b> 7: 6: data @ 5 5:
Joe			

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

**Figure 4:** This figure shows the Bigtable writes performed by a Percolator transaction that mutates two rows. The transaction transfers 7 dollars from Bob to Joe. Each Percolator column is stored as 3 Bigtable columns: data, write metadata, and lock metadata. Bigtable's timestamp dimension is shown within each cell; 12: "data" indicates that "data" has been written at Bigtable timestamp 12. Newly written data is shown in boldface.

Column	Use
<b>c:lock</b>	An uncommitted transaction is writing this cell; contains the location of primary lock
<b>c:write</b>	Committed data present; stores the Bigtable timestamp of the data
<b>c:data</b>	Stores the data itself
<b>c:notify</b>	Hint: observers may need to run
<b>c:ack_O</b>	Observer "O" has run ; stores start timestamp of successful last run

**Figure 5:** The columns in the Bigtable representation of a Percolator column named "c."

mit, the system could mistakenly commit two transactions that should have conflicted. The lock service must provide high throughput; thousands of machines will be requesting locks simultaneously. The lock service should also be low-latency; each Get() operation requires reading locks in addition to data, and we prefer to minimize this latency. Given these requirements, the lock server will need to be replicated (to survive failure), distributed and balanced (to handle load), and write to a persistent data store. Bigtable itself satisfies all of our requirements, and so Percolator stores its locks in special in-memory columns in the same Bigtable that stores data and reads or modifies the locks in a Bigtable row transaction when accessing data in that row.

We'll now consider the transaction protocol in more detail. Figure 6 shows the pseudocode for Percolator transactions, and Figure 4 shows the layout of Percolator data and metadata during the execution of a transaction. These various metadata columns used by the system are described in Figure 5. The transaction's constructor asks the timestamp oracle for a start timestamp (line 6), which determines the consistent snapshot seen by Get(). Calls to Set() are buffered (line 7) until commit time. The basic approach for committing buffered writes is two-phase commit, which is coordinated by the client. Transactions on different machines interact through row transactions on Bigtable tablet servers.

In the first phase of commit ("prewrite"), we try to lock all the cells being written. (To handle client failure, we designate one lock arbitrarily as the primary; we'll discuss this mechanism below.) The transaction reads metadata to check for conflicts in each cell being written. There are two kinds of conflicting metadata: if the transaction sees another write record after its start timestamp, it aborts (line 32); this is the write-write conflict that snapshot isolation guards against. If the transaction sees another lock at any timestamp, it also aborts (line 34). It's possible that the other transaction is just being slow to release its lock after having already committed below our start timestamp, but we consider this unlikely, so we abort. If there is no conflict, we write the lock and

```

1 class Transaction {
2     struct Write { Row row; Column col; string value; };
3     vector<Write> writes_;
4     int start_ts_;
5
6     Transaction() : start_ts_(oracle.GetTimestamp()) {}
7     void Set(Write w) { writes_.push_back(w); }
8     bool Get(Row row, Column c, string* value) {
9         while (true) {
10             bigtable::Txn T = bigtable::StartRowTransaction(row);
11             // Check for locks that signal concurrent writes.
12             if (T.Read(row, c+"lock", [0, start_ts_])) {
13                 // There is a pending lock; try to clean it and wait
14                 BackoffAndMaybeCleanupLock(row, c);
15                 continue;
16             }
17
18             // Find the latest write below our start_timestamp.
19             latest_write = T.Read(row, c+"write", [0, start_ts_]);
20             if (!latest_write.found()) return false; // no data
21             int data_ts = latest_write.start_timestamp();
22             *value = T.Read(row, c+"data", [data_ts, data_ts]);
23             return true;
24         }
25     }
26     // Prewrite tries to lock cell w, returning false in case of conflict.
27     bool Prewrite(Write w, Write primary) {
28         Column c = w.col;
29         bigtable::Txn T = bigtable::StartRowTransaction(w.row);
30
31         // Abort on writes after our start timestamp ...
32         if (T.Read(w.row, c+"write", [start_ts_, infinity])) return false;
33         // ... or locks at any timestamp.
34         if (T.Read(w.row, c+"lock", [0, infinity])) return false;
35
36         T.Write(w.row, c+"data", start_ts_, w.value);
37         T.Write(w.row, c+"lock", start_ts_,
38             {primary.row, primary.col}); // The primary's location.
39         return T.Commit();
40     }
41     bool Commit() {
42         Write primary = writes_[0];
43         vector<Write> secondaries(writes_.begin()+1, writes_.end());
44         if (!Prewrite(primary, primary)) return false;
45         for (Write w : secondaries)
46             if (!Prewrite(w, primary)) return false;
47
48         int commit_ts = oracle_.GetTimestamp();
49
50         // Commit primary first.
51         Write p = primary;
52         bigtable::Txn T = bigtable::StartRowTransaction(p.row);
53         if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
54             return false; // aborted while working
55         T.Write(p.row, p.col+"write", commit_ts,
56             start_ts_); // Pointer to data written at start_ts_
57         T.Erase(p.row, p.col+"lock", commit_ts);
58         if (!T.Commit()) return false; // commit point
59
60         // Second phase: write out write records for secondary cells.
61         for (Write w : secondaries) {
62             bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
63             bigtable::Erase(w.row, w.col+"lock", commit_ts);
64         }
65         return true;
66     }
67 } // class Transaction

```

**Figure 6:** Pseudocode for Percolator transaction protocol.

the data to each cell at the start timestamp (lines 36-38).

If no cells conflict, the transaction may commit and proceeds to the second phase. At the beginning of the second phase, the client obtains the commit timestamp from the timestamp oracle (line 48). Then, at each cell (starting with the primary), the client releases its lock and make its write visible to readers by replacing the lock with a write record. The write record indicates to readers that committed data exists in this cell; it contains a pointer to the start timestamp where readers can find the actual data. Once the primary's write is visible (line 58), the transaction must commit since it has made a write visible to readers.

A Get() operation first checks for a lock in the timestamp range  $[0, \text{start\_timestamp}]$ , which is the range of timestamps visible in the transaction's snapshot (line 12). If a lock is present, another transaction is concurrently writing this cell, so the reading transaction must wait until the lock is released. If no conflicting lock is found, Get() reads the latest write record in that timestamp range (line 19) and returns the data item corresponding to that write record (line 22).

Transaction processing is complicated by the possibility of client failure (tablet server failure does not affect the system since Bigtable guarantees that written locks persist across tablet server failures). If a client fails while a transaction is being committed, locks will be left behind. Percolator must clean up those locks or they will cause future transactions to hang indefinitely. Percolator takes a lazy approach to cleanup: when a transaction A encounters a conflicting lock left behind by transaction B, A may determine that B has failed and erase its locks.

It is very difficult for A to be perfectly confident in its judgment that B is failed; as a result we must avoid a race between A cleaning up B's transaction and a not-actually-failed B committing the same transaction. Percolator handles this by designating one cell in every transaction as a synchronizing point for any commit or cleanup operations. This cell's lock is called the primary lock. Both A and B agree on which lock is primary (the location of the primary is written into the locks at all other cells). Performing either a cleanup or commit operation requires modifying the primary lock; since this modification is performed under a Bigtable row transaction, only one of the cleanup or commit operations will succeed. Specifically: before B commits, it must check that it still holds the primary lock and replace it with a write record. Before A erases B's lock, A must check the primary to ensure that B has not committed; if the primary lock is still present, then it can safely erase the lock.

When a client crashes during the second phase of commit, a transaction will be past the commit point (it has written at least one write record) but will still

have locks outstanding. We must perform roll-forward on these transactions. A transaction that encounters a lock can distinguish between the two cases by inspecting the primary lock: if the primary lock has been replaced by a write record, the transaction which wrote the lock must have committed and the lock must be rolled forward, otherwise it should be rolled back (since we always commit the primary first, we can be sure that it is safe to roll back if the primary is not committed). To roll forward, the transaction performing the cleanup replaces the stranded lock with a write record as the original transaction would have done.

Since cleanup is synchronized on the primary lock, it is safe to clean up locks held by live clients; however, this incurs a performance penalty since rollback forces the transaction to abort. So, a transaction will not clean up a lock unless it suspects that a lock belongs to a dead or stuck worker. Percolator uses simple mechanisms to determine the liveness of another transaction. Running workers write a token into the Chubby lockservice [8] to indicate they belong to the system; other workers can use the existence of this token as a sign that the worker is alive (the token is automatically deleted when the process exits). To handle a worker that is live, but not working, we additionally write the wall time into the lock; a lock that contains a too-old wall time will be cleaned up even if the worker’s liveness token is valid. To handle long-running commit operations, workers periodically update this wall time while committing.

### 2.3 Timestamps

The timestamp oracle is a server that hands out timestamps in strictly increasing order. Since every transaction requires contacting the timestamp oracle twice, this service must scale well. The oracle periodically allocates a range of timestamps by writing the highest allocated timestamp to stable storage; given an allocated range of timestamps, the oracle can satisfy future requests strictly from memory. If the oracle restarts, the timestamps will jump forward to the maximum allocated timestamp (but will never go backwards). To save RPC overhead (at the cost of increasing transaction latency) each Percolator worker batches timestamp requests across transactions by maintaining only one pending RPC to the oracle. As the oracle becomes more loaded, the batching naturally increases to compensate. Batching increases the scalability of the oracle but does not affect the timestamp guarantees. Our oracle serves around 2 million timestamps per second from a single machine.

The transaction protocol uses strictly increasing timestamps to guarantee that `Get()` returns all committed writes before the transaction’s start timestamp. To see how it provides this guarantee, consider a transaction R reading at timestamp  $T_R$  and a transaction W that com-

mitted at timestamp  $T_W < T_R$ ; we will show that R sees W’s writes. Since  $T_W < T_R$ , we know that the timestamp oracle gave out  $T_W$  before or in the same batch as  $T_R$ ; hence, W requested  $T_W$  before R received  $T_R$ . We know that R can’t do reads before receiving its start timestamp  $T_R$  and that W wrote locks before requesting its commit timestamp  $T_W$ . Therefore, the above property guarantees that W must have at least written all its locks before R did any reads; R’s `Get()` will see either the fully-committed write record or the lock, in which case W will block until the lock is released. Either way, W’s write is visible to R’s `Get()`.

### 2.4 Notifications

Transactions let the user mutate the table while maintaining invariants, but users also need a way to trigger and run the transactions. In Percolator, the user writes code (“observers”) to be triggered by changes to the table, and we link all the observers into a binary running alongside every tablet server in the system. Each observer registers a function and a set of columns with Percolator, and Percolator invokes the function after data is written to one of those columns in any row.

Percolator applications are structured as a series of observers; each observer completes a task and creates more work for “downstream” observers by writing to the table. In our indexing system, a MapReduce loads crawled documents into Percolator by running loader transactions, which trigger the document processor transaction to index the document (parse, extract links, etc.). The document processor transaction triggers further transactions like clustering. The clustering transaction, in turn, triggers transactions to export changed document clusters to the serving system.

Notifications are similar to database triggers or events in active databases [29], but unlike database triggers, they cannot be used to maintain database invariants. In particular, the triggered observer runs in a separate transaction from the triggering write, so the triggering write and the triggered observer’s writes are not atomic. Notifications are intended to help structure an incremental computation rather than to help maintain data integrity.

This difference in semantics and intent makes observer behavior much easier to understand than the complex semantics of overlapping triggers. Percolator applications consist of very few observers — the Google indexing system has roughly 10 observers. Each observer is explicitly constructed in the `main()` of the worker binary, so it is clear what observers are active. It is possible for several observers to observe the same column, but we avoid this feature so it is clear what observer will run when a particular column is written. Users do need to be wary about infinite cycles of notifications, but Percolator does nothing to prevent this; the user typically constructs

a series of observers to avoid infinite cycles.

We do provide one guarantee: at most one observer’s transaction will commit for each change of an observed column. The converse is not true, however: multiple writes to an observed column may cause the corresponding observer to be invoked only once. We call this feature message collapsing, since it helps avoid computation by amortizing the cost of responding to many notifications. For example, it is sufficient for <http://google.com> to be reprocessed periodically rather than every time we discover a new link pointing to it.

To provide these semantics for notifications, each observed column has an accompanying “acknowledgment” column for each observer, containing the latest start timestamp at which the observer ran. When the observed column is written, Percolator starts a transaction to process the notification. The transaction reads the observed column and its corresponding acknowledgment column. If the observed column was written after its last acknowledgment, then we run the observer and set the acknowledgment column to our start timestamp. Otherwise, the observer has already been run, so we do not run it again. Note that if Percolator accidentally starts two transactions concurrently for a particular notification, they will both see the dirty notification and run the observer, but one will abort because they will conflict on the acknowledgment column. We promise that at most one observer will *commit* for each notification.

To implement notifications, Percolator needs to efficiently find dirty cells with observers that need to be run. This search is complicated by the fact that notifications are rare: our table has trillions of cells, but, if the system is keeping up with applied load, there will only be millions of notifications. Additionally, observer code is run on a large number of client processes distributed across a collection of machines, meaning that this search for dirty cells must be distributed.

To identify dirty cells, Percolator maintains a special “notify” Bigtable column, containing an entry for each dirty cell. When a transaction writes an observed cell, it also sets the corresponding notify cell. The workers perform a distributed scan over the notify column to find dirty cells. After the observer is triggered and the transaction commits, we remove the notify cell. Since the notify column is just a Bigtable column, not a Percolator column, it has no transactional properties and serves only as a hint to the scanner to check the acknowledgment column to determine if the observer should be run.

To make this scan efficient, Percolator stores the notify column in a separate Bigtable locality group so that scanning over the column requires reading only the millions of dirty cells rather than the trillions of total data cells. Each Percolator worker dedicates several threads to the scan. For each thread, the worker chooses a portion of the

table to scan by first picking a random Bigtable tablet, then picking a random key in the tablet, and finally scanning the table from that position. Since each worker is scanning a random region of the table, we worry about two workers running observers on the same row concurrently. While this behavior will not cause correctness problems due to the transactional nature of notifications, it is inefficient. To avoid this, each worker acquires a lock from a lightweight lock service before scanning the row. This lock server need not persist state since it is advisory and thus is very scalable.

The random-scanning approach requires one additional tweak: when it was first deployed we noticed that scanning threads would tend to clump together in a few regions of the table, effectively reducing the parallelism of the scan. This phenomenon is commonly seen in public transportation systems where it is known as “platooning” or “bus clumping” and occurs when a bus is slowed down (perhaps by traffic or slow loading). Since the number of passengers at each stop grows with time, loading delays become even worse, further slowing the bus. Simultaneously, any bus behind the slow bus speeds up as it needs to load fewer passengers at each stop. The result is a clump of buses arriving simultaneously at a stop [19]. Our scanning threads behaved analogously: a thread that was running observers slowed down while threads “behind” it quickly skipped past the now-clean rows to clump with the lead thread and failed to pass the lead thread because the clump of threads overloaded tablet servers. To solve this problem, we modified our system in a way that public transportation systems cannot: when a scanning thread discovers that it is scanning the same row as another thread, it chooses a new random location in the table to scan. To further the transportation analogy, the buses (scanner threads) in our city avoid clumping by teleporting themselves to a random stop (location in the table) if they get too close to the bus in front of them.

Finally, experience with notifications led us to introduce a lighter-weight but semantically weaker notification mechanism. We found that when many duplicates of the same page were processed concurrently, each transaction would conflict trying to trigger reprocessing of the same duplicate cluster. This led us to devise a way to notify a cell without the possibility of transactional conflict. We implement this weak notification by writing *only* to the Bigtable “notify” column. To preserve the transactional semantics of the rest of Percolator, we restrict these weak notifications to a special type of column that cannot be written, only notified. The weaker semantics also mean that multiple observers may run and commit as a result of a single weak notification (though the system tries to minimize this occurrence). This has become an important feature for managing conflicts; if an observer

frequently conflicts on a hotspot, it often helps to break it into two observers connected by a non-transactional notification on the hotspot.

## 2.5 Discussion

One of the inefficiencies of Percolator relative to a MapReduce-based system is the number of RPCs sent per work-unit. While MapReduce does a single large read to GFS and obtains all of the data for 10s or 100s of web pages, Percolator performs around 50 individual Bigtable operations to process a single document.

One source of additional RPCs occurs during commit. When writing a lock, we must do a read-modify-write operation requiring two Bigtable RPCs: one to read for conflicting locks or writes and another to write the new lock. To reduce this overhead, we modified the Bigtable API by adding conditional mutations which implements the read-modify-write step in a single RPC. Many conditional mutations destined for the same tablet server can also be batched together into a single RPC to further reduce the total number of RPCs we send. We create batches by delaying lock operations for several seconds to collect them into batches. Because locks are acquired in parallel, this adds only a few seconds to the latency of each transaction; we compensate for the additional latency with greater parallelism. Batching also increases the time window in which conflicts may occur, but in our low-contention environment this has not proved to be a problem.

We also perform the same batching when reading from the table: every read operation is delayed to give it a chance to form a batch with other reads to the same tablet server. This delays each read, potentially greatly increasing transaction latency. A final optimization mitigates this effect, however: prefetching. Prefetching takes advantage of the fact that reading two or more values in the same row is essentially the same cost as reading one value. In either case, Bigtable must read the entire SSTable block from the file system and decompress it. Percolator attempts to predict, each time a column is read, what other columns in a row will be read later in the transaction. This prediction is made based on past behavior. Prefetching, combined with a cache of items that have already been read, reduces the number of Bigtable reads the system would otherwise do by a factor of 10.

Early in the implementation of Percolator, we decided to make all API calls blocking and rely on running thousands of threads per machine to provide enough parallelism to maintain good CPU utilization. We chose this thread-per-request model mainly to make application code easier to write, compared to the event-driven model. Forcing users to bundle up their state each of the (many) times they fetched a data item from the table would have made application development much more difficult. Our

experience with thread-per-request was, on the whole, positive: application code is simple, we achieve good utilization on many-core machines, and crash debugging is simplified by meaningful and complete stack traces. We encountered fewer race conditions in application code than we feared. The biggest drawbacks of the approach were scalability issues in the Linux kernel and Google infrastructure related to high thread counts. Our in-house kernel development team was able to deploy fixes to address the kernel issues.

## 3 Evaluation

Percolator lies somewhere in the performance space between MapReduce and DBMSs. For example, because Percolator is a distributed system, it uses far more resources to process a fixed amount of data than a traditional DBMS would; this is the cost of its scalability. Compared to MapReduce, Percolator can process data with far lower latency, but again, at the cost of additional resources required to support random lookups. These are engineering tradeoffs which are difficult to quantify: how much of an efficiency loss is too much to pay for the ability to add capacity endlessly simply by purchasing more machines? Or: how does one trade off the reduction in development time provided by a layered system against the corresponding decrease in efficiency?

In this section we attempt to answer some of these questions by first comparing Percolator to batch processing systems via our experiences with converting a MapReduce-based indexing pipeline to use Percolator. We'll also evaluate Percolator with microbenchmarks and a synthetic workload based on the well-known TPC-E benchmark [1]; this test will give us a chance to evaluate the scalability and efficiency of Percolator relative to Bigtable and DBMSs.

All of the experiments in this section are run on a subset of the servers in a Google data center. The servers run the Linux operating system on x86 processors; each machine is connected to several commodity SATA drives.

### 3.1 Converting from MapReduce

We built Percolator to create Google's large “base” index, a task previously performed by MapReduce. In our previous system, each day we crawled several billion documents and fed them along with a repository of existing documents through a series of 100 MapReduces. The result was an index which answered user queries. Though not all 100 MapReduces were on the critical path for every document, the organization of the system as a series of MapReduces meant that each document spent 2-3 days being indexed before it could be returned as a search result.

The Percolator-based indexing system (known as Caffeine [25]), crawls the same number of documents,

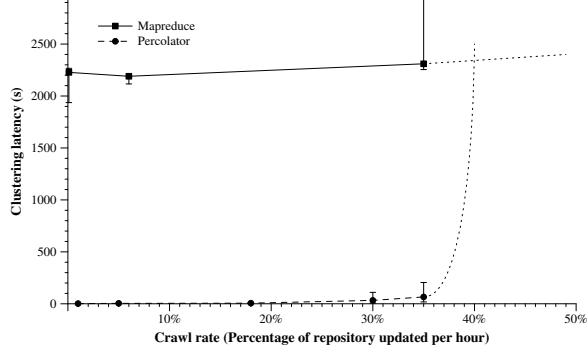
but we feed each document through Percolator as it is crawled. The immediate advantage, and main design goal, of Caffeine is a reduction in latency: the median document moves through Caffeine over 100x faster than the previous system. This latency improvement grows as the system becomes more complex: adding a new clustering phase to the Percolator-based system requires an extra lookup for each document rather than an extra scan over the repository. Additional clustering phases can also be implemented in the same transaction rather than in another MapReduce; this simplification is one reason the number of observers in Caffeine (10) is far smaller than the number of MapReduces in the previous system (100). This organization also allows for the possibility of performing additional processing on only a subset of the repository without rescanning the entire repository.

Adding additional clustering phases isn't free in an incremental system: more resources are required to make sure the system keeps up with the input, but this is still an improvement over batch processing systems where no amount of resources can overcome delays introduced by stragglers in an additional pass over the repository. Caffeine is essentially immune to stragglers that were a serious problem in our batch-based indexing system because the bulk of the processing does not get held up by a few very slow operations. The radically-lower latency of the new system also enables us to remove the rigid distinctions between large, slow-to-update indexes and smaller, more rapidly updated indexes. Because Percolator frees us from needing to process the repository each time we index documents, we can also make it larger: Caffeine's document collection is currently 3x larger than the previous system's and is limited only by available disk space.

Compared to the system it replaced, Caffeine uses roughly twice as many resources to process the same crawl rate. However, Caffeine makes good use of the extra resources. If we were to run the old indexing system with twice as many resources, we could either increase the index size or reduce latency by at most a factor of two (but not do both). On the other hand, if Caffeine were run with half the resources, it would not be able to process as many documents per day as the old system (but the documents it did produce would have much lower latency).

The new system is also easier to operate. Caffeine has far fewer moving parts: we run tablet servers, Percolator workers, and chunkservers. In the old system, each of a hundred different MapReduces needed to be individually configured and could independently fail. Also, the "peaky" nature of the MapReduce workload made it hard to fully utilize the resources of a datacenter compared to Percolator's much smoother resource usage.

The simplicity of writing straight-line code and the ability to do random lookups into the repository makes developing new features for Percolator easy. Under



**Figure 7:** Median document clustering delay for Percolator (dashed line) and MapReduce (solid line). For MapReduce, all documents finish processing at the same time and error bars represent the min, median, and max of three runs of the clustering MapReduce. For Percolator, we are able to measure the delay of individual documents, so the error bars represent the 5th- and 95th-percentile delay on a per-document level.

MapReduce, random lookups are awkward and costly. On the other hand, Caffeine developers need to reason about concurrency where it did not exist in the MapReduce paradigm. Transactions help deal with this concurrency, but can't fully eliminate the added complexity.

To quantify the benefits of moving from MapReduce to Percolator, we created a synthetic benchmark that clusters newly crawled documents against a billion-document repository to remove duplicates in much the same way Google's indexing pipeline operates. Documents are clustered by three clustering keys. In a real system, the clustering keys would be properties of the document like redirect target or content hash, but in this experiment we selected them uniformly at random from a collection of 750M possible keys. The average cluster in our synthetic repository contains 3.3 documents, and 93% of the documents are in a non-singleton cluster. This distribution of keys exercises the clustering logic, but does not expose it to the few extremely large clusters we have seen in practice. These clusters only affect the latency tail and not the results we present here. In the Percolator clustering implementation, each crawled document is immediately written to the repository to be clustered by an observer. The observer maintains an index table for each clustering key and compares the document against each index to determine if it is a duplicate (an elaboration of Figure 2). MapReduce implements clustering of continually arriving documents by repeatedly running a sequence of three clustering MapReduces (one for each clustering key). The sequence of three MapReduces processes the entire repository and any crawled documents that accumulated while the previous three were running.

This experiment simulates clustering documents crawled at a uniform rate. Whether MapReduce or Percolator performs better under this metric is a function of the how frequently documents are crawled (the crawl rate)

and the repository size. We explore this space by fixing the size of the repository and varying the rate at which new documents arrive, expressed as a percentage of the repository crawled per hour. In a practical system, a very small percentage of the repository would be crawled per hour: there are over 1 trillion web pages on the web (and ideally in an indexing system’s repository), far too many to crawl a reasonable fraction of in a single day. When the new input is a small fraction of the repository (low crawl rate), we expect Percolator to outperform MapReduce since MapReduce must map over the (large) repository to cluster the (small) batch of new documents while Percolator does work proportional only to the small batch of newly arrived documents (a lookup in up to three index tables per document). At very large crawl rates where the number of newly crawled documents approaches the size of the repository, MapReduce will perform better than Percolator. This cross-over occurs because streaming data from disk is much cheaper, per byte, than performing random lookups. At the cross-over the total cost of the lookups required to cluster the new documents under Percolator equals the cost to stream the documents and the repository through MapReduce. At crawl rates higher than that, one is better off using MapReduce.

We ran this benchmark on 240 machines and measured the median delay between when a document is crawled and when it is clustered. Figure 7 plots the median latency of document processing for both implementations as a function of crawl rate. When the crawl rate is low, Percolator clusters documents faster than MapReduce as expected; this scenario is illustrated by the leftmost pair of points which correspond to crawling 1 percent of documents per hour. MapReduce requires approximately 20 minutes to cluster the documents because it takes 20 minutes just to process the repository through the three MapReduces (the effect of the few newly crawled documents on the runtime is negligible). This results in an average delay between crawling a document and clustering of around 30 minutes: a random document waits 10 minutes after being crawled for the previous sequence of MapReduces to finish and then spends 20 minutes being processed by the three MapReduces. Percolator, on the other hand, finds a newly loaded document and processes it in two seconds on average, or about 1000x faster than MapReduce. The two seconds includes the time to find the dirty notification and run the transaction that performs the clustering. Note that this 1000x latency improvement could be made arbitrarily large by increasing the size of the repository.

As the crawl rate increases, MapReduce’s processing time grows correspondingly. Ideally, it would be proportional to the combined size of the repository and the input which grows with the crawl rate. In practice, the running time of a small MapReduce like this is limited by strag-

	Bigtable	Percolator	Relative
Read/s	15513	14590	0.94
Write/s	31003	7232	0.23

**Figure 8:** The overhead of Percolator operations relative to Bigtable. Write overhead is due to additional operations Percolator needs to check for conflicts.

glers, so the growth in processing time (and thus clustering latency) is only weakly correlated to crawl rate at low crawl rates. The 6 percent crawl rate, for example, only adds 150GB to a 1TB data set; the extra time to process 150GB is in the noise. The latency of Percolator is relatively unchanged as the crawl rate grows until it suddenly increases to effectively infinity at a crawl rate of 40% per hour. At this point, Percolator saturates the resources of the test cluster, is no longer able to keep up with the crawl rate, and begins building an unbounded queue of unprocessed documents. The dotted asymptote at 40% is an extrapolation of Percolator’s performance beyond this breaking point. MapReduce is subject to the same effect: eventually crawled documents accumulate faster than MapReduce is able to cluster them, and the batch size will grow without bound in subsequent runs. In this particular configuration, however, MapReduce can sustain crawl rates in excess of 100% (the dotted line, again, extrapolates performance).

These results show that Percolator can process documents at orders of magnitude better latency than MapReduce in the regime where we expect real systems to operate (single-digit crawl rates).

### 3.2 Microbenchmarks

In this section, we determine the cost of the transactional semantics provided by Percolator. In these experiments, we compare Percolator to a “raw” Bigtable. We are only interested in the relative performance of Bigtable and Percolator since any improvement in Bigtable performance will translate directly into an improvement in Percolator performance. Figure 8 shows the performance of Percolator and raw Bigtable running against a single tablet server. All data was in the tablet server’s cache during the experiments and Percolator’s batching optimizations were disabled.

As expected, Percolator introduces overhead relative to Bigtable. We first measure the number of random writes that the two systems can perform. In the case of Percolator, we execute transactions that write a single cell and then commit; this represents the worst case for Percolator overhead. When doing a write, Percolator incurs roughly a factor of four overhead on this benchmark. This is the result of the extra operations Percolator requires for commit beyond the single write that Bigtable issues: a read to check for locks, a write to add the lock, and a second write to remove the lock record. The read, in particular, is more expensive than a write and accounts

for most of the overhead. In this test, the limiting factor was the performance of the tablet server, so the additional overhead of fetching timestamps is not measured. We also tested random reads: Percolator performs a single Bigtable operation per read, but that read operation is somewhat more complex than the raw Bigtable operation (the Percolator read looks at metadata columns in addition to data columns).

### 3.3 Synthetic Workload

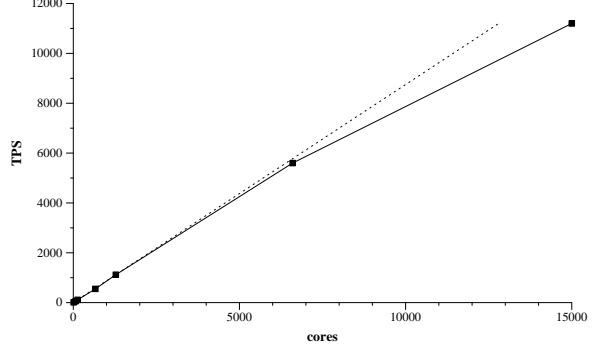
To evaluate Percolator on a more realistic workload, we implemented a synthetic benchmark based on TPC-E [1]. This isn't the ideal benchmark for Percolator since TPC-E is designed for OLTP systems, and a number of Percolator's tradeoffs impact desirable properties of OLTP systems (the latency of conflicting transactions, for example). TPC-E is a widely recognized and understood benchmark, however, and it allows us to understand the cost of our system against more traditional databases.

TPC-E simulates a brokerage firm with customers who perform trades, market search, and account inquiries. The brokerage submits trade orders to a market exchange, which executes the trade and updates broker and customer state. The benchmark measures the number of trades executed. On average, each customer performs a trade once every 500 seconds, so the benchmark scales by adding customers and associated data.

TPC-E traditionally has three components – a customer emulator, a market emulator, and a DBMS running stored SQL procedures. Since Percolator is a client library running against Bigtable, our implementation is a combined customer/market emulator that calls into the Percolator library to perform operations against Bigtable. Percolator provides a low-level Get/Set/iterator API rather than a high-level SQL interface, so we created indexes and did all the ‘query planning’ by hand.

Since Percolator is an incremental processing system rather than an OLTP system, we don't attempt to meet the TPC-E latency targets. Our average transaction latency is 2 to 5 seconds, but outliers can take several minutes. Outliers are caused by, for example, exponential backoff on conflicts and Bigtable tablet unavailability. Finally, we made a small modification to the TPC-E transactions. In TPC-E, each trade result increases the broker's commission and increments his trade count. Each broker services a hundred customers, so the average broker must be updated once every 5 seconds, which causes repeated write conflicts in Percolator. In Percolator, we would implement this feature by writing the increment to a side table and periodically aggregating each broker's increments; for the benchmark, we choose to simply omit this write.

Figure 9 shows how the resource usage of Percolator scales as demand increases. We will measure resource

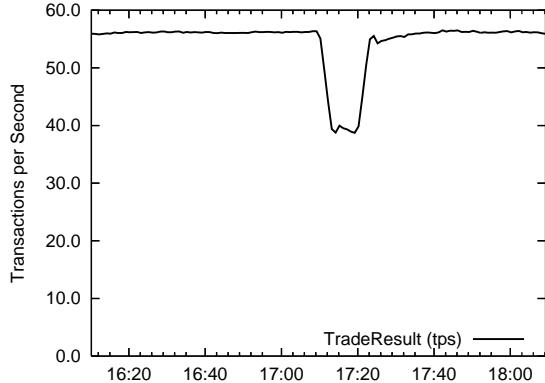


**Figure 9:** Transaction rate on a TPC-E-like benchmark as a function of cores used. The dotted line shows linear scaling.

usage in CPU cores since that is the limiting resource in our experimental environment. We were able to procure a small number of machines for testing, but our test Bigtable cell shares the disk resources of a much larger production cluster. As a result, disk bandwidth is not a factor in the system's performance. In this experiment, we configured the benchmark with increasing numbers of customers and measured both the achieved performance and the number of cores used by all parts of the system including cores used for background maintenance such as Bigtable compactions. The relationship between performance and resource usage is essentially linear across several orders of magnitude, from 11 cores to 15,000 cores.

This experiment also provides an opportunity to measure the overheads in Percolator relative to a DBMS. The fastest commercial TPC-E system today performs 3,183 tpsE using a single large shared-memory machine with 64 Intel Nehalem cores with 2 hyperthreads per core [33]. Our synthetic benchmark based on TPC-E performs 11,200 tps using 15,000 cores. This comparison is very rough: the Nehalem cores in the comparison machine are significantly faster than the cores in our test cell (small-scale testing on Nehalem processors shows that they are 20-30% faster per-thread compared to the cores in the test cluster). However, we estimate that Percolator uses roughly 30 times more CPU per transaction than the benchmark system. On a cost-per-transaction basis, the gap is likely much less than 30 since our test cluster uses cheaper, commodity hardware compared to the enterprise-class hardware in the reference machine.

The conventional wisdom on implementing databases is to “get close to the iron” and use hardware as directly as possible since even operating system structures like disk caches and schedulers make it hard to implement an efficient database [32]. In Percolator we not only interposed an operating system between our database and the hardware, but also several layers of software and network links. The conventional wisdom is correct: this arrangement has a cost. There are substantial overheads in



**Figure 10:** Recovery of tps after 33% tablet server mortality

preparing requests to go on the wire, sending them, and processing them on a remote machine. To illustrate these overheads in Percolator, consider the act of mutating the database. In a DBMS, this incurs a function call to store the data in memory and a system call to force the log to hardware controlled RAID array. In Percolator, a client performing a transaction commit sends multiple RPCs to Bigtable, which commits the mutation by logging it to 3 chunkservers, which make system calls to actually flush the data to disk. Later, that same data will be compacted into minor and major sstables, each of which will be again replicated to multiple chunkservers.

The CPU inflation factor is the cost of our layering. In exchange, we get scalability (our fastest result, though not directly comparable to TPC-E, is more than 3x the current official record [33]), and we inherit the useful features of the systems we build upon, like resilience to failures. To demonstrate the latter, we ran the benchmark with 15 tablet servers and allowed the performance to stabilize. Figure 10 shows the performance of the system over time. The dip in performance at 17:09 corresponds to a failure event: we killed a third of the tablet servers. Performance drops immediately after the failure event but recovers as the tablets are reloaded by other tablet servers. We allowed the killed tablet servers to restart so performance eventually returns to the original level.

#### 4 Related Work

Batch processing systems like MapReduce [13, 22, 24] are well suited for efficiently transforming or analyzing an entire corpus: these systems can simultaneously use a large number of machines to process huge amounts of data quickly. Despite this scalability, re-running a MapReduce pipeline on each small batch of updates results in unacceptable latency and wasted work. Overlapping or pipelining the adjacent stages can reduce latency [10], but straggler shards still set the minimum time to complete the pipeline. Percolator avoids the expense of repeated scans by, essentially, creating indexes

on the keys used to cluster documents; one of criticisms leveled by Stonebraker and DeWitt in their initial critique of MapReduce [16] was that MapReduce did not support such indexes.

Several proposed modifications to MapReduce [18, 26, 35] reduce the cost of processing changes to a repository by allowing workers to randomly read a base repository while mapping over only newly arrived work. To implement clustering in these systems, we would likely maintain a repository per clustering phase. Avoiding the need to re-map the entire repository would allow us to make batches smaller, reducing latency. DryadInc [31] attacks the same problem by reusing identical portions of the computation from previous runs and allowing the user to specify a merge function that combines new input with previous iterations’ outputs. These systems represent a middle-ground between mapping over the entire repository using MapReduce and processing a single document at a time with Percolator.

Databases satisfy many of the requirements of an incremental system: a RDBMS can make many independent and concurrent changes to a large corpus and provides a flexible language for expressing computation (SQL). In fact, Percolator presents the user with a database-like interface: it supports transactions, iterators, and secondary indexes. While Percolator provides distributed transactions, it is by no means a full-fledged DBMS: it lacks a query language, for example, as well as full relational operations such as join. Percolator is also designed to operate at much larger scales than existing parallel databases and to deal better with failed machines. Unlike Percolator, database systems tend to emphasize latency over throughput since a human is often waiting for the results of a database query.

The organization of data in Percolator mirrors that of shared-nothing parallel databases [7, 15, 4]. Data is distributed across a number of commodity machines in shared-nothing fashion: the machines communicate only via explicit RPCs; no shared memory or shared disks are used. Data stored by Percolator is partitioned by Bigtable into tablets of contiguous rows which are distributed among machines; this mirrors the declustering performed by parallel databases.

The transaction management of Percolator builds on a long line of work on distributed transactions for database systems. Percolator implements snapshot isolation [5] by extending multi-version timestamp ordering [6] across a distributed system using two-phase commit.

An analogy can be drawn between the role of observers in Percolator to incrementally move the system towards a “clean” state and the incremental maintenance of materialized views in traditional databases (see Gupta and Mumick [21] for a survey of the field). In practice, while some indexing tasks like clustering documents by

contents could be expressed in a form appropriate for incremental view maintenance it would likely be hard to express the transformation of a raw document into an indexed document in such a form.

The utility of parallel databases and, by extension, a system like Percolator, has been questioned several times [17] over their history. Hardware trends have, in the past, worked against parallel databases. CPUs have become so much faster than disks that a few CPUs in a shared-memory machine can drive enough disk heads to service required loads without the complexity of distributed transactions: the top TPC-E benchmark results today are achieved on large shared-memory machines connected to a SAN. This trend is beginning to reverse itself, however, as the enormous datasets like those Percolator is intended to process become far too large for a single shared-memory machine to handle. These datasets require a distributed solution that can scale to 1000s of machines, while existing parallel databases can utilize only 100s of machines [30]. Percolator provides a system that is scalable enough for Internet-sized datasets by sacrificing some (but not all) of the flexibility and low-latency of parallel databases.

Distributed storage systems like Bigtable have the scalability and fault-tolerance properties of MapReduce but provide a more natural abstraction for storing a repository. Using a distributed storage system allows for low-latency updates since the system can change state by mutating the repository rather than rewriting it. However, Percolator is a data transformation system, not only a data storage system: it provides a way to structure computation to transform that data. In contrast, systems like Dynamo [14], Bigtable, and PNUTS [11] provide highly available data storage without the attendant mechanisms of transformation. These systems can also be grouped with the NoSQL databases (MongoDB [27], to name one of many): both offer higher performance and scale better than traditional databases, but provide weaker semantics.

Percolator extends Bigtable with multi-row, distributed transactions, and it provides the observer interface to allow applications to be structured around notifications of changed data. We considered building the new indexing system directly on Bigtable, but the complexity of reasoning about concurrent state modification without the aid of strong consistency was daunting. Percolator does not inherit all of Bigtable's features: it has limited support for replication of tables across data centers, for example. Since Bigtable's cross data center replication strategy is consistent only on a per-tablet basis, replication is likely to break invariants between writes in a distributed transaction. Unlike Dynamo and PNUTS which serve responses to users, Percolator is willing to accept the lower availability of a single data center in return for stricter consistency.

Several research systems have, like Percolator, extended distributed storage systems to include strong consistency. Sinfonia [3] provides a transactional interface to a distributed repository. Earlier published versions of Sinfonia [2] also offered a notification mechanism similar to the Percolator's observer model. Sinfonia and Percolator differ in their intended use: Sinfonia is designed to build distributed infrastructure while Percolator is intended to be used directly by applications (this probably explains why Sinfonia's authors dropped its notification mechanism). Additionally, Sinfonia's mini-transactions have limited semantics compared to the transactions provided by RDBMSs or Percolator: the user must specify a list of items to compare, read, and write prior to issuing the transaction. The mini-transactions are sufficient to create a wide variety of infrastructure but could be limiting for application builders.

CloudTPS [34], like Percolator, builds an ACID-compliant datastore on top of a distributed storage system (HBase [23] or Bigtable). Percolator and CloudTPS systems differ in design, however: the transaction management layer of CloudTPS is handled by an intermediate layer of servers called local transaction managers that cache mutations before they are persisted to the underlying distributed storage system. By contrast, Percolator uses clients, directly communicating with Bigtable, to coordinate transaction management. The focus of the systems is also different: CloudTPS is intended to be a backend for a website and, as such, has a stronger focus on latency and partition tolerance than Percolator.

ElasTraS [12], a transactional data store, is architecturally similar to Percolator; the Owning Transaction Managers in ElasTraS are essentially tablet servers. Unlike Percolator, ElasTraS offers limited transactional semantics (Sinfonia-like mini-transactions) when dynamically partitioning the dataset and has no support for structuring computation.

## 5 Conclusion and Future Work

We have built and deployed Percolator and it has been used to produce Google's websearch index since April, 2010. The system achieved the goals we set for reducing the latency of indexing a single document with an acceptable increase in resource usage compared to the previous indexing system.

The TPC-E results suggest a promising direction for future investigation. We chose an architecture that scales linearly over many orders of magnitude on commodity machines, but we've seen that this costs a significant 30-fold overhead compared to traditional database architectures. We are very interested in exploring this tradeoff and characterizing the nature of this overhead: how much is fundamental to distributed storage systems, and how much can be optimized away?

## Acknowledgments

Percolator could not have been built without the assistance of many individuals and teams. We are especially grateful to the members of the indexing team, our primary users, and the developers of the many pieces of infrastructure who never failed to improve their services to meet our increasingly large demands.

## References

- [1] TPC benchmark E standard specification version 1.9.0. Tech. rep., Transaction Processing Performance Council, September 2009.
- [2] AGUILERA, M. K., KARAMANOLIS, C., MERCHANT, A., SHAH, M., AND VEITCH, A. Building distributed applications using Sinfonia. Tech. rep., Hewlett-Packard Labs, 2006.
- [3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07* (2007), ACM, pp. 159–174.
- [4] BARU, C., FECTEAU, G., GOYAL, A., HSIAO, H.-I., JHINGRAN, A., PADMANABHAN, S., WILSON, W., AND I HSIAO, A. G. H. DB2 parallel edition, 1995.
- [5] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O’NEIL, E., AND O’NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD* (New York, NY, USA, 1995), ACM, pp. 1–10.
- [6] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Computer Surveys 13*, 2 (1981), 185–221.
- [7] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 4–24.
- [8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *7th OSDI* (Nov. 2006).
- [9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIRES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th OSDI* (Nov. 2006), pp. 205–218.
- [10] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTEN, J. M. MapReduce online. In *7th NSDI* (2010).
- [11] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!’s hosted data serving platform. In *Proceedings of VLDB* (2008).
- [12] DAS, S., AGRAWAL, D., AND ABBADI, A. E. ElasTraS: An elastic transactional data store in the cloud. In *USENIX HotCloud* (June 2009).
- [13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *6th OSDI* (Dec. 2004), pp. 137–150.
- [14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon’s highly available key-value store. In *SOSP ’07* (2007), pp. 205–220.
- [15] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), 44–62.
- [16] DEWITT, D., AND STONEBRAKER, M. MapReduce: A major step backwards. <http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/>.
- [17] DEWITT, D. J., AND GRAY, J. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec.* 19, 4 (1990), 104–112.
- [18] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. Twister: A runtime for iterative MapReduce. In *The First International Workshop on MapReduce and its Applications* (2010).
- [19] GERSHENSON, C., AND PINEDA, L. A. Why does public transport not arrive on time? The pervasiveness of equal headway instability. *PLoS ONE* 4, 10 (10 2009).
- [20] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. vol. 37, pp. 29–43.
- [21] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques, and applications, 1995.
- [22] Hadoop. <http://hadoop.apache.org/>.
- [23] HBase. <http://hbase.apache.org/>.
- [24] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys ’07* (New York, NY, USA, 2007), ACM, pp. 59–72.
- [25] IYER, S., AND CUTTS, M. Help test some next-generation infrastructure. <http://googlewebmastercentral.blogspot.com/2009/08/help-test-some-next-generation.html>, August 2009.
- [26] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *SoCC ’10: Proceedings of the 1st ACM symposium on cloud computing* (2010), pp. 51–62.
- [27] MongoDB. <http://mongodb.org/>.
- [28] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.
- [29] PATON, N. W., AND DÍAZ, O. Active database systems. *ACM Computing Surveys 31*, 1 (1999), 63–103.
- [30] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD ’09* (June 2009), ACM.
- [31] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *USENIX workshop on Hot Topics in Cloud Computing* (2009).
- [32] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM* 24, 7 (1981), 412–418.
- [33] NEC Express5800/A1080a-E TPC-E results. [http://www.tpc.org/tfce/results/tfce\\_result\\_detail.asp?id=110033001](http://www.tpc.org/tfce/results/tfce_result_detail.asp?id=110033001), Mar. 2010.
- [34] WEI, Z., PIERRE, G., AND CHI, C.-H. CloudTPS: Scalable transactions for Web applications in the cloud. Tech. Rep. IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010. <http://www.globule.org/publi/CSTWAC ircs53.html>.
- [35] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *2nd USENIX workshop on Hot Topics in Cloud Computing* (2010).

# Availability in Globally Distributed Storage Systems

Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong,\*

Luiz Barroso, Carrie Grimes, and Sean Quinlan

{ford, flab, florentina, mstokely}@google.com, vtruong@ieor.columbia.edu

{luiz, cgrimes, sean}@google.com

Google, Inc.

## Abstract

Highly available cloud storage is often implemented with complex, multi-tiered distributed systems built on top of clusters of commodity servers and disk drives. Sophisticated management, load balancing and recovery techniques are needed to achieve high performance and availability amidst an abundance of failure sources that include software, hardware, network connectivity, and power issues. While there is a relative wealth of failure studies of individual components of storage systems, such as disk drives, relatively little has been reported so far on the overall availability behavior of large cloud-based storage services.

We characterize the availability properties of cloud storage systems based on an extensive one year study of Google’s main storage infrastructure and present statistical models that enable further insight into the impact of multiple design choices, such as data placement and replication strategies. With these models we compare data availability under a variety of system parameters given the real patterns of failures observed in our fleet.

## 1 Introduction

Cloud storage is often implemented by complex multi-tiered distributed systems on clusters of thousands of commodity servers. For example, in Google we run Bigtable [9], on GFS [16], on local Linux file systems that ultimately write to local hard drives. Failures in any of these layers can cause data unavailability.

Correctly designing and optimizing these multi-layered systems for user goals such as data availability relies on accurate models of system behavior and performance. In the case of distributed storage systems, this includes quantifying the impact of failures and prioritizing hardware and software subsystem improvements in

the datacenter environment.

We present models we derived from studying a year of live operation at Google and describe how our analysis influenced the design of our next generation distributed storage system [22].

Our work is presented in two parts. First, we measured and analyzed the *component availability*, e.g. machines, racks, multi-racks, in tens of Google storage clusters. In this part we:

- Compare mean time to failure for system components at different granularities, including disks, machines and racks of machines. (Section 3)
- Classify the failure causes for storage nodes, their characteristics and contribution to overall unavailability. (Section 3)
- Apply a clustering heuristic for grouping failures which occurs almost simultaneously and show that a large fraction of failures happen in bursts. (Section 4)
- Quantify how likely a failure burst is associated with a given failure domain. We find that most large bursts of failures are associated with rack- or multi-rack level events. (Section 4)

Based on these results, we determined that the critical element in models of availability is their ability to account for the frequency and magnitude of *correlated* failures.

Next, we consider *data availability* by analyzing unavailability at the distributed file system level, where one file system instance is referred to as a *cell*. We apply two models of multi-scale correlated failures for a variety of replication schemes and system parameters. In this part we:

- Demonstrate the importance of modeling correlated failures when predicting availability, and show their

\*Now at Dept. of Industrial Engineering and Operations Research  
Columbia University

impact under a variety of replication schemes and placement policies. (Sections 5 and 6)

- Formulate a Markov model for data availability, that can scale to arbitrary cell sizes, and captures the interaction of failures with replication policies and recovery times. (Section 7)
- Introduce multi-cell replication schemes and compare the availability and bandwidth trade-offs against single-cell schemes. (Sections 7 and 8)
- Show the impact of hardware failure on our cells is significantly smaller than the impact of effectively tuning recovery and replication parameters. (Section 8)

Our results show the importance of considering cluster-wide failure events in the choice of replication and recovery policies.

## 2 Background

We study end to end data availability in a cloud computing storage environment. These environments often use loosely coupled distributed storage systems such as GFS [1, 16] due to the parallel I/O and cost advantages they provide over traditional SAN and NAS solutions. A few relevant characteristics of such systems are:

- Storage server programs running on physical machines in a datacenter, managing local disk storage on behalf of the distributed storage cluster. We refer to the storage server programs as *storage nodes* or *nodes*.
- A pool of storage service masters managing data placement, load balancing and recovery, and monitoring of storage nodes.
- A replication or erasure code mechanism for user data to provide resilience to individual component failures.

A large collection of nodes along with their higher level coordination processes [17] are called a *cell* or *storage cell*. These systems usually operate in a shared pool of machines running a wide variety of applications. A typical cell may comprise many thousands of nodes housed together in a single building or set of colocated buildings.

### 2.1 Availability

A storage node becomes *unavailable* when it fails to respond positively to periodic health checking pings sent

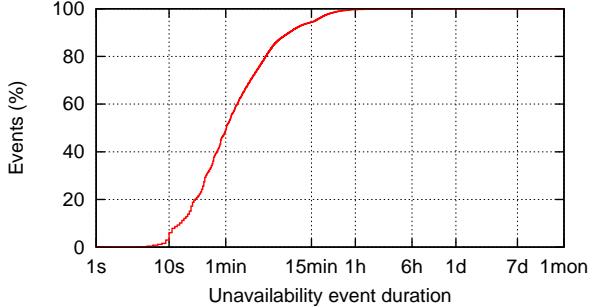


Figure 1: Cumulative distribution function of the duration of node unavailability periods.

by our monitoring system. The node remains unavailable until it regains responsiveness or the storage system reconstructs the data from other surviving nodes.

Nodes can become unavailable for a large number of reasons. For example, a storage node or networking switch can be overloaded; a node binary or operating system may crash or restart; a machine may experience a hardware error; automated repair processes may temporarily remove disks or machines; or the whole cluster could be brought down for maintenance. The vast majority of such unavailability events are transient and do not result in permanent data loss. Figure 1 plots the CDF of node unavailability duration, showing that less than 10% of events last longer than 15 minutes. This data is gathered from tens of Google storage cells, each with 1000 to 7000 nodes, over a one year period. The cells are located in different datacenters and geographical regions, and have been used continuously by different projects within Google. We use this dataset throughout the paper, unless otherwise specified.

Experience shows that while short unavailability events are most frequent, they tend to have a minor impact on cluster-level availability and data loss. This is because our distributed storage systems typically add enough redundancy to allow data to be served from other sources when a particular node is unavailable. Longer unavailability events, on the other hand, make it more likely that faults will overlap in such a way that data could become unavailable at the cluster level for long periods of time. Therefore, while we track unavailability metrics at multiple time scales in our system, in this paper we focus only on events that are 15 minutes or longer. This interval is long enough to exclude the majority of benign transient events while not too long to exclude significant cluster-wide phenomena. As in [11], we observe that initiating recovery after transient failures is inefficient and reduces resources available for other operations. For these reasons, GFS typically waits 15 minutes before commencing recovery of data on unavailable nodes.

We primarily use two metrics throughout this paper. The average availability of all  $N$  nodes in a cell is defined as:

$$A_N = \frac{\sum_{N_i \in N} \text{uptime}(N_i)}{\sum_{N_i \in N} (\text{uptime}(N_i) + \text{downtime}(N_i))} \quad (1)$$

We use  $\text{uptime}(N_i)$  and  $\text{downtime}(N_i)$  to refer to the lengths of time a node  $N_i$  is available or unavailable, respectively. The sum of availability periods over all nodes is called *node uptime*. We define uptime similarly for other component types. We define unavailability as the complement of availability.

*Mean time to failure*, or *MTTF*, is commonly quoted in the literature related to the measurements of availability. We use MTTF for components that suffer transient or permanent failures, to avoid frequent switches in terminology.

$$\text{MTTF} = \frac{\text{uptime}}{\text{number failures}} \quad (2)$$

Availability measurements for nodes and individual components in our system are presented in Section 3.

## 2.2 Data replication

Distributed storage systems increase resilience to failures by using replication [2] or erasure encoding across nodes [28]. In both cases, data is divided into a set of *stripes*, each of which comprises a set of fixed size data and code blocks called *chunks*. Data in a stripe can be reconstructed from some subsets of the chunks. For replication,  $R = n$  refers to  $n$  identical chunks in a stripe, so the data may be recovered from any one chunk. For Reed-Solomon erasure encoding,  $RS(n, m)$  denotes  $n$  distinct data blocks and  $m$  error correcting blocks in each stripe. In this case a stripe may be reconstructed from any  $n$  chunks.

We call a chunk available if the node it is stored on is available. We call a stripe available if enough of its chunks are available to reconstruct the missing chunks, if any.

Data availability is a complex function of the individual node availability, the encoding scheme used, the distribution of correlated node failures, chunk placement, and recovery times that we will explore in the second part of this paper. We do not explore related mechanisms for dealing with failures, such as additional application level redundancy and recovery, and manual component repair.

## 3 Characterizing Node Availability

Anything that renders a storage node unresponsive is a potential cause of unavailability, including hardware

component failures, software bugs, crashes, system reboots, power loss events, and loss of network connectivity. We include in our analysis the impact of software upgrades, reconfiguration, and other maintenance. These planned outages are necessary in a fast evolving datacenter environment, but have often been overlooked in other availability studies. In this section we present data for storage node unavailability and provide some insight into the main causes for unavailability.

### 3.1 Numbers from the fleet

Failure patterns vary dramatically across different hardware platforms, datacenter operating environments, and workloads. We start by presenting numbers for disks.

Disks have been the focus of several other studies, since they are the system component that permanently stores the data, and thus a disk failure potentially results in permanent data loss. The numbers we observe for disk and storage subsystem failures, presented in Table 2, are comparable with what other researchers have measured. One study [29] reports ARR (annual replacement rate) for disks between 2% and 4%. Another study [19] focused on storage subsystems, thus including errors from shelves, enclosures, physical interconnects, protocol failures, and performance failures. They found AFR (annual failure rate) generally between 2% and 4%, but for some storage systems values ranging between 3.9% and 8.3%.

For the purposes of this paper, we are interested in disk errors as perceived by the application layer. This includes latent sector errors and corrupt sectors on disks, as well as errors caused by firmware, device drivers, controllers, cables, enclosures, silent network and memory corruption, and software bugs. We deal with these errors with background scrubbing processes on each node, as in [5, 31], and by verifying data integrity during client reads [4]. Background scrubbing in GFS finds between 1 in  $10^6$  to  $10^7$  of older data blocks do not match the checksums recorded when the data was originally written. However, these cell-wide rates are typically concentrated on a small number of disks.

We are also concerned with node failures in addition to individual disk failures. Figure 2 shows the distribution of three mutually exclusive causes of node unavailability in one of our storage cells. We focus on *node restarts* (software restarts of the storage program running on each machine), *planned machine reboots* (e.g. kernel version upgrades), and *unplanned machine reboots* (e.g. kernel crashes). For the purposes of this figure we do not exclude events that last less than 15 minutes, but we still end the unavailability period when the system reconstructs all the data previously stored on that node. Node restart events exhibit the greatest variability in duration, ranging from less than one minute to well over an

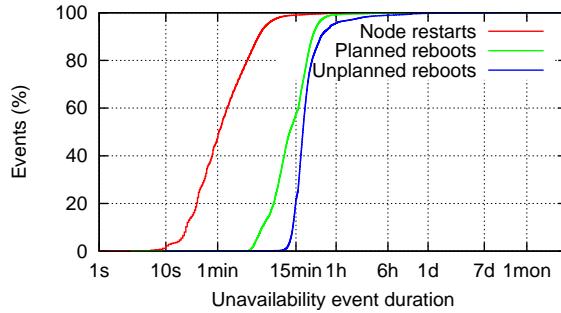


Figure 2: Cumulative distribution function of node unavailability durations by cause.

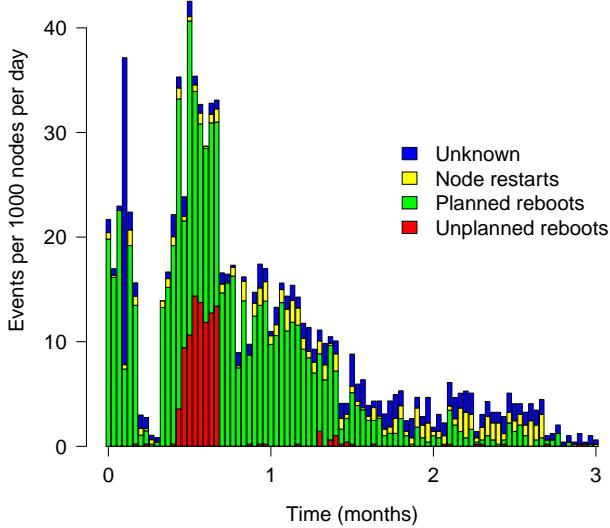


Figure 3: Rate of events per 1000 nodes per day, for one example cell.

hour, though they usually have the shortest duration. Unplanned reboots have the longest average duration since extra checks or corrective action is often required to restore machines to a safe state.

Figure 3 plots the unavailability events per 1000 nodes per day for one example cell, over a period of three months. The number of events per day, as well as the number of events that can be attributed to a given cause vary significantly over time as operational processes, tools, and workloads evolve. Events we cannot classify accurately are labeled *unknown*.

The effect of machine failures on availability is dependent on the rate of failures, as well as on how long the machines stay unavailable. Figure 4 shows the node unavailability, along with the causes that generated the unavailability, for the same cell used in Figure 3. The availability is computed with a one week rolling window, using definition (1). We observe that the majority of unavailability is generated by planned reboots.

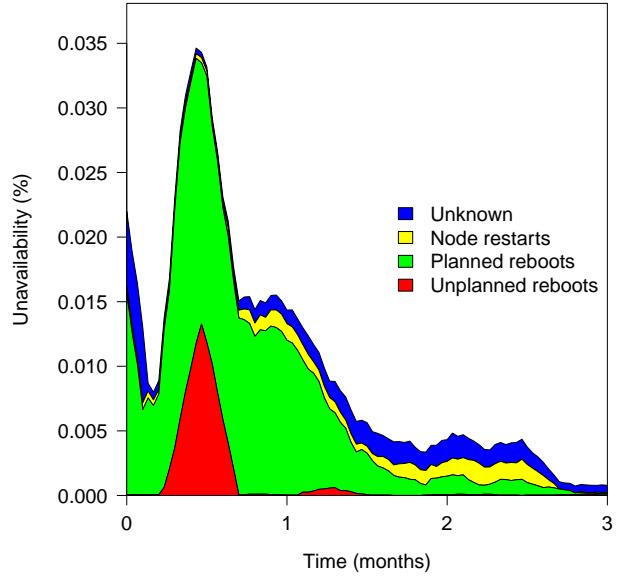


Figure 4: Storage node unavailability computed with a one week rolling window, for one example cell.

Cause	Unavailability (%) average / min / max
Node restarts	0.0139 / 0.0004 / 0.1295
Planned machine reboots	0.0154 / 0.0050 / 0.0563
Unplanned machine reboots	0.0025 / 0.0000 / 0.0122
Unknown	0.0142 / 0.0013 / 0.0454

Table 1: Unavailability attributed to different failure causes, over the full set of cells.

Table 1 shows the unavailability from node restarts, planned and unplanned machine reboots, each of which is a significant cause. The numbers are exclusive, thus the planned machine reboots do not include node restarts.

Table 2 shows the MTTF for a series of important components: disk, nodes, and racks of nodes. The numbers we report for component failures are inclusive of software errors and hardware failures. Though disks failures are permanent and most node failures are transitory, the significantly greater frequency of node failures makes them a much more important factor for system availability (Section 8.4).

## 4 Correlated Failures

The co-occurring failure of a large number of nodes can reduce the effectiveness of replication and encoding schemes. Therefore it is critical to take into account the statistical behavior of correlated failures to understand data availability. In this section we are more concerned with measuring the frequency and severity of such failures rather than root causes.

Component	Disk	Node	Rack
MTTF	10-50 years	4.3 months	10.2 years

Table 2: Component failures across several Google cells.

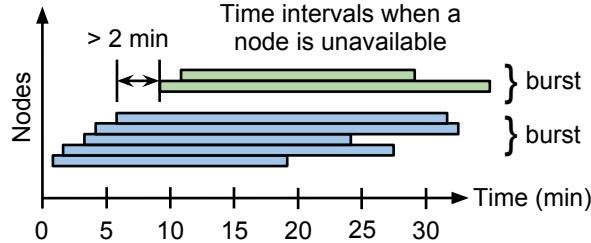


Figure 5: Seven node failures clustered into two failure bursts when the window size is 2 minutes. Note how only the unavailability start times matter.

We define a *failure burst* and examine features of these bursts in the field. We also develop a method for identifying which bursts are likely due to a failure domain. By failure domain, we mean a set of machines which we expect to simultaneously suffer from a common source of failure, such as machines which share a network switch or power cable. We demonstrate this method by validating physical racks as an important failure domain.

#### 4.1 Defining failure bursts

We define a *failure burst* with respect to a window size  $w$  as a maximal sequence of node failures, each one occurring within a time window  $w$  of the next. Figure 5 illustrates the definition. We choose  $w = 120$  s, for several reasons. First, it is longer than the frequency with which nodes are periodically polled in our system for their status. A window length smaller than the polling interval would not make sense as some pairs of events which actually occur within the window length of each other would not be correctly associated. Second, it is less than a tenth of the average time it takes our system to recover a chunk, thus, failures within this window can be considered as nearly concurrent. Figure 6 shows the fraction of individual failures that get clustered into bursts of at least 10 nodes as the window size changes. Note that the graph is relatively flat after 120 s, which is our third reason for choosing this value.

Since failures are clustered into bursts based on their times of occurrence alone, there is a risk that two bursts with independent causes will be clustered into a single burst by chance. The slow increase in Figure 6 past 120 s illustrates this phenomenon. The error incurred is small as long as we keep the window size small. Given a window size of 120 s and the set of bursts obtained from it, the probability that a random failure gets included in a

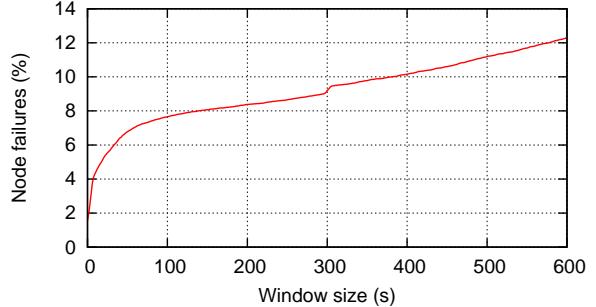


Figure 6: Effect of the window size on the fraction of individual failures that get clustered into bursts of at least 10 nodes.

burst (as opposed to becoming its own singleton burst) is 8.0%. When this inclusion happens, most of the time the random failure is combined with a singleton burst to form a burst of two nodes. The probability that a random failure gets included in a burst of at least 10 nodes is only 0.068%. For large bursts, which contribute most unavailability as we will see in Section 5.2, the fraction of nodes affected is the significant quantity and changes insignificantly if a burst of size one or two nodes is accidentally clustered with it.

Using this definition, we observe that 37% of failures are part of a burst of at least 2 nodes. Given the result above that only 8.0% of non-correlated failures may be incorrectly clustered, we are confident that close to 37% of failures are truly correlated.

#### 4.2 Views of failure bursts

Figure 7 shows the accumulation of individual failures in bursts. For clarity we show all bursts of size at least 10 seen over a 60 day period in an example cell. In the plot, each burst is displayed with a separate shape. The  $n$ -th node failure that joins a burst at time  $t_n$  is said to have ordinal  $n - 1$  and is plotted at point  $(t_n, n - 1)$ . Two broad classes of failure bursts can be seen in the plot:

1. Those failure bursts that are characterized by a large number of failures in quick succession show up as steep lines with a large number of nodes in the burst. Such failures can be seen, for example, following a power outage in a datacenter.
2. Those failure bursts that are characterized by a smaller number of nodes failing at a slower rate at evenly spaced intervals. Such correlated failures can be seen, for example, as part of rolling reboot or upgrade activity at the datacenter management layer.

Figure 8 displays the bursts sorted by the number of nodes and racks that they affect. The size of each bubble

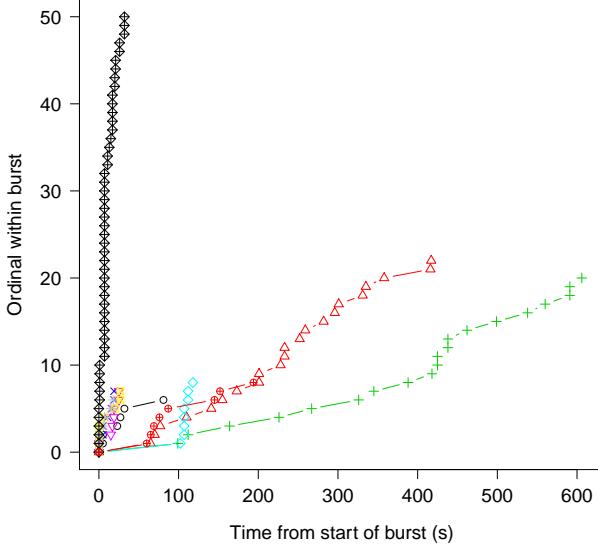


Figure 7: Development of failure bursts in one example cell.

indicates the frequency of each burst group. The grouping of points along the  $45^\circ$  line represent bursts where as many racks are affected as nodes. The points furthest away from this line represent the most rack-correlated failure bursts. For larger bursts of at least 10 nodes, we find only 3% have all their nodes on unique racks. We introduce a metric to quantify this degree of domain correlation in the next section.

### 4.3 Identifying domain-related failures

Domain-related issues, such those associated with physical racks, network switches and power domains, are frequent causes of correlated failure. These problems can sometimes be difficult to detect directly. We introduce a metric to measure the likelihood that a failure burst is domain-related, rather than random, based on the pattern of failure observed. The metric can be used as an effective tool for identifying causes of failures that are connected to domain locality. It can also be used to evaluate the importance of domain diversity in cell design and data placement. We focus on detecting rack-related node failures in this section, but our methodology can be applied generally to any domain and any type of failure.

Let a failure burst be encoded as an  $n$ -tuple  $(k_1, k_2, \dots, k_n)$ , where  $k_1 \leq k_2 \leq \dots \leq k_n$ . Each  $k_i$  gives the number of nodes affected in the  $i$ -th rack affected, where racks are ordered so that these values are increasing. This *rack-based encoding* captures all relevant information about the rack locality of the burst. Let the *size* of the burst be the number of nodes that are affected, i.e.,  $\sum_{i=1}^n k_i$ . We define the *rack-affinity score* of

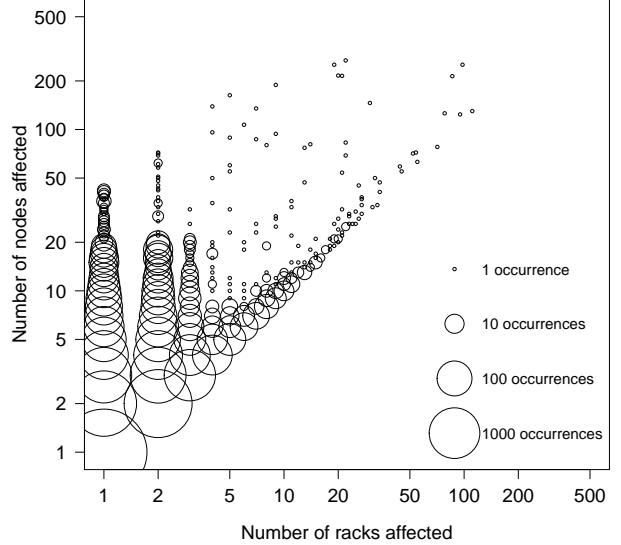


Figure 8: Frequency of failure bursts sorted by racks and nodes affected.

a burst to be

$$\sum_{i=1}^n \frac{k_i(k_i - 1)}{2}$$

Note that this is the number of ways of choosing two nodes from the burst within the same rack. The score allows us to compare the rack concentration of bursts of the same size. For example the burst  $(1, 4)$  has score 6. The burst  $(1, 1, 1, 2)$  has score 1 which is lower. Therefore, the first burst is more concentrated by rack. Possible alternatives for the score include the sum of squares  $\sum_{i=1}^n k_i^2$  or the negative entropy  $\sum_{i=1}^n k_i \log(k_i)$ . The sum of squares formula is equivalent to our chosen score because for a fixed burst size, the two formulas are related by an affine transform. We believe the entropy-inspired formula to be inferior because its log factor tends to downplay the effect of a very large  $k_i$ . Its real-valued score is also a problem for the dynamic program we use later in computation.

We define the *rack affinity* of a burst in a particular cell to be the probability that a burst of the same size affecting randomly chosen nodes in that cell will have a smaller burst score, plus half the probability that the two scores are equal, to eliminate bias. Rack affinity is therefore a number between 0 and 1 and can be interpreted as a vertical position on the cumulative distribution of the scores of random bursts of the same size. It can be shown that for a random burst, the expected value of its rack affinity is exactly 0.5. So we define a rack-correlated burst to be one with a metric close to 1, a rack-uncorrelated burst to be one with a metric close to 0.5, and a rack-anti-correlated burst to be one with a metric close to 0 (we have not observed such a burst). It is possible to ap-

proximate the metric using simulation of random bursts. We choose to compute the metric exactly using dynamic programming because the extra precision it provides allows us to distinguish metric values very close to 1.

We find that, in general, larger failure bursts have higher rack affinity. All our failure bursts of more than 20 nodes have rack affinity greater than 0.7, and those of more than 40 nodes have affinity at least 0.9. It is worth noting that some bursts with high rack affinity do not affect an entire rack and are not caused by common network or power issues. This could be the case for a bad batch of components or new storage node binary or kernel, whose installation is only slightly correlated with these domains.

## 5 Coping with Failure

We now begin the second part of the paper where we transition from node failures to analyzing replicated data availability. Two methods for coping with the large number of failures described in the first part of this paper include data replication and recovery, and chunk placement.

### 5.1 Data replication and recovery

Replication or erasure encoding schemes provide resilience to individual node failures. When a node failure causes the unavailability of a chunk within a stripe, we initiate a recovery operation for that chunk from the other available chunks remaining in the stripe.

Distributed filesystems will necessarily employ queues for recovery operations following node failure. These queues prioritize reconstruction of stripes which have lost the most chunks. The rate at which missing chunks may be recovered is limited by the bandwidth of individual disks, nodes, and racks. Furthermore, there is an explicit design tradeoff in the use of bandwidth for recovery operations versus serving client read/write requests.

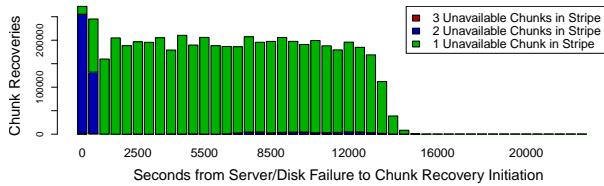


Figure 9: Example chunk recovery after failure bursts.

This limit is particularly apparent during correlated failures when a large number of chunks go missing at the same time. Figure 9 shows the recovery delay after a failure burst of 20 storage nodes affecting millions of stripes. Operators may adjust the rate-limiting seen in the figure.

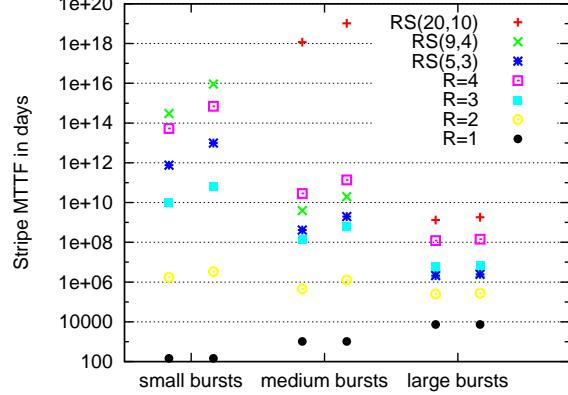


Figure 10: Stripe MTTF due to different burst sizes. Burst sizes are defined as a fraction of all nodes: small (0-0.001), medium (0.001-0.01), large (0.01-0.1). For each size, the left column represents uniform random placement, and the right column represents rack-aware placement.

The models presented in the following sections allow us to measure the sensitivity of data availability to this rate-limit and other parameters, described in Section 8.

### 5.2 Chunk placement and stripe unavailability

To mitigate the effect of large failure bursts in a single failure domain we consider known failure domains when placing chunks within a stripe on storage nodes. For example, racks constitute a significant failure domain to avoid. A rack-aware policy is one that ensures that no two chunks in a stripe are placed on nodes in the same rack.

Given a failure burst, we can compute the expected fraction of stripes made unavailable by the burst. More generally, we compute the probability that exactly  $k$  chunks are affected in a stripe of size  $n$ , which is essential to the Markov model of Section 7. Assuming that stripes are uniformly distributed across nodes of the cell, this probability is a ratio where the numerator is the number of ways to place a stripe of size  $n$  in the cell such that exactly  $k$  of its chunks are affected by the burst, and the denominator is the total number of ways to place a stripe of size  $n$  in the cell. These numbers can be computed combinatorially. The same ratio can be used when chunks are constrained by a placement policy, in which case the numerator and denominator are computed using dynamic programming.

Figure 10 shows the stripe MTTF for three classes of burst size. For each class of bursts we calculate the average fraction of stripes affected per burst and the rate of bursts, to get the combined MTTF due to that class. We see that for all encodings except  $R = 1$ , large failure bursts are the biggest contributor to unavailability

despite the fact that they are much rarer. We also see that for small and medium bursts sizes, and large encodings, using a rack-aware placement policy increases the stripe MTTF by a factor of 3 typically. This is a significant gain considering that in uniform random placement, most stripes end up with their chunks on different racks due to chance.

## 6 Cell Simulation

This section introduces a trace-based simulation method for calculating availability in a cell. The method replays observed or synthetic sequences of node failures and calculates the resulting impact on stripe availability. It offers detailed view of availability in short time frames.

For each node, the recorded events of interest are *down*, *up* and *recovery complete* events. When all nodes are up, they are each assumed to be responsible for an equal number of chunks. When a node goes down it is still responsible for the same number of chunks until 15 minutes later when the chunk recovery process starts. For simplicity and conservativeness, we assume that all these chunks remain unavailable until the *recovery complete* event. A more accurate model could model recovery too, such as by reducing the number of unavailable chunks linearly until the *recovery complete* event, or by explicitly modelling recovery queues.

We are interested in the expected number of stripes that are unavailable for at least 15 minutes, as a function of time. Instead of simulating a large number of stripes, it is more efficient to simulate all possible stripes, and use combinatorial calculations to obtain the expected number of unavailable stripes given a set of down nodes, as was done in Section 5.2.

As a validation, we can run the simulation using the stripe encodings that were in use at the time to see if the predicted number of unavailable stripes matches the actual number of unavailable stripes as measured by our storage system. Figure 11 shows the result of such a simulation. The prediction is a linear combination of the predictions for individual encodings present, in this case mostly  $RS(5, 3)$  and  $R = 3$ .

Analysis of hypothetical scenarios may also be made with the cell simulator, such as the effect of encoding choice and of chunk recovery rate. Although we may not change the frequency and severity of bursts in an observed sequence, bootstrap methods [13] may be used to generate synthetic failure traces with different burst characteristics. This is useful for exploring sensitivity to these events and the impact of improvements in datacenter reliability.

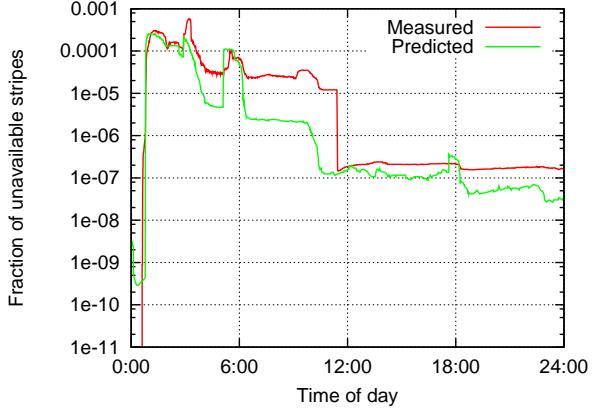


Figure 11: Unavailability prediction over time for a particular cell for a day with large failure bursts.

## 7 Markov Model of Stripe Availability

In this section, we formulate a Markov model of data availability. The model captures the interaction of different failure types and production parameters with more flexibility than is possible with the trace-based simulation described in the previous section. Although the model makes assumptions beyond those in the trace-based simulation method, it has certain advantages. First, it allows us to model and understand the impact of changes in hardware and software on end-user data availability. There are typically too many permutations of system changes and encodings to test each in a live cell. The Markov model allows us to reason directly about the contribution to data availability of each level of the storage stack and several system parameters, so that we can evaluate tradeoffs. Second, the systems we study may have unavailability rates that are so low they are difficult to measure directly. The Markov model handles rare events and arbitrarily low stripe unavailability rates efficiently.

The model focuses on the availability of a representative stripe. Let  $s$  be the total number of chunks in the stripe, and  $r$  be the minimum number of chunks needed to recover that stripe. As described in Section 2.2,  $r = 1$  for replicated data and  $r = n$  for  $RS(n, m)$  encoded data. The state of a stripe is represented by the number of available chunks. Thus, the states are  $s, s-1, \dots, r, r-1$  with the state  $r-1$  representing all of the *unavailable* states where the stripe has less than the required  $r$  chunks available. Figure 12 shows a Markov chain corresponding to an  $R = 2$  stripe.

The Markov chain transitions are specified by the rates at which a stripe moves from one state to another, due to chunk failures and recoveries. Chunk failures reduce the number of available chunks, and several chunks may fail ‘simultaneously’ in a failure burst event. Balancing this, recoveries increase the number of available chunks if any

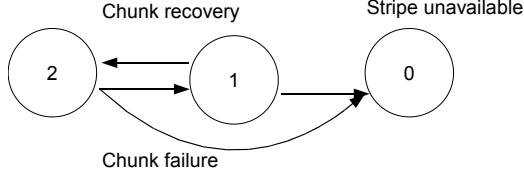


Figure 12: The Markov chain for a stripe encoded using  $R = 2$ .

are unavailable.

A key assumption of the Markov model is that events occur independently and with constant rates over time. This independence assumption, although strong, is not the same as the assumption that individual chunks fail independently of each other. Rather, it implies that failure events are independent of each other, but each event may involve multiple chunks. This allows a richer and more flexible view of the system. It also implies that recovery rates for a stripe depend only on its own current state.

In practice, failure events are not always independent. Most notably, it has been pointed out in [29] that the time between disk failures is not exponentially distributed and exhibits autocorrelation and long-range dependence. The Weibull distribution provides a much better fit for disk MTTF.

However, the exponential distribution is a reasonable approximation for the following reasons. First, the Weibull distribution is a generalization of the exponential distribution that allows the rate parameter to increase over time to reflect the aging of disks. In a large population of disks, the mixture of disks of different ages tends to be stable, and so the average failure rate in a cell tends to be constant. When the failure rate is stable, the Weibull distribution provides the same quality of fit as the exponential. Second, disk failures make up only a small subset of failures that we examined, and model results indicate that overall availability is not particularly sensitive to them. Finally, other authors ([24]) have concluded that correlation and non-homogeneity of the recovery rate and the mean time to a failure event have a much smaller impact on system-wide availability than the size of the event.

## 7.1 Construction of the Markov chain

We compute the transition rate due to failures using observed failure events. Let  $\lambda$  denote the rate of failure events affecting chunks, including node and disk failures. For any observed failure event we compute the probability that it affects  $k$  chunks out of the  $i$  available chunks in a stripe. As in Section 6, for failure bursts this computation takes into account the stripe placement strategy. The rate and severity of bursts, node, disk, and other failures

may be adjusted here to suit the system parameters under exploration.

Averaging these probabilities over all failures events gives the probability,  $p_{i,j}$ , that a random failure event will affect  $i-j$  out of  $i$  available chunks in a stripe. This gives a rate of transition from state  $i$  to state  $j < i$ , of  $\lambda_{i,j} = \lambda p_{i,j}$  for  $s \geq i > j \geq r$  and  $\lambda_{i,r-1} = \lambda \sum_{j=0}^{r-1} p_{i,j}$  for the rate of reaching the unavailable state. Note that transitions from a state to itself are ignored.

For chunk recoveries, we assume a fixed rate of  $\rho$  for recovering a single chunk, i.e. moving from a state  $i$  to  $i+1$ , where  $r \leq i < s$ . In particular, this means we assume that the recovery rate does not depend on the total number of unavailable chunks in the cell. This is justified by setting  $\rho$  to a lower bound for the rate of recovery, based on observed recovery rates across our storage cells or proposed system performance parameters. While parallel recovery of multiple chunks from a stripe is possible,  $\rho_{i,i+1} = (s-i)\rho$ , we model serial recovery to gain more conservative estimates of stripe availability.

As with [12], the distributed systems we study use prioritized recovery for stripes with more than one chunk unavailable. Our Markov model allows state-dependent recovery that captures this prioritization, but for ease of exposition we do not use this added degree of freedom.

Finally, transition rates between pairs of states not mentioned are zero.

With the Markov chain thus completely specified, computing the MTTF of a stripe, as the mean time to reach the ‘unavailable state’  $r-1$  starting from state  $s$ , follows by standard methods [27].

## 7.2 Extension to multi-cell replication

The models introduced so far can be extended to compute the availability of multi-cell replication schemes. An example of such a scheme is  $R = 3 \times 2$ , where six replicas of the data are distributed as  $R = 3$  replication in each of two linked cells. If data becomes unavailable at one cell then it is automatically recovered from another linked cell. These cells may be placed in separate datacenters, even on separate continents. Reed-Solomon codes may also be used, giving schemes such as  $RS(6, 3) \times 3$  for three cells each with a  $RS(6, 3)$  encoding of the data. We do not consider here the case when individual chunks may be combined from multiple cells to recover data, or other more complicated multi-cell encodings.

We compute the availability of stripes that span cells by building on the Markov model just presented. Intuitively, we treat each cell as a ‘chunk’ in the multi-cell ‘stripe’, and compute its availability using the Markov model. We assume that failures at different data centers are independent, that is, that they lack a single point of failure such as a shared power plant or network link. Ad-

ditionally, when computing the cell availability, we account for any cell-level or datacenter-level failures that would affect availability.

We build the corresponding transition matrix that models the resulting multi-cell availability as follows. We start from the transition matrices  $M_i$  for each cell, as explained in the previous section. We then build the transition matrix for the combined scheme as the tensor product of these,  $\bigotimes_i M_i$ , plus terms for whole cell failures, and for cross-cell recoveries if the data becomes unavailable in some cells but is still available in at least one cell. However, it is a fair approximation to simply treat each cell as a highly-reliable chunk in a multi-cell stripe, as described above.

Besides symmetrical cases, such as  $R = 3 \times 2$  replication, we can also model inhomogeneous replication schemes, such as one cell with  $R = 3$  and one with  $R = 2$ . The state space of the Markov model is the product of the state space for each cell involved, but may be approximated again by simply counting how many of each type of cell is available.

A point of interest here is the recovery bandwidth between cells, quantified in Section 8.5. Bandwidth between distant cells has significant cost which should be considered when choosing a multi-cell replication scheme.

## 8 Markov Model Findings

In this section, we apply the Markov models described above to understand how changes in the parameters of the system will affect end-system availability.

### 8.1 Markov model validation

We validate the Markov model by comparing MTTF predicted by the model with actual MTTF values observed in production cells. We are interested in whether the Markov model provides an adequate tool for reasoning about stripe availability. Our main goal in using the model is providing a relative comparison of competing storage solutions, rather than a highly accurate prediction of any particular solution.

We underline two observations that surface from validation. First, the model is able to capture well the effect of failure bursts, which we consider as having the most impact on the availability numbers. For the cells we observed, the model predicted MTTF with the same order of magnitude as the measured MTTF. In one particular cell, besides more regular unavailability events, there was a large failure burst where tens of nodes became unavailable. This resulted in an MTTF of 1.76E+6 days, while the model predicted 5E+6 days. Though the relative error exceeds 100%, we are satisfied with the model

accuracy, since it still gives us a powerful enough tool to make decisions, as can be seen in the following sections.

Second, the model can distinguish between failure bursts that span racks, and thus pose a threat to availability, and those that do not. If one rack goes down, then without other events in the cell, the availability of stripes with  $R=3$  replication will not be affected, since the storage system ensures that chunks in each stripe are placed on different racks. For one example cell, we noticed tens of medium sized failure bursts that affected one or two racks. We expected the availability of the cell to stay high, and indeed we measured MTTF = 29.52E+8 days. The model predicted 5.77E+8 days. Again, the relative error is significant, but for our purposes the model provides sufficiently accurate predictions.

Validating the model for all possible replication and Reed-Solomon encodings is infeasible, since our production cells are not set up to cover the complete space of options. However, because of our large number of production cells we are able to validate the model over a range of encodings and operating conditions.

### 8.2 Importance of recovery rate

To develop some intuition about the sensitivity of stripe availability to recovery rate, consider the situation where there are no failure bursts. Chunks fail independently with rate  $\lambda$  and recover with rate  $\rho$ . As in the previous section, consider a stripe with  $s$  chunks total which can survive losing at most  $s-r$  chunks, such as  $RS(r, s-r)$ . Thus the transition rate from state  $i \geq r$  to state  $i-1$  is  $i\lambda$ , and from state  $i$  to  $i+1$  is  $\rho$  for  $r \geq i < s$ .

We compute the MTTF, given by the time taken to reach state  $r-1$  starting in state  $s$ . Using standard methods related to *Gambler's Ruin*, [8, 14, 15, 26], this comes to:

$$\frac{1}{\lambda} \left( \sum_{k=0}^{s-r} \sum_{i=0}^k \frac{\rho^i}{\lambda^i} \frac{1}{(s-k+i)_{(i+1)}} \right)$$

where  $(a)_{(b)}$  denotes  $(a)(a-1)(a-2)\cdots(a-b+1)$ .

Assuming recoveries take much less time than node MTTF (i.e.  $\rho \gg \lambda$ ), gives a stripe MTTF of:

$$\frac{\rho^{s-r}}{\lambda^{s-r+1}} \frac{1}{(s)_{(s-r+1)}} + O\left(\frac{\rho^{s-r-1}}{\lambda^{s-r}}\right)$$

By similar computations, the recovery bandwidth consumed is approximately  $\lambda s$  per  $r$  data chunks.

Thus, with no correlated failures reducing recovery times by a factor of  $\mu$  will increase stripe MTTF by a factor of  $\mu^2$  for  $R = 3$  and by  $\mu^4$  for  $RS(9, 4)$ .

Reducing recovery times is effective when correlated failures are few. For  $RS(6, 3)$  with no correlated failures, a 10% reduction in recovery time results in a 19% reduction in unavailability. However, when correlated failures

Policy (% overhead)	MTTF(days) with correlated failures	MTTF(days) w/o correlated failures
$R = 2$ (100)	$1.47E + 5$	$4.99E + 05$
$R = 3$ (200)	$6.82E + 6$	$1.35E + 09$
$R = 4$ (300)	$1.40E + 8$	$2.75E + 12$
$R = 5$ (400)	$2.41E + 9$	$8.98E + 15$
$RS(4, 2)$ (50)	$1.80E + 6$	$1.35E + 09$
$RS(6, 3)$ (50)	$1.03E + 7$	$4.95E + 12$
$RS(9, 4)$ (44)	$2.39E + 6$	$9.01E + 15$
$RS(8, 4)$ (50)	$5.11E + 7$	$1.80E + 16$

Table 3: Stripe MTTF in days, corresponding to various data redundancy policies and space overhead.

Policy (recovery time)	MTTF (days)	Bandwidth (per PB)
$R = 2 \times 2$ (1day)	$1.08E + 10$	6.8MB/day
$R = 2 \times 2$ (1hr)	$2.58E + 11$	6.8MB/day
$RS(6, 3) \times 2$ (1day)	$5.32E + 13$	97KB/day
$RS(6, 3) \times 2$ (1hr)	$1.22E + 15$	97KB/day

Table 4: Stripe MTTF and inter-cell bandwidth, for various multi-cell schemes and inter-cell recovery times.

are taken into account, even a 90% reduction in recovery time results in only a 6% reduction in unavailability.

### 8.3 Impact of correlation on effectiveness of data-replication schemes

Table 3 presents stripe availability for several data-replication schemes, measured in MTTF. We contrast this with stripe MTTF when node failures occur at the same total rate but are assumed independent.

Note that failing to account for correlation of node failures typically results in overestimating availability by at least two orders of magnitude, and eight in the case of RS(8,4). Correlation also reduces the benefit of increasing data redundancy. The gain in availability achieved by increasing the replication number, for example, grows much more slowly when we have correlated failures. Reed Solomon encodings achieve similar resilience to failures compared to replication, though with less storage overhead.

### 8.4 Sensitivity of availability to component failure rates

One common method for improving availability is reducing component failure rates. By inserting altered failure rates of hardware into the model we can estimate the impact of potential improvements without actually building or deploying new hardware.

We find that improvements below the node (server)

layer of the storage stack do not significantly improve data availability. Assuming  $R = 3$  is used, a 10% reduction in the latent disk error rate has a negligible effect on stripe availability. Similarly, a 10% reduction in the disk failure rate increases stripe availability by less than 1.5%. On the other hand, cutting node failure rates by 10% can increase data availability by 18%. This holds generally for other encodings.

### 8.5 Single vs multi-cell replication schemes

Table 4 compares stripe MTTF under several multi-cell replication schemes and inter-cell recovery times, taking into consideration the effect of correlated failures within cells.

Replicating data across multiple cells (data centers) greatly improves availability because it protects against correlated failures. For example,  $R = 2 \times 2$  with 1 day recovery time between cells has two orders of magnitude longer MTTF than  $R = 4$ , shown in Table 3.

This introduces a tradeoff between higher replication in a single cell and the cost of inter-cell bandwidth. The extra availability for  $R = 2 \times 2$  with 1 day recoveries versus  $R = 4$  comes at an average cost of 6.8 MB/(user PB) copied between cells each day. This is the inverse MTTF for  $R = 2$ .

It should be noted that most cross-cell recoveries will occur in the event of large failure bursts. This must be considered when calculating expected recovery times between cells and the cost of on-demand access to potentially large amounts of bandwidth.

Considering the relative cost of storage versus recovery bandwidth allows us to choose the most cost effective scheme given particular availability goals.

## 9 Related Work

Several previous studies [3, 19, 25, 29, 30] focus on the failure characteristics of independent hardware components, such as hard drives, storage subsystems, or memory. As we have seen, these must be included when considering availability but by themselves are insufficient.

We focus on failure bursts, since they have a large influence on the availability of the system. Previous literature on failure bursts has focused on methods for discovering the relationship between the size of a failure event and its probability of occurrence. In [10], the existence of near-simultaneous failures in two large distributed systems is reported. The beta-binomial density and the bi-exponential density are used to fit these distributions in [6] and [24], respectively. In [24], the authors further note that using an over-simplistic model for burst size, for example a single size, could result in “dramatic inaccuracies” in practical settings. On the other hand, even

though the mean time to failure and mean time to recovery of system nodes tend to be non-uniform and correlated, this particular correlation effect has only a limited impact on system-wide availability.

There is limited previous work on discovering patterns of correlation in failures. The conditional probability of failures for each pair of nodes in a system has been proposed in [6] as a measure of correlation in the system. This computation extends heuristically to sets of larger nodes. A paradigm for discovering maximally independent groups of nodes in a system to cope with correlated failures is discussed in [34]. That paradigm involves collecting failure statistics on each node in the system and computing a measure of correlation, such as the mutual information, between every pair of nodes. Both of these approaches are computationally intensive and the results found, unlike ours, are not used to build a predictive analytical model for availability.

Models that have been developed to study the reliability of long-term storage fall into two categories, non-Markov and Markov models. Those in the first category tend to be less versatile. For example, in [5] the probability of multiple faults occurring during the recovery period of a stripe is approximated. Correlation is introduced by means of a multiplicative factor that is applied to the mean time to failure of a second chunk when the first chunk is already unavailable. This approach works only for stripes that are replicated and is not easily extendable to Reed-Solomon encoding. Moreover, the factor controlling time correlation is neither measurable nor derivable from other data.

In [33], replication is compared with Reed-Solomon with respect to storage requirement, bandwidth for write and repair and disk seeks for reads. However, the comparison assumes that sweep and repair are performed at regular intervals, as opposed to on demand.

Markov models are able to capture the system much more generally and can be used to model both replication and Reed-Solomon encoding. Examples include [21], [32], [11] and [35]. However, these models all assume independent failures of chunks. As we have shown, this assumption potentially leads to overestimation of data availability by many orders of magnitude. The authors of [20] build a tool to optimize the disaster recovery according to availability requirements, with similar goals as our analysis of multi-cell replication. However, they do not focus on studying the effect of failure characteristics and data redundancy options.

Node availability in our environment is different from previous work, such as [7, 18, 23], because we study a large system that is tightly coupled in a single administrative domain. These studies focus on measuring and predicting availability of individual desktop machines from many, potentially untrusted, domains. Other authors

[11] studied data replication in face of failures, though without considering availability of Reed-Solomon encodings or multi-cell replication.

## 10 Conclusions

We have presented data from Google’s clusters that characterize the sources of failures contributing to unavailability. We find that correlation among node failures dwarfs all other contributions to unavailability in our production environment.

In particular, though disks failures can result in permanent data loss, the multitude of transitory node failures account for most unavailability. We present a simple time-window-based method to group failure events into failure bursts which, despite its simplicity, successfully identifies bursts with a common cause. We develop analytical models to reason about past and future availability in our cells, including the effects of different choices of replication, data placement and system parameters.

Inside Google, the analysis described in this paper has provided a picture of data availability at a finer granularity than previously measured. Using this framework, we provide feedback and recommendations to the development and operational engineering teams on different replication and encoding schemes, and the primary causes of data unavailability in our existing cells. Specific examples include:

- Determining the acceptable rate of successful transfers to battery power for individual machines upon a power outage.
- Focusing on reducing reboot times, because planned kernel upgrades are a major source of correlated failures.
- Moving towards a dynamic delay before initiating recoveries, based on failure classification and recent history of failures in the cell.

Such analysis complements the intuition of the designers and operators of these complex distributed systems.

## Acknowledgments

Our findings would not have been possible without the help of many of our colleagues. We would like to thank the following people for their contributions to data collection: Marc Berhault, Eric Dorland, Sangeetha Eyunni, Adam Gee, Lawrence Greenfield, Ben Kochie, and James O’Kane. We would also like to thank a number of our colleagues for helping us improve the presentation of these results. In particular, feedback from John Wilkes, Tal Garfinkel, and Mike Marty was helpful. We

would also like to thank our shepherd Bianca Schroeder and the anonymous reviewers for their excellent feedback and comments, all of which helped to greatly improve this paper.

## References

- [1] HDFS (Hadoop Distributed File System) architecture. [http://hadoop.apache.org/common/docs/current/hdfs\\_design.html](http://hadoop.apache.org/common/docs/current/hdfs_design.html), 2009.
- [2] ANDREAS, E. S., HAEBERLEN, A., DABEK, F., GON CHUN, B., WEATHERSPOON, H., MORRIS, R., KAASHOEK, M. F., AND KUBIATOWICZ, J. Proactive replication for data durability. In *Proceedings of the 5th Intl Workshop on Peer-to-Peer Systems (IPTPS)* (2006).
- [3] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), pp. 289–300.
- [4] BAIRAVASUNDARAM, L. N., GOODSON, G. R., SCHROEDER, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEA, R. H. An analysis of data corruption in the storage stack. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), pp. 1–16.
- [5] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALO, P. A fresh look at the reliability of long-term digital storage. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), pp. 221–234.
- [6] BAKKALOGLU, M., WYLIE, J. J., WANG, C., AND GANGER, G. R. Modeling correlated failures in survivable storage systems. In *Fast Abstract at International Conference on Dependable Systems & Networks* (June 2002).
- [7] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on measurement and modeling of computer systems* (2000), pp. 34–43.
- [8] BROWN, D. M. The first passage time distribution for a parallel exponential system with repair. In *Reliability and fault tree analysis* (1974), Defense Technical Information Center.
- [9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIJKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Nov. 2006), pp. 205–218.
- [10] CHARACTERISTICS, M. F., YALAG, P., NATH, S., YU, H., GIBBONS, P. B., AND SESAN, S. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *WORLDS '04: First Workshop on Real, Large Distributed Systems* (2004).
- [11] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M. F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation* (2006), pp. 45–58.
- [12] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 1–14.
- [13] EFRON, B., AND TIBSHIRANI, R. *An Introduction to the Bootstrap*. Chapman and Hall, 1993.
- [14] EPSTEIN, R. *The Theory of Gambling and Statistical Logic*. Academic Press, 1977.
- [15] FELLER, W. *An Introduction to Probability Theory and Its Application*. John Wiley and Sons, 1968.
- [16] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 29–43.
- [17] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Opererating Systems Review* 41, 2 (2007), 60–67.
- [18] JAVADI, B., KONDO, D., VINCENT, J.-M., AND ANDERSON, D. Mining for statistical models of availability in large-scale distributed systems: An empirical study of SETI@home (2009). pp. 1–10.
- [19] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures?: a comprehensive study of storage subsystem failure characteristics. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), pp. 1–15.
- [20] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 59–62.
- [21] LIAN, Q., CHEN, W., AND ZHANG, Z. On the impact of replica placement to the reliability of distributed brick storage systems. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems* (2005), pp. 187–196.
- [22] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on fast-forward. *Communications of the ACM* 53, 3 (2010), 42–49.
- [23] MICKENS, J. W., AND NOBLE, B. D. Exploiting availability prediction in distributed systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation* (2006), pp. 73–86.
- [24] NATH, S., YU, H., GIBBONS, P. B., AND SESAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation* (2006), pp. 225–238.
- [25] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies* (2007), pp. 17–23.
- [26] RAMABHADRAN, S., AND PASQUALE, J. Analysis of long-running replicated systems. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications* (2006), pp. 1–9.
- [27] RESNICK, S. I. *Adventures in stochastic processes*. Birkhauser Verlag, 1992.
- [28] RODRIGUES, R., AND LISKOV, B. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems* (2005).
- [29] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies* (2007), pp. 1–16.
- [30] SCHROEDER, B., PINHEIRO, E., AND WEBER, W.-D. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (2009), pp. 193–204.

- [31] SCHWARZ, T. J. E., XIN, Q., MILLER, E. L., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk scrubbing in large archival storage systems. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems* (2004), 409–418.
- [32] T., S. Generalized Reed Solomon codes for erasure correction in SDDS. In *WDAS-4: Workshop on Distributed Data and Structures* (2002).
- [33] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (2002), Springer-Verlag, pp. 328–338.
- [34] WEATHERSPOON, H., MOSCOVITZ, T., AND KUBIATOWICZ, J. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems* (2002), pp. 362–367.
- [35] XIN, Q., MILLER, E. L., SCHWARZ, T., LONG, D. D. E., BRANDT, S. A., AND LITWIN, W. Reliability mechanisms for very large storage systems. In *MSS '03: Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies* (2003), pp. 146–156.

# **Scientific Data Management in the Coming Decade**

Jim Gray, Microsoft

David T. Liu, Berkeley

Maria Nieto-Santisteban & Alexander S. Szalay, Johns Hopkins University

David DeWitt, Wisconsin

Gerd Heber, Cornell

January 2005

Technical Report  
MSR-TR-2005-10

Microsoft Research  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA 98052

# Scientific Data Management in the Coming Decade

Jim Gray, Microsoft

David T. Liu, Berkeley

Maria Nieto-Santisteban & Alexander S. Szalay, Johns Hopkins University

David DeWitt, Wisconsin

Gerd Heber, Cornell

January 2005

## Data-intensive science – a new paradigm

Scientific instruments and computer simulations are creating vast data stores that require new scientific methods to analyze and organize the data. Data volumes are approximately doubling each year. Since these new instruments have extraordinary precision, the data quality is also rapidly improving. Analyzing this data to find the subtle effects missed by previous studies requires algorithms that can simultaneously deal with huge datasets and that can find very subtle effects – finding both needles in the haystack and finding very small haystacks that were undetected in previous measurements.

The raw instrument and simulation data is processed by pipelines that produce standard data products. In the NASA terminology<sup>1</sup>, the raw *Level 0* data is calibrated and rectified to *Level 1* datasets that are combined with other data to make derived *Level 2* datasets. Most analysis happens on these *Level 2* datasets with drill down to *Level 1* data when anomalies are investigated.

We believe that most new science happens when the data is examined in new ways. So our focus here is on data exploration, interactive data analysis, and integration of *Level 2* datasets.

Data analysis tools have not kept pace with our ability to capture and store data. Many scientists envy the pen-and-paper days when all their data used to fit in a notebook and analysis was done with a slide-rule. Things were simpler then; one could focus on the science rather than needing to be an information-technology-professional with expertise in arcane computer data analysis tools.

The largest data analysis gap is in this man-machine interface. How can we put the scientist back in control of his data? How can we build analysis tools that are intuitive and that augment the scientist's intellect rather than adding to the intellectual burden with a forest of arcane user tools? The real challenge is building this *smart notebook* that unlocks the data and makes it easy to capture, organize, analyze, visualize, and publish.

This article is about the data and data analysis layer within such a smart notebook. We argue that *the smart notebook* will access data presented by science centers that will provide the community with analysis tools and computational resources to explore huge data archives.

## New data-analysis methods

The demand for tools and computational resources to perform scientific data-analysis is rising even faster than data volumes. This is a consequence of three phenomena: (1) More sophisticated algorithms consume more instructions to analyze each byte. (2) Many analysis algorithms are super-linear, often needing  $N^2$  or  $N^3$  time to process  $N$  data points. And (3) IO bandwidth has not kept pace with storage capacity. In the last decade, while capacity has grown more than 100-fold, storage bandwidth has improved only about 10-fold.

These three trends: algorithmic intensity, nonlinearity, and bandwidth-limits mean that the analysis is taking longer and longer. To ameliorate these problems, scientists will need better analysis algorithms that can handle extremely large datasets with approximate algorithms (ones with near-linear execution time) and they will need parallel algorithms that can apply many processors and many disks to the problem to meet cpu-density and bandwidth-density demands.

## Science centers

These peta-scale datasets required a new work style. Today the typical scientist copies files to a local server and operates on the datasets using his own resources. Increasingly, the datasets are so large, and the application programs are so complex, that it is much more economical to move the end-user's programs to the data and only communicate questions and answers rather than moving the source data and its applications to the user's local system.

Science data centers that provide access to both the data and the applications that analyze the data are emerging as service stations for one or another scientific domain. Each of these science centers curates one or more massive datasets, curates the applications that provide access to that dataset, and supports a staff that understands the data and indeed is constantly adding to and improving the

---

<sup>1</sup> Committee on Data Management, Archiving, and Computing (CODMAC) Data Level Definitions  
[http://science.hq.nasa.gov/research/earth\\_science\\_formats.html](http://science.hq.nasa.gov/research/earth_science_formats.html)

dataset. One can see this with the SDSS at Fermilab, BaBar at SLAC, BIRN at SDSC, with Entrez-PubMed-GenBank at NCBI, and with many other datasets across other disciplines. These centers federate with others. For example BaBar has about 25 peer sites and CERN LHC expects to have many Tier1 peer sites. NCBI has several peers, and SDSS is part of the International Virtual Observatory.

The new work style in these scientific domains is to send questions to applications running at a data center and get back answers, rather than to bulk-copy raw data from the archive to your local server for further analysis. Indeed, there is an emerging trend to store a *personal workspace* (a *MyDB*) at the data center and deposit answers there. This minimizes data movement and allows collaboration among a group of scientists doing joint analysis. These personal workspaces are also a vehicle for data analysis groups to collaborate. Longer term, personal workspaces at the data center could become a vehicle for data publication – posting both the scientific results of an experiment or investigation along with the programs used to generate them in public read-only databases.

Many scientists will prefer doing much of their analysis at data centers because it will save them having to manage local data and computer farms. Some scientists may bring the small data extracts “home” for local processing, analysis and visualization – but it will be possible to do all the analysis at the data center using the personal workspace.

When a scientist wants to correlate data from two different data centers, then there is no option but to move part of the data from one place to another. If this is common, the two data centers will likely federate with one another to provide mutual data backup since the data traffic will justify making the copy.

Peta-scale data sets will require 1000-10,000 disks and thousands of compute nodes. At any one time some of the disks and some of the nodes will be broken. Such systems have to have a mechanism in place to protect against data loss, and provide availability even with a less than full configuration — a self-healing system is required. Replicating the data in science centers at different geographic locations is implied in the discussion above. Geographic replication provides both data availability and protects against data loss. Within a data center one can combine redundancy with a clever partitioning strategy to protect against failure at the disk controller or server level. While storing the data twice for redundancy, one can use different organizations (e.g. partition by space in one, and by time in the other) to optimize system performance. Failed data can be automatically recovered

from the redundant copies with no interruption to database access, much as RAID5 disk arrays do today.

All these scenarios postulate easy data access, interchange and integration. Data must be self-describing in order to allow this. This self-description, or metadata, is central to all these scenarios; it enables generic tools to understand the data, and it enables people to understand the data.

## Metadata enables data access

Metadata is the descriptive information about data that explains the measured attributes, their names, units, precision, accuracy, data layout and ideally a great deal more. Most importantly, metadata includes the data lineage that describes how the data was measured, acquired or computed.

If the data is to be analyzed by generic tools, the tools need to “understand” the data. You cannot just present a bundle-of-bytes to a tool and expect the tool to intuit where the data values are and what they mean. The tool will want to know the metadata.

To take a simple example, given a file, you cannot say much about it – it could be anything. If I tell you it is a JPEG, you know it is a bitmap in <http://www.jpeg.org/> format. JPEG files start with a header that describes the file layout, and often tells the camera, timestamp, and program that generated the picture. Many programs know how to read JPEG files and also produce new JPEG files that include metadata describing how the new image was produced. MP3 music files and PDF document files have similar roles – each is in a standard format, each carries some metadata, and each has an application suite to process and generate that file class.

If scientists are to read data collected by others, then the data must be carefully documented and must be published in forms that allow easy access and automated manipulation. In an ideal world there would be powerful tools that make it easy to capture, organize, analyze, visualize, and publish data. The tools would do data mining and machine learning on the data, and would make it easy to script workflows that analyze the data. Good metadata for the inputs is essential to make these tools automatic. Preserving and augmenting this metadata as part of the processing (data lineage) will be a key benefit of the next-generation tools.

All the derived data that the scientist produces must also be carefully documented and published in forms that allow easy access. Ideally much of this metadata would be automatically generated and managed as part of the workflow, reducing the scientist’s intellectual burden.

## Semantic convergence: numbers to objects

Much science data is in the form of numeric arrays generated by instruments and simulations. Simple and convenient data models have evolved to represent arrays and relationships among them. These data models can also represent data lineage and other metadata by including narrative text, data definitions, and data tables within the file. HDF<sup>2</sup>, NetCDF<sup>3</sup> and FITS<sup>4</sup> are good examples of such standards. They each include a library that encapsulates the files and provides a platform-independent way to read sub-arrays and to create or update files. Each standard allows easy data interchange among scientists. Generic tools that analyze and visualize these higher-level file formats are built atop each of these standards.

While the commercial world has standardized on the relational data model and SQL, no single standard or tool has critical mass in the scientific community. There are many parallel and competing efforts to build these tool suites – at least one per discipline. Data interchange outside each group is problematic. In the next decade, as data interchange among scientific disciplines becomes increasingly important, a common HDF-like format and package for all the sciences will likely emerge.

Definitions of common terminology (units and measurements) are emerging within each discipline. We are most familiar with the Universal Content Descriptors ([UCD](#)<sup>5</sup>) of the Astronomy community that define about a thousand core astrophysics units, measurements, and concepts. Almost every discipline has an analogous ontology (a.k.a., *controlled vocabulary*) effort. These efforts will likely start to converge over the next decade – probably as part of the converged format standard. This will greatly facilitate tool-building and tools since an agreement on these concepts can help guide analysis tool designs.

In addition to standardization, computerusable ontologies will help build the Semantic Web: applications will be semantically compatible beyond the mere syntactic compatibility that current-generation of Web services offer with type matching interfaces. However, it will take some time before high-performance general-purpose *ontology engines* will be available and integrated with data analysis tools.

Database users on the other hand are well positioned to prototype such applications: a database schema, though not a complete ontology in itself, can be a rich ontology

extract. SQL can be used to implement a rudimentary *semantic algebra*. The XML integration in modern Database Management Systems (DBMS) opens the door for existing standards like RDF and OWL.

Visualization or better *visual exploration* is a prime example of an application where success is determined by the ability to map a question formulated in the conceptual framework of the domain ontology onto the querying capabilities of a (meta-) data analysis backend. For the time being, a hybrid of SQL and XQuery is the only language suitable to serve as the target assembly language in this translation process.

## Metadata enables data independence

The separation of data and programs is artificial – one cannot see the data without using a program and most programs are data driven. So, it is paradoxical that the data management community has worked for 40 years to achieve something called *data independence* – a clear separation of programs from data. Database systems provide two forms of data independence termed *physical data independence* and *logical data independence*.

*Physical data independence* comes in many different forms. However, in all cases the goal is to be able to change the underlying physical data organization without breaking any application programs that depend on the old data format. One example of physical data independence is the ability of a database system to partition the rows of a table across multiple disks and/or multiple nodes of a cluster without requiring that any application programs be modified. The mapping of the fields of each row of a relational table to different disks is another important example of physical data independence. While a database system might choose to map each row to a contiguous storage container (e.g. a record) on a single disk page, it might also choose to store large, possibly infrequently referenced attributes of a table corresponding to large text objects, JPEG images, or multidimensional arrays in separate storage containers on different disk pages and/or different storage volumes in order to maximize the overall performance of the system. Again, such physical storage optimizations are implemented to be completely transparent to application programs except, perhaps, for a change in their performance. In the scientific domain the analogy would be that you could take a working application program that uses a C struct to describe its data records on disk and change the physical layout of the records without having to rewrite or even recompile the application program (or any of the other application programs that access the same data). By allowing such techniques, physical data independence allows performance improvements by reorganizing data for

<sup>2</sup> <http://hdf.ncsa.uiuc.edu/HDF5/>

<sup>3</sup> <http://my.unidata.ucar.edu/content/software/netcdf/>

<sup>4</sup> <http://fits.gsfc.nasa.gov/>

<sup>5</sup> <http://vizier.u-strasbg.fr/doc/UCD.htm>

parallelism—at little or no extra effort on the part of scientists.

Modern database systems also provide *logical data independence* that insulates programs from changes to the logical database design – allowing designers to add or delete relationships and to add information to the database. While physical data independence is used to hide changes in the physical data organizations, logical data independence hides changes in the logical organization of the data. Logical data independence is typically supported using *views*. A view defines a virtual table that is specified using a SQL query over one or more base tables and/or other views. Views serve many purposes including increased security (by hiding attributes from applications and/or users without a legitimate need for access) and enhanced performance (by materializing views defined by complex SQL queries over very large input tables). But views are primarily used to allow old programs to operate correctly even as the underlying database is reorganized and redesigned. For example, consider a program whose correct operation depends on some table T that a database administrator wants to reorganize by dividing vertically into two pieces stored in tables T' and T''. To preserve applications that depend on T, the database administrator can then define a view over T' and T'' corresponding to the original definition of table T, allowing old programs to continue to operate correctly.

In addition, data evolves. Systems evolve from EBCDIC to ASCII to Unicode, from proprietary-float to IEEE-float, from marks to euros, and from 8-character ASCII names to 1,000 character Unicode names. It is important to be able to make these changes without breaking the millions of lines of existing programs that want to see the data in the old way. Views are used to solve these problems by dynamically translating data to the appropriate formats (converting among character and number representations, converting among 6-digit and 9-digit postal codes, converting between long-and-short names, and hiding new information from old programs.) The pain of the Y2K (converting from 2-character to 4-character years) taught most organizations the importance of data independence.

Database systems use a *schema* to implement both logical and physical data independence. The schema for a database holds all metadata including table and view definitions as well as information on what indices exist and how tables are mapped to storage volumes (and nodes in a parallel database environment). Separating the data and the metadata from the programs that manipulate the data is crucial to data independence. Otherwise, it is essentially impossible for other programs to find the metadata which, in turn, makes it essentially impossible

for multiple programs to share a common database. Object-oriented programming concepts have refined the separation of programs and data. Data classes encapsulated with methods provide data independence and make it much easier to evolve the data without perturbing programs. So, these ideas are still evolving.

But the key point of this section is that an explicit and standard data access layer with precise metadata and explicit data access is essential for data independence.

### **Set-oriented data access gives parallelism**

As mentioned earlier, scientists often start with numeric data arrays from their instruments or simulations. Often, these arrays are accompanied by tabular data describing the experimental setup, simulation parameters, or environmental conditions. The data are also accompanied by documents that explain the data.

Many operations take these arrays and produce new arrays, but eventually, the arrays undergo *feature extraction* to produce *objects* that are the basis for further analysis. For example, raw astronomy data is converted to object catalogs of stars and galaxies. Stream-gauge measurements are converted to stream-flow and water-quality time-series data, serum-mass-spectrograms are converted to records describing peptide and protein concentrations, and raw high-energy physics data are converted to events.

Most scientific studies involve exploring and data mining these object-oriented tabular datasets. The scientific file-formats of HDF, NetCDF, and FITS can represent tabular data but they provide minimal tools for searching and analyzing tabular data. Their main focus is getting the tables and sub-arrays into your Fortran/C/Java/Python address space where you can manipulate the data using the programming language.

This Fortran/C/Java/Python file-at-a-time procedural data analysis is nearing the breaking point. The data avalanche is creating billions of files and trillions of events. The file-oriented approach postulates that files are organized into directories. The directories relate all data from some instrument or some month or some region or some laboratory. As things evolve, the directories become hierarchical. In this model, data analysis proceeds by searching all the relevant files – opening each file, extracting the relevant data and then moving onto the next file. When all the relevant data has been gathered in memory (or in intermediate files) the program can begin its analysis. Performing this *filter-then-analyze*, data analysis on large datasets with conventional procedural tools runs slower and slower as data volumes increase. Usually, they use only one-cpu-at-a-time; one-disk-at-a-

time and they do a brute-force search of the data. Scientists need a way (1) to use intelligent indices and data organizations to subset the search, (2) to use parallel processing and data access to search huge datasets within seconds, and (3) to have powerful analysis tools that they can apply to the subset of data being analyzed.

One approach to this is to use the MPI (Message Passing Interface) parallel programming environment to write procedural programs that stream files across a processor array – each node of the array exploring one part of the hierarchy. This is adequate for highly-regular array processing tasks, but it seems too daunting for ad-hoc analysis of tabular data. MPI and the various array file formats lack indexing methods other than partitioned sequential scan. MPI itself lacks any notion of metadata beyond file names.

As file systems grow to petabyte-scale archives with billions of files, the science community must create a synthesis of database systems and file systems. At a minimum, the file hierarchy will be replaced with a database that catalogs the attributes and lineage of each file. Set-oriented file processing will make file names increasingly irrelevant – analysis will be applied to “all data with these attributes” rather than working on a list of file/directory names or name patterns. Indeed, the files themselves may become irrelevant (they are just containers for data.) One can see a harbinger of this idea in the Map-Reduce approach pioneered by Google<sup>6</sup>. From our perspective, the key aspect of Google Map-Reduce is that it applies thousands of processors and disks to explore large datasets in parallel. That system has a very simple data model appropriate for the Google processing, but we imagine it could evolve over the next decade to be quite general.

The database community has provided automatic query processing along with CPU and IO parallelism for over two decades. Indeed, this automatic parallelism allows large corporations to mine 100-Terabyte datasets today using 1000 processor clusters. We believe that many of those techniques apply to scientific datasets<sup>7</sup>.

## Other useful database features

Database systems are also approaching the peta-scale data management problem driven largely by the need to manage huge information stores for the commercial and governmental sectors. They hide the file concept and deal with data collections. They can federate many different

sources letting the program view them all as a single data collection. They also let the program pivot on any data attributes.

Database systems provide very powerful data definition tools to specify the abstract data formats and also specify how the data is organized. They routinely allow the data to be replicated so that it can be organized in several ways (by time, by space, by other attributes). These techniques have evolved from mere indices to materialized views that can combine data from many sources.

Database systems provide powerful associative search (search by value rather than by location) and provide automatic parallel access and execution essential to peta-scale data analysis. They provide non-procedural and parallel data search to quickly find data subsets, and a many tools to automate data design and management.

In addition, data analysis using data cubes has made huge advances, and now efforts are focused on integrating machine learning algorithms that infer trends, do data clustering, and detect anomalies. All these tools are aimed at making it easy to analyze commercial data, but they are equally applicable to scientific data analysis.

## Ending the impedance mismatch

Conventional tabular database systems are adequate for analyzing objects (galaxies, spectra, proteins, events, etc.). But even there, the support for time-sequence, spatial, text and other data types is often awkward. Database systems have not traditionally supported science’s core data type: the N-dimensional array. Arrays have had to masquerade as blobs (binary large objects) in most systems. This collection of problems is generally called the *impedance mismatch* – meaning the mismatch between the programming model and the database capabilities. The impedance mismatch has made it difficult to map many science applications into conventional tabular database systems.

But, database systems are changing. They are being integrated with programming languages so that they can support object-oriented databases. This new generation of object relational database systems treats any data type (be it a native float, an array, a string, or a compound object like an XML or HTML document) as an encapsulated type that can be stored as a value in a field of a record. Actually, these systems allow the values to be either stored directly in the record (embedded) or to be pointed to by the record (linked). This linking-embedding object model nicely accommodates the integration of database systems and file systems – files are treated as linked-objects. Queries can read and write these extended types using the same techniques they use on native types.

<sup>6</sup> “[MapReduce: Simplified Data Processing on Large Clusters](#),” J. Dean, S. Ghemawat, ACM OSDI, Dec. 2004.

<sup>7</sup> “[Parallel Database Systems: the Future of High Performance Database Systems](#)”, D. DeWitt, J. Gray, CACM, Vol. 35, No. 6, June 1992.

Indeed we expect HDF and other file formats to be added as types to most database systems.

Once you can put your types and your programs inside the database you get the parallelism, non-procedural query, and data independence advantages of traditional database systems. We believe this database, file system, and programming language integration will be the key to managing and accessing peta-scale data management systems in the future.

## What's wrong with files?

Everything builds from files as a base. HDF uses files. Database systems use files. But, file systems have no metadata beyond a hierarchical directory structure and file names. They encourage a do-it-yourself- data-model that will not benefit from the growing suite of data analysis tools. They encourage do-it-yourself-access-methods that will not do parallel, associative, temporal, or spatial search. They also lack a high-level query language. Lastly, most file systems can manage millions of files, but by the time a file system can deal with billions of files, it has become a database system.

As you can see, we take an ecumenical view of what a database is. We see NetCDF, HDF, FITS, and Google Map-Reduce as nascent database systems (others might think of them as file systems). They have a schema language (metadata) to define the metadata. They have a few indexing strategies, and a simple data manipulation language. They have the start of non-procedural and parallel programming. And, they have a collection of tools to create, access, search, and visualize the data. So, in our view they are simple database systems.

## Why scientists don't use databases today

Traditional database systems have lagged in supporting core scientific data types but they have a few things scientists desperately need for their data analysis: non-procedural query analysis, automatic parallelism, and sophisticated tools for associative, temporal, and spatial search.

If one takes the controversial view that HDF, NetCDF, FITS, and Root are nascent database systems that provide metadata and portability but lack non-procedural query analysis, automatic parallelism, and sophisticated indexing, then one can see a fairly clear path that integrates these communities.

Some scientists use databases for some of their work, but as a general rule, most scientists do not. Why? Why are tabular databases so successful in commercial applications and such a flop in most scientific applications? Scientific colleagues give one or more of

the following answers when asked why they do not use databases to manage their data:

- We don't see any benefit in them. The cost of learning the tools (data definition and data loading, and query) doesn't seem worth it.
- They do not offer good visualization/plotting tools.
- I can handle my data volumes with my programming language.
- They do not support our data types (arrays, spatial, text, etc.).
- They do not support our access patterns (spatial, temporal, etc.).
- We tried them but they were too slow.
- We tried them but once we loaded our data we could no longer manipulate the data using our standard application programs.
- They require an expensive guru (database administrator) to use.

All these answers are based on experience and considerable investment. Often the experience was with older systems (a 1990 vintage database system) or with a young system (an early object-oriented database or an early version of Postgres or MySQL.) Nonetheless, there is considerable evidence that databases have to improve a lot before they are worth a second look.

## Why things are different now

The thing that forces a second look now is that the file-ftp *modus operandi* just will not work for peta-scale datasets. Some new way of managing and accessing information is needed. We argued that metadata is the key to this and that a non-procedural data manipulation language combined with data indexing is essential to being able to search and analyze the data.

There is a convergence of file systems, database systems, and programming languages. Extensible database systems use object-oriented techniques from programming languages to allow you to define complex objects as native database types. Files (or extended files like HDF) then become part of the database and benefit from the parallel search and metadata management. It seems very likely that these nascent database systems will be integrated with the main-line database systems in the next decade or that some new species of metadata driven analysis and workflow system will supplant both traditional databases and the science-specific file formats and their tool suites.

## Some hints of success

There are early signs that this is a good approach. One of us has shown that the doing analysis atop a database system is vastly simpler and runs much faster than the

corresponding file-oriented approach<sup>8</sup>. The speedup is due to better indexing and parallelism.

We have also had considerable success in adding user defined functions and stored procedures to astronomy databases. The MyDB and CasJobs work for the Sloan Digital Sky Survey give a good example of moving-programs-to-the-database<sup>9</sup>.

The BaBar experiments at SLAC manage a petabyte store of event data. The system uses a combination of Oracle to manage some of the file archive and also a physics-specific data analysis system called Root for data analysis<sup>10</sup>.

Adaptive Finite Element simulations spend considerable time and programming effort on input, output, and checkpointing. We (Heber) use a database to represent large Finite Element models. The initial model is represented in the database and each checkpoint and analysis step is written to the database. Using a database allows queries to define more sophisticated mesh partitions and allows concurrent indexed access to the simulation data for visualization and computational steering. Commercial Finite Element packages each use a proprietary form of a “database”. They are, however, limited in scope, functionality, and scalability, and are typically buried inside the particular application stack. Each worker in the MPI job gets its partition from the database (as a query) and dumps its progress to the database. These dumps are two to four orders of magnitude larger than the input mesh and represent a performance challenge in both traditional and database environments. The database approach has the added benefit that visualization tools can watch and steer the computation by reading and writing the database. Finally, while we have focused on the ability of databases to simplify and speedup the production of raw simulation data, we cannot underestimate its core competency: providing declarative data analysis interfaces. It is with these tools that scientists spend most of their time. We hope to apply similar concepts to some turbulence studies being done at Johns Hopkins.

## Summary

Science centers that curate and serve science data are emerging around next-generation science instruments. The world-wide telescope, GenBank, and the BaBar collaborations are prototypes of this trend. One group of scientists is collecting the data and managing these archives. A larger group of scientists are exploring these archives the way previous generations explored their private data. Often the results of the analysis are fed back to the archive to add to the corpus.

Because data collection is now separated from data analysis, extensive metadata describing the data in standard terms is needed so people and programs can understand the data. Good metadata becomes central for data sharing among different disciplines and for data analysis and visualization tools.

There is a convergence of the nascent-databases (HDF, NetCDF, FITS,...) which focus primarily on the metadata issues and data interchange, and the traditional data management systems (SQL and others) that have focused on managing and analyzing very large datasets. The traditional systems have the virtues of automatic parallelism, indexing, and non-procedural access, but they need to embrace the data types of the science community and need to co-exist with data in file systems. We believe the emphasis on extending database systems by unifying databases with programming languages so that one can either embed or link new object types into the data management system will enable this synthesis.

Three technical advances will be crucial to scientific analysis: (1) extensive metadata and metadata standards that will make it easy to discover what data exists, make it easy for people and programs to understand the data, and make it easy to track data lineage; (2) great analysis tools that allow scientists to easily ask questions, and to easily understand and visualize the answers; and (3) set-oriented data parallelism access supported by new indexing schemes and new algorithms that allow us to interactively explore peta-scale datasets.

The goal is a *smart notebook* that empowers scientists to explore the world’s data. Science data centers with computational resources to explore huge data archives will be central to enabling such notebooks. Because data is so large, and IO bandwidth is not keeping pace, moving code to data will be essential to performance. Consequently, science centers will remain the core vehicle and federations will likely be secondary. Science centers will provide both the archives and the institutional infrastructure to develop these peta-scale archives and the algorithms and tools to analyze them.

<sup>8</sup> “When Database Systems Meet the Grid,” M. Nieto Santisteban et. al., CIDR, 2005,  
<http://www-db.cs.wisc.edu/cidr/papers/P13.pdf>

<sup>9</sup> “Batch is back: CasJobs serving multi-TB data on the Web,” W. O’Mullane, et. al, in preparation.

<sup>10</sup> “Lessons Learned from Managing a Petabyte,” J. Becla and D. L. Wang, CIDR, 2005,  
<http://www-db.cs.wisc.edu/cidr/papers/P06.pdf>

# **What Next?**

## **A Dozen Information-Technology Research Goals**

Jim Gray

June 1999

Technical Report

MS-TR-99-50

Microsoft Research  
Advanced Technology Division  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA 98052

# What Next?

## A Dozen Information-Technology Research Goals<sup>1</sup>

Jim Gray

Microsoft Research

301 Howard St. SF, CA 94105, USA

**Abstract:** Charles Babbage's vision of computing has largely been realized. We are on the verge of realizing Vannevar Bush's Memex. But, we are some distance from passing the Turing Test. These three visions and their associated problems have provided long-range research goals for many of us. For example, the scalability problem has motivated me for several decades. This talk defines a set of fundamental research problems that broaden the Babbage, Bush, and Turing visions. They extend Babbage's computational goal to include highly-secure, highly-available, self-programming, self-managing, and self-replicating systems. They extend Bush's Memex vision to include a system that automatically organizes, indexes, digests, evaluates, and summarizes information (as well as a human might). Another group of problems extends Turing's vision of intelligent machines to include prosthetic vision, speech, hearing, and other senses. Each problem is simply stated and each is orthogonal from the others, though they share some common core technologies

### 1. Introduction

This talk first argues that long-range research has societal benefits, both in creating new ideas and in training people who can make even better ideas and who can turn those ideas into products. The education component is why much of the research should be done in a university setting. This argues for government support of long-term university research. The second part of the talk outlines sample long-term information systems research goals.

I want to begin by thanking the ACM Awards committee for selecting me as the 1998 ACM Turing Award winner. Thanks also to Lucent Technologies for the generous prize.

Most of all, I want to thank my mentors and colleagues. Over the last 40 years, I have learned from many brilliant people. Everything I have done over that time has been a team effort. When I think of any project, it was Mike and Jim, or Don and Jim, or Franco and Jim, or Irv and Jim, or Andrea and Jim, or Andreas and Jim, Dina and Jim, or Tom and Jim, or Robert and Jim, and so on to the present day. In every case it is hard for me to point to anything that I personally did: everything has been a collaborative effort.. It has been a joy to work with these people who are among my closest friends.

More broadly, there has been a large community working on the problems of making automatic and reliable data stores and transaction processing systems. I am proud to have been part of this effort, and I am proud to be chosen to represent the entire community. Thank you all!

---

<sup>1</sup> The Association of Computing Machinery selected me as the 1998 A.M. Turing Award recipient. This is approximately the text of the talk I gave in receipt of that award. The slides for that talk are at <http://research.microsoft.com/~Gray/Talks/Turing2.ppt>

## 1.1. Exponential Growth Means Constant Radical Change.

Exponential growth has been driving the information industry for the last 100 years. Moore's law predicts a doubling every 18 months. This means that in the next 18 months there will be as much new storage as all storage ever built, as much new processing as all the processors ever built. The area under the curve in the next 18 months equals the area under the curve for all human history.

In 1995, George Glider predicted that deployed bandwidth would triple every year, meaning that it doubles every 8 months. So far his prediction has been pessimistic: deployed bandwidth seems to be growing faster than that!

This doubling is only true for the underlying technology, the scientific output of our field is doubling much more slowly. The literature grows at about 15%, per year, doubling every five years.

Exponential growth cannot go on forever. E. coli (bacteria in your stomach) double every 20 minutes. Eventually something happens to limit growth. But, for the last 100 years, the information industry has managed to sustain this doubling by inventing its way around each successive barrier. Indeed, progress seems to be accelerating (see Figure 1). Some argue that this acceleration will continue, while others argue that it may stop soon – certainly if we stop innovating it will stop tomorrow.

These rapid technology doublings mean that information technology must constantly redefine itself: many things that were impossibly hard ten years ago, are now relatively easy. Tradeoffs are different now, and they will be very different in ten years.

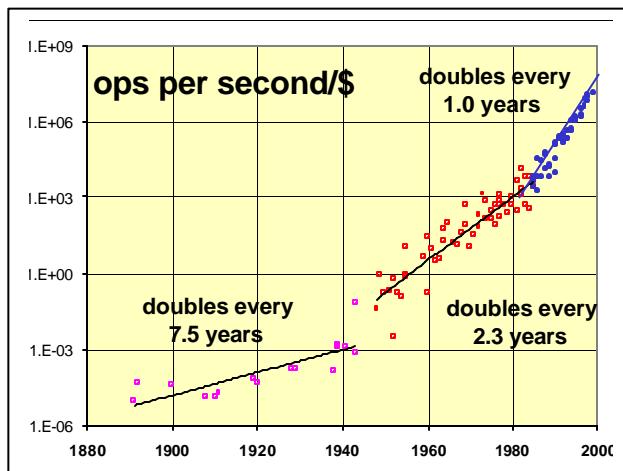


Figure 1: Graph plots performance/price versus time, where  
 $\frac{\text{Performance}}{\text{Price}} = \frac{(\text{operations-per-second})}{(\text{bits-per-op})}$   
Performance = (operations-per-second) x (bits-per-op)  
Price = system price for 3 years  
Performance/price improvements seem to be accelerating.  
There appear to be three growth curves: (1) Before  
transistors (1890-1960), performance/price was doubling  
every seven years. (2) With discrete electronics  
performance/price was doubling every 2.3 years between  
1955 and 1985. (3) Since 1985 performance/price has  
doubled every year with VLSI. Sources Hans Moravec,  
Larry Roberts, and Gordon Bell [1].

## 1.3. Cyberspace is a New World

One way to think of the Information Technology revolution is to think of cyberspace as a new continent -- equivalent to discovery of the Americas 500 years ago. Cyberspace is transforming the old world with new goods and services. It is changing the way we learn, work, and play. It is already a trillion dollar per year industry that has created a trillion dollars of wealth since 1993. Economists believe that 30% of the United States economic growth comes from the IT industry. These are high-paying high-export industries that are credited with the long boom – the US economy has skipped two recessions since this boom started.

With all this money sloshing about, there is a gold rush mentality to stake out territory. There are startups staking claims, and there is great optimism. Overall, this is a very good thing.

## 1.4. This new world needs explorers, pioneers, and settlers

Some have lost sight of the fact that most of the cyberspace territory we are now exploiting was first explored by IT pioneers a few decades ago. Those prototypes are now transforming into products.

The gold rush mentality is casing many research scientists to work on near-term projects that might make them rich, rather than taking a longer term view. Where will the next generation get its prototypes if all the explorers go to startups? Where will the next generation of students come from if the faculty leave the universities for industry?

Many believe that it is time to start Lewis and Clark style expeditions into cyberspace: major university research efforts to explore far-out ideas, and to train the next generation of research scientists. Recall that when Tomas Jefferson bought the Louisiana Territories from France, he was ridiculed for his folly. At the time, Jefferson predicted that the territories would be settled by the year 2000. To accelerate this, he sent out the Lewis & Clark expedition to explore the territories. That expedition came back with maps, sociological studies, and a corps of explorers who led the migratory wave west of the Mississippi [6].

We have a similar opportunity today: we can invest in such expeditions to create the intellectual and human seed corn for the IT industry of 2020. It is the responsibility of government, industry, and private philanthropy to make this investment for our children, much as our parents made this investment for us.

## 1.5. Pioneering research pays off in the long-term

To see what I mean, I recommend you read the NRC Brooks Southerland report, *Evolving the High-Performance Computing and Communications Initiative to Support the nations Information Infrastructure* [2] or more recently: *Funding the Revolution* [3]. Figure 2 is based on a figure that appears in both reports. It shows how government-sponsored and industry-sponsored research in Time Sharing turned into a billion dollar industry after a decade. Similar things happened with research on graphics, networking, user interfaces, and many other fields. Incidentally, much of this research work fits within Pasteur's Quadrant [5], IT

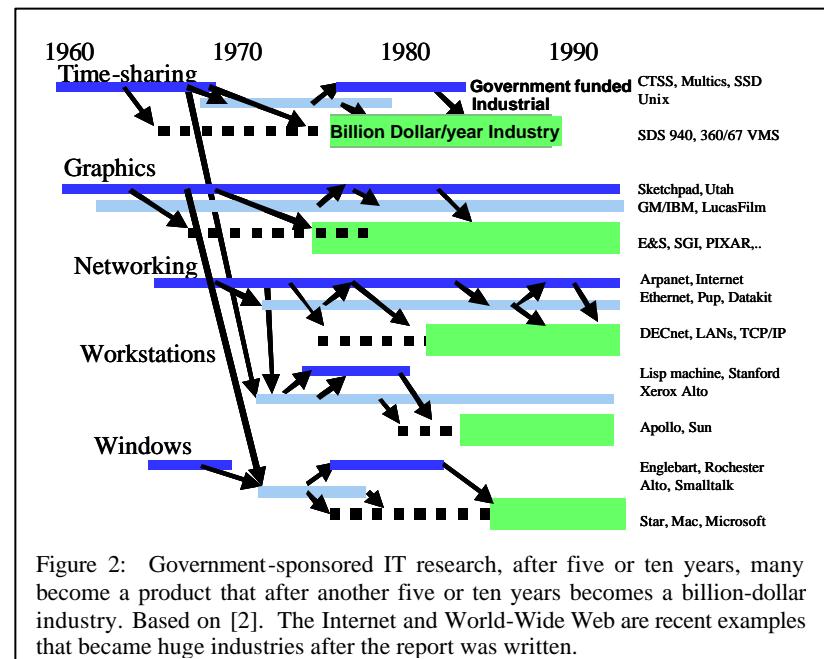


Figure 2: Government-sponsored IT research, after five or ten years, many become a product that after another five or ten years becomes a billion-dollar industry. Based on [2]. The Internet and World-Wide Web are recent examples that became huge industries after the report was written.

research generally focuses on fundamental issues, but the results have had enormous impact on and benefit to society.

Closer to my own discipline, there was nearly a decade of research on relational databases before they became products. Many of these products needed much more research before they could deliver on their usability, reliability, and performance promises. Indeed, active research on each of these problems continues to this day, and new research ideas are constantly feeding into the products. In the mean time, researchers went on to explore parallel and distributed database systems that can search huge databases, and to explore data mining techniques that can quickly summarize data and find interesting patterns, trends, or anomalies in the data. These research ideas are just now creating another billion-dollar-per-year industry.

Research ideas typically need a ten year gestation period to get to products. This time lag is shortening because of the gold rush. Research ideas still need time to mature and develop before they become products.

## **1.6. Long-term research is a public good**

If all these billions are being made, why should government subsidize the research for a trillion-dollar a year industry? After all, these companies are rich and growing fast, why don't they do their own research?

The answer is: "most of them do." The leading IT companies (IBM, Intel, Lucent, Hewlett Packard, Microsoft, Sun, Cisco, AOL, Amazon,...) spend between 5% and 15% of their revenues on Research and Development. About 10% of that R&D is not product development. I guess about 10% of that (1% of the total) is pure long-term research not connected to any near term product (most of the R of R&D is actually advanced development, trying to improve existing products). So, I guess the IT industry spends more than 500 million dollars on long-range research, which funds about 2,500 researchers. This is a conservative estimate, others estimate the number is two or three times as large. By this conservative measure, the scale of long-term industrial IT research is comparable to the number of tenure-track faculty in American computer science departments.

Most of the IT industry does fund long-range IT research; but, to be competitive some companies cannot. MCI-WorldCom has no R&D line item in the annual report, nor does the consulting company EDS. Dell computer has a small R&D budget. In general, service companies and systems integrators have very small R&D budgets.

One reason for this is that long-term research is a social good, not necessarily a benefit to the company. AT&T invented the transistor, UNIX, and the C and C++ languages. Xerox invented Ethernet, bitmap printing, iconic interfaces, and WYSIWYG editing. Other companies like Intel, Sun, 3Com, HP, Apple, and Microsoft got the main commercial benefits from this research. Society got much better products and services -- that is why the research is a public good.

Since long-term research is a public good, it needs to be funded as such: making all the boats rise together. That is why funding should come in part from society: industry is paying a tax by doing long-term research; but, the benefits are so great that society may want to add to that, and fund university research. Funding university research has the added benefit of training the next

generations of researchers and IT workers. I do not advocate Government funding of industrial research labs or government labs without a strong teaching component.

One might argue that US Government funding of long-term research benefits everyone in the world. So why should the US fund long-term research? After all, it is a social good and the US is less than 10% of the world. If the research will help the Europeans and Asians and Africans, the UN should fund long-term research.

The argument here is either altruistic or jingoistic. The altruistic argument is that long-term research is an investment for future generations world-wide. The jingoistic argument is that the US leads the IT industry. US industry is extremely good at transforming research ideas into products – much better than any other nation.

To maintain IT leadership, the US needs people (the students from the universities), and it needs new ideas to commercialize. But, to be clear, this is a highly competitive business, cyberspace is global, and the workers are international. If the United States becomes complacent, IT leadership will move to other nations.

### **1.7. The PITAC report and its recommendations.**

Most of my views on this topic grow out of a two year study by the Presidential IT Advisory Committee (PITAC) <http://www.ccic.gov/ac/report/> [4]. That report recommends that the government sponsor Lewis and Clark style expeditions to the 21<sup>st</sup> century, it recommends that the government double university IT research funding – and that the funding agencies shift the focus to long-term research. By making larger and longer-term grants, we hope that university researchers will be able to attack larger and more ambitious problems.

It also recommends that we fix the near-term staff problem by facilitating immigration of technical experts. Congress acted on that recommendation last year: adding 115,000 extra H1 visas for technical experts. The entire quota was exhausted in 6 months: the last FY99 H1 visas were granted in early June.

## 2. Long Range IT Systems Research Goals

Having made a plea for funding long-term research. What exactly are we talking about? What are examples of long-term research goals that we have in mind? I present a dozen examples of long-term systems research projects. Other Turing lectures have presented research agendas in theoretical computer science. My list complements those others.

### 2.1. What Makes a Good Long Range Research Goal?

Before presenting my list, it is important to describe the attributes of a good goal. A good long-range goal should have five key properties:

**Understandable:** The goal should be simple to state. A sentence, or at most a paragraph should suffice to explain the goal to intelligent people. Having a clear statement helps recruit colleagues and support. It is also great to be able to tell your friends and family what you actually do.

**Challenging:** It should not be obvious how to achieve the goal. Indeed, often the goal has been around for a long time. Most of the goals I am going to describe have been explicit or implicit goals for many years. Often, there is a camp who believe the goal is impossible.

**Useful:** If the goal is achieved, the resulting system should be clearly useful to many people -- I do not mean just computer scientists, I mean people at large.

**Testable:** Solutions to the goal should have a simple test so that one can measure progress and one can tell when the goal is achieved.

**Incremental:** It is very desirable that the goal has intermediate milestones so that progress can be measured along the way. These small steps are what keep the researchers going.

### 2.2. Scalability: a sample goal

To give a specific example, much of my work was motivated by the *scalability* goal described to me by John Cocke. The goal is to devise a software and hardware architecture that scales up without limits. Now, there has to be some kind of limit: billions of dollars, or giga-watts, or just space. So, the more realistic goal is to be able to scale from one node to a million nodes all working on the same problem.

1. **Scalability:** Devise a software and hardware architecture that scales up by a factor for  $10^6$ . That is, an application's storage and processing capacity can automatically grow by a factor of a million, doing jobs faster ( $10^6$ x speedup) or doing  $10^6$  larger jobs in the same time ( $10^6$ x scaleup), just by adding more resources.

Attacking the scalability problem leads to work on all aspects of large computer systems. The system grows by adding modules, each module performing a small part of the overall task. As the system grows, data and computation has to migrate to the new modules. When a module fails, the other modules must mask this failure and continue offering services. Automatic management, fault-tolerance, and load-distribution are still challenging problems.

The benefit of this vision is that it suggests problems and a plan of attack. One can start by working on automatic parallelism and load balancing. Then work on fault tolerance or automatic management. One can start by working on the 10x scaleup problem with an eye to the larger problems.

My particular research focused on building highly-parallel database systems, able to service thousands of transactions per second. We developed a simple model that describes when transactions can be run in parallel, and also showed how to automatically provide this parallelism. This work led to studies of why computers fail, and how to improve computer availability. Lately, I have been exploring very large database applications like <http://terraserver.microsoft.com/> and <http://www.sdss.org/>.

Returning to the scalability goal, how has work on scalability succeeded over the years? Progress has been astonishing, for two reasons.

1. There has been a lot of it.
2. Much of it has come from an unexpected direction – the Internet.

The Internet is a world-scale computer system that surprised us all. A computer system of 100 million nodes, and now merely doubling in size each year. It will probably grow to be much larger. The PITAC worries that we do not know how to scale the network and the servers. I share that apprehension, and think that much more research is needed on protocols and network engineering.

On the other hand, we do know how to build huge servers. Companies have demonstrated single systems that can process a billion transactions per day. That is comparable to all the cash transactions in the US in a day. It is comparable to all the AOL interactions in a day. It is a lot.

In addition, these systems can process a transaction for about a micro-dollar. That is, they are very cheap. It is these cheap transactions that allow free access to the Internet data servers. In essence, accesses can be paid for by advertising.

Through a combination of hardware (60% improvement per year) and software (40% improvement per year) performance and price performance have doubled every year since 1985. Several more years of this progress are in sight (see Figures 1 and 3.)

Still, we have some very dirty laundry. Computer scientists have yet to make parallel programming easy. Most of the scalable systems like databases, file servers, and online transaction processing are embarrassingly parallel. The parallelism comes from the application. We have merely learned how to preserve it, rather

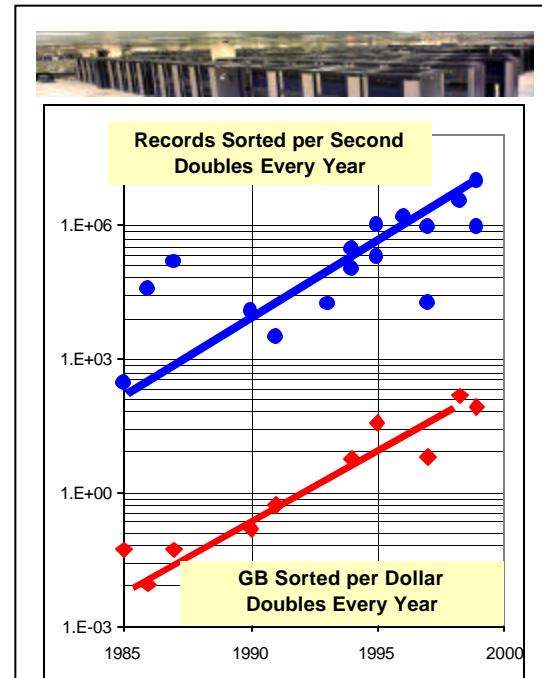


Figure 3: Top: a picture of the 6,000 node computer at LANL. Bottom: Scalability of computer systems on the simple task of sorting data. Sort speed and sorting price-performance has doubled each year over the last 15 years. This is progress is partly due to hardware and partly due to software.

than creating it automatically.

When it comes to running a big monolithic job on a highly parallel computer, there has been modest progress. Parallel database systems that automatically provide parallelism and give answers more quickly, have been very successful. Parallel programming systems that ask the programmer to explicitly write parallel programs have been embraced only as a last resort. The best examples of this are the Beowulf clusters used by scientists to get very inexpensive supercomputers (<http://www.beowulf.org/>), and the huge ASCI machines, consisting of thousands of processors (see top of Figure 3.). Both these groups report spectacular performance, but they also report considerable pain.

Managing these huge clusters is also a serious problem. Only some of the automation postulated for scaleable systems has been achieved. Virtually all the large clusters have a custom-built management system. We will return to this issue later.

The scalability problem will become more urgent in the next decade. It appears that new computer architectures will have multiple execution streams on a single chip: so each processor chip will be an SMP (symmetric multi-processor). Others are pursuing processors imbedded in memories, disks, and network interface cards (e.g. <http://iram.cs.berkeley.edu/istore/>). Still another trend is the movement of processors to Micro-Electro-Mechanical Systems (MEMS). Each 10\$ MEMS will have sensors, effectors, and onboard processing. Programming a collection of a million MEMS systems is a challenge [7].

So, the scaleability problem is still an interesting long-term goal. But, in this lecture, I would like to describe a broad spectrum of systems-research goals.

## 2. Long-term IT Systems Research Goals

In looking for the remaining eleven long-term research problems I read the previous Turing lectures, consulted many people, and ultimately settled on organizing the problems in the context of three seminal visionaries of our field. In the 1870s Charles Babbage had the vision of programmable computers that could store information and could compute much faster than people. In the 1940's Vannevar Bush articulated his vision of a machine that stored all human knowledge. In 1950, Alan Turing argued that machines would eventually be intelligent.

The problems I selected are systems problems. Previous Turing talks have done an excellent job of articulating an IT theory research agenda. Some of the problems here necessarily have an "and prove it" clause. These problems pose challenging theoretical issues. In picking the problems, I tried to avoid specific applications – trying rather to focus on the core issues of information technology that seem generic to all applications.

One area where I wish I had more to say, is the topic of ubiquitous computing. Alan Newell first articulated the vision of an intelligent universe in which every part of our environment is intelligent and networked [8]. Many of the research problems mentioned here bear on this ubiquitous computing vision, but I have been unable to crisply state a specific long-term research goal that is unique to it.

### 3. Turing's vision of machine intelligence

To begin, recall Alan Turing's famous "Computing Machinery and Intelligence" paper published in 1950 [9]. Turing argued that in 50 years, computers would be intelligent.



This was a *very* radical idea at that time. The debate that raged then is largely echoed today: Will computers be tools, or will they be conscious entities, having identity, volition, and free will? Turing was a pragmatist. He was just looking for intelligence, not trying to define or evaluate free will. He proposed a test, now called the *Turing Test*, that for him was an intelligence litmus test.

#### 3.1 The Turing Test

The Turing Test is based on the *Imitation Game*, played by three people. In the imitation game, a man and a woman are in one room, and a judge is in the other. The three cannot see one another, so they communicate via Email. The judge questions them for five minutes, trying to discover which of the two is the man and which is the woman. This would be very easy, except that the man lies and pretends to be a woman. The woman tries to help the judge find the truth. If the man is a really good impersonator, he might fool the judge 50% of the time. In practice, it seems the judge is right about 70% of the time.

Now, the Turning Test replaces the man with a computer pretending to be a woman. If the computer can fool the judge 30% of the time, it passes the Turing Test.

2. **The Turing Test:** Build a computer system that wins the imitation game at least 30% of the time.

Turing's actual text on this matter is worth re-reading. What he said was:

*"I believe that in about fifty years' time it will be possible, to programme computers, with a storage capacity of about  $10^9$ , to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning. The original question, "Can machines think?" I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."*

With the benefit of hindsight, Turing's predictions read very well. His technology forecast was astonishingly accurate, if a little pessimistic. The typical computer has the requisite capacity, and is comparably powerful. Turing estimated that the human memory is between  $10^{12}$  and  $10^{15}$  bytes, and the high end of that estimate stands today.

On the other hand, his forecast for machine intelligence was optimistic. Few people characterize computers as intelligent. You can interview ChatterBots on the Internet (<http://www.loebner.net/Prizef/loebner-prize.html>) and judge for yourself. I think they are still a long way from passing the Turing Test. But, there has been enormous progress in the last 50 years, and I expect that eventually a machine will indeed pass the Turing Test. To be more

specific, I think it will happen within the next 50 years because I am persuaded by the argument that we are nearing parity with the storage and computational power of simple brains.

To date, machine-intelligence has been more of a partnership with scientists: a symbiotic relationship. To give some stunning examples of progress in machine intelligence, computers helped with the proofs of several theorems (the four-color problem is the most famous example [9]), and have solved a few open problems in mathematics. It was front page news when IBM's Deep Blue beat the world chess champion. Computers help design almost everything now – they are used in conceptualization, simulation, manufacturing, testing, and evaluation.

In all these roles, computers are acting as tools and collaborators rather than intelligent machines. Vernor Vinge calls this IA (intelligence amplification) as opposed to AI [11]. These computers are not forming new concepts. They are typically executing static programs with very little adaptation or learning. In the best cases, there is a pre-established structure in which parameters automatically converge to optimal settings for this environment. This is adaptation, but it is not learning new things they way a child, or even a spider seems to.

Despite this progress, there is general pessimism about machine intelligence, and artificial intelligence (AI). We are still in AI winter. The AI community promised breakthroughs, but they did not deliver. Enough people have gotten into enough trouble on the Turing Test, that it has given rise to the expression *Turing Tar Pit* “Where everything is possible but nothing is easy.” *AI complete* is short for even harder than NP complete. This is in part a pun on Turing’s most famous contribution: the proof that very simple computers can compute anything computable.

Paradoxically, today it is much easier to research machine intelligence because the machines are so much faster and so much less expensive. This is the “counting argument” that Turing used. Desktop machines should be about as intelligent as a spider or a frog, and supercomputers ought to be nearing human intelligence.

The argument goes as follows. Various experiments and measures indicate that the human brain stores at most  $10^{14}$  bytes (100 Terabytes). The neurons and synaptic fabric can execute about 100 tera-operations per second. This is about thirty times more powerful than the biggest computers today. So, we should start seeing intelligence in these supercomputers any day now (just kidding). Personal computers are a million times slower and 10,000 times smaller than that.

This is similar to the argument that the human genome is about a billion base pairs. 90% of it is junk, 90% of the residue is in common with chimpanzees, and 90% of that residue is in common with all people. So each individual has just a million unique base pairs (and would fit on a floppy disk).

Both these arguments appear to be true. But both indicate that we are missing something *very* fundamental. There is more going on here than we see. Clearly, there is more than a megabyte difference among babies. Clearly, the software and databases we have for our super-computers is not on a track to pass the Turing Test in the next decade. Something quite different is needed. Out-of-the-box, radical thinking is needed.

We have been handed a puzzle: genomes and brains work. But we are clueless what the solution is. Understanding the answer is a wonderful long-term research goal.

### **3.2. Three more Turing Tests: prosthetic hearing, speech, and vision.**

Implicit in the Turing Test, are two sub-challenges that in themselves are quite daunting: (1) read and understand as well as a human, and (2) think and write as well as a human. Both of these appear to be as difficult as the Turing Test itself.

Interestingly, there are three other problems that appear to be easier, but still very difficult: There has been great progress on computers hearing and identifying natural language, music, and other sounds. Speech-to-text systems are now quite useable. Certainly they have benefited from faster and cheaper computers, but the algorithms have also benefited from deeper language understanding, using dictionaries, good natural language parsers, and semantic nets. Progress in this area is steady, and the error rate is dropping about 10% per year. Right now unlimited-vocabulary, continuous speech with a trained speaker and good microphone recognizes about 95% of the words. I joke that computers understand English much better than most people (note: most people do not understand English at all.) Joking aside, many blind, hearing impaired, and disabled people use speech-to-text and text-to-speech systems for reading, listening, or typing.

Speaking as well as a person, given a prepared text, has received less attention than the speech recognition problem, but it is an important way for machines to communicate with people.

There was a major thrust in language translation in the 1950s, but the topic has fallen out of favor. Certainly simple language translation systems exist today. A system that passes the Turing Test in English, will likely have a very rich internal representation. If one teaches such a system a second language, say Mandarin, then the computer would likely have a similar internal representation for information in that language. This opens up the possibility for faithful translation between languages. There may be a more direct path to good language translation, but so far it is not obvious. Bablefish (<http://babelfish.altavista.com/>) is a fair example of the current state of the art. It translates context-free sentences between English and French, German, Italian, Portuguese, and Spanish. It translates the sentence “Please pass the Turing Test” to “Veuillez passer l'essai de Turing”, which translates back to “Please pass the test of Turing.”

The third area, is visual recognition: build a system that can identify objects and recognize dynamic object behaviors in a scene (horse-running, man-smiling, body gestures,...).

Visual rendering is an area where computers already outshine all but the best of us. Again, this is a man-machine symbiosis, but the “special effects” and characters of from Lucasfilm and Pixar are stunning. Still, the challenge remains to make it easy for kids and adults to create such illusions in real time for fun or to communicate ideas.

The Turing Test also suggests prosthetic memory, but I'll reserve that for Bush's section. So the three additional Turing Tests are:

- 3. **Speech to text:** Hear as well as a native speaker.
- 4. **Text to speech:** Speak as well as a native speaker.
- 5. **See as well as a person:** recognize objects and behavior.

As limited as our current progress is in these three areas, it is still a boon to the handicapped and in certain industrial settings. Optical character recognition is used to scan text and speech

synthesizers read the text aloud. Speech recognition systems are used by deaf people to listen to telephone calls and are used by people with carpal tunnel syndrome and other disabilities to enter text and commands. Indeed, some programmers use voice input to do their programming.

For a majority of the deaf, devices that couple directly to the auditory nerve could convert sounds to nerve impulses thereby replacing the eardrum and the cochlea. Unfortunately, nobody yet understands the coding used by the body. But, it seems likely that this problem will be solved someday.

Longer term these prosthetics will help a much wider audience. They will revolutionize the interface between computers and people. When computers can see and hear, it should be much easier and less intrusive to communicate with them. They will also help us to see better, hear better, and remember better.

I hope you agree that these four tests meet the criterion I set out for a good goal, they are understandable, challenging, useful, testable, and they each have incremental steps.

## 4. Bush's Memex

Vannevar Bush was an early information technologist: he had built analog computers at MIT. During World War II, he ran the Office of Scientific Research and Development. As the war ended he wrote a wonderful piece for the government called the *Endless Frontier* [13][14] that has defined America's science policy for the last fifty years.



In 1945 Bush published a visionary piece "As We May Think" in *The Atlantic Monthly*, <http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm> [14]. In that article, he described *Memex*, a desk that stored "a billion books", newspapers, pamphlets, journals, and other literature, all hyper-linked together. In addition, Bush proposed a set of glasses with an integral camera that would photograph things on demand, and a Dictaphone that would record what was said. All this information was also fed into *Memex*.

*Memex* could be searched by looking up documents, or by following references from one document to another. In addition, anyone could annotate a document with links, and those annotated documents could be shared among users. Bush realized that finding information in *Memex* would be a challenge, so he postulated "association search", finding documents that matched some similarity criteria.

Bush proposed that the machine should recognize spoken commands, and that it type when spoken to. And, if that was not enough, he casually mentioned that "a direct electrical path to the human nervous system" might be a more efficient and effective way to ask questions and get answers.

Well, 50 years later, *Memex* is almost here. Most scientific literature is online. The scientific literature doubles every 8 years, and most of the last 15 years are online. Much of Turing's work and Bush's articles are online. Most literature is also online, but it is protected by copyright and so not visible to the web.

The [Library of Congress](#) is online and gets more web visitors each day than regular visitors: even though just a tiny part of the library is online. Similarly, the [ACM97](#) conference was recorded and converted to a web site. After one month, five times more people had visited the web site than the original conference. After 18 months, 100,000 people had spent a total of 50,000 hours watching the presentations on the web site. This is substantially more people and time than attendees at the actual event. The site now averages 200 visitors and 100 hours per week.

This is all wonderful, but anyone who has used the web is aware of its limitations: (1) it is hard to find things on the web, and (2) many things you want are not yet on the web. Still, the web is very impressive and comes close to Bush's vision. It is the first place I look for information. Information is increasingly migrating online to cyberspace. Most new information is created online. Today, it is about 50 times less expensive to store 100 letters (1 MB) on magnetic disk, than to store them in a file cabinet (ten cents versus 5 dollars.) Similarly, storing a photo online is about five times less expensive than printing it and storing the photo in a shoebox. Every year, the cost of cyberspace drops while the cost of real space rises.

The second reason for migrating information in cyberspace is that it can be searched by robots. Programs can scan large document collections, and find those that match some predicate. This is faster, cheaper, easier, and more reliable than people searching the documents. These searches

can also be done from anywhere -- a document in England can be easily accessed by someone in Australia.

So, why isn't everything in Cyberspace? Well, the simple answer is that most information is valuable property and currently, cyberspace does not have much respect for property rights. Indeed, the cyberspace culture is that all information should be freely available to anyone anytime. Perhaps the information may come cluttered with advertising, but otherwise it should be free. As a consequence, most information on the web is indeed advertising in one form or another.

There are substantial technical issues in protecting intellectual property, but the really thorny issues revolve around the law (e.g., What protection does each party have under the law given that cyberspace is trans-national?), and around business issues (e.g., What are the economic implications of this change?). The latter two issues are retarding the move of "high value" content to the Internet, and preventing libraries from offering Internet access to their collections. Often, customers must come to the physical library to browse the electronic assets.

Several technical solutions to copy-protect intellectual property are on the table. They all allow the property owner to be paid for use of his property on a per view, or subscription, or time basis. They also allow the viewers and listeners to use the property easily and anonymously. But, until the legal and business issues are resolved, these technical solutions will be of little use.

Perhaps better schemes will be devised that protect intellectual property, but in the mean time we as scientists must work to get our scientific literature online and freely available. Much of it was paid for by taxpayers or corporations, so it should not be locked behind publisher's copyrights. To their credit, our technical society the ACM has taken a very progressive view on web publishing. Your ACM technical articles can be posted on your web site, your department's web site, and on the Computer Science Online Research Repository ([CoRR](#)). I hope other societies will follow ACM's lead on this.

## 4.1 Personal Memex

Returning to the research challenges, the sixth problem is to build a personal Memex. A box that records everything you see, hear, or read. Of course it must come with some safeguards so that only *you* can get information out of it. But, it should on command, find the relevant event and display it to you. The key thing about this Memex is that it does not do any data analysis or summarization, it just returns what it sees and hears.

6. **Personal Memex:** Record everything a person sees and hears, and quickly retrieve any item on request.

Since it only records what you see and hear, personal Memex seems not to violate any copyright issues [15]. It still raises some difficult ethical issues. If you and I have a private conversation, does your Memex have the right to disclose our conversation to others? Can you sell the conversation without my permission? But, if one takes a very conservative approach: only record with permission and make everything private, then Memex seems within legal bounds. But the designers must be vigilant on these privacy issues.

Memex seems feasible today for everything but video. A personal record of everything you ever read is about 25 GB. Recording everything you hear is a few terabytes. A personal Memex will grow at 250 megabytes (MB) per year to hold the things you read, and 100 gigabytes (GB) per year to hold the things you hear. This is just the capacity of one modern magnetic tape or 2 modern disks. In three years it should be one disk or tape per year. So, if you start recording now, you should be able to stick with one or two tapes for the rest of your life.

Video Memex seems beyond our technology today, but in a few decades, it will likely be economic. High visual quality would be hundreds times more -- 80 terabytes (TB) per year. That is a lot of storage, eight petabytes (PB) per lifetime. It will continue to be more than most individuals can afford. Of course, people may want very high definition and stereo images of what they see. So, this 8 petabyte could easily rise to ten times that. On the other hand, techniques that recognize objects might give huge image compression. To keep the rate to a terabyte a year, the best we can offer with current compression technology is about ten TV-quality frames per second. Each decade the quality will get at least 100x better. Capturing, storing, organizing, and presenting this information is a fascinating long-term research goal.

## 4.2 World Memex

What about Bush's vision of putting *all* professionally produced information into Memex? Interestingly enough, a book is less than a megabyte of text and all the books and other printed literature is about a petabyte in Unicode. There are about 500,000 movies (most very short). If you record them with DVD quality they come to about a petabyte. If you scanned all the books and other literature in the Library of Congress the images would be a few petabytes. There are 3.5 million sound recordings (most short) which add a few more petabytes. So the consumer-quality digitized contents of the Library of Congress total a few petabytes. Librarians who want to preserve the images and sound want 100x more fidelity in recording and scanning the images, thus getting an exabyte. Recording all TV and radio broadcasts (everywhere) would add 100 PB per year.

Michael Lesk did a nice analysis of the question "How much information is there?" He concludes that there are 10 or 20 exabytes of recorded information (excluding personal and surveillance videotapes) [16]. An interesting fact is that the storage industry shipped exabyte of disk storage in 1999 and about 100 exabytes of tape storage. Near-line (tape) and on-line (disk) storage cost between a 10 k\$ and 100 k\$ per terabyte. Prices are falling faster than Moore's law – storage will likely be a hundred times cheaper in ten years. So, we are getting close to the time when we can record most of what exists very inexpensively. For example, a lifetime cyberspace cemetery plot for your most recent 1 MB research report or photo of your family should cost about 25 cents. That is 10 cents for this year, 5 cents for next year, 5 cents for the successive years, and 5 cents for insurance.

Where does this lead us? If everything will be in cyberspace, how do we find anything? Anyone who has used the web search engines knows both joy and frustration: sometimes they are wonderful and find just what you want. They do some summarization, giving title and first few sentences. But they do very little real analysis or summarization.

So, the next challenge after a personal Memex that just returns exactly what you have seen, undigested, is a Memex that analyzes a large corpus of material and then presents it to you an a convenient way. Raj Reddy described a system that can read a textbook and then answer the

questions at the end of the text as well as a (good) college student [17]. A more demanding task is to take a corpus of text, like the Internet or the Computer Science journals, or Encyclopedia Britannica, and be able to answer summarization questions about it as well as a human expert in that field.

Once we master text, the next obvious step is to build a similar system that can digest a library of sounds (speeches, conversations, music, ...). A third challenge is a system that can absorb and summarize a collection of pictures, movies, and other imagery. The Library of congress has 115 million text and graphic items, the Smithsonian has 140 million items which are 3D (e.g. the Wright Brothers airplane). Moving those items to cyberspace is an interesting challenge. The visible humans ([http://www.nlm.nih.gov/research/visible/visible\\_human.html](http://www.nlm.nih.gov/research/visible/visible_human.html)), millimeter slices versions of two cadavers, give a sense of where this might go. Another exciting project is copying some of Leonardo DeVinci's work to cyberspace.

7. **World Memex:** Build a system that given a text corpus, can answer questions about the text and summarize the text as precisely and quickly as a human expert in that field. Do the same for music, images, art, and cinema.

The challenge in each case is to automatically parse and organize the information. Then when a someone has a question, the question can be posed in a natural interface that combines a language, gesture, graphics, and forms interface. The system should respond with answers which are appropriate to the level of the user.

This is a demanding task. It is probably AI Complete, but it an excellent goal, probably simpler and more useful than a computer that plays the imitation game as well as a human.

### 4.3 Telepresence

One interesting aspect of being able to record everything is that other people can observe the event, either immediately, or retrospectively. I now routinely listen to lectures recorded at another research institution. Sometimes, these are “live”, but usually they are on-demand. This is extraordinarily convenient -- indeed, many of us find this time-shifting to be even more valuable than space-shifting. But, it is fundamentally just television-on-demand; or if it is audio only, just radio-on-demand – turning the Internet into the world’s most expensive VCR.

A much higher-quality experience is possible with the use of computers and virtual reality. By recording an event in high-fidelity from many angles, computers can reconstruct any the scene at high-fidelity from any perspective. This allows a viewer to sit anywhere in the space, or wander around the space. For a sporting event, the spectator can be on the field watching the action close up. For a business meeting, the participant can sit in the meeting and look about to read facial gestures and body language as the meeting progresses.

The challenge is to record events and then create a virtual environment on demand that allows the observer to experience the event as well as actually being there. This is called **Tele-Observer** because it is really geared to a passive observer of an event – either because it is a past event, or because there are so many observers that they must be passive (they are watching, not interacting). Television and radio give a low-quality version of this today, but they are completely passive.

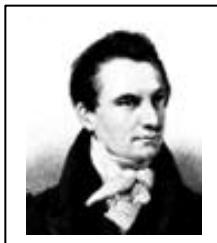
The next challenge is to allow the participant to interact with the other members of the event, i.e. be **Tele-Present**. Tele-presence already exists in the form of telephones, teleconferences, and chat rooms. But, again the experience there is very much lower quality than actually being present. Indeed, people often travel long distances just to get the improved experience. The operational test for Telepresence is that a group of students taking a telepresent class score as well as students who were physically present in the classroom with the instructor. And that the instructor has the same rapport with the telepresent students, as he has with the physically present ones.

8. **TelePresence:** Simulate being some other place retrospectively as an observer (TeleObserver): hear and see as well as actually being there, and as well as a participant, and simulate being some other place as a participant (TelePresent): interacting with others and with the environment as though you are actually there.

There is great interest in allowing a telepresent person to physically interact with the world via a robot. The robot is electrical and mechanical engineering, the rest of the system is information technology. That is why I have left out the robot. As Dave Huffman said: “Computer Science has the patent on the byte and the algorithm. EE has the electron and Physics has energy and matter.”

## 5. Charles Babbage's Computers

Turing's and Bush's visions are heady stuff: machine intelligence, recording everything, and telepresence. Now it is time to consider long-term research issues for traditional computers. Charles Babbage (1791-1871) had two computer designs, a difference engine, that did numeric computations well, and a fully programmable analytical engine that had punched card programs, a 3-address instruction set, and a memory to hold variables. Babbage loved to compute things and was always looking for trends and patterns. machines to help him do his computations.



He wanted these

By 1955, Babbage's vision of a computer had been realized. Computers with the power he envisioned were generating tables of numbers, were doing bookkeeping, and generally doing what computers do. Certainly there is more to do on Babbage's vision. We need better computational algorithms, and better and faster machines.

But I would like to focus on another aspect of Babbage's computers. What happens when computers become free, infinitely fast, with infinite storage, and infinite bandwidth? Now, this is not likely to happen anytime soon. But, computation has gotten a 10 million times cheaper since 1950 and a trillion times cheaper since 1899 (see Figure1). Indeed, in the last decade they have gotten a thousand times cheaper. So, from the perspective of 1950, computers today are almost free, and have almost infinite speed, storage, and bandwidth.

Figure 1 charts how things have changed since Babbage's time. It measures the price-performance of these systems. Larry Roberts proposed a performance/price metric of

$$\text{Performance / Price} = \frac{\text{Operations Per Second} \cdot \text{Bits Per Operation}}{\text{System Price}}$$

This measures *bits-processed per dollar*. In 1969 Roberts observed that this metric was doubling about every 18 months. This was contemporaneous with Gordon Moore's observation about gates-per silicon chip doubling, but was measured at the system level. Using data from Hans' Moravac's web site (<http://www.frc.ri.cmu.edu/~hpm/book98/>), and some corrections from Gordon Bell, we plotted the data from Herman Hollerith forward. Between 1890 and 1945, systems were either mechanical or electro-mechanical. Their performance doubled about every seven years. In the 1950's, computing shifted to tubes and transistors and the doubling time dropped to 2.3 years. In 1985, microprocessors and VLSI came on the scene. Through a combination of lower systems prices and much faster machines, the doubling time has dropped to one year.

This acceleration in performance/price is astonishing, and it changes the rules. Similar graphs apply to the cost of storage and bandwidth.

This is real deflation. When processing, storage, and transmission cost micro-dollars, then the only real value is the data and its organization. But we computer scientists have some dirty laundry: our "best" programs typically have a bug for every thousand lines of code, and our "free" computers cost at least a thousand dollars a year in care-and-feeding known as system administration.

Computer owners pay comparatively little for them today. A few hundred dollars for a palmtop or desktop computer, a few thousand dollars for a workstation, and perhaps a few tens of thousands for a server. These folks do not want to pay a large operations staff to manage their

systems. Rather, they want a self-organizing system that manages itself. For simple systems like handheld computers, the customer just wants the system to work. Always be up, always store the data, and never lose data. When the system needs repairing, it should “call home” and schedule a fix. Either the replacement system arrives in the mail, or a replacement module arrives in the mail – and no information has been lost. If it is a software or data problem, the software or data is just refreshed from the server in the sky. If you buy a new appliance, you just plug it in and it refreshes from the server in the sky (just as though the old appliance had failed).

This is the vision that most server companies are working towards in building information appliances. You can see prototypes of it by looking at WebTVs or your web browser for example.

## 5.1. Trouble-Free Systems

So, who manages the server-in-the sky? Server systems are more complex. They have some semi-custom applications, they have much heavier load, and often they provide the very services the hand-held, appliances, and desktops depend on. To some extent, the complexity has not disappeared, it has just moved.

People who own servers do not mind managing the server content, that is their business. But, they do not want to be systems management experts. So, server systems should be self managing. The human systems manager should set goals, polices, and a budget. The system should do the rest. It should distribute work among the servers. When new modules arrive, they should just add to the cluster when they are plugged in. When a server fails, its storage should have been replicated somewhere else, so the storage and computation can move to those new locations. When some hardware breaks, the system should diagnose itself and order replacement modules which arrive by express mail. Hardware and software upgrades should be automatic.

This suggests the very first of the Babbage goals: trouble-free systems.

9. **Trouble-Free Systems:** Build a system used by millions of people each day and yet administered and managed by a single part-time person.

The operational test for this is that it serves millions of people each day, and yet it is managed by a fraction of a person who does all the administrative tasks. Currently, such a system would need 24-hour a day coverage by a substantial staff. With special expertise required for upgrades, maintenance, system growth, database administration, backups, network management, and the like.

## 5.2. Dependable Systems

Two issues hiding in the previous requirements deserve special attention. There have been a rash of security problems recently: Melissa, Chernobyl, and now a mathematical attack on RSA that makes 512-bit keys seem dangerously small.

We cannot trust our assets to cyberspace if this trend continues. A major challenge for systems designers is to develop a system which only services authorized uses. Service cannot be denied.

Attackers cannot destroy data, nor can they force the system to deny service to authorized users. Moreover, users cannot see data unless they are so authorized.

The added hook here is that most systems are penetrated by stealing passwords and entering as an authorized user. Any authentication based on passwords or other tokens seems too insecure. I believe we will have to go to physio-metric means like retinal scans or some other unforgeable authenticator – and that all software must be signed in an unforgeable way.

The operational test for this research goal is that a tiger team cannot penetrate the system. Unfortunately, that test does not really prove security. So this is one of those instances where the security system must rest on a *proof* that it is secure, and that all the threats are known and are guarded against.

The second attribute is that the system should always be available. We have gone from 90% availability in the 1950s to 99.99% availability today for well managed systems. Web uses experience about 99% availability due to the fragile nature of the web, its protocols, and the current emphasis on time-to-market.

Nonetheless, we have added three 9s in 45 years, or about 15 years per order-of-magnitude improvement in availability. We should aim for five more 9s: an expectation of one second outage in a century. This is an extreme goal, but it seems achievable if hardware is very cheap and bandwidth is very high. One can replicate the services in many places, use transactions to manage the data consistency, use design diversity to avoid common mode failures, and quickly repair nodes when they fail. Again, this is not something you will be able to test: so achieving this goal will require careful analysis and proof.

10. **Secure System:** Assure that the system of problem 9 only services authorized users, service cannot be denied by unauthorized users, and information cannot be stolen (and prove it.)

11. **AlwaysUp:** Assure that the system is unavailable for less than one second per hundred years -- 8 9's of availability (and prove it.)

### 5.3. Automatic Programming.

This brings us to the final problem: Software is expensive to write. It is the only thing in cyberspace that is getting more expensive, and less reliable. Individual pieces of software are not really less reliable, it is just that the typical program has one bug per thousand lines after it has been tested and retested. The typical software product grows fast, and so adds bugs as it grows.

You might ask how programs could be so expensive? It is simple: designing, creating, and documenting a program costs about 20\$ per line. It costs about 150% of that to test the code. Then once the code is shipped, it costs that much again to support and maintain the code over its lifetime.

This is grim news. As computers become cheaper, there will be more and more programs and this burden will get worse and worse.

The solution so far is to write fewer lines of code by moving to high-level non-procedural languages. There have been some big successes. Code reuse from SAP, PeopleSoft, and others are an enormous savings to large companies building semi-custom applications. The companies still write a lot of code, but only a small fraction of what they would have written otherwise.

The user-written code for many database applications and many web applications is tiny. The tools in these areas are very impressive. Often they are based on a scripting language like JavaScript and a set of pre-built objects. Again an example of software reuse. End users are able to create impressive websites and applications using these tools.

If your problem fits one of these pre-built paradigms, then you are in luck. If not, you are back to programming in C++ or Java and producing a 5 to 50 lines of code a day at a cost of 100\$ per line of code.

So, what is the solution? How can we get past this logjam? Automatic programming has been the Holy Grail of programming languages and systems for the last 45 years. Sad to report, there has been relatively little progress -- perhaps a factor of 10, but certainly not a factor of 1,000 improvement in productivity unless your problem fits one of the application-generator paradigms mentioned earlier.

Perhaps the methodical software-engineering approaches will finally yield fruit, but I am pessimistic. I believe that an entirely new approach is needed. Perhaps it is too soon, because this is a Turing Tar Pit. I believe that we have to (1) have a high level specification language that is a thousand times easier and more powerful than the current languages, (2) computers should be able to compile the language, and (3) the language should be powerful enough so that all applications can be described.

We have systems today that do any two of these three things, but none that do all three. In essence this is the imitation game for a programming staff. The customer comes to the programming staff and describes the application. The staff returns with a proposed design. There is discussion, a prototype is built and there is more discussion. Eventually, the desired application is built.

**12. Automatic Programmer:** Devise a specification language or user interface that:

- (a) makes it easy for people to express designs (1,000x easier),
- (b) computers can compile, and
- (c) can describe all applications (is complete).

The system should reason about application, asking questions about exception cases and incomplete specification. But it should not be onerous to use.

The operational test is replace the programming staff with a computer, and produce a result that is better and requires no more time than dealing with a typical human staff. Yes, it will be a while until we have such a system, but if Alan Turing was right about machine intelligence, it's just be a matter of time.

## **6. Summary**

These are a dozen very interesting research problems. Each is a long-term research problem. Now you can see why I want the government to invest in long-term research. I suspect that in 50 years future generations of computer scientists will have made substantial progress on each of these problems. Paradoxically, many (5) of the dozen problems appear to require machine intelligence as envisioned by Alan Turing.

The problems fall in the three broad categories: Turing's intelligent machines improving the human-computer interface, Bush's Memex recording, analyzing, and summarizing everything that happens, and Babbage's computers which will finally be civilized so that they program themselves, never fail, and are safe.

No matter how it turns out, I am sure it will be very exciting. As I said at the beginning, progress appears to be accelerating: the base-technology progress in the next 18 months will equal all previous progress, if Moore's law holds. And there are lots more doublings after that.

## A Dozen Long-Term Systems Research Problems.

1. **Scalability:** Devise a software and hardware architecture that scales up by a factor for  $10^6$ . That is, an application's storage and processing capacity can automatically grow by a factor of a million, doing jobs faster ( $10^6$ x speedup) or doing  $10^6$  larger jobs in the same time ( $10^6$ x scaleup), just by adding more resources.
2. **The Turing Test:** Build a computer system that wins the imitation game at least 30% of the time.
3. **Speech to text:** Hear as well as a native speaker.
4. **Text to speech:** Speak as well as a native speaker.
5. **See as well as a person:** recognize objects and motion.
6. **Personal Memex:** Record everything a person sees and hears, and quickly retrieve any item on request.
7. **World Memex:** Build a system that given a text corpus, can answer questions about the text and summarize the text as precisely and quickly as a human expert in that field. Do the same for music, images, art, and cinema.
8. **TelePresence:** Simulate being some other place retrospectively as an observer (TeleOberserver): hear and see as well as actually being there, and as well as a participant, and simulate being some other place as a participant (TelePresent): interacting with others and with the environment as though you are actually there.
9. **Trouble-Free Systems:** Build a system used by millions of people each day and yet administered and managed by a single part-time person.
10. **Secure System:** Assure that the system of problem 9 only services authorized users, service cannot be denied by unauthorized users, and information cannot be stolen (and prove it).
11. **AlwaysUp:** Assure that the system is unavailable for less than one second per hundred years -- 8 9's of availability (and prove it).
12. **Automatic Programmer:** Devise a specification language or user interface that:
  - (a) makes it easy for people to express designs (1,000x easier),
  - (b) computers can compile, and
  - (c) can describe all applications (is complete).The system should reason about application, asking questions about exception cases and incomplete specification. But it should not be onerous to use.

## 7. References

- [1] Graph based on data in Hans P. Moravec *Robot, Mere Machines to Transcendent Mind*, Oxford, 1999, ISBN 0-19-511630-5, (<http://www.frc.ri.cmu.edu/~hpm/book98/>) personal communication with Larry Roberts who developed the metric in 1969, and personal communication with Gordon Bell who helped analyze the data and corrected some errors.
- [2] CSTB–NRC, *Evolving the High-Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure*, National Academy Press, Washington DC, 1995.
- [3] CSTB–NRC *Funding a Revolution, Government Support for Computing Research*, National Academy Press, Washington DC, 1999. ISBN 0-309-6278-0.
- [4] *Information Technology Research: Investing in Our Future, President's Information Technology Advisory Committee, Report to the President, Feb. 1999*. National Coordination Office for Computing, Information, and Communications, Arlington VA.
- [5] *Donald E. Stokes, Pasteur's Quadrant: Basic Science and Technological Innovation*, Brookings, 1997, ISBN 0-8157-8178-4.
- [6] Stephen E. Ambrose, *Undaunted Courage: Meriwether Lewis, Thomas Jefferson, and the Opening of the American West*, Simon & Schuster, NY, 1996, ISBN: 0684811073
- [7] “From Micro-device to Smart Dust”, *Science News*, 6/26/97, Vol. 152(4), pp 62-63
- [8] Alan Newell, “Fairy Tales,” appears in R. Kruzelweil, *The Age of Intelligent Machines*, MIT Press, 1990, ISBN: 0262610795, pp 420-423
- [9] Alan M. Turing, “Computing Machinery and Intelligence”, *Mind*, Vol. LIX. 433-460, 1950). Also on the web at many sites. K. Appel and W. Haken, “The solution of the four-color-map problem,” *Scientific American*, Oct 1977, 108-121,  
<http://www.math.gatech.edu/~thomas/FC/fourcolor.html> (1995) has a “manual” proof
- [11] Vernor Vinge, “Technological Singularity.” VISION-21 Symposium sponsored by NASA Lewis Research Center and the Ohio Aerospace Institute, March, 1993. also at  
<http://www.frc.ri.cmu.edu/~hpm/book98/com.ch1/vinge.singularity.html>
- [12] *Endless Frontier: Vannevar Bush, Engineer of the American Century*, G. Pascal Zachary, Free Press, 1997 ISBN: 0-684-82821-9
- [13] Vannevar Bush, *Science-The Endless Frontier*, Appendix 3, "Report of the Committee on Science and the Public Welfare," Washington, D.C.: U.S. Government Printing Office, 1945. Reprinted as National Science Foundation Report 1990, online  
[http://rits.stanford.edu/siliconhistory/Bush/Bush\\_text.html](http://rits.stanford.edu/siliconhistory/Bush/Bush_text.html)
- [14] Vannevar Bush ”As We May Think” *The Atlantic Monthly*, July 1945,  
<http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm>
- [15] Anne Wells-Branscomb, *Who Owns Information?: From Privacy to Public Access* Basic Books, 1995, ISBN: 046509144X.
- [16] Micheal Lesk, “How much information is there in the world?”  
<http://www.lesk.com/mlesk/ksg97/ksg.html>
- [17] Raj Reddy, “To Dream The Possible Dream,” CACM, May 1996, Vol. 39, No. 5 pp106-112.

# Volley: Automated Data Placement for Geo-Distributed Cloud Services

Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman

Microsoft Research, {sagarwal, jdunagan, navendu, ssaroiu, alecw}@microsoft.com

Harbinder Bhogan

University of Toronto, hbhogan@cs.toronto.edu

**Abstract:** As cloud services grow to span more and more globally distributed datacenters, there is an increasingly urgent need for automated mechanisms to place application data across these datacenters. This placement must deal with business constraints such as WAN bandwidth costs and datacenter capacity limits, while also minimizing user-perceived latency. The task of placement is further complicated by the issues of shared data, data inter-dependencies, application changes and user mobility. We document these challenges by analyzing month-long traces from Microsoft’s Live Messenger and Live Mesh, two large-scale commercial cloud services.

We present Volley, a system that addresses these challenges. Cloud services make use of Volley by submitting logs of datacenter requests. Volley analyzes the logs using an iterative optimization algorithm based on data access patterns and client locations, and outputs migration recommendations back to the cloud service.

To scale to the data volumes of cloud service logs, Volley is designed to work in SCOPE [5], a scalable MapReduce-style platform; this allows Volley to perform over 400 machine-hours worth of computation in less than a day. We evaluate Volley on the month-long Live Mesh trace, and we find that, compared to a state-of-the-art heuristic that places data closest to the primary IP address that accesses it, Volley simultaneously reduces datacenter capacity skew by over 2 $\times$ , reduces inter-datacenter traffic by over 1.8 $\times$  and reduces 75th percentile user-latency by over 30%.

## 1 Introduction

Cloud services continue to grow rapidly, with ever more functionality and ever more users around the globe. Because of this growth, major cloud service providers now use tens of geographically dispersed datacenters, and they continue to build more [10]. A major unmet challenge in leveraging these datacenters is automatically placing user data and other dynamic application data, so that a single cloud application can serve each of its users from the best datacenter for that user.

At first glance, the problem may sound simple: determine the user’s location, and migrate user data to the closest datacenter. However, this simple heuristic ignores two major sources of cost to datacenter operators: WAN bandwidth between datacenters, and over-provisioning datacenter capacity to tolerate highly skewed datacenter utilization. In this paper, we show that a more sophisticated approach can both dramatically reduce these costs and still further reduce user latency. The more sophisticated approach is motivated by the following trends in modern cloud services:

**Shared Data:** Communication and collaboration are increasingly important to modern applications. This trend is evident in new business productivity software, such as Google Docs [16] and Microsoft Office Online [32], as well as social networking applications such as Facebook [12], LinkedIn [26], and Twitter [43]. These applications have in common that many reads and writes are made to shared data, such as a user’s Facebook wall, and the user experience is degraded if updates to shared data are not quickly reflected to other clients. These reads and writes are made by groups of users who need to collaborate but who may be scattered worldwide, making it challenging to place and migrate the data for good performance.

**Data Inter-dependencies:** The task of placing shared data is made significantly harder by inter-dependencies between data. For example, updating the wall for a Facebook user may trigger updating the data items that hold the RSS feeds of multiple other Facebook users. These connections between data items form a communication graph that represents increasingly rich applications. However, the connections fundamentally transform the problem’s mathematics: in addition to connections between clients and their data, there are connections in the communication graph in-between data items. This motivates algorithms that can operate on these more general graph structures.

**Application Changes:** Cloud service providers want to release new versions of their applications with ever greater frequency [35]. These new application features

can significantly change the patterns of data sharing and data inter-dependencies, as when Facebook released its instant messaging feature.

**Reaching Datacenter Capacity Limits:** The rush in industry to build additional datacenters is motivated in part by reaching the capacity constraints of individual datacenters as new users are added [10]. This in turn requires automatic mechanisms to rapidly migrate application data to new datacenters to take advantage of their capacity.

**User Mobility:** Users travel more than ever today [15]. To provide the same rapid response regardless of a user’s location, cloud services should quickly migrate data when the migration cost is sufficiently inexpensive.

In this paper we present Volley, a system for automatic data placement across geo-distributed datacenters. Volley incorporates an iterative optimization algorithm based on weighted spherical means that handles the complexities of shared data and data inter-dependencies, and Volley can be re-run with sufficient speed that it handles application changes, reaching datacenter capacity limits and user mobility. Datacenter applications make use of Volley by submitting request logs (similar to Pinpoint [7] or X-Trace [14]) to a distributed storage system. These request logs include the client IP addresses, GUIDs identifying the data items accessed by the client requests, and the structure of the request “call tree”, such as a client request updating Facebook wall 1, which triggers requests to data items 2 and 3 handling Facebook user RSS feeds.

Volley continuously analyzes these request logs to determine how application data should be migrated between datacenters. To scale to these data sets, Volley is designed to work in SCOPE [5], a system similar to Map-Reduce [11]. By leveraging SCOPE, Volley performs more than 400 machines hours worth of computation in less than a day. When migration is found to be worthwhile, Volley triggers application-specific data migration mechanisms. While prior work has studied placing static content across CDNs, Volley is the first research system to address placement of user data and other dynamic application data across geographically distributed datacenters.

Datacenter service administrators make use of Volley by specifying three inputs. First, administrators define the datacenter locations and a cost and capacity model (e.g., the cost of bandwidth between datacenters and the maximum amount of data per datacenter). Second, they choose the desired trade-off between upfront migration cost and ongoing better performance, where ongoing performance includes both minimizing user-perceived latency and reducing the costs of inter-datacenter communication. Third, they specify data replication levels and other constraints (e.g., three replicas in three different

datacenters all located within Europe). This allows administrators to use Volley while respecting other external factors, such as contractual agreements and legislation.

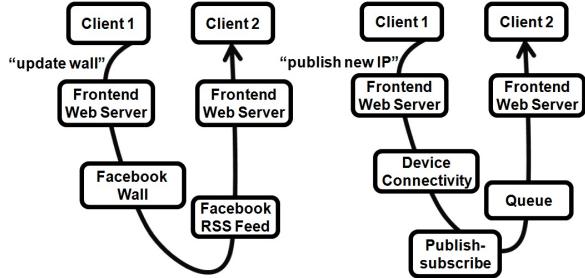
In the rest of this paper, we first quantify the prevalence of trends such as user mobility in modern cloud services by analyzing month-long traces from Live Mesh and Live Messenger, two large-scale commercial datacenter services. We then present the design and implementation of the Volley system for computing data placement across geo-distributed datacenters. Next, we evaluate Volley analytically using the month-long Live Mesh trace, and we evaluate Volley on a live testbed consisting of 20 VMs located in 12 commercial datacenters distributed around the world. Previewing our results, we find that compared to a state-of-the-art heuristic, Volley can reduce skew in datacenter load by over 2 $\times$ , decrease inter-datacenter traffic by over 1.8 $\times$ , and reduce 75th percentile latency by over 30%. Finally, we survey related work and conclude.

## 2 Analysis of Commercial Cloud-Service Traces

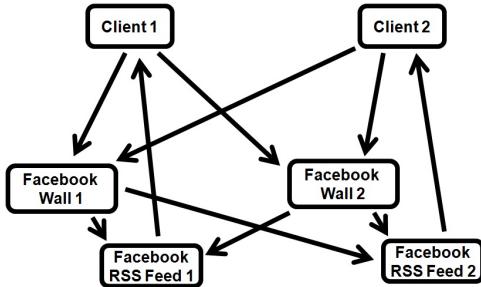
We begin by analyzing workload traces collected by two large datacenter applications, Live Mesh [28] and Live Messenger [29]. Live Mesh provides a number of communication and collaboration features, such as file sharing and synchronization, as well as remote access to devices running the Live Mesh client. Live Messenger is an instant messaging application. In our presentation, we also use Facebook as a source for examples due to its ubiquity.

The Live Mesh and Live Messenger traces were collected during June 2009, and they cover all users and devices that accessed these services over this entire month. The Live Mesh trace contains a log entry for every modification to hard state (such as changes to a file in the Live Mesh synchronization service) and user-visible soft state (such as device connectivity information stored on a pool of in-memory servers [1]). The Live Messenger trace contains all login and logoff events, all IM conversations and the participants in each conversation, and the total number of messages in each conversation. The Live Messenger trace does not specify the sender or the size of individual messages, and so for simplicity, we model each participant in an IM conversation as having an equal likelihood of sending each message, and we divide the total message bytes in this conversation equally among all messages. A prior measurement study describes many aspects of user behavior in the Live Messenger system [24]. In both traces, clients are identified by application-level unique identifiers.

To estimate client location, we use a standard commercial geo-location database [34] as in prior work [36].



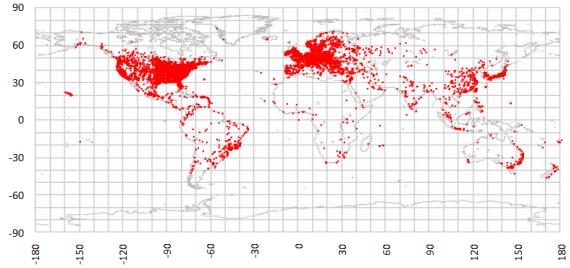
**Figure 1.** Simplified data inter-dependencies in Facebook (left) and Live Mesh (right). In Facebook, an “updated wall” request arrives at a Facebook wall data item, and this data item sends the request to an RSS feed data item, which then sends it to the other client. In Live Mesh, a “publish new IP” request arrives at a Device Connectivity data item, which forwards it to a Publish-subscribe data item. From there, it is sent to a Queue data item, which finally sends it on to the other client. These pieces of data may be in different datacenters, and if they are, communication between data items incurs expensive inter-datacenter traffic.



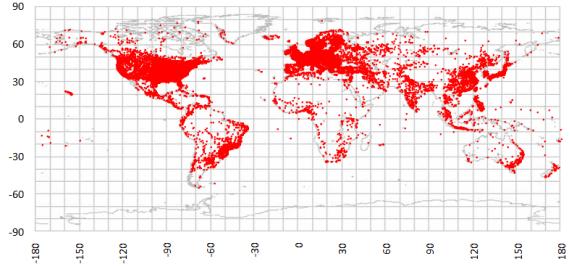
**Figure 2.** Two clients, their four data items and the communication between them in the simplified Facebook example. Data placement requires appropriately mapping the four data items to datacenters so as to simultaneously achieve low inter-datacenter traffic, low datacenter capacity skew, and low latency.

The database snapshot is from June 30th 2009, the very end of our trace period.

We use the traces to study three of the trends motivating Volley: shared data, data inter-dependencies, and user mobility. The other motivating trends for Volley, rapid application changes and reaching datacenter capacity limits, are documented in other data sources, such as developers describing how they build cloud services and how often they have to release updates [1, 35]. To provide some background on how data inter-dependencies arise in commercial cloud services, Figure 1 shows simplified examples from Facebook and Live Mesh. In the Facebook example, Client 1 updates its Facebook wall, which is then published to Client 2; in Facebook, this allows users to learn of each other’s activities. In the Live Mesh example, Client 1 publishes its new IP address,



**Figure 3.** Distribution of clients in the Mesh trace.



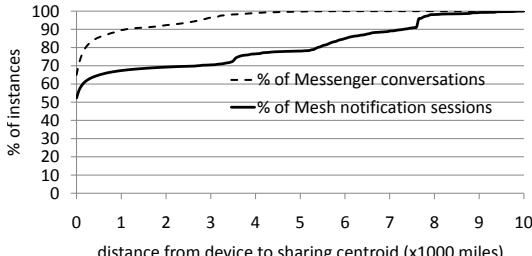
**Figure 4.** Distribution of clients in the Messenger trace.

which is routed to Client 2, enabling Client 2 to connect directly to Client 1; in Live Mesh, this is referred to as a notification session, and it enables both efficient file sharing and remote device access. The Figure caption provides additional details, as do other publications [1]. In both cases, the client operations involve multiple datacenter items; inter-datacenter traffic is minimized by co-locating these items, while the latency of this particular request is minimized by placing the data items as close as possible to the two clients.

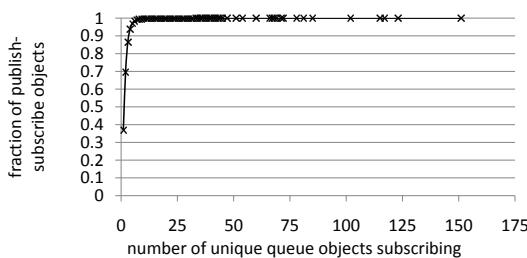
Figure 2 attempts to convey some intuition for why data sharing and inter-dependencies make data placement challenging. The figure shows the web of connections between just two clients in the simplified Facebook example; these inter-connections determine whether a mapping of data items to datacenters achieves low inter-datacenter traffic, low datacenter capacity skew, and low latency. Actual cloud services face this problem with hundreds of millions of clients. Each client may access many data items, and these data items may need to communicate with each other to deliver results to clients. Furthermore, the clients may access the data items from a variety of devices at different locations. This leads to a large, complicated graph.

In order to understand the potential for this kind of inter-connection to occur between clients that are quite distant, we begin by characterizing the geographic diversity of clients in the traces.

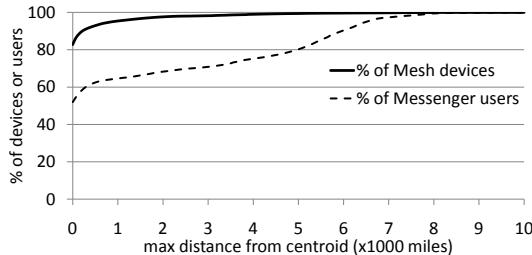
**Client Geographic Diversity:** We first study the traces to understand the geographic diversity of these services’ client populations. Figures 3 and 4 show the distribution of clients in the two traces on a map of the world. The figures show that both traces contain a geographi-



**Figure 5.** Sharing of data between geographically distributed clients in the Messenger and Mesh traces. Large amounts of sharing occur between distant clients.



**Figure 6.** Data inter-dependencies in Live Mesh between Publish-subscribe objects and Queue objects. A user that updates a hard state data item, such as a document stored in Live Mesh, will cause an update message to be generated at the Publish-subscribe object for that document, and all Queue objects that subscribe to it will receive a copy of the message. Each user or device that is sharing that document will have a unique Queue. Many Publish-subscribe objects are subscribed to by a single Queue, but there is a long tail of popular objects that are subscribed to by many Queues.



**Figure 7.** Mobility of clients in the Messenger and Mesh traces. Most clients do not travel. However, a significant fraction do travel quite far.

cally diverse set of clients, and thus these service’s performance may significantly benefit from intelligent data placement.

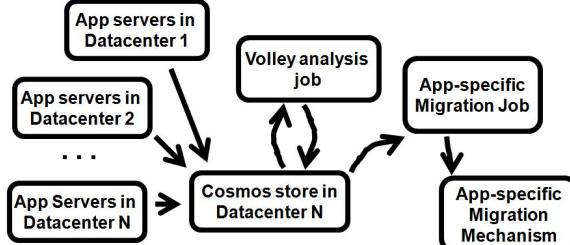
**Geographically Distant Data Sharing:** We next study the traces to understand whether there is significant data sharing among distant users. For each particular data item, we compute its centroid (centroid on a sphere is computed using the weighted spherical mean methodology, which we describe in detail in Section 3). Fig-

ure 5 shows a CDF for the distance over which clients access data placed according to its centroid; data that is not shared has an access distance of 0, as does data shared by users whose IP addresses map to the same geographic location. Given the amount of collaboration across nations both within corporations and between them, it is perhaps not surprising that large amounts of sharing happens between very distant clients. This data suggests that even for static clients, there can be significant benefits to placing data closest to those who use it most heavily, rather than just placing it close to some particular client that accesses the data.

**Data Inter-dependencies:** We proceed to study the traces to understand the prevalence of data inter-dependencies. Our analysis focuses on Live Mesh because data inter-dependencies in Live Messenger have been documented in detail in prior work [24]. Figure 6 shows the number of Queue objects subscribing to receive notifications from each Publish-subscribe object; each such subscription creates a data inter-dependency where the Publish-subscribe object sends messages to the Queue object. We see that some Publish-subscribe objects send out notifications to only a single Queue object, but there is a long tail of popular Publish-subscribe objects. The presence of such data inter-dependencies motivates the need to incorporate them in Volley.

**Client Mobility:** We finally study the traces to understand the amount of client mobility in these services’ client populations. Figure 7 shows a CDF characterizing client mobility over the month of the trace. To compute this CDF, we first computed the location of each client at each point in time that it contacted the Live Mesh or Live Messenger application using the previously described methodology, and we then compute the client’s centroid. Next, we compute the maximum distance between each client and its centroid. As expected, we observe that most clients do not move. However, a significant fraction do move (more in the Messenger trace than the Mesh trace), and these movements can be quite dramatic – for comparison purposes, antipodal points on the earth are slightly more than 12,000 miles apart.

From these traces, we cannot characterize the reason for the movement. For example, it could be travel, or it could be that the clients are connecting in through a VPN to a remote office, causing their connection to the public Internet to suddenly emerge in a dramatically different location. For Volley’s goal of reducing client latency, there is no need to distinguish between these different causes; even though the client did not physically move in the VPN case, client latency is still minimized by moving data closer to the location of the client’s new connection to the public Internet. The long tail of client mobility suggests that for some fraction of clients, the ideal data placement changes significantly during this month.



**Figure 8.** Dataflow for an application using Volley.

This data does leave open the possibility that some fraction of the observed clients are bots that do not correspond to an actual user (i.e., they are modified clients driven by a program). The current analysis does filter out the automated clients that the service itself uses for doing performance measurement from various locations. Prior work has looked at identifying bots automatically [45], and Volley might benefit from leveraging such techniques.

### 3 System Design and Implementation

The overall flow of data in the system is shown in Figure 8. Applications make use of Volley by logging data to the Cosmos [5] distributed storage system. The administrator must also supply some inputs, such as a cost and capacity model for the datacenters. The Volley system frequently runs new analysis jobs over these logs, and computes migration decisions. Application-specific jobs then feed these migration decisions into application-specific data migration mechanisms. We now describe these steps in greater detail.

#### 3.1 Logging Requests

To utilize Volley, applications have to log information on the requests they process. These logs must enable correlating requests into “call trees” or “runtime paths” that capture the logical flow of control across components, as in Pinpoint [7] or X-Trace [14]. If the source or destination of a request is movable (i.e., because it is a data item under the control of the cloud service), we log a GUID identifier rather than its IP address; IP addresses are only used for endpoints that are not movable by Volley, such as the location that a user request came from. Because Volley is responsible for placing all the data named by GUIDs, it already knows their current locations in the steady state. It is sometimes possible for both the source and destination of a request to be referred to by GUIDs—this would happen, for example, in Figure 1, where the GUIDs would refer to Client 1’s Facebook wall and Client 2’s Facebook RSS feed. The exact fields in the Volley request logs are shown in Table 1. In total, each record requires only 100 bytes.

There has been substantial prior work modifying applications to log this kind of information, and many com-

mercial applications (such as the Live Mesh and Live Messenger services analyzed in Section 2) already log a superset of this data. For such applications, Volley can incorporate simple filters to extract out the relevant subset of the logs.

For the Live Mesh and Live Messenger commercial cloud services, the data volumes from generating Volley logs are much less than the data volumes from processing user requests. For example, recording Volley logs for all the requests for Live Messenger, an IM service with hundreds of millions of users, only requires hundreds of GB per day, which leads to an average bandwidth demand in the tens of Mbps [24]. Though we cannot reveal the exact bandwidth consumption of the Live Mesh and Live Messenger services due to confidentiality concerns, we can state that tens of Mbps is a small fraction of the total bandwidth demands of the services themselves. Based on this calculation, we centralize all the logs in a single datacenter; this then allows Volley to run over the logs multiple times as part of computing a recommended set of migrations.

#### 3.2 Additional Inputs

In addition to the request logs, Volley requires four inputs that change on slower time scales. Because they change on slower time scales, they do not noticeably contribute to the bandwidth required by Volley. These additional inputs are (1) the requirements on RAM, disk, and CPU per transaction for each type of data handled by Volley (e.g., a Facebook wall), (2) a capacity and cost model for all the datacenters, (3) a model of latency between datacenters and between datacenters and clients, and (4) optionally, additional constraints on data placement (e.g., legal constraints). Volley also requires the current location of every data item in order to know whether a computed placement keeps an item in place or requires migration. In the steady state, these locations are simply remembered from previous iterations of Volley.

In the applications we have analyzed thus far, the administrator only needs to estimate the average requirements on RAM, disk and CPU per data item; the administrator can then rely on statistical multiplexing to smooth out the differences between data items that consume more or fewer resources than average. Because of this, resource requirements can be estimated by looking at OS-provided performance counters and calculating the average resource usage for each piece of application data hosted on a given server.

The capacity and cost models for each datacenter specify the RAM, disk and CPU provisioned for the service in that datacenter, the available network bandwidth for both egress and ingress, and the charging model for service use of network bandwidth. While energy usage is a significant cost for datacenter owners, in our expe-

Request Log Record Format

Field	Meaning
Timestamp	Time in seconds when request was received (4B)
Source-Entity	A GUID if the source is another data item, an IP address if it is a client (40B)
Request-Size	Bytes in request (8B)
Destination-Entity	Like Source-Entity, either a GUID or an IP address (40B)
Transaction-Id	Used to group related requests (8B)

**Table 1.** To use Volley, the application logs a record with these fields for every request. The meaning and size in bytes of each field are also shown.

Migration Proposal Record Format

Field	Meaning
Entity	The GUID naming the entity (40B)
Datacenter	The GUID naming the new datacenter for this entity (40B)
Latency-Change	The average change in latency per request to this object (4B)
Ongoing-Bandwidth-Change	The change in egress and ingress bandwidth per day (4B)
Migration-Bandwidth	The one-time bandwidth required to migrate (4B)

**Table 2.** Volley constructs a set of proposed migrations described using the records above. Volley then selects the final set of migrations according to the administrator-defined trade-off between performance and cost.

rience this is incorporated as a fixed cost per server that is factored in at the long timescale of server provisioning. Although datacenter owners may be charged based on peak bandwidth usage on individual peering links, the unpredictability of any given service’s contribution to a datacenter-wide peak leads datacenter owners to charge services based on total bandwidth usage, as in Amazon’s EC2 [2]. Accordingly, Volley helps services minimize their total bandwidth usage. We expect the capacity and cost models to be stable at the timescale of migration. For fluid provisioning models where additional datacenter capacity can be added dynamically as needed for a service, Volley can be trivially modified to ignore provisioned capacity limits.

Volley needs a latency model to make placement decisions that reduce user perceived latency. It allows different static or dynamic models to be plugged in. Volley migrates state at large timescales (measured in days) and hence it should use a latency model that is stable at that timescale. Based on the large body of work demonstrating the effectiveness of network coordinate systems, we designed Volley to treat latencies between IPs as distances in some n-dimensional space specified by the model. For the purposes of evaluation in the paper, we rely on a static latency model because it is stable over these large timescales. This model is based on a linear regression of great-circle distance between geographic coordinates; it was developed in prior work [36], where it was compared to measured round trip times across millions of clients and shown to be reasonably accurate. This latency model requires translating client IP addresses to geographic coordinates, and for this purpose we rely on the geo-location database mentioned in Section 2. This geo-location database is updated every two

weeks. In this work, we focus on improving latency to users and not bandwidth to users. Incorporating bandwidth would require both specifying a desired latency bandwidth tradeoff and a model for bandwidth between arbitrary points in the Internet.

Constraints on data placement can come in many forms. They may reflect legal constraints that data be hosted only in a certain jurisdiction, or they may reflect operational considerations requiring two replicas to be physically located in distant datacenters. Volley models such replicas as two distinct data items that may have a large amount of inter-item communication, along with the constraint that they be located in different datacenters. Although the commercial cloud service operators we spoke with emphasized the need to accommodate such constraints, the commercial applications we study in this paper do not currently face constraints of this form, and so although Volley can incorporate them, we did not explore this in our evaluation.

### 3.3 Volley Algorithm

Once the data is in Cosmos, Volley periodically analyzes it for migration opportunities. To perform the analysis, Volley relies on the SCOPE [5] distributed execution infrastructure, which at a high level resembles MapReduce [11] with a SQL-like query language. In our current implementation, Volley takes approximately 14 hours to run through one month’s worth of log files; we analyze the demands Volley places on SCOPE in more detail in Section 4.4.

Volley’s SCOPE jobs are structured into three phases. The search for a solution happens in Phase 2. Prior work [36] has demonstrated that starting this search in a good location improves convergence time, and hence

### Recursive Step:

$$wsm(\{w_i, \vec{x}_i\}_{i=1}^N) = \\ interp\left(\frac{w_N}{\sum w_i}, \vec{x}_N, wsm(\{w_i, \vec{x}_i\}_{i=1}^{N-1})\right)$$

### Base Case:

$$\begin{aligned} interp(w, \vec{x}_A, \vec{x}_B) &= (\phi_C, \lambda_C) = \vec{x}_C \\ d &= \cos^{-1} [\cos(\phi_A) \cos(\phi_B) + \\ &\quad \sin(\phi_A) \sin(\phi_B) \cos(\lambda_B - \lambda_A)] \\ \gamma &= \tan^{-1} \left[ \frac{\sin(\phi_B) \sin(\phi_A) \sin(\lambda_B - \lambda_A)}{\cos(\phi_A) - \cos(d) \cos(\phi_B)} \right] \\ \beta &= \tan^{-1} \left[ \frac{\sin(\phi_B) \sin(wd) \sin(\gamma)}{\cos(wd) - \cos(\phi_A) \cos(\phi_B)} \right] \\ \phi_C &= \cos^{-1} [\cos(wd) \cos(\phi_B) + \\ &\quad \sin(wd) \sin(\phi_B) \cos(\gamma)] \\ \lambda_C &= \lambda_B - \beta \end{aligned}$$

**Figure 9.** Weighted spherical mean calculation. The weighted spherical mean ( $wsm()$ ) is defined recursively as a weighted interpolation ( $interp()$ ) between pairs of points. Here,  $w_i$  is the weight assigned to  $\vec{x}_i$ , and  $\vec{x}_i$  (the coordinates for node  $i$ ) consists of  $\phi_i$ , the latitudinal distance in radians between node  $i$  and the North Pole, and  $\lambda_i$ , the longitude in radians of node  $i$ . The new (interpolated) node  $C$  consists of  $w$  parts node  $A$  and  $1 - w$  parts node  $B$ ;  $d$  is the current distance in radians between  $A$  and  $B$ ;  $\gamma$  is the angle from the North Pole to  $B$  to  $A$  (which stays the same as  $A$  moves);  $\beta$  is the angle from  $B$  to the North Pole to  $A$ 's new location. These are used to compute  $\vec{x}_C$ , the result of the  $interp()$ . For simplicity of presentation, we omit describing the special case for antipodal nodes.

Phase 1 computes a reasonable initial placement of data items based on client IP addresses. Phase 2 iteratively improves the placement of data items by moving them freely over the surface of the earth—this phase requires the bulk of the computational time and the algorithm code. Phase 3 does the needed fix up to map the data items to datacenters and to satisfy datacenter capacity constraints. The output of the jobs is a set of potential migration actions with the format described in Table 2. Many adaptive systems must incorporate explicit elements to prevent oscillations. Volley does not incorporate an explicit mechanism for oscillation damping. Oscillations would occur only if *user behavior* changed in response to Volley migration in such a way that Volley needed to move that user's state back to a previous location.

**Phase 1: Compute Initial Placement.** We first map each client to a set of geographic coordinates using the commercial geo-location database mentioned earlier. This IP-to-location mapping may be updated between Volley jobs, but it is not updated within a single Volley job. We then map each data item that is directly accessed by a client to the weighted average of the geographic coordinates for the client IPs that access it. This is done using the weighted spherical mean calculation shown in Figure 9. The weights are given by the amount of communication between the client nodes and the data item whose initial location we are calculating. The weighted spherical mean calculation can be thought of as drawing an arc on the earth between two points, and then finding the point on the arc that interpolates between the two initial points in proportion to their weight. This operation is then repeated to average in additional points. The recursive definition of weighted spherical mean in Figure 9 is conceptually similar to defining the more familiar weighted mean recursively, e.g.,

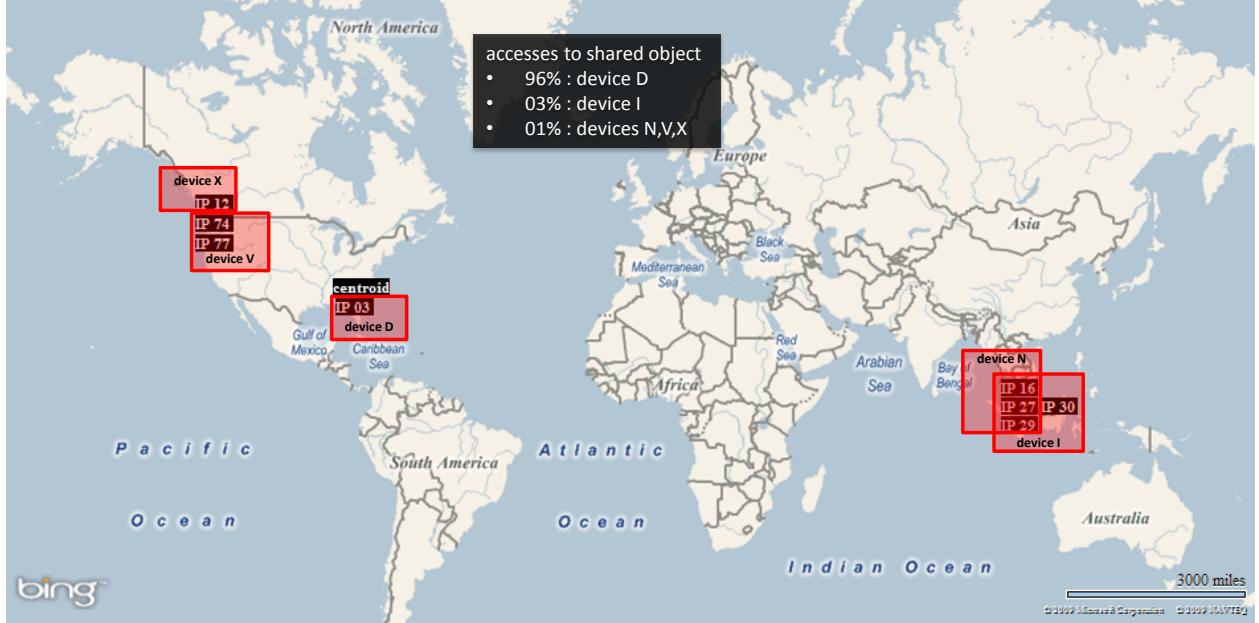
$$\begin{aligned} \text{weighted-mean}(\{3, x_k\}, \{2, x_j\}, \{1, x_i\}) &= \\ \left( \frac{3}{6} \cdot x_k + \frac{3}{6} \cdot \text{weighted-mean}(\{2, x_j\}, \{1, x_i\}) \right) \end{aligned}$$

Compared to weighted mean, weighted spherical mean has the subtlety that the rule for averaging two individual points has to use spherical coordinates.

Figure 10 shows an example of this calculation using data from the Live Mesh trace: five different devices access a single shared object from a total of eight different IP addresses; device D accesses the shared object far more than the other devices, and this leads to the weighted spherical mean (labeled “centroid” in the figure) being placed very close to device D.

Finally, for each data item that is never accessed directly by clients (e.g., the Publish-subscribe data item in the Live Mesh example of Figure 1), we map it to the weighted spherical mean of the data items that communicate with it using the positions these other items were already assigned.

**Phase 2: Iteratively Move Data to Reduce Latency.** Volley iteratively moves data items closer to both clients and to the other data items that they communicate with. This iterative update step incorporates two earlier ideas: a weighted spring model as in Vivaldi [9] and spherical coordinates as in Htrae [36]. Spherical coordinates define the locations of clients and data items in a way that is more conducive to incorporating a latency model for geographic locations. The latency distance between two nodes and the amount of communication between them increase the spring force that is pulling them together. However, unlike a network coordinate system, nodes in



**Figure 10.** An example of a shared object being placed at its weighted spherical mean (labeled “centroid” in the Figure). This particular object, the locations of the clients that access it, and their access ratios are drawn from the Live Mesh trace. Because device D is responsible for almost all of the accesses, the weighted spherical mean placement for the object is very close to device D’s location.

$$w = \frac{1}{1 + \kappa \cdot d \cdot l_{AB}}$$

$$\bar{x}_A^{new} = \text{interp}(w, \bar{x}_A^{current}, \bar{x}_B^{current})$$

**Figure 11.** Update rule applied to iteratively move nodes with more communication closer together. Here,  $w$  is a fractional weight that determines how much node  $A$  is moved towards node  $B$ ,  $l_{AB}$  is the amount of communication between the two nodes,  $d$  is the distance between nodes  $A$  and  $B$ ,  $\bar{x}_A^{current}$  and  $\bar{x}_B^{current}$  are the current locations of node  $A$  and  $B$ ,  $\bar{x}_A^{new}$  is the location of  $A$  after the update, and  $\kappa$  is an algorithmic constant.

Volley only experience contracting forces; the only factor preventing them from collapsing to a single location is the fixed nature of client locations. This yields the update rule shown in Figure 11. In our current implementation, we simply run a fixed number of iterations of this update rule; we show in Section 4 that this suffices for good convergence.

Intuitively, Volley’s spring model attempts to bring data items closer to users and to other data items that they communicate with regularly. Thus it is plausible that Volley’s spring model will simultaneously reduce latency and reduce inter-datacenter traffic; we show in Section 4 that this is indeed the case for the commercial cloud services that we study.

**Phase 3: Iteratively Collapse Data to Datacenters.** After computing a nearly ideal placement of the data

items on the surface of the earth, we have to modify this placement so that the data items are located in datacenters, and the set of items in each datacenter satisfies its capacity constraints. Like Phase 2, this is done iteratively: initially, every data item is mapped to its closest datacenter. For datacenters that are over their capacity, Volley identifies the items that experience the fewest accesses, and moves all of them to the next closest datacenter. Because this may still exceed the total capacity of some datacenter due to new additions, Volley repeats the process until no datacenter is over capacity. Assuming that the system has enough capacity to successfully host all items, this algorithm always terminates in at most as many iterations as there are datacenters in the system.

For each data item that has moved, Volley outputs a migration proposal containing the new datacenter location, the new values for latency and ongoing inter-datacenter bandwidth, and the one-time bandwidth required for this migration. This is a straightforward calculation using the old data locations, the new data locations, and the inputs supplied by the datacenter service administrator, such as the cost model and the latency model. These migration proposals are then consumed by application-specific migration mechanisms.

### 3.4 Application-specific Migration

Volley is designed to be usable by many different cloud services. For Volley to compute a recommended placement, the only requirement it imposes on the cloud

service is that it logs the request data described in Table 1. Given these request logs as input, Volley outputs a set of migration proposals described in Table 2, and then leaves the actual migration of the data to the cloud service itself. If the cloud service also provides the initial location of data items, then each migration proposal will include the bandwidth required to migrate, and the expected change in latency and inter-datacenter bandwidth after migration.

Volley’s decision to leave migration to application-specific migration mechanisms allows Volley to be more easily applied to a diverse set of datacenter applications. For example, some datacenter applications use migration mechanisms that follow the pattern of marking data read-only in the storage system at one location, copying the data to a new location, updating an application-specific name service to point to the new copy, marking the new copy as writeable, and then deleting the old copy. Other datacenter applications maintain multiple replicas in different datacenters, and migration may simple require designating a different replica as the primary. Independent of the migration mechanism, datacenter applications might desire to employ application-specific throttling policies, such as only migrating user state when an application-specific predictive model suggests the user is unlikely to access their state in the next hour. Because Volley does not attempt to migrate the data itself, it does not interfere with these techniques or any other migration technique that an application may wish to employ.

## 4 Evaluation

In our evaluation, we compare Volley to three heuristics for where to place data and show that Volley substantially outperforms all of them on the metrics of datacenter capacity skew, inter-datacenter traffic, and user-perceived latency. We focus exclusively on the month-long Live Mesh trace for conciseness. For both the heuristics and Volley, we first compute a data placement using a week of data from the Live Mesh trace, and then evaluate the quality of the resulting placement on the following three weeks of data. For all four placement methodologies, any data that appears in the three-week evaluation window but not in the one-week placement computation window is placed in a single datacenter located in the United States (in production, this new data will be handled the next time the placement methodology is run). Placing all previously unseen data in one datacenter penalizes the different methodologies equally for such data.

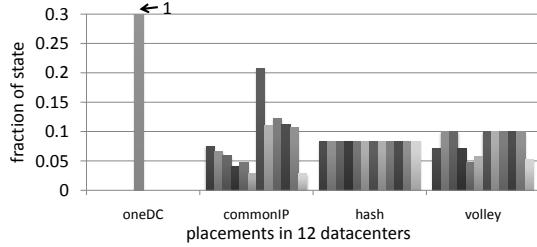
The first heuristic we consider is *commonIP* – place data as close as possible to the IP address that most commonly accesses it. The second heuristic is *oneDC* – put all data in one datacenter, a strategy still taken by many companies due to its simplicity. The third heuristic is

*hash* – hash data to datacenters so as to optimize for load-balancing. These three heuristics represent reasonable approaches to optimizing for the three different metrics we consider—*oneDC* and *hash* optimize for inter-datacenter traffic and datacenter capacity skew respectively, while *commonIP* is a reasonably sophisticated proposal for optimizing latency.

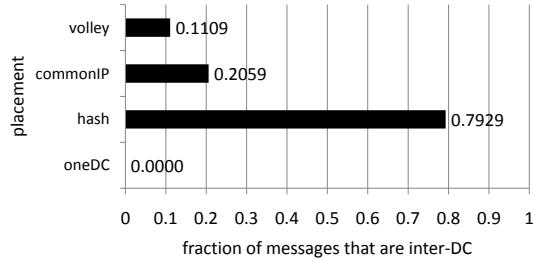
Throughout our evaluation, we use 12 commercial datacenters as potential locations. These datacenters are distributed across multiple continents, but their exact locations are confidential. Confidentiality concerns also prevent us from revealing the exact amount of bandwidth consumed by our services. Thus, we present the inter-datacenter traffic from different placements using the metric “fraction of messages that are inter-datacenter.” This allows an apples-to-apples comparison between the different heuristics and Volley without revealing the underlying bandwidth consumption. The bandwidth consumption from centralizing Volley logs, needed for Volley and *commonIP*, is so small compared to this inter-datacenter traffic that it does not affect graphs comparing this metric among the heuristics. We configure Volley with a datacenter capacity model such that no one of the 12 datacenters can host more than 10% of all data, a reasonably balanced use of capacity.

All latencies that we compute analytically use the latency model described in Section 3. This requires using the client’s IP address in the trace to place them at a geographic location. In this Live Mesh application, client requests require sending a message to a first data item, which then sends a second message to a second data item; the second data item sends a reply, and then the first data item sends the client its reply. If the data items are in the same datacenter, latency is simply the round trip time between the client and the datacenter. If the data items are in separate datacenters, latency is the sum of four one-way delays: client to datacenter 1, datacenter 1 to datacenter 2, datacenter 2 back to datacenter 1, and datacenter 1 back to the client. These latency calculations leave out other potential protocol overheads, such as the need to initially establish a TCP connection or to authenticate; any such protocol overheads encountered in practice would magnify the importance of latency improvements by incurring the latency multiple times. For clarity of presentation, we consistently group latencies into 10 millisecond bins in our graphs. The graphs only present latency up to 250 milliseconds because the better placement methodologies all achieve latency well under this for almost all requests.

Our evaluation begins by comparing Volley and the three heuristics on the metrics of datacenter capacity skew and inter-datacenter traffic (Section 4.1). Next, we evaluate the impact of these placements on the latency of client requests, including evaluating Volley in the con-



**Figure 12.** Datacenter capacity required by three different placement heuristics and Volley.

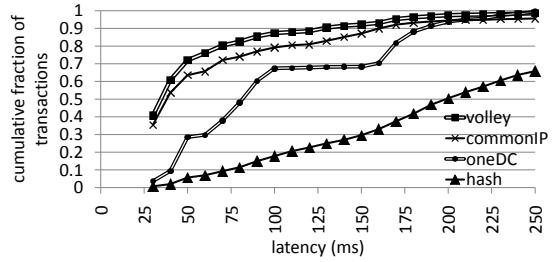


**Figure 13.** Inter-datacenter traffic under three different placement heuristics and Volley.

text of a simple, hypothetical example to understand this impact in detail (Section 4.2). We then evaluate the incremental benefit of Volley as a function of the number of Volley iterations (Section 4.3). Next, we evaluate the resource demands of running Volley on the SCOPE distributed execution infrastructure (Section 4.4). Finally, we evaluate the impact of running Volley more frequently or less frequently (Section 4.5).

#### 4.1 Impact on Datacenter Capacity Skew and Inter-datacenter Traffic

We now compare Volley to the three heuristics for where to place data and show that Volley substantially outperforms all of them on the metrics of datacenter capacity skew and inter-datacenter traffic. Figures 12 and 13 show the results: hash has perfectly balanced use of capacity, but high inter-datacenter traffic; oneDC has zero inter-datacenter traffic (the ideal), but extremely unbalanced use of capacity; and commonIP has a modest amount of inter-datacenter traffic, and capacity skew where 1 datacenter has to support more than twice the load of the average datacenter. Volley is able to meet a reasonably balanced use of capacity while keeping inter-datacenter traffic at a very small fraction of the total number of messages. In particular, compared to commonIP, Volley reduces datacenter skew by over 2 $\times$  and reduces inter-datacenter traffic by over 1.8 $\times$ .



**Figure 14.** Client request latency under three different placement heuristics and Volley.

#### 4.2 Impact on Latency of Client Requests

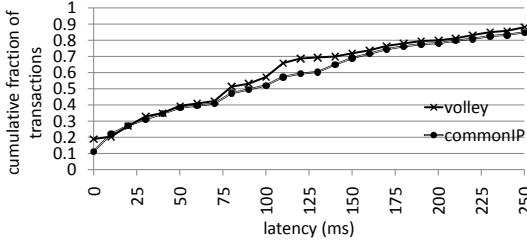
We now compare Volley to the three heuristics on the metric of user-perceived latency. Figure 14 shows the results: hash has high latency; oneDC has mediocre latency; and commonIP has the best latency among the three heuristics. Although commonIP performs better than oneDC and hash, Volley performs better still, particularly on the tail of users that experience high latency even under the commonIP placement strategy. Compared to commonIP, Volley reduces 75th percentile latency by over 30%.

##### 4.2.1 Multiple Datacenter Testbed

Previously, we evaluated the impact of placement on user-perceived latency analytically use the latency model described in Section 3. In this section, we evaluate Volley’s latency impact on a live system using a prototype cloud service. We use the prototype cloud service to emulate Live Mesh for the purpose of replaying a subset of the Live Mesh trace. We deployed the prototype cloud service across 20 virtual machines spread across the 12 geographically distributed datacenters, and we used one node at each of 109 Planetlab sites to act as clients of the system.

The prototype cloud service consists of four components: the frontend, the document service, the publish-subscribe service, and the message queue service. Each of these components run on every VM so as to have every service running in every datacenter. These components of our prototype map directly to the actual Live Mesh component services that run in production. The ways in which the production component services cooperate to provide features in the Live Mesh service is described in detail elsewhere [1], and we provide only a brief overview here.

The prototype cloud service exposes a simple frontend that accepts client requests and routes them to the appropriate component in either its own or another datacenter. In this way, each client can connect directly to any datacenter, and requests that require an additional step (e.g., updating an item, and then sending the update to others) will be forwarded appropriately. This design



**Figure 15.** Comparing Volley to the commonIP heuristic on a live system spanning 12 geographically distributed datacenters and accessed by Planetlab clients. In this Figure, we use a random sample of the Live Mesh trace. We see that Volley provides moderately better latency than the commonIP heuristic.

allows clients to cache the location of the best datacenter to connect to for any given operation, but requests still succeed if a client request arrives at the wrong datacenter due to cache staleness.

We walk through an example of how two clients can rendezvous by using the document, publish-subscribe, and message queue services. The document service can store arbitrary data; in this case, the first client can store its current IP address, and a second client can then read that IP address from the document service and contact the first client directly. The publish-subscribe service is used to send out messages when data in the document service changes; for example, if the second client subscribes to updates for the first client’s IP address, these updates will be proactively sent to the second client, instead of the second client having to poll the document service to see if there have been any changes. Finally, the message queue service buffers messages for clients from the publish-subscribe service. If the client goes offline and then reconnects, it can connect to the queue service and dequeue these messages.

To evaluate both Volley and the commonIP heuristic’s latency on this live system, we used the same data placements computed on the first week of the Live Mesh trace. Because the actual Live Mesh service requires more than 20 VMs, we had to randomly sample requests from the trace before replaying it. We also mapped each client IP in the trace subset to the closest Planetlab node, and replayed the client requests from these nodes.

Figure 15 shows the measured latency on the sample of the Live Mesh trace; recall that we are grouping latencies into 10 millisecond bins for clarity of presentation. We see that Volley consistently provides better latency than the commonIP placement. These latency benefits are visible despite a relatively large number of external sources of noise, such as the difference between the actual client locations and the Planetlab locations, differences between typical client connectivity (that Volley’s

Timestamp	Source-Entity	Request-Size	Destination-Entity	Transaction-Id
$T_0$	$PSS^a$	100 B	$Q^1$	1
$T_0$	$Q^1$	100 B	$IP^1$	1
$T_0 + 1$	$PSS^b$	100 B	$Q^1$	2
$T_0 + 1$	$Q^1$	100 B	$IP^2$	2
$T_0 + 2$	$PSS^b$	100 B	$Q^1$	3
$T_0 + 2$	$Q^1$	100 B	$IP^2$	3
$T_0 + 5$	$PSS^b$	100 B	$Q^2$	4
$T_0 + 5$	$Q^2$	100 B	$IP^1$	4

**Table 3.** Hypothetical application logs. In this example,  $IP^1$  is located at geographic coordinates (10,110) and  $IP^2$  at (10,10).

Data	commonIP	Volley Phase 1	Volley Phase 2
$PSS^a$	(10,110)	(14.7,43.1)	(15.1,49.2)
$PSS^b$	(10,10)	(15.3,65.6)	(15.3,63.6)
$Q^1$	(10,10)	(14.7,43.1)	(15.1,50.5)
$Q^2$	(10,110)	(10,110)	(13.6,88.4)

**Table 4.** CommonIP and Volley placements computed using Table 3, assuming a datacenter at every point on Earth and ignoring capacity constraints and inter-datacenter traffic.

Transaction-Id	commonIP		Volley Phase 2	
	distance	latency	distance	latency
1	27,070 miles	396 ms	8,202 miles	142 ms
2	0 miles	31 ms	7,246 miles	129 ms
3	0 miles	31 ms	7,246 miles	129 ms
4	13,535 miles	182 ms	6,289 miles	116 ms

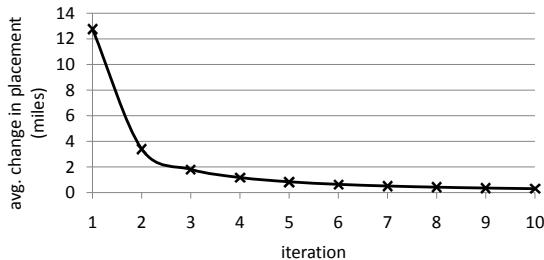
**Table 5.** Distances traversed and latencies of user requests in Table 3 using commonIP and Volley Phase 2 placements in Table 4. Note that our latency model [36] includes an empirically-determined access penalty for all communication involving a client.

latency model relies on) and Planetlab connectivity, and occasional high load on the Planetlab nodes leading to high slice scheduling delays.

Other than due to sampling of the request trace, the live experiment has no impact on the datacenter capacity skew and inter-datacenter traffic differences between the two placement methodologies. Thus, Volley offers an improvement over commonIP on every metric simultaneously, with the biggest benefits coming in reduced inter-datacenter traffic and reduced datacenter capacity skew.

#### 4.2.2 Detailed Examination of Latency Impact

To examine in detail how placement decisions impact latencies experienced by user requests, we now consider a simple example. Table 3 lists four hypothetical Live Mesh transactions involving four data objects and clients behind two IP addresses. For the purposes of this simple example, we assume there is a datacenter at every point on Earth with infinite capacity and no inter-datacenter traffic costs. We pick the geographic coor-



**Figure 16.** Average distance traveled by each object during successive Volley Phase 2 iterations. The average incorporates some objects traveling quite far, while many travel very little.

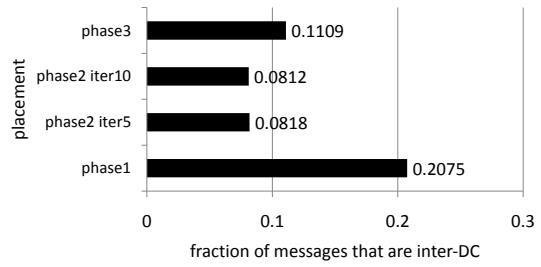
dinates of (10,110) and (10,10) for ease of examining how far each object’s placement is from the client IP addresses. Table 4 shows the placements calculated by commonIP and Volley in Phases 1 and 2. In Phase 1, Volley calculates the weighted spherical mean of the geographic coordinates for the client IPs that access each “Q” object. Hence  $Q^1$  is placed roughly two-thirds along the great-circle segment from  $IP^1$  to  $IP^2$ , while  $Q^2$  is placed at  $IP^1$ . Phase 1 similarly calculates the placement of each “PSS” object using these coordinates for “Q” objects. Phase 2 then iteratively refines these coordinates.

We now consider the latency impact of these placements on the *same* set of user requests in Table 3. Table 5 shows for each user request, the physical distance traversed and the corresponding latency (round trip from PSS to Q and from Q to IP). CommonIP optimizes for client locations that are most frequently used, thereby driving down latency to the minimum for some user requests but at a significant expense to others. Volley considers all client locations when calculating placements, and in doing so drives down the worst cases by more than the amount it drives up the common case, leading to an overall better latency distribution. Note that in practice, user requests change over time after placement decisions have been made and our trace-based evaluation does use later sets of user requests to evaluate placements based on earlier requests.

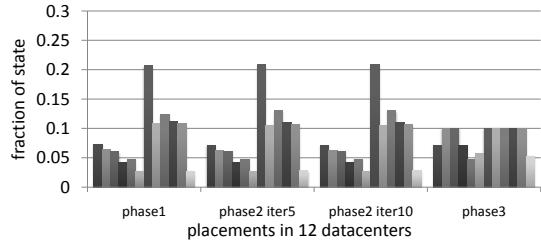
### 4.3 Impact of Volley Iteration Count

We now show that Volley converges after a small number of iterations; this will allow us to establish in Section 4.5 that Volley runs quickly (i.e., less than a day), and thus can be re-run frequently. Figures 16, 17, 18 and 19 show the performance of Volley as the number of iterations varies. Figure 16 shows that the distance that Volley moves data significantly decreases with each Volley iteration, showing that Volley relatively quickly converges to its ideal placement of data items.

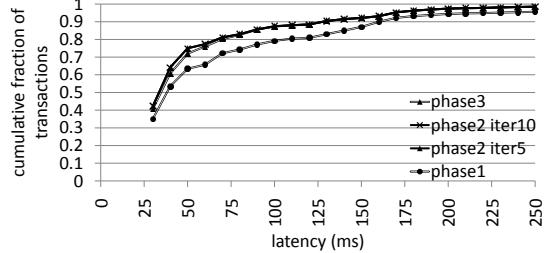
Figures 17, 18 and 19 further break down the changes



**Figure 17.** Inter-datacenter traffic at each Volley iteration.



**Figure 18.** Datacenter capacity at each Volley iteration.



**Figure 19.** Client request latency at each Volley iteration.

in Volley’s performance in each iteration. Figure 17 shows that inter-datacenter traffic is reasonably good after the initial placement of Phase 1, and is quite similar to the commonIP heuristic. In contrast, recall that the hash heuristic led to almost 80% of messages crossing datacenter boundaries. Inter-datacenter traffic then decreases by over a factor of 2 during the first 5 Phase 2 iterations, decreases by a small amount more during the next 5 Phase 2 iterations, and finally goes back up slightly when Volley’s Phase 3 balances the items across datacenters. Of course, the point of re-balancing is to avoid the kind of capacity skew seen in the commonIP heuristic, and in this regard a small increase in inter-datacenter traffic is acceptable.

Turning now to datacenter capacity, we see that Volley’s placement is quite skewed (and by an approximately constant amount) until Phase 3, where it smooths out datacenter load according to its configured capacity

Volley Phase	Elapsed Time in Hours	SCOPE Stages	SCOPE Vertices	CPU Hours
1	1:22	39	20,668	89
2	14:15	625	255,228	386
3	0:10	16	200	0:07

**Table 6.** Volley’s demands on the SCOPE infrastructure to analyze 1 week’s worth of traces.

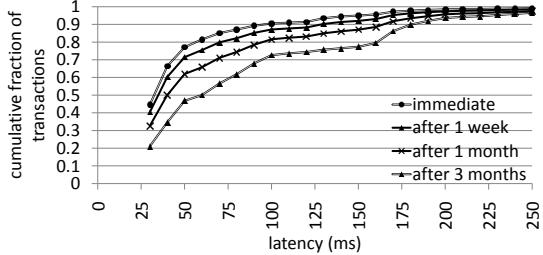
model (i.e., such that no one of the 12 datacenters hosts more than 10% of the data). Turning finally to latency, Figure 19 shows that latency has reached its minimum after only five Phase 2 iterations. In contrast to the impact on inter-datacenter traffic, there is almost no latency penalty from Phase 3’s data movement to satisfy datacenter capacity.

#### 4.4 Volley Resource Demands

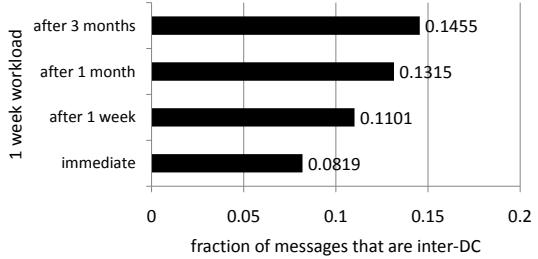
Having established that Volley converges after a small number of iterations, we now analyze the resource requirements for this many iterations; this will allow us to conclude that Volley completes quickly and can be re-run frequently. The SCOPE cluster we use consists of well over 1,000 servers. Table 6 shows Volley’s demands on the SCOPE infrastructure broken down by Volley’s different phases. The elapsed time, SCOPE stages, SCOPE vertices and CPU hours are cumulative over each phase – Phase 1 has only one iteration to compute the initial placement, while Phase 2 has ten iterations to improve the placement, and Phase 3 has 12 iterations to balance out usage over the 12 datacenters. Each SCOPE stage in Table 6 corresponds approximately to a single map or reduce step in MapReduce [11]. There are 680 such stages overall, leading to lots of data shuffling; this is one reason why the total elapsed time is not simply CPU hours divided by the degree of possible parallelism. Every SCOPE vertex in Table 6 corresponds to a node in the computation graph that can be run on a single machine, and thus dividing the total number of vertices by the total number of stages yields the average degree of parallelism within Volley: the average stage parallelizes out to just over 406 machines (some run on substantially more). The SCOPE cluster is not dedicated for Volley but rather is a multi-purpose cluster used for several tasks. The operational cost of using the cluster for 16 hours every week is small compared to the operational savings in bandwidth consumption due to improved data placement. The data analyzed by Volley is measured in the terabytes. We cannot reveal the exact amount because it could be used to infer confidential request volumes since every Volley log record is 100 bytes.

#### 4.5 Impact of Rapid Volley Re-Computation

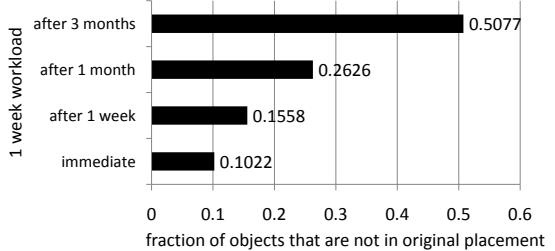
Having established that Volley can be re-run frequently, we now show that Volley provides substantially



**Figure 20.** Client request latency with stale Volley placements.

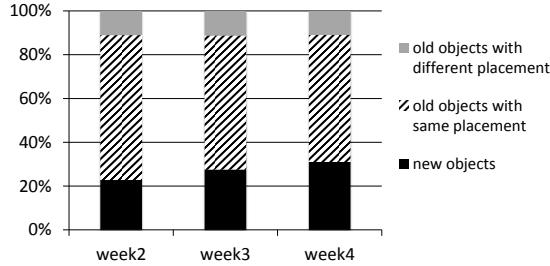


**Figure 21.** Inter-datacenter traffic with stale Volley placements.



**Figure 22.** Previously unseen objects over time.

better performance by being re-run frequently. For these experiments, we use traces from the Live Mesh service extending from the beginning of June 2009 all the way to the beginning of September 2009. Figures 20, 21 and 22 show the impact of rapidly re-computing placements: Volley computes a data placement using the trace from the first week of June, and we evaluate the performance of this placement on a trace from the immediately following week, the week after the immediately following week, a week starting a month later, and a week starting three months after Volley computed the data placement. The better performance of the placement on the immediately following week demonstrates the significant benefits of running Volley frequently with respect to both latency and inter-datacenter traffic. Figure 20 shows that running Volley even every two weeks is noticeably worse than having just run Volley, and this latency penalty keeps increasing as the Volley placement



**Figure 23.** Fraction of objects moved compared to first week.

becomes increasingly stale. Figure 21 shows a similar progressively increasing penalty to inter-datacenter traffic; running Volley frequently results in significant inter-datacenter traffic savings.

Figure 22 provides some insight into why running Volley frequently is so helpful; the number of previously unseen objects increases rapidly with time. When run frequently, Volley detects accesses to an object sooner. Note that this inability to intelligently place previously unseen objects is shared by the commonIP heuristic, and so we do not separately evaluate the rate at which it degrades in performance.

In addition to new objects that are created and accessed, previously placed objects may experience significantly different access patterns over time. Running Volley periodically provides the added benefit of migrating these objects to locations that can better serve new access patterns. Figure 23 compares a Volley placement calculated from the first week of June to a placement calculated in the second week, then the first week to the third week, and finally the first week to the fourth week. About 10% of the objects in any week undergo migrations, either as a direct result of access pattern changes or due to more important objects displacing others in capacity-limited datacenters. The majority of objects retain their placement compared to the first week. Running Volley periodically has a third, but minor advantage. Some client requests come from IP addresses that are not present in geo-location databases. Objects that are accessed solely from such locations are not placed by Volley. If additional traces include accesses from other IP addresses that are present in geo-location databases, Volley can then place these objects based on these new accesses.

## 5 Related Work

The problem of automatic placement of application data re-surfaces with every new distributed computing environment, such as local area networks (LANs), mobile computing, sensor networks, and single cluster web sites. In characterizing related work, we first focus on the mechanisms and policies that were developed for these

other distributed computing environments. We then describe prior work that focused on placing static content on CDNs; compared to this prior work, Volley is the first research system to address placement of dynamic application data across geographically distributed datacenters. We finally describe prior work on more theoretical approaches to determining an optimal data placement.

### 5.1 Placement Mechanisms

Systems such as Emerald [20], SOS [37], Globe [38], and Legion [25] focused on providing location-independent programming abstractions and migration mechanisms for moving data and computation between locations. Systems such as J-Orchestra [42] and Addistant [41] have examined distributed execution of Java applications through rewriting of byte code, but have left placement policy decisions to the user or developer. In contrast, Volley focuses on placement policy, not mechanism. Some prior work incorporated both placement mechanism and policy, e.g., Coign [18], and we characterize its differences with Volley’s placement policy in the next subsection.

### 5.2 Placement Policies for Other Distributed Computing Environments

Prior work on automatic data placement can be broadly grouped by the distributed computing environment that it targeted. Placing data in a LAN was tackled by systems such as Coign [18], IDAP [22], ICOPS [31], CAGES [17], Abacus [3] and the system of Stewart et al [39]. Systems such as Spectra [13], Slingshot [40], MagnetOS [27], Pleaides [23] and Wishbone [33] explored data placement in a wireless context, either between mobile clients and more powerful servers, or in ad hoc and sensor networks. Hilda [44] and Doloto [30] explored splitting data between web clients and web servers, but neither assumed there were multiple geographic locations that could host the web server.

Volley differs from these prior systems in several ways. First, the scale of the data that Volley must process is significantly greater. This required designing the Volley algorithm to work in a scalable data analysis framework such as SCOPE [5] or MapReduce [11]. Second, Volley must place data across a large number of datacenters with widely varying latencies both between datacenters and clients, and between the datacenters themselves; this aspect of the problem is not addressed by the algorithms in prior work. Third, Volley must continuously update its measurements of the client workload, while some (though not all) of these prior approaches used an upfront profiling approach.

### 5.3 Placement Policies for Static Data

Data placement for Content Delivery Networks (CDNs) has been explored in many pieces of prior work [21, 19]. These systems have focused on static data – the HTTP caching header should be honored, but no other more elaborate synchronization between replicas is needed. Because of this, CDNs can easily employ decentralized algorithms e.g., each individual server or a small set of servers can independently make decisions about what data to cache. In contrast, Volley’s need to deal with dynamic data would make a decentralized approach challenging; Volley instead opts to collect request data in a single datacenter and leverage the SCOPE distributed execution framework to analyze the request logs within this single datacenter.

### 5.4 Optimization Algorithms

Abstractly, Volley seeks to map objects to locations so as to minimize a cost function. Although there are no known approximation algorithms for this general problem, the theory community has developed approximation algorithms for numerous more specialized settings, such as sparsest cut [4] and various flavors of facility location [6, 8]. To the best of our knowledge, the problem in Volley does not map to any of these previously studied specializations. For example, the problem in Volley differs from facility location in that there is a cost associated with placing two objects at different datacenters, not just costs between clients and objects. This motivates Volley’s choice to use a heuristic approach and to experimentally validate the quality of the resulting data placement.

Although Volley offers a significant improvement over a state-of-the-art heuristic, we do not yet know how close it comes to an optimal placement; determining such an optimal placement is challenging because standard commercial optimization packages simply do not scale to the data sizes of large cloud services. This leaves open the tantalizing possibility that further improvements are possible beyond Volley.

## 6 Conclusion

Cloud services continue to grow to span large numbers of datacenters, making it increasingly urgent to develop automated techniques to place application data across these datacenters. Based on the analysis of month-long traces from two large-scale commercial cloud services, Microsoft’s Live Messenger and Live Mesh, we built the Volley system to perform automatic data placement across geographically distributed datacenters. To scale to the large data volumes of cloud service logs, Volley is designed to work in the SCOPE [5] scalable data analysis framework.

We evaluate Volley analytically and on a live system consisting of a prototype cloud service running on a geographically distributed testbed of 12 datacenters. Our evaluation using one of the month-long traces shows that, compared to a state-of-the-art heuristic, Volley simultaneously reduces datacenter capacity skew by over  $2\times$ , reduces inter-datacenter traffic by over  $1.8\times$ , and reduces 75th percentile latency by over 30%. This shows the potential of Volley to simultaneously improve the user experience and significantly reduce datacenter costs.

While in this paper we have focused on using Volley to optimize data placement in existing datacenters, service operators could also use Volley to explore future sites for datacenters that would improve performance. By including candidate locations for datacenters in Volley’s input, the operator can identify which combination of additional sites improve latency at modest costs in greater inter-datacenter traffic. We hope to explore this more in future work.

### Acknowledgments

We greatly appreciate the support of Microsoft’s ECN team for donating usage of their VMs, and the support of the Live Mesh and Live Messenger teams in sharing their data with us. We thank our shepherd, Dejan Kostic, and the anonymous reviewers for their detailed feedback and help in improving this paper.

### References

- [1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrating Lease Management and Partitioning for Cloud Services. In *NSDI*, 2010.
- [2] Amazon Web Services. <http://aws.amazon.com>.
- [3] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *USENIX Annual Technical Conference*, 2000.
- [4] S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. In *STOC*, 2004.
- [5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.
- [6] M. Charikar and S. Guha. Improved Combinatorial Algorithms for the Facility Location and k-Median Problems. In *FOCS*, 1999.
- [7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *DSN*, 2002.
- [8] F. Chudak and D. Williamson. Improved approximation algorithms for capacitated facility location problems. *Mathematical Programming*, 102(2):207–222, 2005.
- [9] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, 2004.
- [10] Data Center Global Expansion Trend.

- http://www.datacenterknowledge.com/archives/2008/03/27/google-data-center-faq/.
- [11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *OSDI*, 2004.
  - [12] Facebook. http://www.facebook.com.
  - [13] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *ICDCS*, 2002.
  - [14] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.
  - [15] Global Air travel Trends. http://www.zinnov.com/presentation/Global\_Aviation-Markets-An\_Analysis.pdf.
  - [16] Google Apps. http://apps.google.com.
  - [17] G. Hamlin Jr and J. Foley. Configurable applications for graphics employing satellites (CAGES). *ACM SIGGRAPH Computer Graphics*, 9(1):9–19, 1975.
  - [18] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *OSDI*, 1999.
  - [19] S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constrained Mirror Placement on the Internet. In *INFOCOM*, 2001.
  - [20] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1996.
  - [21] M. Karlsson and M. Mahalingam. Do We Need Replica Placement Algorithms in Content Delivery Networks? In *International Workshop on Web Content Caching and Distribution (WCW)*, 2002.
  - [22] D. Kimelman, V. Rajan, T. Roth, M. Wegman, B. Lindsey, H. Lindsey, and S. Thomas. Dynamic Application Partitioning in VisualAge Generator Version 3.0. *Lecture Notes In Computer Science; Vol. 1543*, pages 547–548, 1998.
  - [23] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *PLDI*, 2007.
  - [24] J. Leskovec and E. Horvitz. Planetary-Scale Views on an Instant-Messaging Network. In *WWW*, 2008.
  - [25] M. Lewis and A. Grimshaw. The core Legion object model. In *HPDC*, 1996.
  - [26] LinkedIn. http://linkedin.com.
  - [27] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Sirer. Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. In *MobiSys*, 2005.
  - [28] Live Mesh. http://www.mesh.com.
  - [29] Live Messenger. http://messenger.live.com.
  - [30] B. Livshits and E. Kiciman. Doloto: Code Splitting for Network-bound Web 2.0 Applications. In *SIGSOFT FSE*, 2008.
  - [31] J. Michel and A. van Dam. Experience with distributed processing on a host/satellite graphics system. *ACM SIGGRAPH Computer Graphics*, 10(2):190–195, 1976.
  - [32] Microsoft Office Online. http://office.microsoft.com/en-us/office\_live/default.aspx.
  - [33] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *NSDI*, 2009.
  - [34] Quova IP Geo-Location Database. http://www.quova.com.
  - [35] Rapid Release Cycles. http://www.ereleases.com/pr/20070716009.html.
  - [36] S. Agarwal and J. Lorch. Matchmaking for Online Games and Other Latency-Sensitive P2P Systems. In *SIGCOMM*, 2009.
  - [37] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, and M. Ruffin. SOS: An object-oriented operating system—assessment and perspectives. *Computing Systems*, 1989.
  - [38] M. Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, 1999.
  - [39] C. Stewart, K. Shen, S. Dwarkadas, M. Scott, and J. Yin. Profile-driven component placement for cluster-based online services. *IEEE Distributed Systems Online*, 5(10):1–1, 2004.
  - [40] Y. Su and J. Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *MobiSys*, 2005.
  - [41] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A byte-code translator for distributed execution of “legacy” java software. In *ECOOP*, pages 236–255. Springer-Verlag, 2001.
  - [42] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP*, pages 178–204. Springer-Verlag, 2002.
  - [43] Twitter. http://www.twitter.com.
  - [44] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A High-Level Language for Data-DrivenWeb Applications. *ICDE*, 2006.
  - [45] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: Large Scale Spamming Botnet Detection. In *NSDI*, 2009.

# Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,  
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall  
and Werner Vogels

Amazon.com

## ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

## Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;

## General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

## 1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP '07, October 14–17, 2007, Stevenson, Washington, USA.  
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications require a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

## 2. BACKGROUND

Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

## 2.1 System Assumptions and Requirements

The storage system for this class of services has the following requirements:

*Query Model:* simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

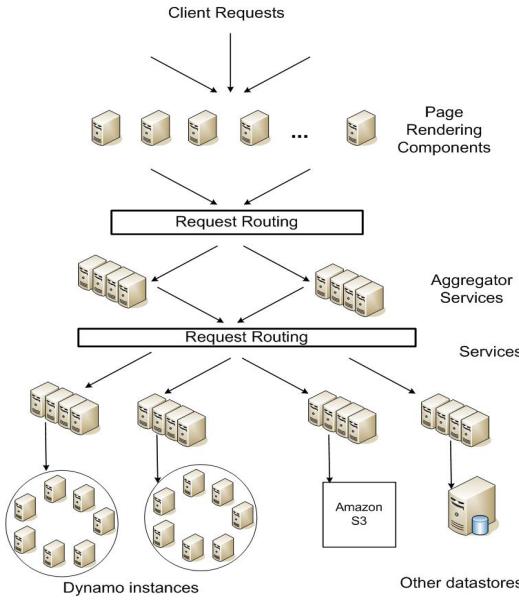
*ACID Properties:* ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

*Efficiency:* The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9<sup>th</sup> percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

*Other Assumptions:* Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

## 2.2 Service Level Agreements (SLA)

To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will



**Figure 1: Service-oriented architecture of Amazon’s platform**

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon’s decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon’s platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9<sup>th</sup> percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon’s

production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9<sup>th</sup> percentile of distributions, which reflects Amazon engineers’ relentless focus on performance from the perspective of the customers’ experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon’s engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9<sup>th</sup> percentile.

Storage systems often play an important role in establishing a service’s SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service’s SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

### 2.3 Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an “always writeable” data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as “last write wins” [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client’s experience. For instance, the application that maintains customer shopping carts can choose to “merge” the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as “last write wins”.

Other key principles embraced in the design are:

*Incremental scalability*: Dynamo should be able to scale out one storage host (henceforth, referred to as “*node*”) at a time, with minimal impact on both operators of the system and the system itself.

*Symmetry*: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

*Decentralization*: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

*Heterogeneity*: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

## 3. RELATED WORK

### 3.1 Peer to Peer Systems

There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella<sup>1</sup>, were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

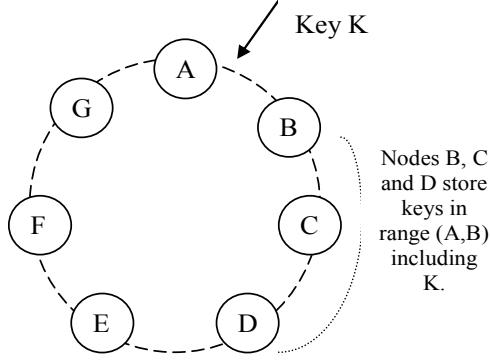
### 3.2 Distributed File Systems and Databases

Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google’s internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.

---

<sup>1</sup> <http://freenetproject.org/>, <http://www.gnutella.org>



**Figure 2: Partitioning and replication of keys in Dynamo ring.**

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

### 3.3 Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an “always writeable” data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

## 4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

**Table 1: Summary of techniques used in *Dynamo* and their advantages.**

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

### 4.1 System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: `get()` and `put()`. The `get(key)` operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The `put(key, context, object)` operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the `put` request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

### 4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo’s partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or “ring” (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its “position” on the ring. Each data item identified by a key is assigned to a node by hashing the data item’s key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item’s position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of “virtual nodes”. A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, “tokens”) in the ring. The process of fine-tuning Dynamo’s partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

### 4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured “per-instance”. Each key,  $k$ , is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its  $N^{\text{th}}$  predecessor. In Figure 2, node B replicates the key  $k$  at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

### 4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may

return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

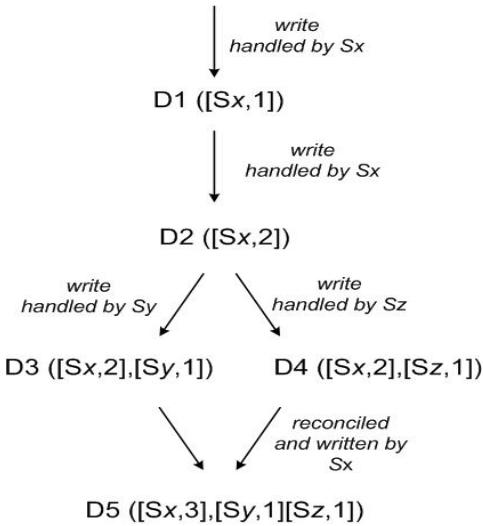
There is a category of applications in Amazon’s platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an “Add to Cart” operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn’t supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both “add to cart” and “delete item from cart” operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is “merging” different versions of a customer’s shopping cart. Using this reconciliation mechanism, an “add to cart” operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examining their vector clocks. If the counters on the first object’s clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if



**Figure 3: Version evolution of an object over time.**

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock  $[(Sx, 1)]$ . The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock  $[(Sx, 2)]$ . D2 descends from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock  $[(Sx, 2), (Sy, 1)]$ .

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is  $[(Sx, 2), (Sz, 1)]$ . A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely  $[(Sx, 2), (Sy, 1), (Sz, 1)]$ . If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock:  $[(Sx, 3), (Sy, 1), (Sz, 1)]$ .

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an

object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

## 4.5 Execution of get() and put() operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that  $R + W > N$  yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with

the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

## 4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a “sloppy quorum”; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

## 4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to

the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are “out of sync”. Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

## 4.8 Membership and Failure Detection

### 4.8.1 Ring Membership

In Amazon’s environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and

initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

#### 4.8.2 External Discovery

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

#### 4.8.3 Failure Detection

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during `get()` and `put()` operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

### 4.9 Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let

us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges  $(F, G]$ ,  $(G, A]$  and  $(A, X]$ . As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

## 5. IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

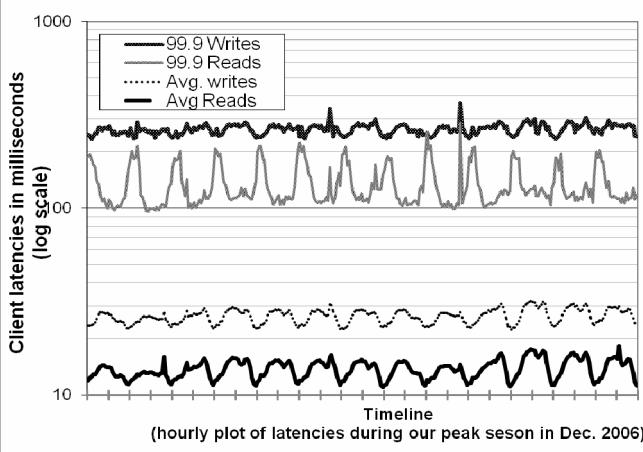
Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store<sup>2</sup>, BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

---

<sup>2</sup><http://www.oracle.com/database/berkeley-db.html>



**Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006.** The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

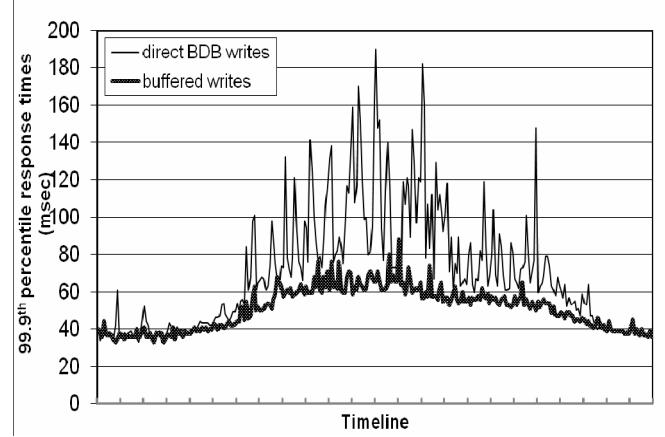
machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting “read-your-writes” consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

## 6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer’s shopping cart.

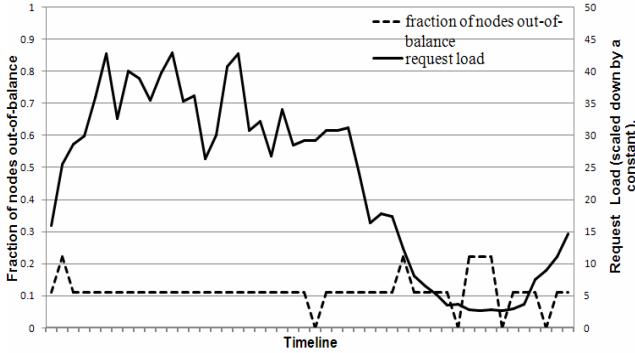


**Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours.** The intervals between consecutive ticks in the x-axis correspond to one hour.

- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of “last write wins”; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer’s session information is a good example of a service that uses this mode.
- *High performance read engine:* While Dynamo is built to be an “always writeable” data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo’s users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.



**Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.**

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing  $W$ . This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common  $(N, R, W)$  configuration used by several instances of Dynamo is  $(3, 2, 2)$ . These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of  $(3, 2, 2)$  and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response  $R$  (or  $W$ ) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

## 6.1 Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9<sup>th</sup> or 99.99<sup>th</sup> percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the  $R$  or  $W$  replicas. Figure 4 shows the average and 99.9<sup>th</sup> percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a

significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9<sup>th</sup> percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9<sup>th</sup> percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

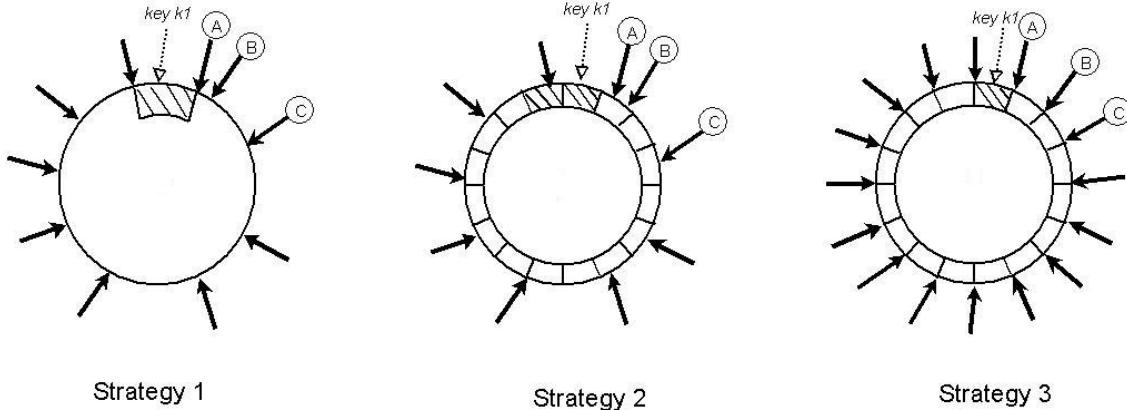
While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9<sup>th</sup> percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the  $N$  replicas to perform a "durable write". Since the coordinator waits only for  $W$  responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

## 6.2 Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8<sup>th</sup>



**Figure 7: Partitioning and placement of keys in the three strategies.** A, B, and C depict the three unique nodes that form the preference list for the key  $k_1$  on the consistent hashing ring ( $N=3$ ). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

**Strategy 1: T random tokens per node and partition by token value:** This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned  $T$  tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

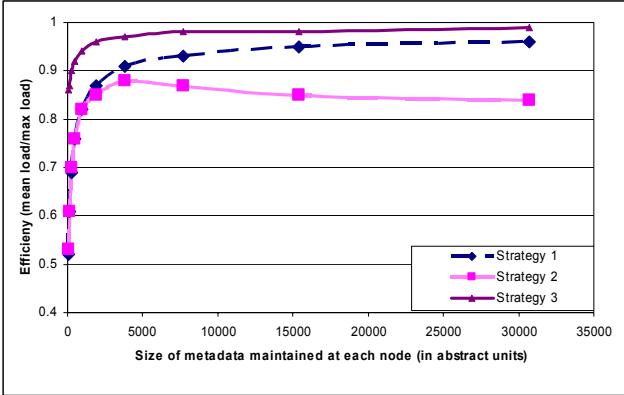
While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

**Strategy 2: T random tokens per node and equal sized partitions:** In this strategy, the hash space is divided into  $Q$  equally sized partitions/ranges and each node is assigned  $T$  random tokens.  $Q$  is usually set such that  $Q \gg N$  and  $Q \gg S*T$ , where  $S$  is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first  $N$  unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for  $N=3$ . In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key  $k_1$ . The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

**Strategy 3: Q/S tokens per node, equal-sized partitions:** Similar to strategy 2, this strategy divides the hash space into  $Q$  equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned  $Q/S$  tokens where  $S$  is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with  $S=30$  and  $N=3$ . However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e.,  $T$ ) while strategy 3 depends on the number of partitions (i.e.,  $Q$ ). One fair way to compare these strategies is to



**Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.**

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters ( $T$  and  $Q$ ). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery*: Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

### 6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

### 6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

**Table 2: Performance of client-driven and server-driven coordination approaches.**

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9<sup>th</sup> percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9<sup>th</sup> percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9<sup>th</sup> percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9<sup>th</sup> percentile than the average.

## 6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database),

shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99<sup>th</sup> percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

## 6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

## 7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current

request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

## ACKNOWLEDGEMENTS

The authors would like to thank Pat Helland for his contribution to the initial design of Dynamo. We would also like to thank Marvin Theimer and Robert van Renesse for their comments. Finally, we would like to thank our shepherd, Jeff Mogul, for his detailed comments and inputs while preparing the camera ready version that vastly improved the quality of the paper.

## REFERENCES

- [1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1-14.
- [2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Systems*, 9(4):596-615, December 1984
- [3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.
- [4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.
- [5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.
- [6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.
- [7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.
- [8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.
- [9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.
- [10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.
- [11] Lindsay, B.G., et al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979
- [12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *ACM Communications*, 21(7), pp. 558-565, 1978.
- [13] Merkle, R. A digital signature based on a conventional encryption function. *Proceedings of CRYPTO*, pages 369-378. Springer-Verlag, 1988.
- [14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 29 - 31, 2004.
- [15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..
- [16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *Proceedings of Middleware*, pages 329-350, November, 2001.
- [17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of Symposium on Operating Systems Principles*, October 2001.
- [18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.
- [19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. *IEEE Workshop on Workstation Operating Systems*, Nov. 1987.
- [20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

- [21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.
- [22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.
- [23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.
- [24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.

# Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach

Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

*Google, Inc.*

## Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

## 1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable’s performance. Section 7 provides measurements of Bigtable’s performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

## 2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

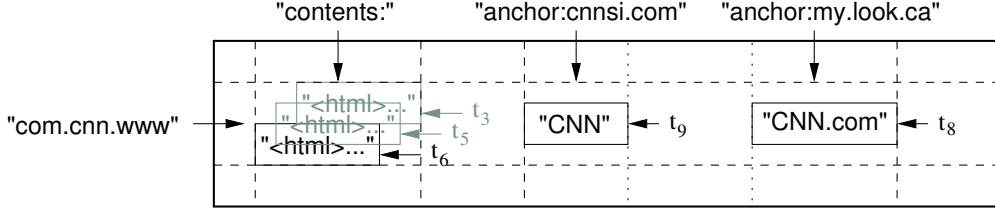


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The **contents** column family contains the page contents, and the **anchor** column family contains the text of any anchors that reference the page. CNN’s home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named **anchor:cnnsi.com** and **anchor:my.look.ca**. Each anchor cell has one version; the contents column has three versions, at timestamps  $t_3$ ,  $t_5$ , and  $t_6$ .

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions, suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the **contents:** column under the timestamps when they were fetched, as illustrated in Figure 1.

## Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system’s behavior in the presence of concurrent updates to the same row.

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for `maps.google.com/index.html` under the key `com.google.maps/index.html`. Storing pages from the same domain near each other makes some host and domain analyses more efficient.

## Column Families

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns.

A column key is named using the following syntax: *family:qualifier*. Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is `language`, which stores the language in which a web page was written. We use only one column key in the `language` family, and it stores each web page’s language ID. Another useful column family for this table is `anchor`; each column key in this family represents a single anchor, as shown in Figure 1. The qualifier is the name of the referring site; the cell contents is the link text.

Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

## Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent “real time” in microseconds, or be explicitly assigned by client

```

// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);

```

Figure 2: Writing to Bigtable.

applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last  $n$  versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

In our Webtable example, we set the timestamps of the crawled pages stored in the `contents`: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.

### 3 API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a `RowMutation` abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to `Apply` performs an atomic mutation to the Webtable: it adds one anchor to `www.cnn.com` and deletes a different anchor.

Figure 3 shows C++ code that uses a `Scanner` abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan. For example, we could restrict the scan above to only produce anchors whose columns match the regular expression `anchor:*.cnn.com`, or to only produce anchors whose timestamps fall within ten days of the current time.

```

Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
    printf("%s %s %lld %s\n",
        scanner.RowName(),
        stream->ColumnName(),
        stream->MicroTimestamp(),
        stream->Value());
}

```

Figure 3: Reading from Bigtable.

Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall [28]. At the moment, our Sawzall-based API does not allow client scripts to write back into Bigtable, but it does allow various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.

Bigtable can be used with MapReduce [12], a framework for running large-scale parallel computations developed at Google. We have written a set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

### 4 Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google *SSTable* file format is used internally to store Bigtable data. An *SSTable* provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified

key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a *session* with a Chubby service. A client’s session expires if it is unable to renew its session lease within the lease expiration time. When a client’s session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section 5.1); to discover tablet servers and finalize tablet server deaths (see Section 5.2); to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

## 5 Implementation

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be

dynamically added (or removed) from a cluster to accommodate changes in workloads.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

As with many single-master distributed storage systems [17, 21], client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

### 5.1 Tablet Location

We use a three-level hierarchy analogous to that of a B<sup>+</sup>-tree [10] to store tablet location information (Figure 4).

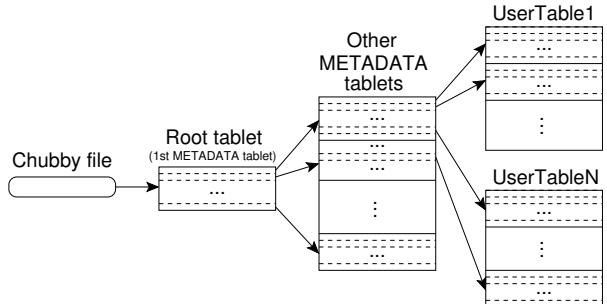


Figure 4: Tablet location hierarchy.

The first level is a file stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The *root tablet* is just the first tablet in the METADATA table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet’s table

identifier and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address  $2^{34}$  tablets (or  $2^{61}$  bytes in 128 MB tablets).

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

## 5.2 Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the *servers directory*) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an efficient mechanism that allows a tablet server to check whether it still holds its lock without incurring network traffic.) A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's file. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server file. Once a server's file has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will be assigned. Because the root tablet contains the names of all METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it notifies the master. In case the split notification is lost (either

because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry it finds in the METADATA table will specify only a portion of the tablet that the master asked it to load.

### 5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server

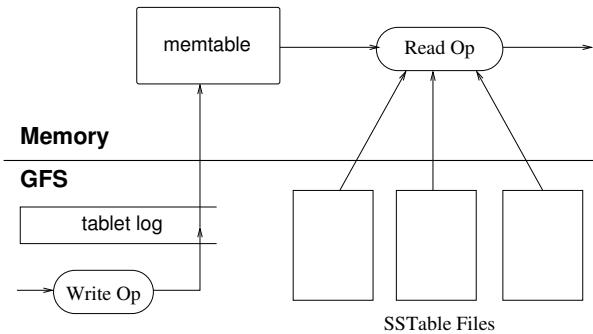


Figure 5: Tablet Representation

reads its metadata from the METADATA table. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points.

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations [13, 16]. After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently.

Incoming read and write operations can continue while tablets are split and merged.

### 5.4 Compactions

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such files by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactations can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactations to them. These major compactations allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

## 6 Refinements

The implementation described in the previous section required a number of refinements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these refinements.

### Locality groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an ap-

plication that wants to read the metadata does not need to read through all of the page contents.

In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the METADATA table.

## Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specific tuning parameter). Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [6], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines.

Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well. For example, in Webtable, we use this compression scheme to store Web page contents. In one experiment, we stored a large number of documents in a compressed locality group. For the purposes of the experiment, we limited ourselves to one version of each document instead of storing all versions available to us. The scheme achieved a 10-to-1 reduction in space. This is much better than typical Gzip reductions of 3-to-1 or 4-to-1 on HTML pages because of the way Webtable rows are laid out: all pages from a single host are stored close to each other. This allows the Bentley-McIlroy algorithm to identify large amounts of shared boilerplate in pages from the same host. Many applications, not just Webtable, choose their row names so that similar data ends up clustered, and therefore achieve very good compression ratios. Compression ratios get even better when we store multiple versions of the same value in Bigtable.

## Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTable blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

## Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

## Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file [18, 20].

Using one log provides significant performance benefits during normal operation, but it complicates recovery. When a tablet server dies, the tablets that it served will be moved to a large number of other tablet servers: each server typically loads a small number of the original server's tablets. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. However, the mutations for these tablets

were co-mingled in the same physical log file. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

We avoid duplicating log reads by first sorting the commit log entries in order of the keys *(table, row name, log sequence number)*. In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

Writing commit logs to GFS sometimes causes performance hiccups for a variety of reasons (e.g., a GFS server machine involved in the write crashes, or the network paths traversed to reach the particular set of three GFS servers is suffering network congestion, or is heavily loaded). To protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file; only one of these two threads is actively in use at a time. If writes to the active log file are performing poorly, the log file writing is switched to the other thread, and mutations that are in the commit log queue are written by the newly active log writing thread. Log entries contain sequence numbers to allow the recovery process to elide duplicated entries resulting from this log switching process.

### Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor compaction on that tablet. This compaction reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. After finishing this compaction, the tablet server stops serving the tablet. Before it actually unloads the tablet, the tablet server does another (usually very fast) minor compaction to eliminate any remaining uncompacted state in the tablet server's log that arrived while the first minor compaction was being performed. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

### Exploiting immutability

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all

of the SSTables that we generate are immutable. For example, we do not need any synchronization of accesses to the file system when reading from SSTables. As a result, concurrency control over rows can be implemented very efficiently. The only mutable data structure that is accessed by both reads and writes is the memtable. To reduce contention during reads of the memtable, we make each memtable row copy-on-write and allow reads and writes to proceed in parallel.

Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. Each tablet's SSTables are registered in the METADATA table. The master removes obsolete SSTables as a mark-and-sweep garbage collection [25] over the set of SSTables, where the METADATA table contains the set of roots.

Finally, the immutability of SSTables enables us to split tablets quickly. Instead of generating a new set of SSTables for each child tablet, we let the child tablets share the SSTables of the parent tablet.

## 7 Performance Evaluation

We set up a Bigtable cluster with  $N$  tablet servers to measure the performance and scalability of Bigtable as  $N$  is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each.  $N$  client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

$R$  is the distinct number of Bigtable row keys involved in the test.  $R$  was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

The *sequential write* benchmark used row keys with names 0 to  $R - 1$ . This space of row keys was partitioned into  $10N$  equal-sized ranges. These ranges were assigned to the  $N$  clients by a central scheduler that as-

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

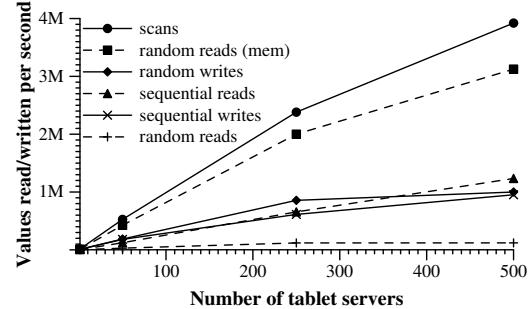
signed the next available range to a client as soon as the client finished processing the previous range assigned to it. This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible. In addition, strings under different row key were distinct, so no cross-row compression was possible. The *random write* benchmark was similar except that the row key was hashed modulo  $R$  immediately before writing so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

The *sequential read* benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark). Similarly, the *random read* benchmark shadowed the operation of the random write benchmark.

The *scan* benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range. Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

The *random reads (mem)* benchmark is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read. For just this benchmark, we reduced the amount of data per tablet server from 1 GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.



### Single tablet-server performance

Let us first consider performance with just one tablet server. Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Random reads from memory are much faster since each 1000-byte read is satisfied from the tablet server's local memory without fetching a large 64 KB block from GFS.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes; in both cases, all writes to the tablet server are recorded in the same commit log.

Sequential reads perform better than random reads since every 64 KB SSTable block that is fetched from GFS is stored into our block cache, where it is used to serve the next 64 read requests.

Scans are even faster since the tablet server can return a large number of values in response to a single client RPC, and therefore RPC overhead is amortized over a large number of values.

### Scaling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

The random read benchmark shows the worst scaling (an increase in aggregate throughput by only a factor of 100 for a 500-fold increase in number of servers). This behavior occurs because (as explained above) we transfer one large 64KB block over the network for every 1000-byte read. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

## 8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

Table 2 provides some data about a few of the tables currently in use. Some tables store data that is served to users, whereas others store data for batch processing; the tables range widely in total size, average cell size,

percentage of data served from memory, and complexity of the table schema. In the rest of this section, we briefly describe how three product teams use Bigtable.

### 8.1 Google Analytics

Google Analytics ([analytics.google.com](http://analytics.google.com)) is a service that helps webmasters analyze traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page.

To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters.

We briefly describe two of the tables used by Google Analytics. The raw click table (~200 TB) maintains a row for each end-user session. The row name is a tuple containing the website’s name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

The summary table (~20 TB) contains various predefined summaries for each website. This table is generated from the raw click table by periodically scheduled MapReduce jobs. Each MapReduce job extracts recent session data from the raw click table. The overall system’s throughput is limited by the throughput of GFS. This table compresses to 29% of its original size.

### 8.2 Google Earth

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world’s surface, both through the web-based Google Maps interface ([maps.google.com](http://maps.google.com)) and through the Google Earth ([earth.google.com](http://earth.google.com)) custom client software. These products allow users to navigate across the world’s surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to preprocess data, and a different set of tables for serving client data.

The preprocessing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into final serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are efficiently compressed already, so Bigtable compression is disabled.

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	—	9	8	3	0%	No
Orkut	9	—	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

The preprocessing pipeline relies heavily on MapReduce over Bigtable to transform data. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs.

The serving system uses one table to index data stored in GFS. This table is relatively small (~500 GB), but it must serve tens of thousands of queries per second per datacenter with low latency. As a result, this table is hosted across hundreds of tablet servers and contains in-memory column families.

### 8.3 Personalized Search

Personalized Search ([www.google.com/psearch](http://www.google.com/psearch)) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

Personalized Search stores each user’s data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

## 9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons.

One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. For example, we have seen problems due to all of the following causes: memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance. As we have gained more experience with these problems, we have addressed them by changing various protocols. For example, we added checksumming to our RPC mechanism. We also handled

some problems by removing assumptions made by one part of the system about another part. For example, we stopped assuming a given Chubby operation could return only one of a fixed set of errors.

Another lesson we learned is that it is important to delay adding new features until it is clear how the new features will be used. For example, we initially planned to support general-purpose transactions in our API. Because we did not have an immediate use for them, however, we did not implement them. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions. Where people have requested distributed transactions, the most important use is for maintaining secondary indices, and we plan to add a specialized mechanism to satisfy this need. The new mechanism will be less general than distributed transactions, but will be more efficient (especially for updates that span hundreds of rows or more) and will also interact better with our scheme for optimistic cross-data-center replication.

A practical lesson that we learned from supporting Bigtable is the importance of proper system-level monitoring (i.e., monitoring both Bigtable itself, as well as the client processes using Bigtable). For example, we extended our RPC system so that for a sample of the RPCs, it keeps a detailed trace of the important actions done on behalf of that RPC. This feature has allowed us to detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the METADATA table when METADATA tablets are unavailable. Another example of useful monitoring is that every Bigtable cluster is registered in Chubby. This allows us to track down all clusters, discover how big they are, see which versions of our software they are running, how much traffic they are receiving, and whether or not there are any problems such as unexpectedly large latencies.

The most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging. One example of this is our tablet-server membership protocol. Our first protocol was simple: the master periodically issued leases to tablet servers, and tablet servers killed themselves if their lease expired. Unfortunately, this protocol reduced availability significantly in the presence of network problems, and was also sensitive to master recovery time. We redesigned the protocol several times until we had a protocol that performed well. However, the resulting protocol was too complex and depended on

the behavior of Chubby features that were seldom exercised by other applications. We discovered that we were spending an inordinate amount of time debugging obscure corner cases, not only in Bigtable code, but also in Chubby code. Eventually, we scrapped this protocol and moved to a newer simpler protocol that depends solely on widely-used Chubby features.

## 10 Related Work

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at “Internet scale.” This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle’s Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM’s DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benefits observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partitioning into flat files and achieves good data compression ratios is AT&T’s Daytona database [19]. Locality groups do not support CPU-cache-level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a “read-optimized relational DBMS”, whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable’s load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

## 11 Conclusions

We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.

Given the unusual interface to Bigtable, an interesting question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demonstrates that our design works well in practice.

We are in the process of implementing several additional Bigtable features, such as support for secondary indices and infrastructure for building cross-data-center replicated Bigtables with multiple master replicas. We have also begun deploying Bigtable as a service to product groups, so that individual groups do not need to maintain their own clusters. As our service clusters scale, we will need to deal with more resource-sharing issues within Bigtable itself [3, 5].

Finally, we have found that there are significant advantages to building our own storage solution at Google. We have gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable’s implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise.

## Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwieginczew.

## References

- [1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in column-oriented database systems. *Proc. of SIGMOD* (2006).
- [2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169–180.
- [3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb. 1999), pp. 45–58.
- [4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND,

- G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal* 34, 2 (1995), 292–322.
- [5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st OSDI* (Mar. 2004), pp. 253–266.
  - [6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287–295.
  - [7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *CACM* 13, 7 (1970), 422–426.
  - [8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI* (Nov. 2006).
  - [9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J. Paxos made live — An engineering perspective. In *Proc. of PODC* (2007).
  - [10] COMER, D. Ubiquitous B-tree. *Computing Surveys* 11, 2 (June 1979), 121–137.
  - [11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In *Proc. of SIGMOD* (1988), pp. 99–108.
  - [12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 137–150.
  - [13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proc. of SIGMOD* (June 1984), pp. 1–8.
  - [14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. *CACM* 35, 6 (June 1992), 85–98.
  - [15] FRENCH, C. D. One size fits all database architectures do not work for DSS. In *Proc. of SIGMOD* (May 1995), pp. 449–450.
  - [16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Engineering Bulletin* 8, 2 (1985), 3–10.
  - [17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 19th ACM SOSP* (Dec. 2003), pp. 29–43.
  - [18] GRAY, J. Notes on database operating systems. In *Operating Systems — An Advanced Course*, vol. 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.
  - [19] GREER, R. Daytona and the fourth-generation language Cymbal. In *Proc. of SIGMOD* (1999), pp. 525–526.
  - [20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th SOSP* (Dec. 1987), pp. 155–162.
  - [21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In *Proc. of the 14th SOSP* (Asheville, NC, 1993), pp. 29–43.
  - [22] KX.COM. [kx.com/products/database.php](http://kx.com/products/database.php). Product page.
  - [23] LAMPORT, L. The part-time parliament. *ACM TOCS* 16, 2 (1998), 133–169.
  - [24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 105–120.
  - [25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *CACM* 3, 4 (Apr. 1960), 184–195.
  - [26] O’NEIL, P., CHENG, E., GAWLICK, D., AND O’NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), 351–385.
  - [27] ORACLE.COM. [www.oracle.com/technology/products/database/clustering/index.html](http://www.oracle.com/technology/products/database/clustering/index.html). Product page.
  - [28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal* 13, 4 (2005), 227–298.
  - [29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug. 2001), pp. 161–172.
  - [30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001* (Nov. 2001), pp. 329–350.
  - [31] SENSAge.COM. [sensage.com/products-sensage.htm](http://sensage.com/products-sensage.htm). Product page.
  - [32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM* (Aug. 2001), pp. 149–160.
  - [33] STONEBRAKER, M. The case for shared nothing. *Database Engineering Bulletin* 9, 1 (Mar. 1986), 4–9.
  - [34] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O’NEIL, E., O’NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A column-oriented DBMS. In *Proc. of VLDB* (Aug. 2005), pp. 553–564.
  - [35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In *Proc. of the Tenth ICDE* (1994), IEEE Computer Society, pp. 54–65.
  - [36] SYBASE.COM. [www.sybase.com/products/database-servers/sybaseiq](http://www.sybase.com/products/database-servers/sybaseiq). Product page.
  - [37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.
  - [38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 — A DBMS in the CPU cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.

# The Collective: A Cache-Based System Management Architecture

Ramesh Chandra

Nickolai Zeldovich

Constantine Sapuntzakis

Monica S. Lam

Computer Science Department

Stanford University

Stanford, CA 94305

{rameshch, nickolai, csapuntz, lam}@cs.stanford.edu

## Abstract

This paper presents the Collective, a system that delivers managed desktops to personal computer (PC) users. System administrators are responsible for the creation and maintenance of the desktop environments, or *virtual appliances*, which include the operating system and all installed applications. PCs run client software, called the *virtual appliance transceiver*, that caches and runs the latest copies of appliances locally and continuously backs up changes to user data to a network repository. This model provides the advantages of central management, such as better security and lower cost of management, while leveraging the cost-effectiveness of commodity PCs.

With a straightforward design, this model provides a comprehensive suite of important system functions including machine lockdown, system updates, error recovery, backups, and support for mobility. These functions are made available to all desktop environments that run on the x86 architecture, while remaining protected from the environments and their many vulnerabilities. The model is suitable for managing computers on a LAN, WAN with broadband, or even computers occasionally disconnected from the network like a laptop. Users can access their desktops from any Collective client; they can also carry a bootable drive that converts a PC into a client; finally, they can use a remote display client from a browser to access their desktop running on a remote server.

We have developed a prototype of the Collective system and have used it for almost a year. We have found the system helpful in simplifying the management of our desktops while imposing little performance overhead.

## 1 Introduction

With the transition from mainframe computing to personal computing, the administration of systems shifted from central to distributed management. With mainframes, professionals were responsible for creating and maintaining the single environment that all users accessed. With the advent of personal computing, users got to define their environment by installing any software that fit their fancy. Unfortunately, with this freedom also came the tedious, difficult task of system manage-

ment: purchasing the equipment and software, installing the software, troubleshooting errors, performing upgrades and re-installing operating systems, performing backups, and finally recovering from problems caused by mistakes, viruses, worms and spyware.

Most users are not professionals and, as such, do not have the wherewithal to maintain systems. As a result, most personal computers are not backed up and not up to date with security patches, leaving users vulnerable to data loss and the Internet vulnerable to worms that can infect millions of computers in minutes [16]. In the home, the challenges we have outlined above lead to frustration; in the enterprise, the challenges cost money.

The difficulties in managing distributed PCs has prompted a revival in interest in thin-client computing, both academically [15, 8] and commercially [2]. Reminiscent of mainframe computing, computation is performed on computers centralized in the data center. On the user's desk is either a special-purpose remote display terminal or a general-purpose personal computer running remote display software. Unfortunately, this model has higher hardware costs and does not perform as well. Today, the cheapest thin clients are PCs without a hard disk, but unlike a stand-alone PCs, a thin client also needs a server that does the computation, increasing hardware costs. The service provider must provision enough computers to handle the peak load; users cannot improve their computing experience by going to the store and buying a faster computer. The challenge of managing multiple desktops remains, even if it is centralized in the data center. Finally, remote display, especially over slower links, cannot deliver the same interactivity as local applications.

### 1.1 Cache-based System Management

This paper presents a cache-based system management model that combines the advantages of centralized management while taking advantage of inexpensive PCs. Our model delivers instances of the same software environments to desktop computers automatically, thereby amortizing the cost of the management. This design trades off users' ability to customize their own environment in return for uniformity, scalability, better security and lower cost of management.

In our model, we separate the state in a computer into two parts: system state and user state. The system state consists of an operating system and all installed applications. We refer to the system state as an *appliance* to emphasize that only the administrator is allowed to modify the system function; thus, to the user the system state defines a fixed function, just like any appliance. Note that these appliances are *virtual appliances* because unlike real appliances, they do not come with dedicated hardware. User state consists of a user's profile, preferences, and data files.

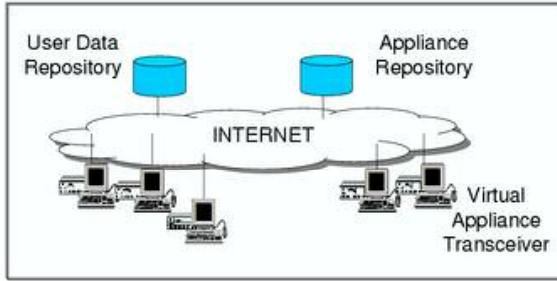


Figure 1: Architecture of the Collective system

In the cache-based system management model, appliances and user state are stored separately in network-accessible *appliance repositories* and *data repositories*, as shown in Figure 1. PCs in this model are fixed-function devices called *virtual appliance transceivers* (VATs).

In this model, a user can walk up to any of these clients, log in, and get access to any appliance he is entitled to. The VAT performs the following functions:

1. authenticates users.
2. fetches and runs the latest copies of appliances locally.
3. backs up user state changes to the data repository continuously; only administrators are allowed to update appliances in the appliance repository.
4. optimizes the system by managing a cache to reduce the amount of the data that needs to be fetched over the network.

Because appliances run locally, rather than on some central server, users experience performance and interactivity similar to their current PC environment; this approach provides the benefits of central management while leveraging commodity PCs.

## 1.2 System Highlights

We have developed a prototype based on this model which we call the Collective [14, 12]. By using x86 virtualization technology provided by the VMware GSX Server [20], the Collective client can manage and run most

software that runs on an x86 PC. We have used the system daily since June 2004. Throughout this period, we have extended our system and rewritten parts of it several times. The system that we are describing is the culmination of many months of experience. This paper presents the design and rationale for cache-based system management, as well as the detailed design and implementation of the Collective. We also measure our prototype and describe our experiences with it.

Our cache-based system management model has several noteworthy characteristics. First, the VAT is a separate layer in the software stack devoted to management. The VAT is protected from the appliances it manages by the virtual machine monitor, increasing our confidence in the VAT's security and reliability in the face of appliance compromise or malfunction. Finally, the VAT automatically updates itself, requiring little to no management on the part of the user.

Second, the system delivers a comprehensive suite of critical system management functions automatically and efficiently, including disk imaging, machine lockdown, software updates, backups, system and data recovery, mobile computing, and disconnected operation, through a simple and unified design based on caching. Our cache design is kept simple with the use of a versioning scheme where every data item is referred to by a unique name. In contrast, these management functions are currently provided by a host of different software packages, often requiring manual intervention.

Third, our design presents a uniform user interface, while providing performance and security, across computers with different network connectivities and even on computers not running the VAT software. The design works on computers connected to a LAN or WAN with broadband bandwidth; it even works when computers are occasionally disconnected. It also enables a new mobility model, where users carry a portable storage device such as an 1.8-inch disk. With the disk, they can boot any compatible PC and run the VAT. Finally, in the case where users can only get access to a conventional PC, they can access their environment using a browser, albeit only at remote access speeds.

In contrast, without a solution that delivers performance across different network connectivities, enterprises often resort to using a complicated set of techniques as a compromise. In a large organization, users in the main campus may use PCs since they have enough IT staff to manage them; remote offices may use thin clients instead, trading performance for reduced IT staff. Employees may manage their home PCs, installing corporate applications, and accessing corporate infrastructure via VPN. Finally, ad hoc solutions may be used to manage laptops, as they are seldom connected to the network unless they are in use.

The combination of centralized management and cheap PCs, provided by the Collective, offers a number of practical benefits. This approach lowers the management cost at the cost of a small performance overhead. More importantly, this approach improves security by keeping software up to date and locking down desktops. Recovery from failures, errors, and attacks is made possible by continuous backup. And, the user sees the same computing environment at the office, at home, or on the road.

### 1.3 Paper Organization

The rest of the paper is structured as follows. Section 2 presents an overview of the system. Section 3 discusses the design in more detail. We present quantitative evaluation in section 4 and qualitative user experience in Section 5. Section 6 discusses the related work and Section 7 concludes the paper.

## 2 System Overview

This section provides an overview of the Collective. We start by presenting the data types in the system: appliances, repositories and subscriptions. We then show how the Collective works from a user's perspective. We describe how the Collective's architecture provides mobility and management functions and present optimizations that allow the Collective to perform well under different connectivities.

### 2.1 Appliances

An appliance encapsulates a computer state into a virtual machine which consists of the following:

- *System disks* are created by administrators and hold the appliance's operating system and applications. As part of the appliance model, the contents of the system disk at every boot are made identical to the contents published by the administrator. As the appliance runs, it may mutate the disk.
- *User disks* hold persistent data private to a user, such as user files and settings.
- *Ephemeral disks* hold user data that is not backed up. They are used to hold ephemeral data such browser caches and temporary files; there is little reason to incur additional traffic to back up such data over the network.
- A *memory image* holds the state of a suspended appliance.

### 2.2 Repositories

Many of the management benefits of the Collective derive from the versioning of appliances in network repositories. In the Collective, each appliance has a repository; updates

to that appliance get published as a new version in that repository. This allows the VAT to automatically find and retrieve the latest version of the appliance.

To keep consistency simple, versions are immutable. To save space and to optimize data transfer, we use copy-on-write (COW) disks to express the differences between versions.

### 2.3 Subscriptions

Users have accounts, which are used to perform access control and keep per-user state. Associated with a user's account is the user's Collective profile which exists on network storage. In the user's Collective profile is a list of appliances that the user has access to. When the user first runs an appliance, a *subscription* is created in the profile to store the user state associated with that appliance.

To version user disks, each subscription in the user's Collective profile contains a repository for the user disks associated with the appliance. The first version in the repository is a copy of an initial user disk published in the appliance repository.

Other state associated with the subscription is stored only on storage local to the VAT. When an appliance starts, a COW copy of the system disk is created locally. Also, an ephemeral disk is instantiated, if the appliance requires one but it does not already exist. When an appliance is suspended, a memory image is written to the VAT's storage.

Since we do not transfer state between VATs directly, we cannot migrate suspended appliances between VATs. This is mitigated by the user's ability to carry their VAT with them on a portable storage device.

To prevent the complexity of branching the user disk, the user should not start a subscribed appliance on a VAT while it is running on another VAT. So that we can, in many cases, detect this case and warn the user, a VAT will attempt to acquire and hold a lock for the subscription while the appliance is running.

### 2.4 User Interface

The VAT's user interface is very simple—it authenticates the user and allows him to perform a handful of operations on appliances. On bootup, the VAT presents a Collective log-in window, where a user can enter his username and password. The system then presents the user with a list of appliances that he has access to, along with their status. Choosing from a menu, the user can perform any of the operations on his appliances:

- *start* boots the latest version of the appliance, or if a suspended memory image is available locally, resumes the appliance.
- *stop* shuts down the appliance.
- *suspend* suspends the appliance to a memory image.

- *reset* destroys all ephemeral disks and the memory image but retains the user disks
- *delete* destroys the subscription including the user disks
- *user disk undo* allows the user to go back to a previous snapshot of their user disk.
- *publish* allows the administrator to save the current version of the system disk as the latest version of the appliance.

When a user starts an appliance, the appliance takes over the whole screen once it runs; at any time, the user can hit a hot key sequence (Ctrl-Alt-Shift) and return to the list of appliances to perform other operations. The VAT user interface also indicates the amount of data that remains to be backed up from the local storage device. When this hits zero, the user can log out of the VAT and safely shift to using the virtual machines on another device.

## 2.5 Management Functions

We now discuss how the various management functions are implemented using caching and versioning.

### 2.5.1 System Updates

Desktop PCs need to be updated constantly. These upgrades include security patches to operating systems or installed applications, installations of new software, upgrades of the operating system to a new version, and finally re-installation of the operating system from scratch. All software upgrades, be they small or big, are accomplished in our system with the same mechanism. The system administrator prepares a new version of the appliance and deposits it in the appliance repository. The user gets the latest copy of the system disks the next time they reboot the appliance. The VAT can inform the user that a new version of the system disk is available, encouraging the user to reboot, or the VAT can even force a reboot to disallow use of the older version. From a user's standpoint, upgrade involves minimal work; they just reboot their appliances. Many software updates and installations on Windows already require the computer to be rebooted to take effect.

This update approach has some advantages over package and patch systems like yum [24], RPM [1], Windows Installer [22], and Windows Update. First, patches may fail on some users' computers because of interactions with user-installed software. Our updates are guaranteed to move the appliance to a new consistent state. Users running older versions are unaffected until they reboot. Even computers that have been off or disconnected get the latest software updates when restarted; with many patch management deployments, this is not the case. Finally, our

update approach works no matter how badly the software in the appliance is functioning, since the VAT is protected from the appliance software. However, our model requires the user to subscribe to an entire appliance; patching works with individual applications and integrates well in environments where users mix and match their applications.

Fully automating the update process, the VAT itself is managed as an appliance. It automatically updates itself from images hosted in a repository. This is described in more detail in Section 3. By making software updates easy, we expect environments to be much more up to date with the Collective and hence more secure.

### 2.5.2 Machine Lockdown

Our scheme locks down user desktops because changes made to the system disks are saved in a new version of the disk and are discarded when the appliance is shut down. This means that if a user accidentally installs undesirable software like spyware into the system state, these changes are wiped out.

Of course, undesirable software may still install itself into the user's state. Even in this case, the Collective architecture provides the advantage of being able to reboot to an uncompromised system image with uncompromised virus scanning tools. If run before accessing ephemeral and user data, the uncompromised virus scanner can stay uncompromised and hopefully clean the changed state.

### 2.5.3 Backup

The VAT creates a COW snapshot of the user disk whenever the appliance is rebooted and also periodically as the appliance runs. The VAT backs up the COW disks for each version to the user repository. The VAT interface allows users to roll back changes made since the last reboot or to return to other previous versions. This allows the user to recover from errors, no matter the cause, be it spyware or user error.

When the user uses multiple VATs to access his appliances without waiting for backup to complete, potential for conflicts in the user disk repository arises. The backup protocol ensures that only one VAT can upload user disk snapshots into the repository at a time. If multiple VATs attempt to upload user disks at the same time, the user is first asked to choose which VAT gets to back up into the subscription and then given the choice of terminating the other backups or creating additional subscriptions for them.

If an appliance is writing and overwriting large quantities of data, and there is insufficient network bandwidth for backup, snapshots can accumulate at the client, buffered for upload. In the extreme, they can potentially fill the disk. We contemplate two strategies to deal with accumulating snapshots: collapse multiple snapshots to-

gether and reduce the frequency of snapshots. In the worst case, the VAT can stop creating new snapshots and collapse all of the snapshots into one; the amount of disk space taken by the snapshot is then bounded by the size of the virtual disk.

#### 2.5.4 Hardware Management

Hardware management becomes simpler in the Collective because PCs running the VAT are interchangeable; there is no state on a PC that cannot be discarded. Deploying hardware involves loading PCs with the VAT software. Provisioning is easy because users can get access to their environments on any of the VATs. Faulty computers can be replaced without manually customizing the new computer.

### 2.6 Optimizations for Different Network Connectivities

To reduce network and server usage and improve performance, the VAT includes a large on-disk cache, on the order of gigabytes. The cache keeps local copies of the system and user disk blocks from the appliance and data repositories, respectively. Besides fetching data on demand, our system also prefetches data into the cache. To ensure good write performance even on low-bandwidth connections, all appliance writes go to the VAT's local disk; the backup process described in section 2.5.3 sends updates back in the background.

The cache makes it possible for this model to perform well under different network connectivities. We shall show below how our system allows users to access their environments on any computer, even computers with no pre-installed VAT client software, albeit with reduced performance.

#### 2.6.1 LAN

On low-latency, high bandwidth (e.g., 100 Mbps) networks, the system performs reasonably well even if the cache is empty. Data can be fetched from the repositories fast enough to sustain good responsiveness. The cache is still valuable for reducing network and server bandwidth requirements. The user can easily move about in a LAN environment, since it is relatively fast to fetch data from a repository.

#### 2.6.2 WAN with Broadband

By keeping a local copy of data from the repository, the cache reduces the need for data accesses over the network. This is significant because demand-fetching every block at broadband bandwidth and latency would make the system noticeably sluggish to the user. We avoid this worst-case scenario with the following techniques. First, at the time the VAT client software is installed on the hard disk, we also populate the cache with blocks of the appliances most likely to be used. This way only updates

need to be fetched. Second, the VAT prefetches data in the background whenever updates for the appliances in use are available. If, despite these optimizations, the user wishes to access an appliance that has not been cached, the user will find the application sluggish when using a feature for the first time. The performance of the feature should subsequently improve as its associated code and data get cached. In this case, although the system may try the user's patience, the system is guaranteed to work without the user knowing details about installing software and other system administration information.

#### 2.6.3 Disconnected Operation with Laptops

A user can ensure access to a warm cache of their appliances by carrying the VAT on a laptop. The challenge here is that a laptop is disconnected from the network from time to time. By hoarding all the blocks of the appliances the user wishes to use, we can keep operating even while disconnected.

#### 2.6.4 Portable VATs

The self-containment of the VAT makes possible a new model of mobility. Instead of carrying a laptop, we can carry the VAT, with a personalized cache, on a bootable, portable storage device. Portable storage devices are fast, light, cheap, and small. In particular, we can buy a 1.8-inch, 40GB, 4200 rpm portable disk, weighing about 2 ounces, for about \$140 today. Modern PCs can boot from a portable drive connected via USB.

The portable VAT has these advantages:

1. *Universality and independence of the computer hosts.* Eliminating dependences on the software of the hosting computer, the device allows us to convert any x86 PC into a Collective VAT. This approach leaves the hosting computer undisturbed, which is a significant benefit to the hosting party. Friends and relatives need not worry about their visitors modifying their computing environments accidentally, although malicious visitors can still wreak havoc on the disks in the computers.
2. *Performance.* The cache in the portable VAT serves as a network accelerator. This is especially important if we wish to use computers on low-bandwidth networks.
3. *Fault tolerance.* Under typical operation, the VAT does not contain any indispensable state when not in use; thus, in the event the portable drive is lost or forgotten, the user gets access to his data by inserting another generic VAT and continuing to work, albeit at a slower speed.
4. *Security and privacy.* This approach does not disturb the hosting computer nor does it leave any trace

of its execution on the hosting computer. Data on the portable drive can be encrypted to maintain secrecy if the portable drive is lost or stolen. However, there is always the possibility that the firmware of the computer has been doctored to spy on the computations being performed. Trusted computing techniques [18, 5] can be applied here to provide more security; hardware could in theory attest to the drive the identity of the firmware.

### 2.6.5 Remote Display

Finally, in case the users do not have access to any VATs, they can access their environments using remote display. We recreate a user experience similar to the one with the VAT; the user logs in, is presented with a list of appliances and can click on them to begin using them. The appliances are run on a server and a window appears with an embedded Java remote display applet that communicates with the server.

## 3 Design of the VAT

In this section, we present the design of the appliance transceiver. The VAT's major challenges include running on as many computers as possible, automatically updating itself, authenticating users and running their appliances, and working well on slow networks.

### 3.1 Hardware Abstraction

We would like the VAT image to run on as many different hardware configurations as possible. This allows users with a bootable USB drive to access their state from almost any computer that they might have available to them. It also reduces the number of VAT images we need to maintain, simplifying our administration burden.

To build a VAT that would support a wide range of hardware, we modified KNOPPIX [6], a *Live CD* version of Linux that automatically detects available hardware at boot time and loads the appropriate Linux drivers. KNOPPIX includes most of the drivers available for Linux today.

KNOPPIX's ability to quickly auto-configure itself to a computer's hardware allows the same VAT software to be used on many computers without any per-computer modification or configuration, greatly simplifying the management of an environment with diverse hardware. We have found only one common situation where the VAT cannot configure itself without the user's help: to join a wireless network, the user may need to select a network and provide an encryption key.

If a VAT successfully runs on a computer, we can be reasonably sure the appliances running on it will. The appliances run by the VAT see a reasonably uniform set of virtual devices; VMware GSX server takes advantage of the VAT's device drivers to map these virtual devices to a wide range of real hardware devices.

### 3.2 User Authentication

To maintain security, users must identify themselves to the VAT and provide credentials that will be used by the VAT to access their storage on their behalf. As part of logging in, the user enters his username and password. The VAT then uses SSH and the password to authenticate the user to the server storing the user's Collective profile. To minimize the lifetime of the user's password in memory, the VAT sets up a key pair with the storage server so that it can use a private key, rather than a password, to access storage on behalf of the user.

Disconnected operation poses a challenge, as there is no server to contact. However, if a user has already logged in previously, the VAT can authenticate him. When first created, the private key mentioned in the previous paragraph is stored encrypted with the user's password. On a subsequent login, if the password entered successfully decrypts the private key, the user is allowed to login and access the cached appliances and data.

### 3.3 VAT Maintenance

As mentioned earlier, the VAT is managed as an appliance, and needs zero maintenance from the end user; it automatically updates itself from a repository managed by a VAT administrator.

For most problems with the VAT software, a reboot restores the software to a working state. This is because the VAT software consists largely of a read-only file system image. Any changes made during a session are captured in separate file systems on a ramdisk. As a result, the VAT software does not drift from the published image.

All VATs run an update process which checks a repository for new versions of the VAT software and downloads them to the VAT disk when they become available. To ensure the integrity and authenticity of the VAT software, each version is signed by the VAT publisher. The repository location and the public key of publisher are stored on the VAT disk.

After downloading the updated image, the update process verifies the signature and atomically changes the boot sector to point to the new version. To guarantee progress, we allocate enough room for three versions of the VAT image: the currently running version, a potentially newer version that is pointed to by the boot sector which will be used at next reboot, and an even newer, incomplete version that is in the process of being downloaded or verified.

The VAT image is about 350MB uncompressed; downloading a completely new image wastes precious network capacity on broadband links. To reduce the size to about 160MB, we use the `cloop` tools that come with KNOPPIX to generate a compressed disk image; by using the `cloop` kernel driver, the VAT can mount the root file system from the compressed file system directly. Even so, we expect most updates to touch only a few files; transferring

an entire 160MB image is inefficient. To avoid transferring blocks already at the client, the update process uses rsync; for small changes, the size of the update is reduced from 160MB to about 10MB.

### 3.4 Storage Access

Our network repositories have the simplest layout we could imagine; we hope this will let us use a variety of access protocols. Each repository is a directory; each version is a subdirectory whose name is the version number. The versioned objects are stored as files in the subdirectories. Versions are given whole numbers starting at 1. Since some protocols (like HTTP) have no standard directory format, we keep a `latest` file in the repository's main directory that indicates the highest version number.

To keep consistency simple, we do not allow a file to be changed once it has been published into a repository. However, it should be possible to reclaim space of versions that are old; as such, files and versions can be deleted from the repository. Nothing prevents the deletion of an active version; the repository does not keep track of the active users.

When reading from network storage, we wanted a simple, efficient protocol that could support demand paging of large objects, like disk images. For our prototype, we use NFS. NFS has fast, reliable servers and clients. To work around NFS's poor authentication, we tunnel NFS over SSH.

While demand paging a disk from the repository, we may become disconnected. If a request takes too long to return, the disk drivers in the appliance's OS will timeout and return an I/O error. This can lead to file system errors which can cause the system to panic. In some cases, suspending the OS before the timeout and resuming it once we have the block can prevent these errors. A better solution is to try to make sure this does not happen in the first place by aggressively caching blocks; we will discuss this approach more in Section 3.6.

We also use NFS to write to network storage. To atomically add a new version to a repository, the VAT first creates and populates the new version under a one-time directory name. As part of the process, it places a nonce in the directory. The VAT then renames the directory to its final name and checks the nonce to see if it succeeded.

We would also like to set the priority of the writes to user data repositories with respect to other network traffic. Currently, the VAT approximates this by mounting the NFS server again on a different mount point; this in turn uses a separate TCP connection, which is given a different priority. Another consideration is that when an NFS server is slow or disconnected, the NFS in-kernel client will buffer writes in memory, eventually filling memory with dirty blocks and degrading performance. To limit

the quantity of dirty data, the VAT performs an fsync after every 64 kilobytes of writes to the user data repository.

### 3.5 Caching

Our cache is designed to mask the high latency and low bandwidth of wide-area communication by taking advantage of large, persistent local storage, like hard disks and flash drives. At the extreme, the cache allows the client to operate disconnected.

COW disks can be gigabytes in size; whole file caching them would lead to impractical startup times. On the other hand, most of the other data in our system, like the virtual machine description, is well under 25 kilobytes. As a result, we found it easiest to engineer two caches: a small object cache for small data and meta-data and a COW cache for COW disk blocks.

To simplify disconnected operation, small objects, like the user's list of appliances or the meta-data associated with a repository, are replicated in their entirety. All the code reads the replicated copy directly; a background process periodically polls the servers for updates and integrates them into the local replica. User data snapshots, which are not necessarily small, are also stored in their entirety before being uploaded to the server.

The COW cache is designed to cache immutable versions of disks from repositories; as such the name of a disk in the cache includes an ID identifying the repository (currently the URL), the disk ID (currently the disk file name), and the version number. To name a specific block on a disk, the offset on disk is added. Since a data block can change location when COW disk chains are collapsed, we use the offset in the virtual disk, not the offset in the COW disk file.

One of the challenges of on-disk data structures is dealing with crashes, which can happen at any time, leading to partial writes and random data in files. With a large cache, scanning the disk after a crash is unattractive. To cope with partial writes and other errors introduced by the file system, each 512-byte sector stored in the cache is protected by an MD5 hash over its content and its address. If the hash fails, the cache assumes the data is not resident in the cache.

A traditional challenge with file caches has been invalidation; however, our cache needs no invalidation protocol. The names used when storing and retrieving data from the cache include the version number; since any given version of an object is immutable, no invalidation is necessary.

Our cache implementation does not currently make an effort to place sequential blocks close to each other on disk. As a result, workloads that are optimized for sequential disk access perform noticeably slower with our cache, due to the large number of incurred seeks. One such common workload is system bootup; we have implemented a *bootup block optimization* for this case. Since the block

access pattern during system bootup is highly predictable, a trace of the accessed blocks is saved along with each virtual appliance. When an appliance is started, the trace is replayed, bringing the blocks into the buffer cache before the appliance OS requests them. This optimization significantly reduces bootup time. A more general version of this technique can be applied to other predictable block access patterns, such as those associated with starting large applications.

### 3.6 Prefetching

To minimize cache misses, the VAT runs a prefetcher process to fetch useful appliance blocks in the background. The prefetcher checks for updates to appliances used by the VAT user, and populates the cache with blocks from the updated appliances. One optimization is to prioritize the blocks using access frequency so that the more important data can be prefetched first.

The user can use an appliance in disconnected mode by completely prefetching a version of an appliance into the cache. The VAT user interface indicates to the user what versions of his appliances have been completely prefetched. The user can also manually issue a command to the prefetcher if he explicitly wants to save a complete version of the appliance in the cache.

The prefetcher reduces interference with other processes by rate-limiting itself. It maintains the latencies of recent requests and uses these to determine the extent of contention for network or disk resources. The prefetcher halves its rate if the percentage of recent requests experiencing a high latency exceeds a threshold; otherwise, it doubles its rate when it finds a large percentage experience a low latency. If none of these apply, it increases or decreases request rate by a small constant based on the latency of the last request.

Prefetching puts spare resources to good use by utilizing them to provide better user experience in the future: when a user accesses an appliance version for the first time, it is likely that the relevant blocks would already be cached. Prefetching hides network latency from the appliance, and better utilizes network bandwidth by streaming data rather than fetching it on demand.

## 4 Evaluation

We provide some quantitative measurements of the system to give a sense of how the system behaves. We perform four sets of experiments. We first use a set of benchmarks to characterize the overhead of the system and the effect of using different portable drives. We then present some statistics on how three appliances we have created have evolved over time. Next, we evaluate prefetching. We show that a small amount of data accounts for most of the accesses, and that prefetching can greatly improve

the responsiveness of an interactive workload. Finally, we study the feasibility of continuous backups.

### 4.1 Run-Time Performance

We first establish some basic parameters of our system by running a number of benchmarks under different conditions. All of the experiments, unless noted otherwise, are run on 2.4GHz Pentium IV machines with 1GB of memory and a 40GB Hitachi 1.8" hard drive connected via Prolific Technology's PL-2507 USB-to-IDE bridge controller. VAT software running on the experimental machines is based on Linux kernel 2.6.11.4 and VMware GSX server version 3.1. The file server is a 2.4GHz Pentium IV with 1GB of memory and a Linux software RAID, consisting of four 160GB IDE drives. We use FreeBSD's *dummynet* [11] network simulator to compare the performance of our system over a 100 Mbps LAN to that over a 1.5 Mbps downlink / 384 Kbps uplink DSL connection with 40 msec round-trip delay.

#### 4.1.1 Effects of Caching

To evaluate caching, we use three repeatable workloads: bootup and shutdown of a Linux VM, bootup and shutdown of a Windows XP VM, and building the Linux 2.4.23 kernel in a VM. The runtime of each workload is measured in different network and cache configurations to illustrate how caching and network connectivity affect performance. All workloads are run with both an empty and a fully prefetched initial cache. We also repeat the workloads with a fully prefetched cache but without the bootup block optimization, to show the optimization's effect on startup performance.

By running the same virtual machine workloads on an unmodified version of VMware's GSX server, we quantify the benefits and overheads imposed by the Collective caching system. In particular, we run two sets of experiments using unmodified VMware without the Collective cache:

- *Local*, where the entire VM is copied to local disk and executes without demand-fetching. The COW disks of each VM disk are collapsed into a flat disk for this experiment. We expect this to provide a bound on VMware performance.
- *NFS*, where the VM is stored on an NFS file server and is demand-fetched by VMware without additional caching. This is expected to be slow in the DSL case and shows the need for caching.

Figure 2 summarizes the performance of these benchmarks. Workloads running with a fully prefetched cache are slower than the *local* workload, due to additional seek overhead imposed by the layout of blocks in our cache. The bootup block prefetching optimization, described in

Section 3.5, largely compensates for the suboptimal block layout.

As expected, the performance of our workloads is bad in both the NFS and the empty cache scenario, especially in the case of a DSL network, thus underscoring the need for caching.

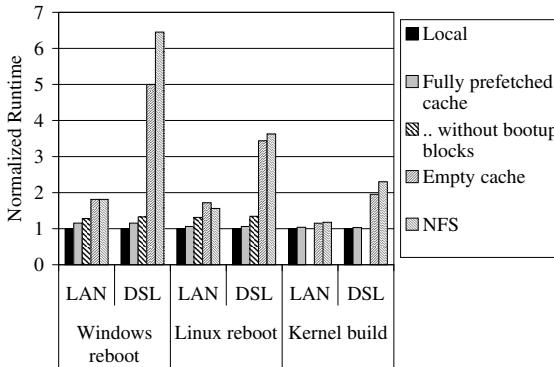


Figure 2: Runtime of workload experiments on different cache configurations when run over a 100 Mbps LAN, and a simulated DSL link with 1.5 Mbps downlink / 384 Kbps uplink and 40 msec RTT latency. The runtimes are normalized to the runtime in the local experiment. The local runtimes are 64 sec, 32 sec, and 438 sec, respectively for the Windows reboot, Linux reboot, and Linux kernel build experiments.

#### 4.1.2 Effects of disk performance

As a first test to evaluate the performance of different disks, we measured the time taken to boot the VAT software on an IBM Thinkpad T42p laptop, since our standard experimental desktop machine did not have USB 2.0 support in its BIOS. The results, shown in the first column of Figure 3 indicate that the VAT boot process is reasonably fast across different types of drives we tried.

For our second test, we run the same micro-benchmark workloads as above; to emphasize disk performance rather than network performance, the VMs are fully prefetched into the cache, and the machines are connected over a 100Mbps LAN. The results are shown in Figure 3. The flash drive performs well on this workload, because of its good read performance with zero seek time, but has limited capacity, which would prevent it from running larger applications well. The microdrive is relatively slow, largely due to its high seek time and rotational latency. In our opinion, the 1.8" hard drive offers the best price / performance / form factor combination.

## 4.2 Maintaining Appliances

We have created and maintained three virtual machine appliances over a period of time:

- a Windows XP environment. Over the course of half a year, the Windows appliance has gone through two service packs and many security updates. The ap-

	VAT startup	Windows reboot	Linux reboot	Kernel build
Lexar 1GB Flash Drive	53	129	42	455
IBM 4GB Microdrive	65	158	53	523
Hitachi 40GB 1.8" Drive	61	84	43	457
Fujitsu 60GB 2.5" Drive	52	65	40	446

Figure 3: Performance characteristics of four different VAT disks. All the numbers are in seconds. The first column shows the VAT boot times on an IBM Thinkpad T42p, from the time BIOS transfers control to the VAT's boot block to the VAT being fully up and running. In all cases the BIOS takes an additional 8 seconds initializing the system before transferring control to the VAT. The rest of the table shows results for the micro-benchmarks run with fully-primed caches when run over a 100 Mbps network.

pliance initially contained Office 2000 and was upgraded to Office 2003. The appliance includes a large number of applications such as Adobe Photoshop, FrameMaker, and Macromedia DreamWeaver.

- a Linux environment, based on Red Hat's Fedora Core, that uses NFS to access our home directories on our group file server. Over a period of eight months, the NFS Linux appliance required many security updates, which replaced major subsystems like the kernel and X server. Software was added to the NFS Linux appliance as it was found to be needed.
- a Linux environment also based on Fedora, that stores the user's home directory in a user disk. This Linux appliance included all the programs that came with the distribution and was therefore much larger. We used this appliance for two months.

Some vital statistics of these appliances are shown in Figure 4. We show the number of versions created, either due to software installations or security patches. Changes to the system happen frequently; we saved a lot of time by having to just update one instance of each appliance.

Appliance	Number of versions	Total size	Active size	Cache size
Windows XP	31	16.5	4.5	3.1
NFS Linux	20	5.7	2.8	1.4
User-disk Linux	8	7.0	4.9	3.7

Figure 4: Statistics of three appliances. Sizes are in GB.

We also measure the size of all the COW disks for each appliance ("Total size") and the size of the latest version ("Active size"). The last column of the table, "Cache size", shows an example of the cache size of an active user of each appliance. We observe from our usage that the cache size grows quickly and stabilizes within a short amount of time. It grows whenever major system updates are performed and when new applications are used for the first time. The sizes shown here represent all the blocks ever cached and may include disk blocks that may have

since been made obsolete. We have not needed to evict any blocks from our 40GB disks.

### 4.3 Effectiveness of Prefetching

In the following, we first measure the access profile to establish that prefetching a small amount of data is useful. Second, we measure the effect of prefetching on the performance of an interactive application.

#### 4.3.1 Access Profile

In this experiment, we measure the access profile of appliance blocks, to understand the effectiveness of prefetching based on the popularity of blocks. We took 15 days of usage traces from 9 users using the three appliances described above in their daily work. Note that during this period some of the appliances were updated, so the total size of data accessed was greater than the size of a single active version. For example, the Windows XP appliance had an active size of 4.5 GB and seven updates of 4.4 GB combined, for a total of 8.9 GB of accessible appliance data.

Figure 5 shows each appliance’s effective size, the size of all the accesses to the appliance in the trace, and the size of unique accesses. The results suggest that only a fraction of the appliance data is ever accessed by any user. In this trace, users access only 10 to 30% of the accessible data in the appliances.

Appliance	Accessible Size	Accesses in Traces	Unique data Accessed
Windows XP	8.9 GB	31.1 GB	2.4 GB
NFS Linux	3.4 GB	6.8 GB	1.0 GB
User-disk Linux	6 GB	5.9 GB	0.5 GB

Figure 5: Statistics of appliances in the trace.

Figure 6 shows the percentage of accesses that are satisfied by the cache (Y-axis) if a given percentage of the most popular blocks are cached (X-axis). The results show that a large fraction of data accesses are to a small fraction of the data. For example, more than 75% of data accesses in the Windows XP appliance are to less than 20% of the accessed data, which is about 5% of the total appliance size. These preliminary results suggest that popularity of accessed appliance data is a good heuristic for prefetching, and that prefetching a small fraction of the appliance’s data can significantly reduce the chances of a cache miss.

#### 4.3.2 Interactive Performance

The responsiveness of an interactive application can be severely affected by cache miss delays. Our next experiment attempts to measure the effects of prefetching on an application’s response time.

To simulate interactive workloads, we created a VNC [10] *recorder* to record user mouse and keyboard input events, and a VNC *player* to play them back to reproduce user’s actions [25]. Using VNC provides us with

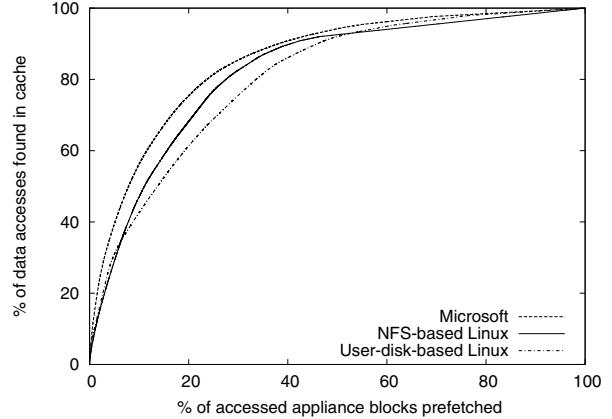


Figure 6: Block access profile: cache hit rate as a function of prefetched appliance data. Most frequently used appliance data is prefetched first.

a platform-independent mechanism for interacting with the desktop environment. Furthermore, it allows us to use VMware’s built-in VNC interface to the virtual machine console.

Other tools [19, 23] try to do this, but play back is not always correct when the system is running significantly slower (or faster) than during recording. This is especially true for mouse click events. To reliably replay user actions, our VNC recorder takes screen snapshots along with mouse click events. When replaying input events, the VNC player waits for the screen snapshot taken during recording to match the screen contents during replay before sending the mouse click.

Our replay works only on systems with little or no non-deterministic behavior. Since we use virtual machines, we can easily ensure that the initial state is the same for each experiment.

We use the Windows XP appliance to record a VNC session of a user creating a PowerPoint presentation for approximately 8 minutes in a LAN environment. This session is then replayed in the following experimental configurations:

- Local: the entire appliance VM is copied to the VAT disk and executed with unmodified VMware, without demand-fetching or caching.
- Prefetched: some of the virtual machine’s blocks are prefetched into the VAT’s cache, and the VM is then executed on top of that cache. The VAT is placed behind a simulated 1.5 Mbps / 384 Kbps DSL connection.

For the prefetched experiments, we asked four users to use various programs in our appliance, to model other people’s use of the same appliance; their block access patterns are used for prefetching blocks in the experiment.

Prefetching measures the amount of data transferred over the network; due to compression, the amount of raw disk data transferred is approximately 1.6 times more. The amount of prefetching goes up to a maximum of 420 MB, which includes all of the blocks accessed in the appliance by our users.

The total runtimes for the replayed sessions are within approximately 5% of each other – the additional latency imposed by demand-fetching disk blocks over DSL is absorbed by long periods of user think time when the system is otherwise idle. To make a meaningful comparison of the results, we measure the response time latency for each mouse click event, and plot the distribution of response times over the entire workload in Figure 7. For low response times, the curves are virtually indistinguishable. This region of the graph corresponds to events that do not result in any disk access, and hence are quick in all the scenarios. As response time increases, the curves diverge; this corresponds to events which involve accessing disk – the system takes noticeably longer to respond in this case, when disk blocks need to be demand-fetched over the network. The figure shows that PowerPoint running in the Collective is as responsive as running in a local VM, except for times when new features have to be loaded from disk – similar to Windows taking a while to start any given application for the first time.

The most commonly accessed blocks are those used in the bootup process. This experiment only measures the time taken to complete the PowerPoint workload after the system has been booted up, and therefore the benefit of prefetching the startup blocks is not apparent in the results shown in the figure. However, prefetching the startup blocks (approximately 100 MB) improves startup time from 391 seconds in the no prefetching case to 127 seconds when 200 MB of data is prefetched.

The results show that prefetching improves interactive performance. In the case of full prefetching, the performance matches that of a local VM. Partial prefetching is also beneficial – we can see that prefetching 200 MB significantly improves the interactive performance of PowerPoint.

#### 4.4 Feasibility of Online Backup

Ideally, in our system, user data should always be backed up onto network storage. To determine whether online backup works for real workloads, we collected usage traces for three weeks on personal computers of ten users running Windows XP. These users included office workers, home users, and graduate students. The traces contain information on disk block reads and writes, file opens and start and end of processes. We also monitored idle times of keyboard and mouse; we assume the user to be idle if the idle time exceeds five minutes.

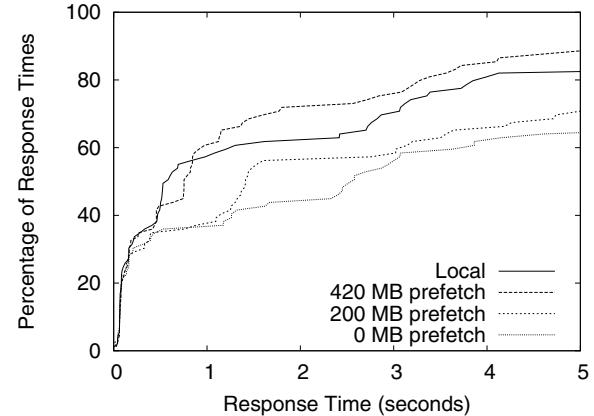


Figure 7: CDF plot of response times observed by the user during a PowerPoint session, for different levels of prefetching.

We expect that in our system the user would log out and possibly shut down his VAT soon after he completes his work. So, the measure we are interested in is whether there is any data that is not backed up when he becomes idle. If all the data is backed up, then the user can log in from any other VAT and get his most recent user data; if the user uses a portable VAT, he could lose it with no adverse effects.

To quantify this measure we simulated the usage traces on our cache running over a 384 Kbps DSL uplink. To perform the simulation we divided the disk writes from the usage data into writes to system data, user data, and ephemeral data. These correspond to the system disk, user disk, and ephemeral disk that were discussed earlier. System data consists of the writes that are done in the normal course by an operating system that need not be backed up. Examples of this include paging, defragmentation, NTFS metadata updates to system disk, and virus scans. User data consists of the data that the user would want to be backed up. This includes email documents, office documents, etc., We categorize internet browser cache, and media objects such as mp3 files, that are downloaded from the web as ephemeral data and do not consider them for backup. In our traces there were a total of about 300GB worth of writes of which about 3.3% were to user data, 3.4% were to ephemeral data and the rest to program data. Users were idle 1278 times in the trace, and in our simulation, backup stops during idle periods. We estimate the size of dirty data in the cache when users become idle.

The results are presented in Figure 8. The x-axis shows the size of data that is not backed up, and the y-axis shows the percentage of idle periods. From the figure we see that most of the time there is very little data to be backed up by the time the user becomes idle. This suggests that interactive users have large amounts of think time and generate little backup traffic. This also shows that online backup,

as implemented in the Collective, works well even on a DSL link. Even in the worst case, the size of dirty data is only about 35 MB, which takes less than 15 minutes to backup on DSL.

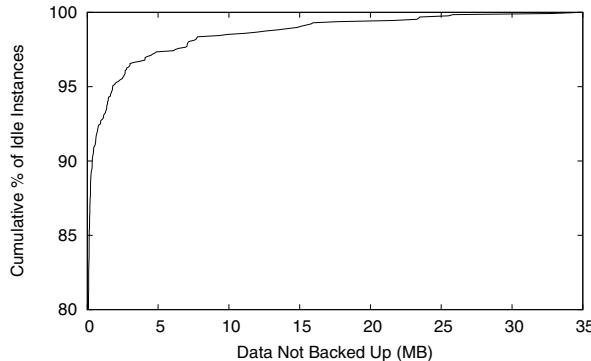


Figure 8: Size of data that is not backed up when a user becomes idle. The graph shows the fraction of times the user would have less than a certain amount of dirty data in his cache at the end of his session.

The results presented in this section illustrate that the system performs well over different network connections, and that it provides a good interactive user experience. Further, the results support our use of prefetching for reducing cache misses, and show that continuous backup is feasible for most users.

## 5 Experiences

We have been using the Collective for our daily work since June 2004. Based on this and other experiences, we describe how the Collective helped reduce the burden of administering software and computers.

### 5.1 Uses of the System

At first, members of our research group were using the prototype for the sake of understanding how our system behaves. As the system stabilized, more people started using the Collective because it worked better than their current setup. The following real-life scenarios we encountered illustrate some of the uses of our system:

*Deploying new equipment.* Before the Collective, when we needed to set up a new desktop or laptop, it would take a couple of hours to install the operating system, applications, and configure the computer. By plugging in and booting from USB disk containing the VAT, we were able to start using the computer immediately, starting up appliances we had previously used on other computers. We also used the VAT to configure the new computer's internal hard drive to be a VAT; all it takes is one user command and, in less than 5 minutes, the computer is assimilated into the Collective.

*Fixing broken software setups.* In one case, a student adopted the Collective after he botched the upgrade of the

Linux kernel on his laptop. As a result of the failed upgrade, the laptop did not even boot. In other cases, we had lent machines to other groups and received them back with less than useful software setups. The Collective allowed us to resume work quickly by placing a VAT on the computer.

*Distributing a complex computing environment.* Over the summer, two undergraduates participated in a compiler research project that required many tools including Java, Eclipse, the JoeQ Java compiler, BDD libraries, etc. Since the students were not familiar with those tools, it would have taken each of the students a couple of days to create a working environment. Instead, an experienced graduate student created an appliance that he shared with both students, enabling both of them to start working on the research problems.

*Using multiple environments.* Our Linux appliance users concurrently start up a Windows appliance for the occasional tasks, like visiting certain web pages and running Powerpoint, that work better or require using Windows applications.

*Distributing a centrally maintained infrastructure.* Our university maintains a pool of computers that host the software for course assignments. Towards the end of the term, these computers become over-subscribed and slow. While the course software and the students' home directories are all available over a distributed file system (AFS), most students do not want to risk installing Linux and configuring AFS on their laptops. We gave students external USB drives with a VAT and created a Linux appliance that uses AFS to access course software and their home directories. The students used the VAT and the appliance to take advantage of the ample cycles on their laptops, while leaving the Windows setup on their internal drive untouched.

### 5.2 Lessons from our Experience

We appreciate that we only need to update an appliance once and all of the users can benefit from it. The authors would not be able to support all the users of the system otherwise.

The design of the VAT as a portable, self-contained, fixed-function device contributes greatly to our ability to carry out our experiments.

1. Auto-update. It is generally hard to conduct experiments involving distributed users because the software being tested needs to be fixed and improved frequently, especially at the beginning. Our system automatically updates itself allowing us to make quick iterations in the experiment without having to recall the experiment. The user needs to take no action, and the system has the appearance of healing itself upon a reboot.

- Self-containment. It is easy to get users to try out the system because we give them an external USB drive from which to boot their computer. The VAT does not disturb the computing environment stored on their internal hard drive.

The system also makes us less wary of taking actions that may compromise an appliance. For example, we can now open email attachments more willingly because our system is up to date with security patches, and we can roll back the system should the email contain a new virus. As a trial, we opened up a message containing the BagleJ email virus in a system that had not yet been patched. Because BagleJ installed itself onto the system disk, it was removed when we rebooted. We have had similar experiences with spyware; a reboot removes the spyware executables, leaving only some icons on the user's desktop to clean up.

We observed that the system can be slow when it is used to access appliance versions that have not yet been cached. This is especially true over a DSL network. Prefetching can be useful in these cases. Prefetching on a LAN is fast; on a DSL network, it is useful to leave the computer connected to the network even when it is not in use, to allow prefetching to complete. The important point to note here is that this is fully automatic and hands-free, and it is much better than having to baby-sit the software installation process. Our experience suggests that it is important to prioritize between the different kinds of network traffic performed on behalf of the users; background activities like prefetching new appliance versions or backing up user data snapshots should not interfere with normal user activity.

We found that the performance of the Collective is not satisfactory for I/O intensive applications such as software builds, and graphics intensive applications such as video games. The virtualization overhead, along with the I/O overhead of our cache makes the Collective not suitable for these applications.

Finally, many software licenses restrict the installation of software to a single computer. Software increasingly comes with activation and other copy protection measures. Being part of a large organization that negotiates volume licenses, we avoided these licensing issues. However, the current software licensing model will have to change for the Collective model to be widely adopted.

## 6 Related Work

To help manage software across wide-area grids, GVFS [26] transfers hardware-level virtual machines. Their independent design shares many similarities to our design, including on-disk caches, NFS over SSH, and VMM-specific cache coherence. The Collective evaluates a broader system, encompassing portable storage, user

data, virtual appliance transceiver, and initial user experiences.

Internet Suspend/Resume (ISR) [7] uses virtual machines and a portable cache to provide mobility; the Collective architecture also provides management features like rollback and automatic update, in addition to mobility. Similar to our previous work [13], ISR uses a cache indexed by content hash. In contrast, the current Collective prototype uses COW disks and a cache indexed by location. We feel that any system like the Collective needs COW disks to succinctly express versions; also, indexing the cache by location was straightforward to implement. Index by hash does have the advantage of being able to use a cached block from an unrelated disk image. Our previous work [13] suggests that there is promise in combining COW disks and index by hash. In the case a user does not wish to carry portable storage, ISR also implements *proactive* prefetching, which sends updated blocks to the computers the user uses commonly in anticipation of the user arriving there. The Collective uses prefetching of data from repositories to improve the performance at VATs where the user is already logged in. The two approaches are complementary.

Managing software using disk images is common; a popular tool is Symantec Ghost [17]. Unlike our system, a compromised operating system can disable Ghost since the operating system has full access to the raw hardware. In addition, since Ghost does not play copy-on-write tricks, roll back involves rewriting the whole partition. This potentially lengthy process limits the frequency of ghosting. Finally, Ghost leaves it to the administrator and other tools to address how to manage user data.

Using network repositories for disk images and expressing updates compactly using differences are explored by Rauch et al [9]. A different way of distributing disk images is Live CDs, bootable CDs with a complete software environment. Live CDs provide lock down and can easily roll back changes to operating systems. However, they do not provide automatic updates and management of user data.

Various solutions for transparent install and update exist for platforms other than x86 hardware. Java has Java Web Start [21]; some Windows games use Valve Steam; Konvalo and Zero Install manage Linux applications. The Collective uses virtual machine technology and an automatically updating virtual appliance transceiver to manage the entire software stack.

Like the Collective, MIT's Project Athena [4] provides the management benefits of centralized computing while using the power of distributed desktop computers. In Athena, management is a service that runs alongside applications; in contrast, the Collective's management software are protected from the applications by a virtual machine monitor. The Collective uses a disk-based ab-

straction to distribute software and user data; in contrast, Athena uses a distributed file system. By explicitly exposing multiple versions of disk images through repositories, the Collective can provide consistent snapshots of software and does not force users to start using the new version immediately. In contrast, software run from a network file system must be carefully laid out and managed to provide similar semantics. In Athena, users can mix and match software from many providers; in our model, an appliance is a monolithic unit created and tested by an administrator.

Candea et al [3] have explored rebooting components of a running system as a simple, consistent, and fast method of recovery. The Collective uses reboots to rollback changes and provide upgrades, providing similar advantages.

## 7 Conclusions

This paper presents the Collective, a prototype of a system management architecture for managing desktop computers. This paper concentrates on the design issues of a complete system. By combining simple concepts of caching, separation of system and user state, network storage, and versioning, the Collective provides several management benefits, including centralized management, atomic updates, and recovery via rollback.

Our design of a portable, self-managing virtual appliance transceiver makes the Collective infrastructure itself easy to deploy and maintain. Caching in the Collective helps provide good interactive performance even over wide-area networks. Our experience and the experimental data gathered on the system suggest that the Collective system management architecture can provide a practical solution to the complex problem of system management.

## 8 Acknowledgments

The work for this paper was funded in part by the National Science Foundation under Grant No. 0121481. We would like to thank Ben Pfaff, Chris Unkel, Emre Kiciman, George Candea, Arvind Arasu, Mendel Rosenblum, Eu-jin Goh, our shepherd Ed Lazowska, and the anonymous reviewers for their comments.

## References

- [1] E. Bailey. *Maximum RPM*. Sams, 1st edition, 1997.
- [2] W. M. Bulkeley. The office PC slims down. *The Wall Street Journal*, January 2005.
- [3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, December 2004.
- [4] G. Champie, J. Daniel Geer, and W. Ruh. Project Athena as a distributed computer system. *IEEE Computer Magazine*, 23(9):40–51, September 1990.
- [5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, pages 193–206, October 2003.
- [6] KNOPPIX Live CD Linux distribution. <http://www.knoppix.org/>.
- [7] M. Kozuch, M. Satyanarayanan, T. Bressoud, C. Helfrich, and S. Sinnamohideen. Seamless mobile computing on fixed infrastructure. Technical Report 28, Intel Research Pittsburgh, 2004.
- [8] J. Nieh, S. J. Yang, and N. Novik. Measuring thin-client performance using slow-motion benchmarking. *ACM Transactions on Computer Systems*, 21(1), February 2003.
- [9] F. Rauch, C. Kurmann, and T. Stricker. Partition repositories for partition cloning—OS independent software maintenance in large clusters of PCs. In *Proceedings of the IEEE International Conference on Cluster Computing 2000*, pages 233–242, November/December 2000.
- [10] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, January/February 1998.
- [11] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.
- [12] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, J. Norris, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of Seventeenth USENIX Large Installation System Administration Conference*, pages 181–194, October 2003.
- [13] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 377–390, December 2002.
- [14] C. Sapuntzakis and M. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Workshop on Hot Topics in Operating Systems*, pages 55–60, May 2003.
- [15] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 32–47, December 1999.
- [16] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.
- [17] Symantec Ghost. <http://www.ghost.com/>.
- [18] TCPA. <http://www.trustedcomputing.org/>.
- [19] Rational VisualTest. <http://www.ibm.com/software/awdtools/tester/robot/>.
- [20] VMware GSX server. [http://www.vmware.com/products/server/gsx\\_features.html](http://www.vmware.com/products/server/gsx_features.html).
- [21] Java web start. <http://java.sun.com/j2se/1.5.0/docs/guide/javaws/>.
- [22] P. Wilson. *Definitive Guide to Windows Installer*. Apress, 1st edition, 2004.
- [23] Xnee home page. <http://www.gnu.org/software/xnee/www>.
- [24] Yellowdog updater modified (yum). <http://linux.duke.edu/projects/yum/>.
- [25] N. Zeldovich and R. Chandra. Interactive performance measurement with VNCplay. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 153–162, April 2005.
- [26] M. Zhao, J. Zhang, and R. Figueiredo. Distributed file system support for virtual machines in grid computing. In *Proceedings of the Thirteenth IEEE Symposium on High-Performance Distributed Computing*, June 2004.

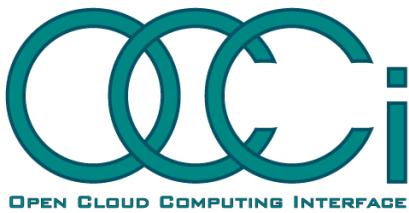


---

September 2009

# Cloud Storage for Cloud Computing

This paper is a joint production of the Storage Networking Industry Association and the Open Grid Forum. Copyright © 2009 Open Grid Forum, Copyright © 2009 Storage Networking Industry Association. All rights Reserved.



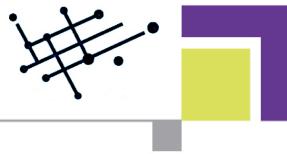
## Table of Contents

<b>Introduction.....</b>	<b>4</b>
<b>Cloud Computing Overview.....</b>	<b>4</b>
From Server Consolidation to Cloud Computing.....	4
The role of server virtualization software .....	4
How is all this managed?.....	4
<b>Standardizing Cloud Computing Interfaces.....</b>	<b>5</b>
Introducing OCCI.....	5
The OCCI Reference Architecture.....	5
<b>Cloud Storage Overview.....</b>	<b>7</b>
Some of the Use Cases.....	7
Web facing applications .....	7
Storage for Cloud Computing.....	8
What makes Cloud Storage different? .....	8
<b>Introducing CDMI .....</b>	<b>8</b>
<b>Using CDMI and OCCI for a Cloud Computing Infrastructure.....</b>	<b>9</b>
How it works .....	11
<b>Standards Coordination .....</b>	<b>11</b>
<b>About the SNIA .....</b>	<b>11</b>

## List of Figures

---

<b>Figure 1: The OCCI API.....</b>	<b>5</b>
<b>Figure 2: Alignment of OCCI URI to IaaS Resources .....</b>	<b>6</b>
<b>Figure 3: The OCCI Lifecycle Model.....</b>	<b>7</b>
<b>Figure 4: The Cloud Storage Reference Model .....</b>	<b>9</b>
<b>Figure 5: CDMI and OCCI in an integrated cloud computing environment.....</b>	<b>10</b>



## Cloud Storage for Cloud Computing

### Introduction

The Cloud has become a new vehicle for delivering resources such as computing and storage to customers on demand. Rather than being a new technology in itself, the cloud is a new business model wrapped around new technologies such as server virtualization that take advantage of economies of scale and multi-tenancy to reduce the cost of using information technology resources.

This paper discusses the business drivers in the Cloud delivery mechanism and business model, what the requirements are in this space, and how standard interfaces, coordinated between different organizations can meet the emerging needs for interoperability and portability of data between clouds.

### Cloud Computing Overview

Recent interest in Cloud Computing has been driven by new offerings of computing resources that are attractive due to per-use pricing and elastic scalability, providing a significant advantage over the typical acquisition and deployment of equipment that was previously required. The effect has been a shift to outsourcing of not only equipment setup, but also the ongoing IT administration of the resources as well.

### From Server Consolidation to Cloud Computing

The needed changes to applications, in order to take advantage of this model, are the same as those required for server consolidation – which had already been taking place for several years prior to the advent of the Cloud. The increased resource utilization and reduction in power and cooling requirements achieved by server consolidation are now being expanded into the cloud.

#### The role of server virtualization software

The new technology underlying this is the **system virtual machine** that allows multiple instances of an operating system and associated applications to run on single physical machine. Delivering this over the network, on demand, is termed **Infrastructure as a Service** (IaaS). The IaaS offerings on the market today allow quick provisioning and deployment of applications and their underlying operating systems onto an infrastructure that expands and contracts as needed to handle the load. Thus the resources that are used can be better matched to the demand on the applications.

#### How is all this managed?

IaaS offerings typically provide an interface that allows the deployment and management of virtual images onto their infrastructure. The lifecycle of these image instances, the amount of resources allocated to these instances and the storage that they use can all be managed through these interfaces. In many cases, this interface is based on REST (short for REpresentational State Transfer) HTTP operations. Without the overhead of many similar protocols the REST approach allows users to easily access their services. Every resource is uniquely addressed using a Uniform Resource Identifier (URI). Based on a set of operations – create, retrieve, update and delete – resources can be managed. Currently three types of resources are considered: storage, network and compute resources. Those resources can be linked together to form a virtual machine with assigned attributes. For example, it is possible to provision a machine that has 2GB of RAM, one hard disk and one network interface.



### Standardizing Cloud Computing Interfaces

Having a programmable interface to the IaaS infrastructure means that you can write client software that uses this interface to manage your use of the Cloud. Many cloud providers have licensed their proprietary APIs freely allowing anyone to implement a similar cloud infrastructure. Despite the accessibility of open APIs, cloud community members have been slow to uniformly adopt any proprietary interface controlled by a single company. The Open Source community has attempted responses, but this has done little to stem the tide of API proliferation. In fact, Open Source projects have increased the tally of interfaces to navigate in a torrent of proprietary APIs.

What is needed instead is a vendor neutral, standard API for cloud computing that all vendors can implement with minimal risk and assured stability. This will allow customers to move their application stacks from one cloud vendor to another, avoiding lock-in and reducing costs.

### Introducing OCCI

The Open Grid Forum™ has created a working group to standardize such an interface. The Open Cloud Computing Interface (OCCI) is a free, open, community consensus driven API, targeting cloud infrastructure services. The API shields IT data centers and cloud partners from the disparities existing between the lineup of proprietary and open cloud APIs.

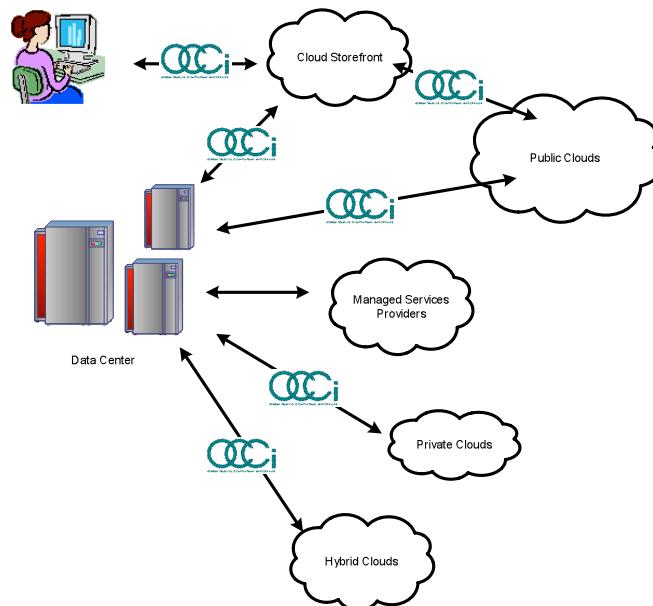


Figure 1: The OCCI API

### The OCCI Reference Architecture

The OCCI has adopted a "Resource Oriented Architecture (ROA)" to represent key components comprising cloud infrastructure services. Each resource (identified by a canonical URI) can have multiple representations that may or may not be hypertext (e.g. HTML). The OCCI working group is



## Cloud Storage for Cloud Computing

planning mappings of the API to several formats. Atom/Pub, JSON and Plain Text are planned for the initial release of the standard. A single URI entry point defines an OCCI interface. Interfaces expose "nouns" which have "attributes" and on which "verbs" can be performed.

Figure 1 shows how the components of an OCCI URI aligns to IaaS Resources:

operation: GET <http://abc.com/compute/uid123foobar>

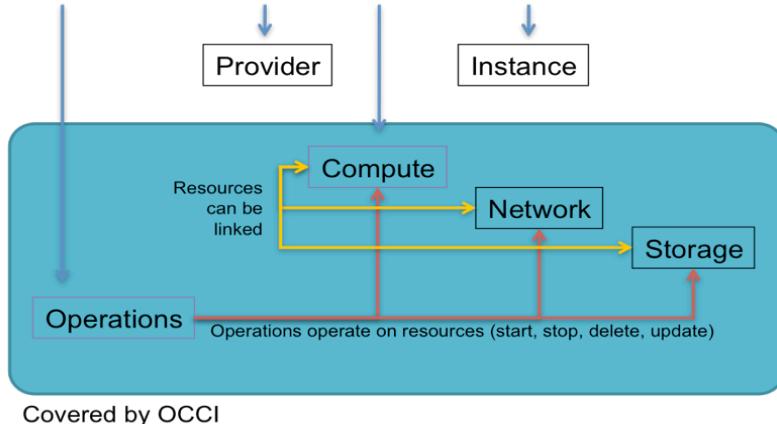


Figure 2: Alignment of OCCI URI to IaaS Resources

Attributes are exposed as key-value pairs and the appropriate verbs as links. The attributes may be described as a URI. Adopting URI support affords the convenience of referencing (linking to) other interfaces including SNIA's Cloud Data Management Interface (CDMI), for example.

The API implements CRUD operations: Create, Retrieve, Update and Delete. Each is mapped to HTTP verbs POST, GET, PUT and DELETE respectively. HEAD and OPTIONS verbs may be used to retrieve metadata and valid operations without the entity body to improve performance. All HTTP functionality can take full advantage of existing internet infrastructure including caches, proxies, gateways and other advanced functionality.

All metadata, including associations between resources is exposed via HTTP headers (e.g. the Link: header). The interface, natively expressed as Atom, executes as close as possible to the underlying Hyper Text Transfer Protocol (HTTP). In one case where the HTTP protocol did not explicitly support Atom collections, an Internet Draft ([draft-johnston-http-category-header-00.txt](https://datatracker.ietf.org/doc/draft-johnston-http-category-header-00.txt)) for a new HTTP header supporting Atom collections, has been submitted by an OCCI working group coordinator to the IETF for standardization.

OCCI provides the capabilities to govern the definition, creation, deployment, operation and retirement of infrastructures services. Using a simplified service lifecycle model, it supports the most common life cycle states offered by cloud providers. In the event providers do not support or report service life cycle states, OCCI does not mandate compliance, defining the life cycle model as only a recommendation. Cloud providers wishing to do so, can comply with the OCCI service life cycle recommendations.

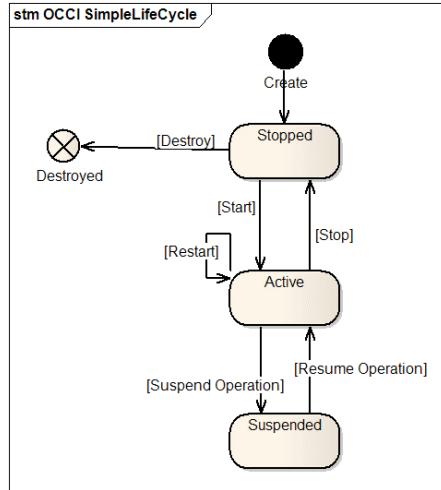


Figure 3: The OCCI Lifecycle Model

With OCCI, cloud computing clients can invoke a new application stack, manage its lifecycle and manage the resources that it uses. The OCCI interface can also be used to assign storage to a virtual machine in order to run the application stack such as that exported by SNIA's CDMI interface. Next we will examine the means of managing that storage and the data in it.

## Cloud Storage Overview

Just like Cloud Computing, Cloud Storage has also been increasing in popularity recently due to many of the same reasons as Cloud Computing. Cloud Storage delivers virtualized storage on demand, over a network based on a request for a given quality of service (QoS). There is no need to purchase storage or in some cases even provision it before storing data. You only pay for the amount of storage your data is actually consuming.

## Some of the Use Cases

Cloud storage is used in many different ways. For example: local data (such as on a laptop) can be backed up to cloud storage; a virtual disk can be “synched” to the cloud and distributed to other computers; and the cloud can be used as an archive to retain (under policy) data for regulatory or other purposes.

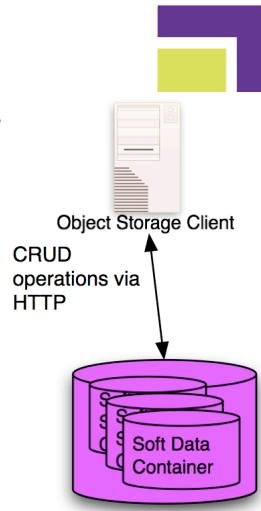
### Web facing applications

For applications that provide data directly to their clients via the network, cloud storage can be used to store that data and the client can be redirected to a location at the cloud storage provider for the data. Media such as audio and video files are an example of this, and the network requirements for streaming data files can be made to scale in order to meet the demand without affecting the application.

The type of interface used for this is just HTTP. Fetching the file can be done from a browser without having to do any special coding, and the correct application is invoked automatically. But how do you get the file there in the first place and how do you make sure the storage you use is of the right type

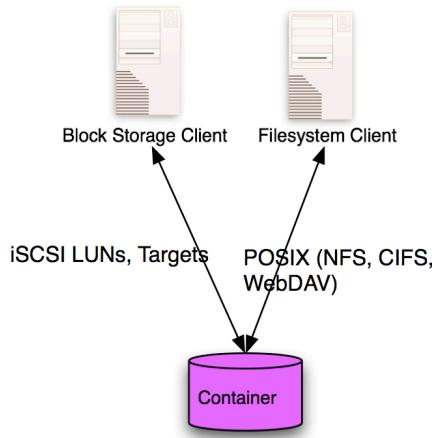
## Cloud Storage for Cloud Computing

and QoS? Again many offerings expose an interface for these operations, and it's not surprising that many of these interfaces use REST principals as well. This is typically a data object interface with operations for creating, reading, updating and deleting the individual data objects via HTTP operations.



### Storage for Cloud Computing

For cloud computing boot images, cloud storage is almost always offered via traditional block and file interfaces such as iSCSI or NFS. These are then



mounted by the virtual machine and attached to a guest for use by cloud computing. Additional drives and filesystems can be similarly provisioned. Of course cloud computing applications can use the data object interface as well, once they are running.

### What makes Cloud Storage different?

The difference between the purchase of a dedicated appliance and that of cloud storage is not the functional interface, but merely the fact that the storage is delivered on demand. The customer pays for either what they actually use or in other

cases, what they have allocated for use. In the case of block storage, a LUN or virtual volume is the granularity of allocation. For file protocols, a filesystem is the unit of granularity. In either case, the actual storage space can be thin provisioned and billed for based on actual usage. Data services such as compression and deduplication can be used to further reduce the actual space consumed.

The management of this storage is typically done out of band of these standard Data Storage interfaces, either through an API, or more commonly, though an administrative browser based user interface. This interface may be used to invoke other data services as well, such as snapshot and cloning.

## Introducing CDMI

The Storage Networking Industry Association™ has created a technical work group to address the need for a cloud storage standard. The new Cloud Data Management Interface (CDMI) is meant to enable interoperable cloud storage and data management. In CDMI, the underlying storage space exposed by the above interfaces is abstracted using the notion of a container. A container is not only a useful abstraction for storage space, but also serves as a grouping of the data stored in it, and a point of control for applying data services in the aggregate.

## Cloud Storage for Cloud Computing



Clients can be in the cloud and providing additional services (computing, data, etc.)

### Clients acting in the role of using a Data Storage Interface

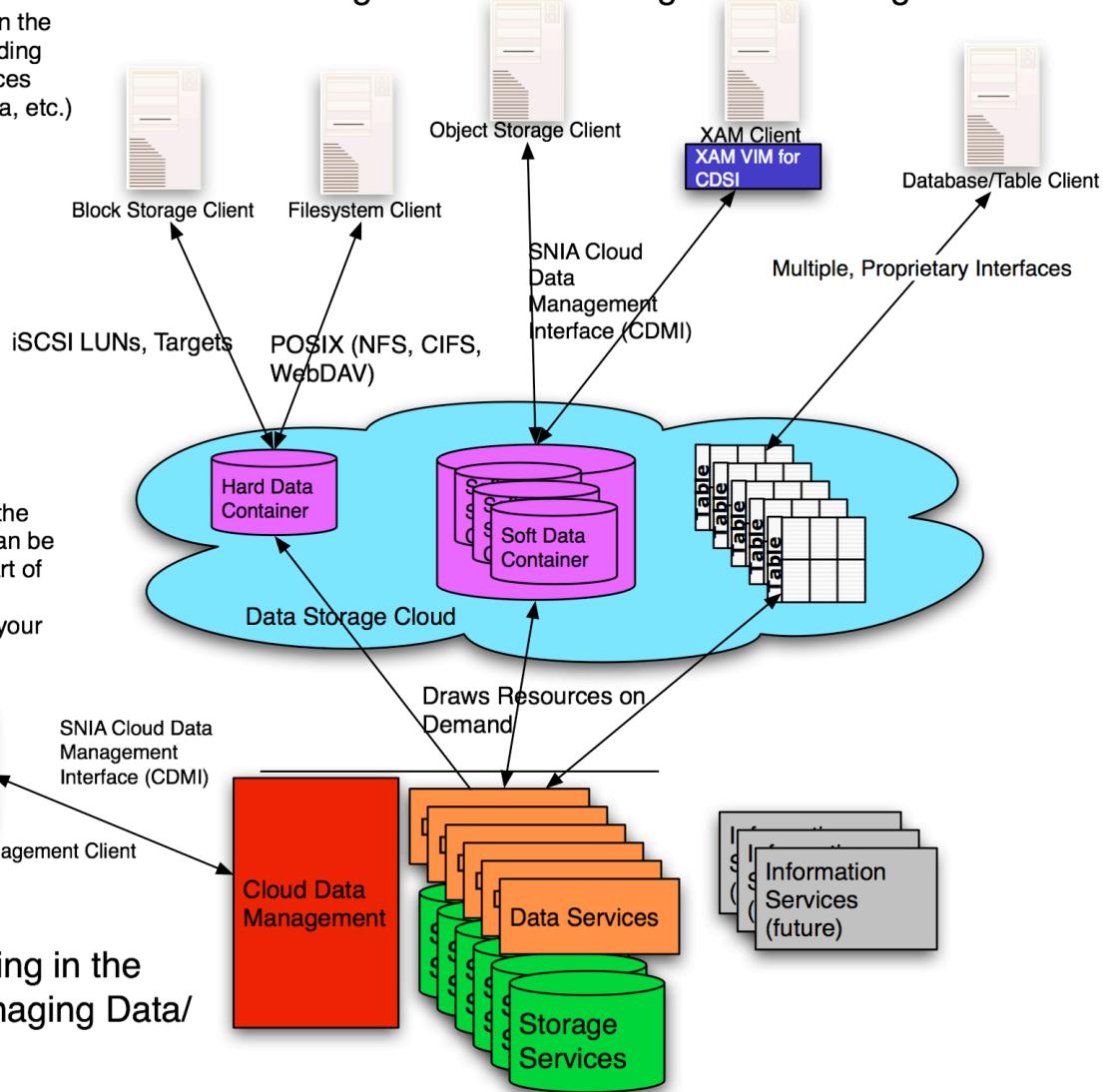


Figure 4: The Cloud Storage Reference Model

CDMI provides not only a data object interface with CRUD semantics; it also can be used to manage containers exported for use by cloud computing infrastructures as shown above in Figure 4.

CDMI for Cloud Computing

With a common cloud computing management infrastructure

## Using CDMI and OCCI for a Cloud Computing Infrastructure

CDMI Containers are accessible not only via CDMI as a data path, but other protocols as well. This is especially useful for using CDMI as the storage interface for a cloud computing environment as shown in Figure 5 below:



## Cloud Storage for Cloud Computing

OCCI  $\leftrightarrow$  CDMI Interface Diagram

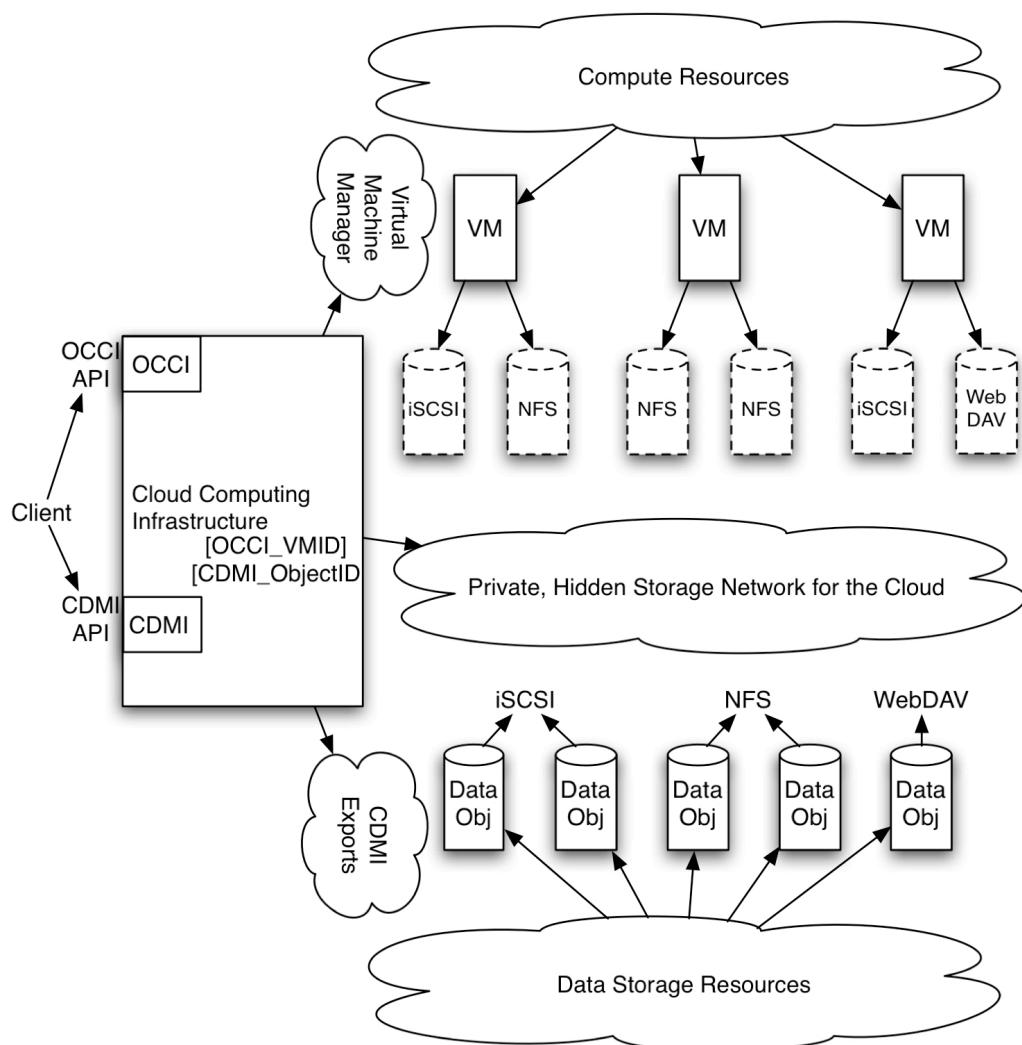


Figure 5: CDMI and OCCI in an integrated cloud computing environment

The exported CDMI containers can be used by the Virtual Machines in the Cloud Computing environment as virtual disks on each guest as shown. With the internal knowledge of the network and the Virtual Machine, the cloud infrastructure management application can attach exported CDMI containers to the Virtual Machines.



## Cloud Storage for Cloud Computing

### How it works

The cloud computing infrastructure management shown above supports both OCCI and CDMI interfaces. To achieve interoperably, CDMI provides a type of export that contains information obtained via the OCCI interface. In addition, OCCI provides a type of storage that corresponds to exported CDMI containers.

OCCI and CDMI can achieve interoperability initiating storage export configurations from either OCCI or CDMI interfaces as starting points. Although the outcome is the same, there are differences between the procedures using CDMI's interface over the OCCI's as a starting point. Below, we present examples of interoperability initiating storage export from both CDMI and OCCI approaches

A client of both interfaces would perform the following operations as an example:

- The Client creates a CDMI Container through the CDMI interface and exports it as an OCCI export type. The CDMI Container ObjectID is returned as a result.
- The Client then creates a Virtual Machine through the OCCI interface and attaches a storage volume of type CDMI using the ObjectID. The OCCI Virtual Machine ID is returned as a result.
- The Client then updates the CDMI Container object export information with the OCCI Virtual Machine ID to allow the Virtual Machine access to the container.
- The Client then starts the Virtual Machine through the OCCI interface.

### Standards Coordination

As can be seen above OCCI and CDMI are standards working towards interoperable cloud computing and cloud storage. The standards are being coordinated through an alliance between the OGF and the SNIA as well as through a cross-SDO cloud standards collaboration group at <http://cloud-standards.org>. OCCI will take advantage of the storage that CDMI has provisioned and configured.

Since both interfaces use similar principles and technologies, it is likely that a single client could manage both the computing and storage needs of an application, scaling both to meet the demands placed on them.

### About the SNIA

The Storage Networking Industry Association (SNIA) is a not-for-profit global organization, made up of some 400 member companies spanning virtually the entire storage industry. SNIA's mission is to lead the storage industry worldwide in developing and promoting standards, technologies, and educational services to empower organizations in the management of information. To this end, the SNIA is uniquely committed to delivering standards, education, and services that will propel open storage networking solutions into the broader market. For additional information, visit the SNIA web site at [www.snia.org](http://www.snia.org).

### About Open Grid Forum:

OGF is the premier world-wide community for the development and adoption of best practices and standards for applied distributed computing technologies. OGF's open forum and process enable communities of users,



## Cloud Storage for Cloud Computing

infrastructure providers, and software developers from around the globe in research, business and government to work together on key issues and promote interoperable solutions.

<http://www.ogf.org>

# **Data-Intensive Supercomputing: The case for DISC**

**Randal E. Bryant**

May 10, 2007  
CMU-CS-07-128

School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

## **Abstract**

Google and its competitors have created a new class of large-scale computer systems to support Internet search. These “Data-Intensive Super Computing” (DISC) systems differ from conventional supercomputers in their focus on data: they acquire and maintain continually changing data sets, in addition to performing large-scale computations over the data. With the massive amounts of data arising from such diverse sources as telescope imagery, medical records, online transaction records, and web pages, DISC systems have the potential to achieve major advances in science, health care, business efficiencies, and information access. DISC opens up many important research topics in system design, resource management, programming models, parallel algorithms, and applications. By engaging the academic research community in these issues, we can more systematically and in a more open forum explore fundamental aspects of a societally important style of computing.

**Keywords:** parallel computing, data storage, web search

*When a teenage boy wants to find information about his idol by using Google with the search query “Britney Spears,” he unleashes the power of several hundred processors operating on a data set of over 200 terabytes. Why then can’t a scientist seeking a cure for cancer invoke large amounts of computation over a terabyte-sized database of DNA microarray data at the click of a button?*

*Recent papers on parallel programming by researchers at Google [13] and Microsoft [19] present the results of using up to 1800 processors to perform computations accessing up to 10 terabytes of data. How can university researchers demonstrate the credibility of their work without having comparable computing facilities available?*

## 1 Background

This document describes an evolving set of ideas about a new form of high-performance computing facility that places emphasis on *data*, rather than raw computation, as the core focus of the system. The system is responsible for the acquisition, updating, sharing, and archiving of the data, and it supports sophisticated forms of computation over its data. We refer to such such system as *Data-Intensive Super Computer* systems, or “DISC.” We believe that DISC systems could yield breakthroughs in a number of scientific disciplines and other problems of societal importance.

Much of our inspiration for DISC comes from the server infrastructures that have been developed to support search over the worldwide web. Google and its competitors have created DISC systems providing very high levels of search quality, response time, and availability. In providing this service, they have created highly profitable businesses, enabling them to build ever larger and more powerful systems. We believe the style of computing that has evolved to support web search can be generalized to encompass a much wider set of applications, and that such systems should be designed and constructed for use by the larger research community.

DISC opens up many important research topics in system design, resource management, programming models, parallel algorithms, and applications. By engaging the academic research community in these issues, we can more systematically and in a more open forum explore fundamental aspects of a societally important style of computing. University faculty members can also incorporate the ideas behind DISC into their courses, ensuring that students will learn material that will be increasingly important to the IT industry.

Others have observed that scientific research increasingly relies on computing over large data sets, sometimes referring to this as “e-Science” [18]. Our claim is that these applications call for a new form of system design, where storage and computation are colocated, and the systems are designed, programmed, and operated to enable users to interactively invoke different forms of computation over large-scale data sets. In addition, our entire world has been increasingly data intensive, as sensor, networking, and storage technology makes it possible to collect and store information ranging from retail transactions to medical imagery. DISC systems will enable us to create new efficiencies and new capabilities well beyond those already achieved by today’s information technology.

This paper outlines the case for DISC as an important direction for large-scale computing systems. It also argues for the need to create university research and educational projects on the design, programming, and applications of such systems. Many people have contributed and helped refine the ideas presented here, as acknowledged in Appendix A.

## 1.1 Motivation

The following applications come from very different fields, but they share in common the central role of data in their computation:

- *Web search without language barriers.* The user can type a query in any (human) language. The engine retrieves relevant documents from across the worldwide web in all languages and translates them into the user’s preferred language. Key to the translation is the creation of sophisticated statistical language models and translation algorithms. The language model must be continuously updated by crawling the web for new and modified documents and recomputing the model. By this means, the translation engine will be updated to track newly created word patterns and idioms, such as “improvised explosive devices.”

Google already demonstrated the value of applying massive amounts of computation to language translation in the 2005 NIST machine translation competition. They won all four categories of the competition in the first year they entered, translating Arabic to English and Chinese to English [1]. Their approach was purely statistical. They trained their program using, among other things, multilingual United Nations documents comprising over 200 billion words, as well as English-language documents comprising over one trillion words. No one in their machine translation group knew either Chinese or Arabic. During the competition, they applied the collective power of 1000 processors to perform the translations.

- *Inferring biological function from genomic sequences.* Increasingly, computational biology involves comparing genomic data from different species and from different organisms of the same species to determine how information is encoded in DNA. Ever larger data sets are being collected as new sequences are discovered, and new forms of derived data are computed. The National Center for Biotechnology Innovation maintains the GenBank database of nucleotide sequences, which has been doubling in size every 10 months. As of August, 2006, it contained over 65 billion nucleotide bases from more than 100,000 distinct organisms.

Although the total volume of data is less than one terabyte, the computations performed are very demanding. In addition, the amount of genetic information available to researchers will increase rapidly once it becomes feasible to sequence the DNA of individual organisms, for example to enable *pharmacogenomics*, predicting a patient’s response to different drugs based on his or her genetic makeup.

- *Predicting and modeling the effects of earthquakes.* Scientists are creating increasingly detailed and accurate finite-element meshes representing the geological properties of the earth’s crust, enabling them to model the effect of a geological disturbance and the probabilities of

earthquakes occurring in different regions of the world [3]. These models are continually updated as actual earthquake data are analyzed and as more sophisticated modeling techniques are devised. The models are an important shared resource among geologists, computational scientists, and civil engineers.

- *Discovering new astronomical phenomena from telescope imagery data.* Massive amounts of imagery data are collected daily, and additional results are derived from computation applied to that data [26]. Providing this information in the form of a shared global database would reduce the redundant storage and computation required to maintain separate copies.
- *Synthesizing realistic graphic animations.* The system stores large amounts of motion capture data and uses this to generate high quality animations [23]. Over time, the motion data can be expanded and refined by capturing more subjects performing more tasks, yielding richer and more realistic animations.
- *Understanding the spatial and temporal patterns of brain behavior based on MRI data.* Information from multiple data sets, measured on different subjects and at different time periods, can be jointly analyzed to better understand how brains function [22]. This data must be updated regularly as new measurements are made.

These and many other tasks have the properties that they involve collecting and maintaining very large data sets and applying vast amounts of computational power to the data. An increasing number of *data-intensive* computational problems are arising as technology for capturing and storing data becomes more widespread and economical, and as the web provides a mechanism to retrieve data from all around the world. Quite importantly, the relevant data sets are not static. They must be updated on a regular basis, and new data derived from computations over the raw information should be updated and saved.

## 2 Data-Intensive Super Computing

We believe that these data-intensive computations call for a new class of machine we term a *data-intensive super computing* system, abbreviated “DISC”. A DISC system is a form of supercomputer, but it incorporates a fundamentally different set of principles than mainstream supercomputers. Current supercomputers are evaluated largely on the number of arithmetic operations they can supply each second to the application programs. This is a useful metric for problems that require large amounts of computation over highly structured data, but it also creates misguided priorities in the way these machines are designed, programmed, and operated. The hardware designers pack as much arithmetic hardware into the systems they can, disregarding the importance of incorporating computation-proximate, fast-access data storage, and at the same time creating machines that are very difficult to program effectively. The programs must be written in terms of very low-level primitives, and the range of computational styles is restricted by the system structure. The systems are operated using a batch-processing style, representing a trade off that favors high utilization of the computing resource over the productivity of the people using the machines.

Below, we enumerate the key principles of DISC and how they differ from conventional supercomputer systems.

1. *Intrinsic, rather than extrinsic data.* Collecting and maintaining data are the duties of the system, rather than for the individual users. The system must retrieve new and updated information over networks and perform derived computations as background tasks. Users can use rich queries based on content and identity to access the data. Reliability mechanisms (e.g., replication, error correction) ensure data integrity and availability as part of the system function. By contrast, current supercomputer centers provide short-term, large-scale storage to their users, high-bandwidth communication to get data to and from the system, and plenty of computing power, but they provide no support for data management. Users must collect and maintain data on their own systems, ship them to the supercomputer for evaluation, and then return the results back for further analysis and updating of the data sets.
2. *High-level programming models for expressing computations over the data.* Current supercomputers must be programmed at a very low level to make maximal use of the resources. Wresting maximum performance from the machine requires hours of tedious optimization. More advanced algorithms, such as ones using sophisticated and irregular data structures, are avoided as being too difficult to implement even though they could greatly improve application performance. With DISC, the application developer is provided with powerful, high-level programming primitives that express natural forms of parallelism and that do not specify a particular machine configuration (e.g., the number of processing elements). It is then the job of the compiler and runtime system to map these computations onto the machine efficiently.
3. *Interactive access.* DISC system users are able to execute programs interactively and with widely varying computation and storage requirements. The system responds to user queries and simple computations on the stored data in less than one second, while more involved computations take longer but do not degrade performance for the queries and simple computations of others. By contrast, existing supercomputers are operated in batch mode to maximize processor utilization. Consequently, users generally maintain separate, smaller cluster systems to do their program development, where greater interaction is required. Inherently interactive tasks, such as data visualization, are not well supported. In order to support interactive computation on a DISC system, there must be some over-provisioning of resources. We believe that the consequent increased cost of computing resources can be justified based on the increased productivity of the system users.
4. *Scalable mechanisms to ensure high reliability and availability.* Current supercomputers provide reliability mainly by periodically checkpointing the state of a program, and then rolling back to the most recent checkpoint when an error occurs. More serious failures require bringing the machine down, running diagnostic tests, replacing failed components, and only then restarting the machine. This is an inefficient, nonscalable mechanism that is not suitable for interactive use. Instead, we believe a DISC system should employ nonstop reliability mechanisms, where all original and intermediate data are stored in redundant forms,

and selective recomputation can be performed in event of component or data failures. Furthermore, the machine should automatically diagnose and disable failed components; these would only be replaced when enough had accumulated to impair system performance. The machine should be available on a 24/7 basis, with hardware and software replacements and upgrades performed on the live system. This would be possible with a more distributed and less monolithic structure than is used with current supercomputers.

## 3 Comparison to Other Large-Scale Computer Systems

There are many different ways large-scale computer systems are organized, but DISC systems have a unique set of characteristics. We compare and contrast some of the other system types.

### 3.1 Current Supercomputers

Although current supercomputers have enabled fundamental scientific breakthroughs in many disciplines, such as chemistry, biology, physics, and astronomy, large portions of the scientific world really have not benefited from the supercomputers available today. For example, most computer scientists make use of relatively impoverished computational resources—ranging from their laptops to clusters of 8–16 processors. We believe that DISC systems could serve a much larger part of the research community, including many scientific areas, as well as other information-intensive disciplines, such as public health, political science, and economics. Moreover, even many of the disciplines that are currently served by supercomputers would benefit from the flexible usage model, the ease of programming, and the managed data aspects of DISC.

### 3.2 Transaction Processing Systems

Large data centers, such as those maintained by financial institutions, airlines, and online retailers, provide another model for large-scale computing systems. We will refer to these as *transaction processing systems*, since the term “data center” is too generic. Like a DISC system, the creation and maintenance of data plays a central role in how a transaction processing system is conceived and organized. High data availability is of paramount importance, and hence the data are often replicated at geographically distributed sites. Precise consistency requirements must be maintained, for example, so that only one person can withdraw the last dollar from a joint bank account. The creation and accessing of data at transaction processing systems occur largely in response to single transactions, each performing limited computation over a small fraction of the total data.

The strong set of reliability constraints placed on transaction systems [15], and the potentially severe consequences of incorrect operation (e.g., if the electronic banking system does not maintain consistent account balances), tend to narrow the range of implementation options and encourage fairly conservative design approaches. Large amounts of research have been done on increasing the amount of concurrency through distribution and replication of the data, but generally very complex

and expensive systems are required to meet the combined goals of high capacity, availability, and reliability.

Search engines have very different characteristics from transaction processing systems that make it easier to get high performance at lower cost. A search engine need not track every change to every document on the web in real time. Users are satisfied if they have access to a reasonably large fraction of the available documents, and that the documents are reasonably up to date. Thus, search can be performed on cached copies of the web, with the updating of the caches performed as a background activity. In addition, the updating of the state is not done in response to individual transactions, but rather by an independent web crawling process. This decoupling between the agents that read data and those that update them makes it much easier to scale the size of the systems at a reasonable cost.

We envision that most DISC systems will have requirements and characteristics more similar to those of search engines than to transaction processing systems. Data sets for scientific applications typically have higher needs for accuracy and consistency than do search engines, but we anticipate they will not be updated as often or by as many agents as is the case for a transaction processing system. In addition, many DISC applications may be able to exploit relaxed consistency constraints similar to those found with web search. For those that extract statistical patterns from massive data sets, their outcomes will be relatively insensitive to small errors or inconsistencies in the data.

On the other hand, in contrast to transaction processing, many DISC operations will invoke large-scale computations over large amounts of data, and so they will require the ability to schedule and coordinate many processors working together on a single computation. All of these differences will lead to very different choices in DISC hardware, programming models, reliability mechanisms, and operating policies than is found in transaction processing systems.

### 3.3 Grid Systems

Many research communities have embraced *grid computing* to enable a sharing of computational resources and to maintain shared data repositories [7]. Although the term “grid” means different things to different people, its primary form is a *computational grid* enabling a number of computers, often distributed geographically and organizationally, to work together on a single computational task. The low-bandwidth connectivity between machines typically limits this style of computation to problems that require little or no communication between the different subtasks. In addition, a *data grid* enables a shared data repository to be distributed across a number of machines, often in combination with a computational grid to operate on this data.

In many ways, our conception for DISC has similar objectives to a combined computational and data grid: we want to provide one or more research communities with an actively managed repository of shared data, along with computational resources that can create and operate on this data. The main difference between DISC systems and grid systems is physical: we believe that a DISC system benefits by locating substantial storage and computational power in a single facility, enabling much faster and much greater movement of data within the system. This makes it possible to support forms of computation that are completely impractical with grid systems. It also enables

the system to be much more aggressive in using different forms of scheduling and load balancing to achieve interactive performance for many users, and to provide higher degrees of reliability and availability. We believe that we can provide more powerful programming models and better economies of scale by taking the more centralized approach to resource location and management represented by DISC.

Alex Szalay and Jim Gray stated in a commentary on 2020 Computing [25]:

“In the future, working with large data sets will typically mean sending computations to the data, rather than copying the data to your workstation.”

Their arguments were based on shear numbers: even the most optimistic predictions of network bandwidth do not allow transferring petabyte data sets from one site to another in a reasonable amount of time. The complexities of managing and computing over such data sets lend further impetus to the need to build systems centered around specific data repositories.

Over time, as multiple organizations set up DISC systems, we could well imagine creating a grid system with DISCs as the nodes. Through different forms of data mirroring and load sharing, we could mitigate the impact of major outages due to power failures, earthquakes, and other disasters.

## 4 Google: A DISC Case Study

We draw much of our inspiration for DISC from the infrastructure that companies have created to support web search. Many credit Inktomi (later acquired by Yahoo) for initiating the trend of constructing specialized, large-scale systems to support web search [9]. Their 300-processor system in 1998 pointed the way to the much larger systems used today. Google has become the most visible exemplar of this approach, and so we focus on their system as a case study. Their system demonstrates how a DISC system can be designed and utilized, although our view of DISC is much broader and envisions a more complex usage model than Google’s. Our ideal system will almost certainly not match all of the characteristics of Google’s.

Google has published a small, but high quality set of papers about their system design [6, 10, 13, 12, 14]. Although these papers set out a number of important concepts in system design, Google is fairly secretive about many specific details of their systems. Some of what is stated below is a bit speculative and may be wrong or out of date. Most likely, both Yahoo and Microsoft have comparable server infrastructure for supporting their search tools, but they are even more secretive than Google.

Google does not disclose the size of their server infrastructure, but reports range from 450,000 [21] to several million [20] processors, spread around at least 25 data centers worldwide. Machines are grouped into clusters of “a few thousand processors,” with disk storage associated with each processor. The system makes use of low-cost, commodity parts to minimize per unit costs, including using processors that favor low power over maximum speed. Standard Ethernet communication links are used to connect the processors. This style of design stands in sharp contrast to the exotic technology found in existing supercomputers, using the fastest possible processors and

specialized, high-performance interconnection networks, consuming large amounts of power and requiring costly cooling systems. Supercomputers have much higher interconnection bandwidth between their processors, which can dramatically improve performance on some applications, but this capability comes at a very high cost.

The Google system actively maintains cached copies of every document it can find on the Internet (around 20 billion), to make it possible to effectively search the entire Internet in response to each query. These copies must be updated on an ongoing basis by having the system *crawl* the web, looking for new or updated documents. In addition to the raw documents, the system constructs complex *index* structures, summarizing information about the documents in forms that enable rapid identification of the documents most relevant to a particular query. When a user submits a query, the front end servers direct the query to one of the clusters, where several hundred processors work together to determine the best matching documents based on the index structures. The system then retrieves the documents from their cached locations, creates brief summaries of the documents, orders them with the most relevant documents first, and determines which sponsored links should be placed on the page.

Processing a single query requires a total of around  $10^{10}$  CPU cycles, not even counting the effort spent for web crawling and index creation. This would require around 10 seconds of computer time on a single machine (or more, when considering the time for disk accesses), but by using multiple processors simultaneously, Google generates a response in around 0.1 seconds [6].

It is interesting to reflect on how the Google server structure gets used, as indicated by our initial observation about the level of resources Google applies in response to often-mundane search queries. Some estimates say that Google earns an average of \$0.05 in advertising revenue for every query to which it responds [17]. It is remarkable that they can maintain such a complex infrastructure and provide that level of service for such a low price. Surely we could make it a national priority to provide the scientific community with equally powerful computational capabilities over large data sets.

The Google hardware design is based on a philosophy of using components that emphasize low cost and low power over raw speed and reliability. They typically stay away from the highest speed parts, because these carry a price premium and consume greater power. In addition, whereas many processors designed for use in servers employ expensive hardware mechanisms to ensure reliability (e.g., the processors in IBM mainframes perform every computation on two separate data paths and compare the results), Google keeps the hardware as simple as possible. Only recently have they added error-correcting logic to their DRAMs. Instead, they make extensive use of redundancy and software-based reliability, following the lead set by Inktomi [9]. Multiple copies of all data are stored, and many computations are performed redundantly. The system continually runs diagnostic programs to identify and isolate faulty components. Periodically, these failed components are removed and replaced without turning the system off. (In the original server, the disk drives were held in with Velcro to facilitate easy replacement.) This software-based reliability makes it possible to provide different levels of reliability for different system functions. For example, there is little harm if the system occasionally fails to respond to a search query, but it must be meticulous about accounting for advertising revenue, and it must ensure high integrity of the index structures.

Google has significantly lower operating costs in terms of power consumption and human labor than do other data centers.

Although much of the software to support web crawling and search is written at a low level, they have implemented a programming abstraction, known as *MapReduce* [13], that supports powerful forms of computation performed in parallel over large amounts of data. The user needs only specify two functions: a *map* function that generates values and associated keys from each document, and a *reduction* function that describes how all the data matching each possible key should be combined. MapReduce can be used to compute statistics about documents, to create the index structures used by the search engine, and to implement their PageRank algorithm for quantifying the relative importance of different web documents. The runtime system implements MapReduce, handling details of scheduling, load balancing, and error recovery [12].

More recently, researchers at Google have devised programming support for distributed data structures they call *BigTable* [10]. Whereas MapReduce is purely a functional notation, generating new files from old ones, BigTable provides capabilities similar to those seen in database systems. Users can record data in tables that are then stored and managed by the system. BigTable does not provide the complete set of operations supported by relational databases, striking a balance between expressive power and the ability to scale for very large databases in a distributed environment.

In summary, we see that the Google infrastructure implements all the features we have enumerated for data-intensive super computing in a system tailored for web search. More recently, they have expanded their range of services to include email and online document creation. These applications have properties more similar to transaction processing than to web search. Google has been able to adapt its systems to support these functions successfully, although it purportedly has been very challenging for them.

## 5 Possible Usage Model

We envision that different research communities will emerge to use DISC systems, each organized around a particular shared data repository. For example, natural language researchers will join together to develop and maintain corpora from a number of different sources and in many different languages, plus derived statistical and structural models, as well as annotations relating the correspondences between phrases in different languages. Other communities might maintain finite-element meshes describing physical phenomena, copies of web documents, etc. These different communities will devise different policies for how data will be collected and maintained, what computations can be performed and how they will be expressed, and how different people will be given different forms of access to the data.

One useful perspective is to think of a DISC system as supporting a powerful form of database. Users can invoke operations on the database that can be simple queries or can require complex computations accessing large amounts of data. Some would be read-only, while others would create or update the stored data. As mentioned earlier, we anticipate that most applications of DISC will not have to provide the strong consistency guarantees found in the database support for

transactions processing.

Unlike traditional databases, which support only limited forms of operations, the DISC operations could include user-specified functions in the style of Google’s MapReduce programming framework. As with databases, different users will be given different authority over what operations can be performed and what modifications can be made.

## 6 Constructing a General-Purpose DISC System

Suppose we wanted to construct a general purpose DISC system that could be made available to the research community for solving data-intensive problems. Such a system could range from modest, say 1000 processors, to massive, say 50,000 processors or more. We have several models for how to build large-scale systems, including current supercomputers, transaction processing systems, and search-engine systems.

Assembling a system that can perform web search could build on standard hardware and a growing body of available software. The open source project *Hadoop* implements capabilities similar to the Google file system and support for MapReduce. Indeed, a near-term project to provide this capability as soon as possible would be worth embarking on, so that the university research community can become more familiar with DISC systems and their applications. Beyond web search, a system that performs web crawling and supports MapReduce would be useful for many applications in natural language processing and machine learning.

Scaling up to a larger and more general purpose machine would require a significant research effort, but we believe the computer science community would embrace such an effort as an exciting research opportunity. Below we list some of the issues to be addressed

- *Hardware Design.* There are a wide range of choices here, from assembling a system out of low-cost commodity parts, à la Google, to using off-the-shelf systems designed for data centers, to using supercomputer-class hardware, with more processing power, memory, and disk storage per processing node, and a much higher bandwidth interconnection network. These choices could greatly affect the system cost, with prices ranging between around \$2,000 to \$10,000 per node. In any case, the hardware building blocks are all available commercially. One fundamental research question is to understand the tradeoffs between the different hardware configurations and how well the system performs on different applications. Google has made a compelling case for sticking with low-end nodes for web search applications, but we need to consider other classes of applications as well. In addition, the Google approach requires much more complex system software to overcome the limited performance and reliability of the components. That might be fine for a company that hires computer science PhDs at the rate Google does, and for which saving a few dollars per node can save the company millions, but it might not be the most cost-effective solution for a smaller operation when personnel costs are considered.
- *Programming Model.* As Google has demonstrated with MapReduce and BigTable, there

should be a small number of program abstractions that enable users to specify their desired computations at a high level, and then the runtime system should provide an efficient and reliable implementation, handling such issues as scheduling, load balancing, and error recovery. Some variations of MapReduce and BigTable would be good starts, but it is likely that multiple such abstractions will be required to support the full range of applications we propose for the system, and for supporting active collection and management of different forms of data.

One important software concept for scaling parallel computing beyond 100 or so processors is to incorporate error detection and recovery into the runtime system and to isolate programmers from both transient and permanent failures as much as possible. Historically, most work on and implementations of parallel programming assumes that the hardware operates without errors. By assuming instead that every computation or information retrieval step can fail to complete or can return incorrect answers, we can devise strategies to correct or recover from errors that allow the system to operate continuously. Work on providing fault tolerance in a manner invisible to the application programmer started in the context of grid-style computing [5], but only with the advent of MapReduce [13] and in recent work by Microsoft [19] has it become recognized as an important capability for parallel systems.

We believe it is important to avoid the tightly synchronized parallel programming notations used for current supercomputers. Supporting these forces the system to use resource management and error recovery mechanisms that would be hard to integrate with the interactive scheduling and flexible error handling schemes we envision. Instead, we want programming models that dynamically adapt to the available resources and that perform well in a more asynchronous execution environment. Parallel programs based on a task queue model [8] do a much better job of adapting to available resources, and they enable error recovery by re-execution of failed tasks. For example, Google’s implementation of MapReduce partitions a computation into a number of map and reduce tasks that are then scheduled dynamically onto a number of “worker” processors. They cite as typical parameters having 200,000 map tasks, 4,000 reduce tasks, and 2,000 workers [12].

- *Resource Management.* A very significant set of issues concern how to manage the computing and storage resources of a DISC system. We want it to be available in an interactive mode and yet able to handle very large-scale computing tasks. In addition, even though it would be feasible to provide multiple petabytes of storage, some scientific applications, such as astronomy, could easily soak up all of this. Different approaches to scheduling processor and storage resources can be considered, with the optimal decisions depending on the programming models and reliability mechanisms to be supported.

As described earlier, we anticipate multiple, distinct research communities to make use of DISC systems, each centered around a particular collection of data. Some aspects of the system hardware and support software will be common among these communities, while others will be more specialized.

- *Supporting Program Development.* Developing parallel programs is notoriously difficult, both in terms of correctness and to get good performance. Some of these challenges can

be reduced by using and supporting high-level programming abstractions, but some issues, especially those affecting performance, affect how application programs should be written.

We must provide software development tools that allow correct programs to be written easily, while also enabling more detailed monitoring, analysis, and optimization of program performance. Most likely, DISC programs should be written to be “self-optimizing,” adapting strategies and parameters according to the available processing, storage, and communications resources, and also depending on the rates and nature of failing components. Hopefully, much of this adaptation can be built into the underlying runtime support, but some assistance may be required from application programmers.

- *System Software.* Besides supporting application programs, system software is required for a variety of tasks, including fault diagnosis and isolation, system resource control, and data migration and replication. Many of these issues are being addressed by the Self-\* systems project at Carnegie Mellon [2], but the detailed solutions will depend greatly on the specifics of the system organization.

Designing and implementing a DISC system requires careful consideration of a number of issues, and a collaboration between a number of disciplines within computer science and computer engineering. We are optimistic that we can form a team of researchers and arrive at a successful system design. After all, Google and its competitors provide an existence proof that DISC systems can be implemented using available technology.

Over the long term, there are many research topics that could be addressed by computer scientists and engineers concerning DISC systems. The set of issues listed previously will all require ongoing research efforts. Some additional topics include:

- *How should the processors be designed for use in cluster machines?* Existing microprocessors were designed to perform well as desktop machines. Some of the design choices, especially the exotic logic used to exploit instruction-level parallelism, may not make the best use of hardware and energy for systems that can make greater use of data parallelism. For example, a study by researchers at Google [6], indicated that their most critical computations did not perform well on existing microprocessors. Perhaps the chip area and power budgets would be better served by integrating many simpler processor cores on a single chip [4].
- *How can we effectively support different scientific communities in their data management and applications?* Clearly, DISC works for web search applications, but we need to explore how far and how well these ideas extend to other data-intensive disciplines.
- *Can we radically reduce the energy requirements for large-scale systems?* The power needs of current systems are so high that Google has set up a major new facility in Oregon, while Microsoft and Yahoo are building ones in Eastern Washington, to be located near inexpensive hydroelectric power [21]. Would a combination of better hardware design and better resource management enable us to reduce the required power by a factor of 10 or more?

- *How do we build large-scale computing systems with an appropriate balance of performance and cost?* The IT industry has demonstrated that they can build and operate very large and complex data centers, but these systems are very expensive to build (both machines and infrastructure) and operate (both personnel and energy). We need to create a framework by which system designers can rigorously evaluate different design alternatives in terms of their reliability, cost, and ability to support the desired forms of computation.
- *How can very large systems be constructed given the realities of component failures and repair times?* Measurements indicate that somewhere between 4% and 7% of the disks in a data center must be replaced each year [16, 24]. In a system with 50,000 disks, that means that disks will be failing every few hours. Even once a new unit is installed, it can take multiple hours to reconstruct its contents from the redundant copies on other disks, and so the system will always be involved in data recovery activities. Furthermore, we run the risk that all copies of some data item could be lost or corrupted due to multiple component failures. Creating reliable systems of such scale will require careful analysis of failure modes and frequencies, and devising a number of strategies for mitigating the effects of failures. We will require ways to assess the levels of criticality of different parts of the data sets in order to apply differential replication and recovery strategies.
- *Can we support a mix of computationally intensive jobs with ones requiring interactive response?* In describing our ideas to users of current supercomputers, this possibility has proved to be the one they find the most intriguing. It requires new ways of structuring and programming systems, and new ways to schedule their resources.
- *How do we control access to the system while enabling sharing?* Our system will provide a repository of data that is shared by many users. We cannot implement security by simply imposing complete isolation between users. In addition, we want more sophisticated forms of access control than simply whether a user can read or write some part of the data, since improper updating of the data could impede the efforts of other users sharing the data. We must guard against both accidental and malicious corruption.
- *Can we deal with bad or unavailable data in a systematic way?* When operating on very large data sets distributed over many disk drives, it is inevitable that some of the data will be corrupted or will be unavailable in a timely manner. Many applications can tolerate small amounts of data loss, and so they should simply skip over corrupted records, as is done in Google’s implementation of MapReduce [13], and they should be allowed to proceed when enough data have been retrieved. Providing the right set of mechanisms to allow the application programmer to implement such strategies while maintaining acceptable accuracy requires ways to quantify acceptable data loss and clever design of the application-program interface.
- *Can high performance systems be built from heterogenous components?* Traditionally, most high-performance systems have been built using identical processors, disks, etc., in order to simplify issues of scheduling, control, and maintenance. Such homogeneity is required

to support the tightly synchronized parallel programming models used in these systems, but would not be required for a more loosely coupled task queue model. Allowing heterogeneous components would enable incremental upgrading of the system, adding or replacing a fraction of the processors or disks at a time.

Although the major search engine companies are examining many of these issues with their own systems, it is important that the university research community gets involved. First, there are many important disciplines beyond web search and related services that can benefit from the DISC principles. Second, academic researchers are uniquely equipped to bring these issues into a public forum where they can be systematically and critically evaluated by scholars from around the world. Companies are too driven by deadlines and too wary of protecting their proprietary advantages to serve this role.

In addition to being able to contribute to the progress of DISC, academics need to engage in this area to guarantee their future relevance. It is important that our students learn about the systems they will encounter in their careers, and that our research work addresses problems of real importance to the IT industry. There is a large and growing gap between the scale of systems found in academia compared to those of the numerous data centers worldwide supporting web search, electronic commerce, and business processes. Although some universities have large-scale systems in the form of supercomputers, only a small subset of academic computer scientists are involved in high performance computing, and these systems have very different characteristics from commercial data centers. Bringing DISC projects into university environments would provide new opportunities for research and education that would have direct relevance to the current and future IT industry.

## 7 Turning Ideas into Reality

We are convinced that DISC provides an important area for university-based research and education. It could easily spawn projects at multiple institutions and involve researchers in computer engineering, computer science, computational science, and other disciplines that could benefit from the availability of DISC systems.

How then should we proceed? There are many possible paths to follow, involving efforts of widely varying scale and scope. Choosing among these will depend on the availability of research funding, how many institutions will be involved, and how collaborative their efforts will be. Rather than pin down a specific plan, we simply describe possible options here.

One factor is certain in our planning—there are many researchers who are eager to get involved. I have spoken with researchers in a number of companies and universities, and there is a clear consensus that there are ample opportunities for exciting work ranging across the entire spectrum of computing research disciplines.

## 7.1 Developing a Prototype System

One approach is to start by constructing a prototype system of around 1000 processing nodes. Such a system would be large enough to demonstrate the performance potential of DISC and to encounter some of the challenges in resource management and error handling. For example, if we provision each node with at least one terabyte of storage (terabyte disks will be available within the next year or so), the system would have a storage capacity of over one petabyte. This would easily provide enough storage to hold replicated copies of every document available over the worldwide web. By having two dual-core processors in each node, the resulting machine would have 4,000 total processor cores.

In order to support both system and application researchers simultaneously, we propose constructing a system that can be divided into multiple partitions, where the different partitions could operate independently without any physical reconfiguration of the machines or interconnections.

Typically, we would operate two types of partitions: some for application development, focusing on gaining experience with the different programming techniques, and others for systems research, studying fundamental issues in system design. This multi-partition strategy would resolve the age-old dilemma of how to get systems and applications researchers working together on a project. Application developers want a stable and reliable machine, but systems researchers keep changing things.

For the program development partitions, we would initially use available software, such as the open source code from the Hadoop project, to implement the file system and support for application programming.

For the systems research partitions, we would create our own design, studying the different layers of hardware and system software required to get high performance and reliability. As mentioned earlier, there is a range of choices in the processor, storage, and interconnection network design that greatly affects the system cost. During the prototyping phases of the project, we propose using relatively high-end hardware—we can easily throttle back component performance to study the capabilities of lesser hardware, but it is hard to conduct experiments in the reverse direction. As we gain more experience and understanding of tradeoffs between hardware performance and cost, and as we develop better system software for load balancing and error handling, we may find that we can build systems using lower-cost components.

Over time, we would migrate the software being developed as part of the systems research to the partitions supporting applications programming. In pursuing this evolutionary approach, we must decide what forms of compatibility we would seek to maintain. Our current thinking is that any compatibility should only be provided at a very high level, such that an application written in terms of MapReduce and other high-level constructs can continue to operate with minimal modifications, but complete compatibility is not guaranteed. Otherwise, our systems researchers would be overly constrained to follow nearly the exact same paths set by existing projects.

We can estimate the hardware cost of a prototype machine based on per-node costs. Our current thinking is that it would be best to use powerful nodes, each consisting of a high-end rack-mounted server, with one or two multicore processors, several terabytes of disk, and one or more high-

performance communications interfaces. Going toward the high end would create a more general prototyping facility. We estimate such nodes, plus the cost of the high performance communication network, would cost around \$10,000, yielding a hardware cost of around \$10 million for the machine.

In addition to the cost of procuring hardware, we would incur costs for personnel, infrastructure, and energy. Since one goal is to develop and maintain software of sufficiently high quality to provide a reliable computing platform, we must plan for a large staff (i.e., not just graduate students and faculty), including software architects, programmers, project managers, and system operators. The exact costs depend to a large extent on issues such as where the work is performed, where the system is located, and how many researchers get involved.

## 7.2 Jump Starting

Instead of waiting until hardware can be funded, procured, and installed, we could begin application development by renting much of the required computing infrastructure. Recently, Amazon has begun marketing network-accessible storage, via its Simple Storage System (S3) service, and computing cycles via its Elastic Computing Cloud (EC2) service [11]. The current pricing for storage is \$0.15 per gigabyte per day (\$1,000 per terabyte per year), with addition costs for reading or writing the data. Computing cycles cost \$0.10 per CPU hour (\$877 per year) on a virtual Linux machine. As an example, it would be possible to have 100 processors running continuously, maintaining a 50 TB data set, updated at a rate of 1 TB per day at a cost of \$214,185 per year. That, for example, would be enough to collect, maintain, and perform computations over a substantial fraction of the available web documents.

We view this rental approach as a stopgap measure, not as a long-term solution. For one thing, the performance of such a configuration is likely to be much less than what could be achieved by a dedicated facility. There is no way to ensure that the S3 data and the EC2 processors will be in close enough proximity to provide high speed access. In addition, we would lose the opportunity to design, evaluate, and refine our own system. Nevertheless, the view this capability as an intriguing direction for computing services and a way to move forward quickly.

## 7.3 Scaling Up

An important goal in building a 1000-node prototype would be to determine the feasibility and study the issues in constructing a much larger system, say 10,000 to 50,000 nodes. Scaling up to such a large system only makes sense if we can clearly demonstrate the ability of such a system to solve problems having high societal importance, and to do so more effectively than would be possible with other approaches. We would also need to understand whether it is best to create a small number of very large machines, a larger number of more modest machines, or some combination of the two. Having the 1000-node prototype would enable us to study these issues and make projections on the scalability and performance of our designs.

## 8 Conclusion

As we have described the ideas behind DISC to other researchers, we have found great enthusiasm among both potential system users and system developers. We believe it is time for the computer science community to step up their level of thinking about the power of data-intensive computing and the scientific advances it can produce. Just as web search has become an essential tool in the lives of people ranging from schoolchildren to academic researchers to senior citizens, we believe that DISC systems could change the face of scientific research worldwide.

We are also confident that any work in this area would have great impact on the many industries that benefit from more powerful and more capable information technology. In domains ranging from retail services to health care delivery, vast amounts of data are being collected and analyzed. Information can be extracted from these data that makes companies better serve their customers while running more efficiently and that detects long term health trends in different populations. The combination of sensors and networks to collect data, inexpensive disks to store data, and the benefits derived by analyzing data causes our society to be increasingly data intensive. DISC will help realize the potential all these data provides.

Universities cannot come close to matching the capital investments that industry is willing to undertake. In 2006, Microsoft announced plans to invest \$2 billion in server infrastructure for 2007 [21], and Google \$1.5 billion. It should be realized, though, that these companies are trying to serve the needs of millions of users, while we are only supporting hundreds or thousands. In addition, the federal government spends billions of dollars per year for high-performance computing. Over the long term, we have the opportunity to help that money be invested in systems that better serve the needs of their users and our society. Thus, getting involved in DISC is within the budgetary reach of academic computer scientists.

In this research area, universities are in the unusual position of following a lead set by industry, rather than the more normal reverse situation. Google and its competitors have demonstrated a new style of computing, and it is important for universities to adopt and build on these ideas. We have the ability to develop and evaluate ideas systematically and without proprietary constraints. We can apply these ideas to domains that are unlikely to produce any commercial value in the near term, while also generating technology that has long-term economic impact. We also have a duty to train students to be at the forefront of computer technology, a task we can only do by first moving to the frontier of computer systems ourselves.

## References

- [1] Google tops translation ranking. *News@Nature*, Nov. 6, 2006.
- [2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Early experiences on the journey towards self-\* storage. *IEEE Data Eng. Bulletin*, 29(3):55–62, 2006.

- [3] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. R. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of SC2003*, November 2003.
- [4] K. Asanovic, R. Bodik, B. C. Catanzo, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, U. C., Berkeley, December 18 2006.
- [5] Ö. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli, and L. A. Giachini. Paralex: an environment for parallel programming in distributed systems. In *6th ACM International Conference on Supercomputing*, pages 178–187, 1992.
- [6] L. A. Barroso, J. Dean, and U. Hölze. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.
- [7] F. Berman, G. Fox, and T. Hey. The Grid: Past, present, and future. In F. Berman, G. C. Fix, and A. J. G. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 9–50. Wiley, 2003.
- [8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995.
- [9] E. A. Brewer. Delivering high availability for Inktomi search engines. In L. M. Haas and A. Tiwary, editors, *ACM SIGMOD International Conference on Management of Data*, page 538. ACM, 1998.
- [10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A distributed storage system for structured data. In *Operating Systems Design and Implementation*, 2006.
- [11] T. Claburn. In Web 2.0 keynote, Jeff Bezos touts Amazon's on-demand services. *Information Week*, Apr. 17, 2007.
- [12] J. Dean. Experiences with MapReduce, an abstraction for large-scale computation. In *International Conference on Parallel Architecture and Compilation Techniques*. ACM, 2006.
- [13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, 2004.
- [14] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Symposium on Operating Systems Principles*, pages 29–43. ACM, 2003.
- [15] J. Gray. The transaction concept: Virtues and limitations. In *Very Large Database Conference*, pages 144–154, 1981.

- [16] J. Gray and C. van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft Research, 2005.
- [17] M. Helft. A long-delayed ad system has Yahoo crossing its fingers. *New York Times*, Feb. 5, 2007.
- [18] T. Hey and A. Trefethen. The data deluge: an e-Science perspective. In F. Berman, G. C. Fix, and A. J. G. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 809–824. Wiley, 2003.
- [19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys 2007*, March 2007.
- [20] J. Markoff. Sun and IBM offer new class of high-end servers. *New York Times*, Apr. 26, 2007.
- [21] J. Markoff and S. Hansell. Hiding in plain sight, Google seeks more power. *New York Times*, June 14, 2006.
- [22] T. M. Mitchell, R. Hutchinson, R. S. Niculescu, F. Pereira, X. Wang, M. Just, and S. Newman. Learning to decode cognitive states from brain images. *Machine Learning*, 57(1–2):145–175, October 2004.
- [23] L. Ren, G. Shakhnarovich, J. K. Hodgins, H. Pfister, and P. Viola. Learning silhouette features for control of human motion. *ACM Transactions on Graphics*, 24(4), October 2005.
- [24] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST’07: Fifth USENIX Conference on File and Storage Technologies*, 2007.
- [25] A. Szalay and J. Gray. Science in an exponential world. *Nature*, 440, March 23 2006.
- [26] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: The Sloan Digital Sky Survey. In *SIGMOD International Conference on Management of Data*, pages 451–462. ACM, 2000.

## A Acknowledgments

A number of people have contributed their ideas and helped refine my ideas about DISC. The following is a partial list of people who have been especially helpful.

- Carnegie Mellon University
  - Guy Blelloch, for insights on parallel programming
  - Jamie Callan and Jaime Carbonell, for explaining the requirements and opportunities for language translation and information retrieval.

- Greg Ganger and Garth Gibson, for all aspects of DISC system design and operation
  - Peter Lee, Todd Mowry, and Jeannette Wing, for high-level guidance and support
  - Tom Mitchell, for insights into the needs and capabilities of machine learning
  - David O’Hallaron, for devising the domain-specific database perspective of DISC operation
  - Anthony Tomasic, for information on current commercial data centers.
- Other universities
  - David Patterson (Berkeley), Ed Lazowska (Washington) for insights into the role of university research in this area.
  - Tom Andersen (Washington), regarding how this project relates to the proposed GENI project.
- Google
  - Urs Hözle, for convincing us that it would be feasible for universities to build systems capable of supporting web crawling and search.
  - Andrew Moore, for advice on the resources required to create and operate a DISC system
- Intel
  - George Cox, for describing how important cluster computing has become to Intel.
- Microsoft
  - Sailesh Chutani, for ideas on some major challenges facing large-scale system design, such as the high power requirements.
  - Tony Hey, for a perspective on scientific computing
  - Roy Levin, for a historical perspective and for calling for a more precise characterization of the nature of problems suitable for DISC.
  - Rick Rashid, for a “reality check” on the challenges of creating and maintaining large-scale computing facilities
- Sun
  - Jim Waldo for encouraging us to think about heterogenous systems.
- Elsewhere
  - Bwolen Yang, for an appreciation of the need to create systems that are resilient to failures.
  - Tom Jordan (Southern California Earthquake Center), for insights on operating a multi-institutional, data and computation-intensive project.

# MapReduce Online

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein  
*UC Berkeley*

Khaled Elmeleegy, Russell Sears  
*Yahoo! Research*

## Abstract

MapReduce is a popular framework for data-intensive distributed computing of batch jobs. To simplify fault tolerance, many implementations of MapReduce *materialize* the entire output of each map and reduce task before it can be consumed. In this paper, we propose a modified MapReduce architecture that allows data to be *pipelined* between operators. This extends the MapReduce programming model beyond batch processing, and can reduce completion times and improve system utilization for batch jobs as well. We present a modified version of the Hadoop MapReduce framework that supports *online aggregation*, which allows users to see “early returns” from a job as it is being computed. Our Hadoop Online Prototype (*HOP*) also supports *continuous queries*, which enable MapReduce programs to be written for applications such as event monitoring and stream processing. *HOP* retains the fault tolerance properties of Hadoop and can run unmodified user-defined MapReduce programs.

## 1 Introduction

MapReduce has emerged as a popular way to harness the power of large clusters of computers. MapReduce allows programmers to think in a *data-centric* fashion: they focus on applying transformations to sets of data records, and allow the details of distributed execution, network communication and fault tolerance to be handled by the MapReduce framework.

MapReduce is typically applied to large batch-oriented computations that are concerned primarily with time to job completion. The Google MapReduce framework [6] and open-source Hadoop system reinforce this usage model through a batch-processing implementation strategy: the entire output of each map and reduce task is *materialized* to a local file before it can be consumed by the next stage. Materialization allows for a simple and elegant checkpoint/restart fault tolerance mechanism

that is critical in large deployments, which have a high probability of slowdowns or failures at worker nodes.

We propose a modified MapReduce architecture in which intermediate data is *pipelined* between operators, while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. To validate this design, we developed the Hadoop Online Prototype (*HOP*), a pipelining version of Hadoop.<sup>1</sup>

Pipelining provides several important advantages to a MapReduce framework, but also raises new design challenges. We highlight the potential benefits first:

- Since reducers begin processing data as soon as it is produced by mappers, they can generate and refine an approximation of their final answer during the course of execution. This technique, known as *online aggregation* [12], can provide initial estimates of results several orders of magnitude faster than the final results. We describe how we adapted online aggregation to our pipelined MapReduce architecture in Section 4.
- Pipelining widens the domain of problems to which MapReduce can be applied. In Section 5, we show how *HOP* can be used to support *continuous queries*: MapReduce jobs that run continuously, accepting new data as it arrives and analyzing it immediately. This allows MapReduce to be used for applications such as event monitoring and stream processing.
- Pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization, and reduce response time. A thorough performance study is a topic for future work; however, in Section 6 we present some initial performance results which demonstrate that pipelining can reduce job completion times by up to 25% in some scenarios.

<sup>1</sup>The source code for *HOP* can be downloaded from <http://code.google.com/p/hop/>

Pipelining raises several design challenges. First, Google’s attractively simple MapReduce fault tolerance mechanism is predicated on the materialization of intermediate state. In Section 3.3, we show that this can co-exist with pipelining, by allowing producers to periodically ship data to consumers in parallel with their materialization. A second challenge arises from the greedy communication implicit in pipelines, which is at odds with batch-oriented optimizations supported by “combiners”: map-side code that reduces network utilization by performing pre-aggregation before communication. We discuss how the HOP design addresses this issue in Section 3.1. Finally, pipelining requires that producers and consumers are co-scheduled intelligently; we discuss our initial work on this issue in Section 3.4.

## 1.1 Structure of the Paper

In order to ground our discussion, we present an overview of the Hadoop MapReduce architecture in Section 2. We then develop the design of HOP’s pipelining scheme in Section 3, keeping the focus on traditional batch processing tasks. In Section 4 we show how HOP can support online aggregation for long-running jobs and illustrate the potential benefits of that interface for MapReduce tasks. In Section 5 we describe our support for continuous MapReduce jobs over data streams and demonstrate an example of near-real-time cluster monitoring. We present initial performance results in Section 6. Related and future work are covered in Sections 7 and 8.

## 2 Background

In this section, we review the MapReduce programming model and describe the salient features of Hadoop, a popular open-source implementation of MapReduce.

### 2.1 Programming Model

To use MapReduce, the programmer expresses their desired computation as a series of *jobs*. The input to a job is an input specification that will yield key-value pairs. Each job consists of two stages: first, a user-defined *map* function is applied to each input record to produce a list of intermediate key-value pairs. Second, a user-defined *reduce* function is called once for each distinct key in the map output and passed the list of intermediate values associated with that key. The MapReduce framework automatically parallelizes the execution of these functions and ensures fault tolerance.

Optionally, the user can supply a *combiner* function [6]. Combiners are similar to reduce functions, except that they are not passed *all* the values for a given key: instead, a combiner emits an output value that summarizes the

```
public interface Mapper<K1, V1, K2, V2> {
    void map(K1 key, V1 value,
             OutputCollector<K2, V2> output);
    void close();
}
```

Figure 1: Map function interface.

input values it was passed. Combiners are typically used to perform map-side “pre-aggregation,” which reduces the amount of network traffic required between the map and reduce steps.

## 2.2 Hadoop Architecture

Hadoop is composed of *Hadoop MapReduce*, an implementation of MapReduce designed for large clusters, and the *Hadoop Distributed File System* (HDFS), a file system optimized for batch-oriented workloads such as MapReduce. In most Hadoop jobs, HDFS is used to store both the input to the map step and the output of the reduce step. Note that HDFS is *not* used to store intermediate results (e.g., the output of the map step): these are kept on each node’s local file system.

A Hadoop installation consists of a single master node and many worker nodes. The master, called the *JobTracker*, is responsible for accepting jobs from clients, dividing those jobs into *tasks*, and assigning those tasks to be executed by worker nodes. Each worker runs a *TaskTracker* process that manages the execution of the tasks currently assigned to that node. Each TaskTracker has a fixed number of slots for executing tasks (two maps and two reduces by default).

## 2.3 Map Task Execution

Each map task is assigned a portion of the input file called a *split*. By default, a split contains a single HDFS block (64MB by default), so the total number of file blocks determines the number of map tasks.

The execution of a map task is divided into two phases.

1. The *map* phase reads the task’s split from HDFS, parses it into records (key/value pairs), and applies the map function to each record.
2. After the map function has been applied to each input record, the *commit* phase registers the final output with the TaskTracker, which then informs the JobTracker that the task has finished executing.

Figure 1 contains the interface that must be implemented by user-defined map functions. After the *map* function has been applied to each record in the split, the *close* method is invoked.

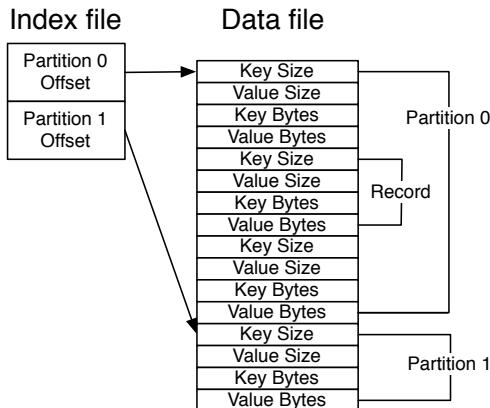


Figure 2: Map task index and data file format (2 partition/reduce case).

The third argument to the *map* method specifies an *OutputCollector* instance, which accumulates the output records produced by the map function. The output of the map step is consumed by the reduce step, so the *OutputCollector* stores map output in a format that is easy for reduce tasks to consume. Intermediate keys are assigned to reducers by applying a partitioning function, so the *OutputCollector* applies that function to each key produced by the map function, and stores each record and partition number in an in-memory buffer. The *OutputCollector* spills this buffer to disk when it reaches capacity.

A spill of the in-memory buffer involves first sorting the records in the buffer by partition number and then by key. The buffer content is written to the local file system as an index file and a data file (Figure 2). The index file points to the offset of each partition in the data file. The data file contains only the records, which are sorted by the key within each partition segment.

During the *commit* phase, the final output of the map task is generated by merging all the spill files produced by this task into a single pair of data and index files. These files are registered with the TaskTracker before the task completes. The TaskTracker will read these files when servicing requests from reduce tasks.

## 2.4 Reduce Task Execution

The execution of a reduce task is divided into three phases.

1. The *shuffle* phase fetches the reduce task's input data. Each reduce task is assigned a partition of the key range produced by the map step, so the reduce task must fetch the content of this partition from every map task's output.
2. The *sort* phase groups records with the same key together.

```
public interface Reducer<K2, V2, K3, V3> {
    void reduce(K2 key, Iterator<V2> values,
               OutputCollector<K3, V3> output);

    void close();
}
```

Figure 3: Reduce function interface.

3. The *reduce* phase applies the user-defined reduce function to each key and corresponding list of values.

In the *shuffle* phase, a reduce task fetches data from each map task by issuing HTTP requests to a configurable number of TaskTrackers at once (5 by default). The JobTracker relays the location of every TaskTracker that hosts map output to every TaskTracker that is executing a reduce task. Note that a reduce task cannot fetch the output of a map task until the map has finished executing and committed its final output to disk.

After receiving its partition from all map outputs, the reduce task enters the *sort* phase. The map output for each partition is already sorted by the reduce key. The reduce task merges these runs together to produce a single run that is sorted by key. The task then enters the *reduce* phase, in which it invokes the user-defined reduce function for each distinct key in sorted order, passing it the associated list of values. The output of the reduce function is written to a temporary location on HDFS. After the reduce function has been applied to each key in the reduce task's partition, the task's HDFS output file is atomically renamed from its temporary location to its final location.

In this design, the output of both map and reduce tasks is written to disk before it can be consumed. This is particularly expensive for reduce tasks, because their output is written to HDFS. Output materialization simplifies fault tolerance, because it reduces the amount of state that must be restored to consistency after a node failure. If any task (either map or reduce) fails, the JobTracker simply schedules a new task to perform the same work as the failed task. Since a task never exports any data other than its final answer, no further recovery steps are needed.

## 3 Pipelined MapReduce

In this section we discuss our extensions to Hadoop to support pipelining between tasks (Section 3.1) and between jobs (Section 3.2). We describe how our design supports fault tolerance (Section 3.3), and discuss the interaction between pipelining and task scheduling (Section 3.4). Our focus here is on batch-processing workloads; we discuss online aggregation and continuous queries in Section 4 and Section 5. We defer performance results to Section 6.

### 3.1 Pipelining Within A Job

As described in Section 2.4, reduce tasks traditionally issue HTTP requests to *pull* their output from each TaskTracker. This means that map task execution is completely decoupled from reduce task execution. To support pipelining, we modified the map task to instead *push* data to reducers as it is produced. To give an intuition for how this works, we begin by describing a straightforward pipelining design, and then discuss the changes we had to make to achieve good performance.

#### 3.1.1 Naïve Pipelining

In our naïve implementation, we modified Hadoop to send data directly from map to reduce tasks. When a client submits a new job to Hadoop, the JobTracker assigns the map and reduce tasks associated with the job to the available TaskTracker slots. For purposes of discussion, we assume that there are enough free slots to assign all the tasks for each job. We modified Hadoop so that each reduce task contacts every map task upon initiation of the job, and opens a TCP socket which will be used to pipeline the output of the map function. As each map output record is produced, the mapper determines which partition (reduce task) the record should be sent to, and immediately sends it via the appropriate socket.

A reduce task accepts the pipelined data it receives from each map task and stores it in an in-memory buffer, spilling sorted runs of the buffer to disk as needed. Once the reduce task learns that every map task has completed, it performs a final merge of all the sorted runs and applies the user-defined reduce function as normal.

#### 3.1.2 Refinements

While the algorithm described above is straightforward, it suffers from several practical problems. First, it is possible that there will not be enough slots available to schedule every task in a new job. Opening a socket between every map and reduce task also requires a large number of TCP connections. A simple tweak to the naïve design solves both problems: if a reduce task has not yet been scheduled, any map tasks that produce records for that partition simply write them to disk. Once the reduce task is assigned a slot, it can then pull the records from the map task, as in regular Hadoop. To reduce the number of concurrent TCP connections, each reducer can be configured to pipeline data from a bounded number of mappers at once; the reducer will pull data from the remaining map tasks in the traditional Hadoop manner.

Our initial pipelining implementation suffered from a second problem: the map function was invoked by the same thread that wrote output records to the pipeline sockets. This meant that if a network I/O operation blocked

(e.g., because the reducer was over-utilized), the mapper was prevented from doing useful work. Pipeline stalls should not prevent a map task from making progress—especially since, once a task has completed, it frees a TaskTracker slot to be used for other purposes. We solved this problem by running the map function in a separate thread that stores its output in an in-memory buffer, and then having another thread periodically send the contents of the buffer to the connected reducers.

#### 3.1.3 Granularity of Map Output

Another problem with the naïve design is that it eagerly sends each record as soon as it is produced, which prevents the use of map-side combiners. Imagine a job where the reduce key has few distinct values (e.g., gender), and the reduce applies an aggregate function (e.g., count). As discussed in Section 2.1, combiners allow map-side “pre-aggregation”: by applying a reduce-like function to each distinct key at the mapper, network traffic can often be substantially reduced. Eagerly pipelining each record as it is produced prevents the use of map-side combiners.

A related problem is that eager pipelining moves some of the sorting work from the mapper to the reducer. Recall that in the blocking architecture, map tasks generate sorted spill files: all the reduce task must do is merge together the pre-sorted map output for each partition. In the naïve pipelining design, map tasks send output records in the order in which they are generated, so the reducer must perform a full external sort. Because the number of map tasks typically far exceeds the number of reduces [6], moving more work to the reducer increased response time in our experiments.

We addressed these issues by modifying the in-memory buffer design described in Section 3.1.2. Instead of sending the buffer contents to reducers directly, we wait for the buffer to grow to a threshold size. The mapper then applies the combiner function, sorts the output by partition and reduce key, and writes the buffer to disk using the spill file format described in Section 2.3.

Next, we arranged for the TaskTracker at each node to handle pipelining data to reduce tasks. Map tasks register spill files with the TaskTracker via RPCs. If the reducers are able to keep up with the production of map outputs and the network is not a bottleneck, a spill file will be sent to a reducer soon after it has been produced (in which case, the spill file is likely still resident in the map machine’s kernel buffer cache). However, if a reducer begins to fall behind, the number of unsent spill files will grow.

When a map task generates a new spill file, it first queries the TaskTracker for the number of unsent spill files. If this number grows beyond a certain threshold (two unsent spill files in our experiments), the map task does not immediately register the new spill file with the

TaskTracker. Instead, the mapper will accumulate multiple spill files. Once the queue of unsent spill files falls below the threshold, the map task merges and combines the accumulated spill files into a single file, and then resumes registering its output with the TaskTracker. This simple flow control mechanism has the effect of *adaptively* moving load from the reducer to the mapper or vice versa, depending on which node is the current bottleneck.

A similar mechanism is also used to control how aggressively the combiner function is applied. The map task records the ratio between the input and output data sizes whenever it invokes the combiner function. If the combiner is effective at reducing data volumes, the map task accumulates more spill files (and applies the combiner function to all of them) before registering that output with the TaskTracker for pipelining.<sup>2</sup>

The connection between pipelining and adaptive query processing techniques has been observed elsewhere (e.g., [2]). The adaptive scheme outlined above is relatively simple, but we believe that adapting to feedback along pipelines has the potential to significantly improve the utilization of MapReduce clusters.

## 3.2 Pipelining Between Jobs

Many practical computations cannot be expressed as a single MapReduce job, and the outputs of higher-level languages like Pig [20] typically involve multiple jobs. In the traditional Hadoop architecture, the output of each job is written to HDFS in the reduce step and then immediately read back from HDFS by the map step of the next job. Furthermore, the JobTracker cannot schedule a consumer job until the producer job has completed, because scheduling a map task requires knowing the HDFS block locations of the map’s input split.

In our modified version of Hadoop, the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job, sidestepping the need for expensive fault-tolerant storage in HDFS for what amounts to a temporary file. Unfortunately, the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped: the final result of the reduce step cannot be produced until all map tasks have completed, which prevents effective pipelining. However, in the next sections we describe how online aggregation and continuous query pipelines can publish “snapshot” outputs that can indeed pipeline between jobs.

---

<sup>2</sup>Our current prototype uses a simple heuristic: if the combiner reduces data volume by  $\frac{1}{k}$  on average, we wait until  $k$  spill files have accumulated before registering them with the TaskTracker. A better heuristic would also account for the computational cost of applying the combiner function.

## 3.3 Fault Tolerance

Our pipelined Hadoop implementation is robust to the failure of both map and reduce tasks. To recover from map task failures, we added bookkeeping to the reduce task to record which map task produced each pipelined spill file. To simplify fault tolerance, the reducer treats the output of a pipelined map task as “tentative” until the JobTracker informs the reducer that the map task has committed successfully. The reducer can merge together spill files generated by the same uncommitted mapper, but will not combine those spill files with the output of other map tasks until it has been notified that the map task has committed. Thus, if a map task fails, each reduce task can ignore any tentative spill files produced by the failed map attempt. The JobTracker will take care of scheduling a new map task attempt, as in stock Hadoop.

If a reduce task fails and a new copy of the task is started, the new reduce instance must be sent all the input data that was sent to the failed reduce attempt. If map tasks operated in a purely pipelined fashion and discarded their output after sending it to a reducer, this would be difficult. Therefore, map tasks retain their output data on the local disk for the complete job duration. This allows the map’s output to be reproduced if any reduce tasks fail. For batch jobs, the key advantage of our architecture is that reducers are not blocked waiting for the complete output of the task to be written to disk.

Our technique for recovering from map task failure is straightforward, but places a minor limit on the reducer’s ability to merge spill files. To avoid this, we envision introducing a “checkpoint” concept: as a map task runs, it will periodically notify the JobTracker that it has reached offset  $x$  in its input split. The JobTracker will notify any connected reducers; map task output that was produced before offset  $x$  can then be merged by reducers with other map task output as normal. To avoid duplicate results, if the map task fails, the new map task attempt resumes reading its input at offset  $x$ . This technique would also reduce the amount of redundant work done after a map task failure or during speculative execution of “backup” tasks [6].

## 3.4 Task Scheduling

The Hadoop JobTracker had to be retrofitted to support pipelining between jobs. In regular Hadoop, job are submitted one at a time; a job that consumes the output of one or more other jobs cannot be submitted until the producer jobs have completed. To address this, we modified the Hadoop job submission interface to accept a list of jobs, where each job in the list depends on the job before it. The client interface traverses this list, annotating each job with the identifier of the job that it depends on. The

JobTracker looks for this annotation and co-schedules jobs with their dependencies, giving slot preference to “upstream” jobs over the “downstream” jobs they feed. As we note in Section 8, there are many interesting options for scheduling pipelines or even DAGs of such jobs that we plan to investigate in future.

## 4 Online Aggregation

Although MapReduce was originally designed as a batch-oriented system, it is often used for interactive data analysis: a user submits a job to extract information from a data set, and then waits to view the results before proceeding with the next step in the data analysis process. This trend has accelerated with the development of high-level query languages that are executed as MapReduce jobs, such as Hive [27], Pig [20], and Sawzall [23].

Traditional MapReduce implementations provide a poor interface for interactive data analysis, because they do not emit any output until the job has been executed to completion. In many cases, an interactive user would prefer a “quick and dirty” approximation over a correct answer that takes much longer to compute. In the database literature, online aggregation has been proposed to address this problem [12], but the batch-oriented nature of traditional MapReduce implementations makes these techniques difficult to apply. In this section, we show how we extended our pipelined Hadoop implementation to support online aggregation within a single job (Section 4.1) and between multiple jobs (Section 4.2). In Section 4.3, we evaluate online aggregation on two different data sets, and show that it can yield an accurate approximate answer long before the job has finished executing.

### 4.1 Single-Job Online Aggregation

In HOP, the data records produced by map tasks are sent to reduce tasks shortly after each record is generated. However, to produce the final output of the job, the reduce function cannot be invoked until the entire output of every map task has been produced. We can support online aggregation by simply applying the reduce function to the data that a reduce task has received so far. We call the output of such an intermediate reduce operation a *snapshot*.

Users would like to know how accurate a snapshot is: that is, how closely a snapshot resembles the final output of the job. Accuracy estimation is a hard problem even for simple SQL queries [15], and particularly hard for jobs where the map and reduce functions are opaque user-defined code. Hence, we report job *progress*, not accuracy: we leave it to the user (or their MapReduce code) to correlate progress to a formal notion of accuracy. We give a simple progress metric below.

Snapshots are computed periodically, as new data arrives at each reducer. The user specifies how often snapshots should be computed, using the progress metric as the unit of measure. For example, a user can request that a snapshot be computed when 25%, 50%, and 75% of the input has been seen. The user may also specify whether to include data from tentative (unfinished) map tasks. This option does not affect the fault tolerance design described in Section 3.3. In the current prototype, each snapshot is stored in a directory on HDFS. The name of the directory includes the progress value associated with the snapshot. Each reduce task runs independently, and at a different rate. Once a reduce task has made sufficient progress, it writes a snapshot to a temporary directory on HDFS, and then atomically renames it to the appropriate location.

Applications can consume snapshots by polling HDFS in a predictable location. An application knows that a given snapshot has been completed when every reduce task has written a file to the snapshot directory. Atomic rename is used to avoid applications mistakenly reading incomplete snapshot files.

Note that if there are not enough free slots to allow all the reduce tasks in a job to be scheduled, snapshots will not be available for reduce tasks that are still waiting to be executed. The user can detect this situation (e.g., by checking for the expected number of files in the HDFS snapshot directory), so there is no risk of incorrect data, but the usefulness of online aggregation will be reduced. In the current prototype, we manually configured the cluster to avoid this scenario. The system could also be enhanced to avoid this pitfall entirely by optionally waiting to execute an online aggregation job until there are enough reduce slots available.

#### 4.1.1 Progress Metric

Hadoop provides support for monitoring the progress of task executions. As each map task executes, it is assigned a *progress score* in the range [0,1], based on how much of its input the map task has consumed. We reused this feature to determine how much progress is represented by the current input to a reduce task, and hence to decide when a new snapshot should be taken.

First, we modified the spill file format depicted in Figure 2 to include the map’s current progress score. When a partition in a spill file is sent to a reducer, the spill file’s progress score is also included. To compute the progress score for a snapshot, we take the average of the progress scores associated with each spill file used to produce the snapshot.

Note that it is possible that a map task might not have pipelined *any* output to a reduce task, either because the map task has not been scheduled yet (there are no free TaskTracker slots), the map tasks does not produce any

output for the given reduce task, or because the reduce task has been configured to only pipeline data from at most  $k$  map tasks concurrently. To account for this, we need to scale the progress metric to reflect the portion of the map tasks that a reduce task has pipelined data from: if a reducer is connected to  $\frac{1}{n}$  of the total number of map tasks in the job, we divide the average progress score by  $n$ .

This progress metric could easily be made more sophisticated: for example, an improved metric might include the selectivity ( $|output|/|input|$ ) of each map task, the statistical distribution of the map task’s output, and the effectiveness of each map task’s combine function, if any. Although we have found our simple progress metric to be sufficient for most experiments we describe below, this clearly represents an opportunity for future work.

## 4.2 Multi-Job Online Aggregation

Online aggregation is particularly useful when applied to a long-running analysis task composed of multiple MapReduce jobs. As described in Section 3.2, our version of Hadoop allows the output of a reduce task to be sent directly to map tasks. This feature can be used to support online aggregation for a sequence of jobs.

Suppose that  $j_1$  and  $j_2$  are two MapReduce jobs, and  $j_2$  consumes the output of  $j_1$ . When  $j_1$ ’s reducers compute a snapshot to perform online aggregation, that snapshot is written to HDFS, and also sent directly to the map tasks of  $j_2$ . The map and reduce steps for  $j_2$  are then computed as normal, to produce a snapshot of  $j_2$ ’s output. This process can then be continued to support online aggregation for an arbitrarily long sequence of jobs.

Unfortunately, inter-job online aggregation has some drawbacks. First, the output of a reduce function is not “monotonic”: the output of a reduce function on the first 50% of the input data may not be obviously related to the output of the reduce function on the first 25%. Thus, as new snapshots are produced by  $j_1$ ,  $j_2$  must be recomputed from scratch using the new snapshot. As with inter-job pipelining (Section 3.2), this could be optimized for reduce functions that are declared to be distributive or algebraic aggregates [9].

To support fault tolerance for multi-job online aggregation, we consider three cases. Tasks that fail in  $j_1$  recover as described in Section 3.3. If a task in  $j_2$  fails, the system simply restarts the failed task. Since subsequent snapshots produced by  $j_1$  are taken from a superset of the mapper output in  $j_1$ , the next snapshot received by the restarted reduce task in  $j_2$  will have a higher progress score. To handle failures in  $j_1$ , tasks in  $j_2$  cache the most recent snapshot received by  $j_1$ , and replace it when they receive a new snapshot with a higher progress metric. If tasks from both jobs fail, a new task in  $j_2$  recovers the most

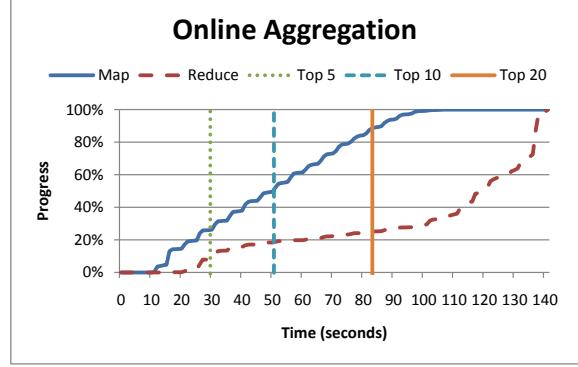


Figure 4: Top-100 query over 5.5GB of Wikipedia article text. The vertical lines describe the increasing accuracy of the approximate answers produced by online aggregation.

recent snapshot from  $j_1$  that was stored in HDFS and then wait for snapshots with a higher progress score.

## 4.3 Evaluation

To evaluate the effectiveness of online aggregation, we performed two experiments on Amazon EC2 using different data sets and query workloads. In our first experiment, we wrote a “Top- $K$ ” query using two MapReduce jobs: the first job counts the frequency of each word and the second job selects the  $K$  most frequent words. We ran this workload on 5.5GB of Wikipedia article text stored in HDFS, using a 128MB block size. We used a 60-node EC2 cluster; each node was a “high-CPU medium” EC2 instance with 1.7GB of RAM and 2 virtual cores. A virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor. A single EC2 node executed the Hadoop Job-Tracker and the HDFS NameNode, while the remaining nodes served as slaves for running the TaskTrackers and HDFS DataNodes.

Figure 4 shows the results of inter-job online aggregation for a Top-100 query. Our accuracy metric for this experiment is post-hoc — we note the time at which the Top- $K$  words in the snapshot are the Top- $K$  words in the final result. Although the final result for this job did not appear until nearly the end, we did observe the Top-5, 10, and 20 values at the times indicated in the graph. The Wikipedia data set was biased toward these Top- $K$  words (e.g., “the”, “is”, etc.), which remained in their correct position throughout the lifetime of the job.

### 4.3.1 Approximation Metrics

In our second experiment, we considered the effectiveness of the job progress metric described in Section 4.1.1. Unsurprisingly, this metric can be inaccurate when it is used to estimate the accuracy of the approximate answers produced by online aggregation. In this experiment, we com-

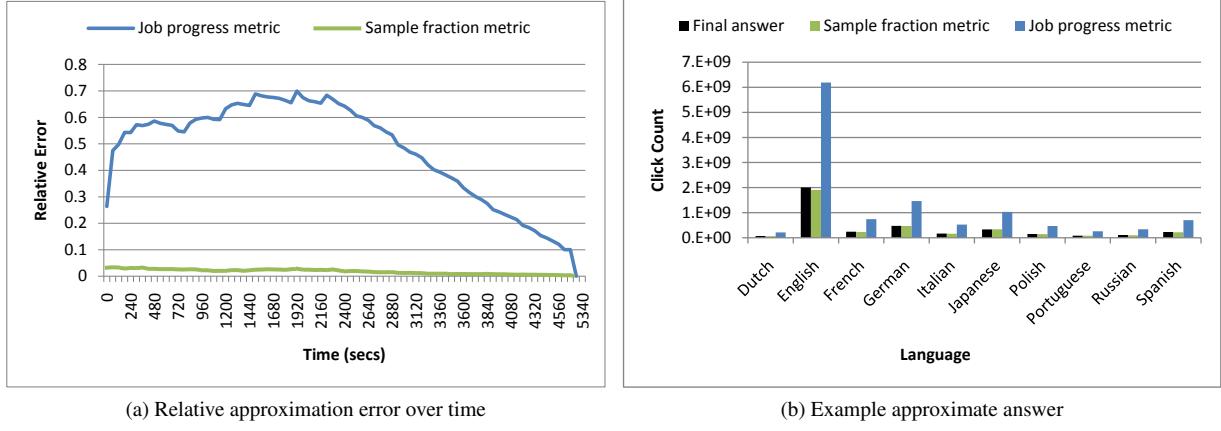


Figure 5: Comparison of two approximation metrics. Figure (a) shows the relative error for each approximation metric over the runtime of the job, averaged over all groups. Figure (b) compares an example approximate answer produced by each metric with the final answer, for each language and for a single hour.

pared the job progress metric with a simple user-defined metric that leverages knowledge of the query and data set. HOP allows such metrics, although developing such a custom metric imposes more burden on the programmer than using the generic progress-based metric.

We used a data set containing seven months of hourly page view statistics for more than 2.5 million Wikipedia articles [26]. This constituted 320GB of compressed data (1TB uncompressed), divided into 5066 compressed files. We stored the data set on HDFS and assigned a single map task to each file, which was decompressed before the map function was applied.

We wrote a MapReduce job to count the total number of page views for each language and each hour of the day. In other words, our query grouped by language and hour of day, and summed the number of page views that occurred in each group. To enable more accurate approximate answers, we modified the map function to include the fraction of a given hour that each record represents. The reduce function summed these fractions for a given hour, which equated to one for all records from a single map task. Since the total number of hours was known ahead of time, we could use the result of this sum over all map outputs to determine the total fraction of each hour that had been sampled. We call this user-defined metric the “sample fraction.”

To compute approximate answers, each intermediate result was scaled up using two different metrics: the generic metric based on job progress and the sample fraction described above. Figure 5a reports the relative error of the two metrics, averaged over all groups. Figure 5b shows an example approximate answer for a single hour using both metrics (computed two minutes into the job runtime). This figure also contains the final answer for comparison. Both results indicate that the sample fraction metric pro-

vides a much more accurate approximate answer for this query than the progress-based metric.

Job progress is clearly the wrong metric to use for approximating the final answer of this query. The primary reason is that it is too coarse of a metric. Each intermediate result was computed from some fraction of each hour. However, the job progress assumes that this fraction is uniform across all hours, when in fact we could have received much more of one hour and much less of another. This assumption of uniformity in the job progress resulted in a significant approximation error. By contrast, the sample fraction scales the approximate answer for each group according to the actual fraction of data seen for that group, yielding much more accurate approximations.

## 5 Continuous Queries

MapReduce is often used to analyze streams of constantly-arriving data, such as URL access logs [6] and system console logs [30]. Because of traditional constraints on MapReduce, this is done in large batches that can only provide periodic views of activity. This introduces significant latency into a data analysis process that ideally should run in near-real time. It is also potentially inefficient: each new MapReduce job does not have access to the computational state of the last analysis run, so this state must be recomputed from scratch. The programmer can manually save the state of each job and then reload it for the next analysis operation, but this is labor-intensive.

Our pipelined version of Hadoop allows an alternative architecture: MapReduce jobs that run *continuously*, accepting new data as it becomes available and analyzing it immediately. This allows for near-real-time analysis of data streams, and thus allows the MapReduce programming model to be applied to domains such as environment

monitoring and real-time fraud detection.

In this section, we describe how HOP supports continuous MapReduce jobs, and how we used this feature to implement a rudimentary cluster monitoring tool.

## 5.1 Continuous MapReduce Jobs

A bare-bones implementation of continuous MapReduce jobs is easy to implement using pipelining. No changes are needed to implement continuous map tasks: map output is already delivered to the appropriate reduce task shortly after it is generated. We added an optional “flush” API that allows map functions to force their current output to reduce tasks. When a reduce task is unable to accept such data, the mapper framework stores it locally and sends it at a later time. With proper scheduling of reducers, this API allows a map task to ensure that an output record is promptly sent to the appropriate reducer.

To support continuous reduce tasks, the user-defined reduce function must be periodically invoked on the map output available at that reducer. Applications will have different requirements for how frequently the reduce function should be invoked; possible choices include periods based on wall-clock time, logical time (e.g., the value of a field in the map task output), and the number of input rows delivered to the reducer. The output of the reduce function can be written to HDFS, as in our implementation of online aggregation. However, other choices are possible; our prototype system monitoring application (described below) sends an alert via email if an anomalous situation is detected.

In our current implementation, the number of map and reduce tasks is fixed, and must be configured by the user. This is clearly problematic: manual configuration is error-prone, and many stream processing applications exhibit “bursty” traffic patterns, in which peak load far exceeds average load. In the future, we plan to add support for elastic scaleup/scaledown of map and reduce tasks in response to variations in load.

### 5.1.1 Fault Tolerance

In the checkpoint/restart fault-tolerance model used by Hadoop, mappers retain their output until the end of the job to facilitate fast recovery from reducer failures. In a continuous query context, this is infeasible, since mapper history is in principle unbounded. However, many continuous reduce functions (e.g., 30-second moving average) only require a suffix of the map output stream. This common case can be supported easily, by extending the JobTracker interface to capture a rolling notion of reducer consumption. Map-side spill files are maintained in a ring buffer with unique IDs for spill files over time. When a reducer commits an output to HDFS, it informs the Job-

Tracker about the *run* of map output records it no longer needs, identifying the run by spill file IDs and offsets within those files. The JobTracker can then tell mappers to garbage collect the appropriate data.

In principle, complex reducers may depend on very long (or infinite) histories of map records to accurately reconstruct their internal state. In that case, deleting spill files from the map-side ring buffer will result in potentially inaccurate recovery after faults. Such scenarios can be handled by having reducers checkpoint internal state to HDFS, along with markers for the mapper offsets at which the internal state was checkpointed. The MapReduce framework can be extended with APIs to help with state serialization and offset management, but it still presents a programming burden on the user to correctly identify the sensitive internal state. That burden can be avoided by more heavyweight process-pair techniques for fault tolerance, but those are quite complex and use significant resources [24]. In our work to date we have focused on cases where reducers can be recovered from a reasonable-sized history at the mappers, favoring minor extensions to the simple fault-tolerance approach used in Hadoop.

## 5.2 Prototype Monitoring System

Our monitoring system is composed of *agents* that run on each monitored machine and record statistics of interest (e.g., load average, I/O operations per second, etc.). Each agent is implemented as a continuous map task: rather than reading from HDFS, the map task instead reads from various system-local data streams (e.g., `/proc`).

Each agent forwards statistics to an *aggregator* that is implemented as a continuous reduce task. The aggregator records how agent-local statistics evolve over time (e.g., by computing windowed-averages), and compares statistics between agents to detect anomalous behavior. Each aggregator monitors the agents that report to it, but might also report statistical summaries to another “upstream” aggregator. For example, the system might be configured to have an aggregator for each rack and then a second level of aggregators that compare statistics between racks to analyze datacenter-wide behavior.

## 5.3 Evaluation

To validate our prototype system monitoring tool, we constructed a scenario in which one member of a MapReduce cluster begins thrashing during the execution of a job. Our goal was to test how quickly our monitoring system would detect this behavior. The basic mechanism is similar to an alert system one of the authors implemented at an Internet search company.

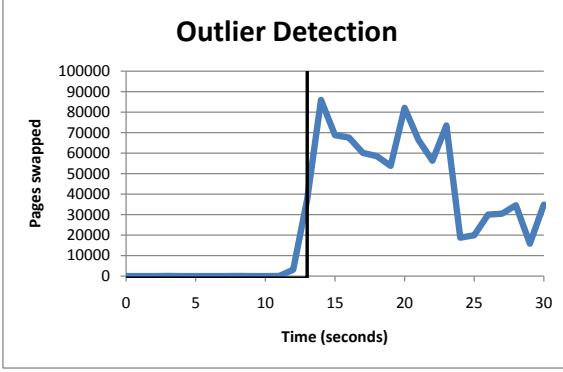


Figure 6: Number of pages swapped over time on the thrashing host, as reported by `vmstat`. The vertical line indicates the time at which the alert was sent by the monitoring system.

We used a simple load metric (a linear combination of CPU utilization, paging, and swap activity). The continuous reduce function maintains windows over samples of this metric: at regular intervals, it compares the 20 second moving average of the load metric for each host to the 120 second moving average of all the hosts in the cluster *except* that host. If the given host’s load metric is more than two standard deviations above the global average, it is considered an outlier and a tentative alert is issued. To dampen false positives in “bursty” load scenarios, we do not issue an alert until we have received 10 tentative alerts within a time window.

We deployed this system on an EC2 cluster consisting of 7 “large” nodes (large nodes were chosen because EC2 allocates an entire physical host machine to them). We ran a wordcount job on the 5.5GB Wikipedia data set, using 5 map tasks and 2 reduce tasks (1 task per host). After the job had been running for about 10 seconds, we selected a node running a task and launched a program that induced thrashing.

We report detection latency in Figure 6. The vertical bar indicates the time at which the monitoring tool fired a (non-tentative) alert. The thrashing host was detected very rapidly—notably faster than the 5-second TaskTracker-JobTracker heartbeat cycle that is used to detect straggler tasks in stock Hadoop. We envision using these alerts to do early detection of stragglers within a MapReduce job: HOP could make scheduling decisions for a job by running a secondary continuous monitoring query. Compared to out-of-band monitoring tools, this economy of mechanism—reusing the MapReduce infrastructure for reflective monitoring—has benefits in software maintenance and system management.

## 6 Performance Evaluation

A thorough performance comparison between pipelining and blocking is beyond the scope of this paper. In this section, we instead demonstrate that pipelining can reduce job completion times in some configurations.

We report performance using both large (512MB) and small (32MB) HDFS block sizes using a single workload (a wordcount job over randomly-generated text). Since the words were generated using a uniform distribution, map-side combiners were ineffective for this workload. We performed all experiments using relatively small clusters of Amazon EC2 nodes. We also did not consider performance in an environment where multiple concurrent jobs are executing simultaneously.

### 6.1 Background and Configuration

Before diving into the performance experiments, it is important to further describe the division of labor in a HOP job, which is broken into task phases. A map task consists of two work phases: *map* and *sort*. The majority of work is performed in the *map* phase, where the map function is applied to each record in the input and subsequently sent to an output buffer. Once the entire input has been processed, the map task enters the *sort* phase, where a final merge sort of all intermediate spill files is performed before registering the final output with the TaskTracker. The progress reported by a map task corresponds to the *map* phase only.

A reduce task in HOP is divided into three work phases: *shuffle*, *reduce*, and *commit*. In the *shuffle* phase, reduce tasks receive their portion of the output from each map. In HOP, the *shuffle* phase consumes 75% of the overall reduce task progress while the remaining 25% is allocated to the *reduce* and *commit* phase.<sup>3</sup> In the *shuffle* phase, reduce tasks periodically perform a merge sort on the already received map output. These intermediate merge sorts decrease the amount of sorting work performed at the end of the *shuffle* phase. After receiving its portion of data from all map tasks, the reduce task performs a final merge sort and enters the *reduce* phase.

By pushing work from map tasks to reduce tasks more aggressively, pipelining can enable better overlapping of map and reduce computation, especially when the node on which a reduce task is scheduled would otherwise be underutilized. However, when reduce tasks are already the bottleneck, pipelining offers fewer performance benefits, and may even hurt performance by placing additional load on the reduce nodes.

<sup>3</sup>The stock version of Hadoop divides the reduce progress evenly among the three phases. We deviated from this approach because we wanted to focus more on the progress during the *shuffle* phase.

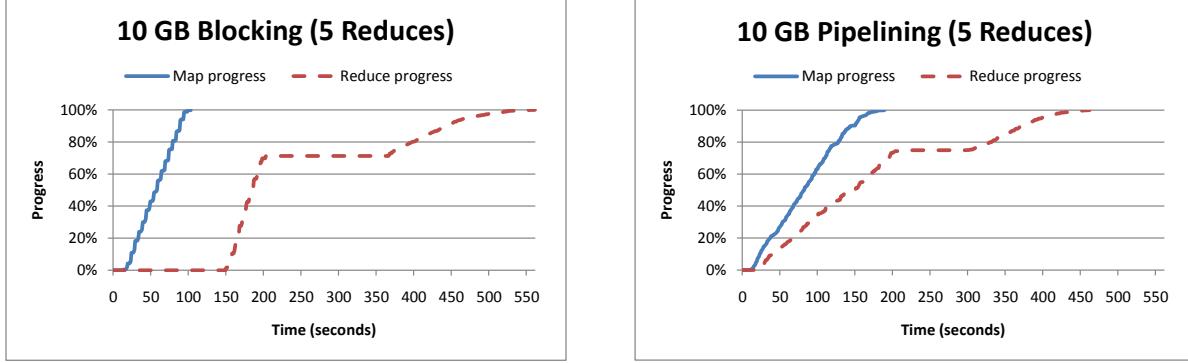


Figure 7: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 5 reduce tasks (512MB block size). The total job runtimes were 561 seconds for blocking and 462 seconds for pipelining.

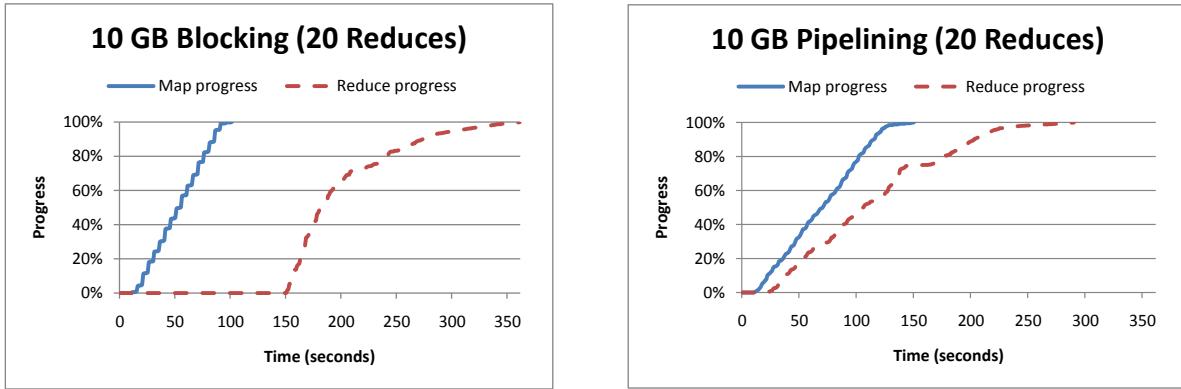


Figure 8: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 20 reduce tasks (512MB block size). The total job runtimes were 361 seconds for blocking and 290 seconds for pipelining.

The *sort* phase in the map task minimizes the merging work that reduce tasks must perform at the end of the *shuffle* phase. When pipelining is enabled, the *sort* phase is avoided since map tasks have already sent some fraction of the spill files to concurrently running reduce tasks. Therefore, pipelining increases the merging workload placed on the reducer. The adaptive pipelining scheme described in Section 3.1.3 attempts to ensure that reduce tasks are not overwhelmed with additional load.

We used two Amazon EC2 clusters depending on the size of the experiment: “small” jobs used 10 worker nodes, while “large” jobs used 20. Each node was an “extra large” EC2 instances with 15GB of memory and four virtual cores.

## 6.2 Small Job Results

Our first experiment focused on the performance of small jobs in an underutilized cluster. We ran a 10GB wordcount with a 512MB block size, yielding 20 map tasks. We used 10 worker nodes and configured each worker to execute at most two map and two reduce tasks simultaneously. We ran several experiments to compare the

performance of blocking and pipelining using different numbers of reduce tasks.

Figure 7 reports the results with five reduce tasks. A plateau can be seen at 75% progress for both blocking and pipelining. At this point in the job, all reduce tasks have completed the *shuffle* phase; the plateau is caused by the time taken to perform a final merge of all map output before entering the *reduce* phase. Notice that the plateau for the pipelining case is shorter. With pipelining, reduce tasks receive map outputs earlier and can begin sorting earlier, thereby reducing the time required for the final merge.

Figure 8 reports the results with twenty reduce tasks. Using more reduce tasks decreases the amount of merging that any one reduce task must perform, which reduces the duration of the plateau at 75% progress. In the blocking case, the plateau is practically gone.

Note that in both experiments, the map phase finishes faster with blocking than with pipelining. This is because pipelining allows reduce tasks to begin executing more quickly; hence, the reduce tasks compete for resources with the map tasks, causing the map phase to take slightly

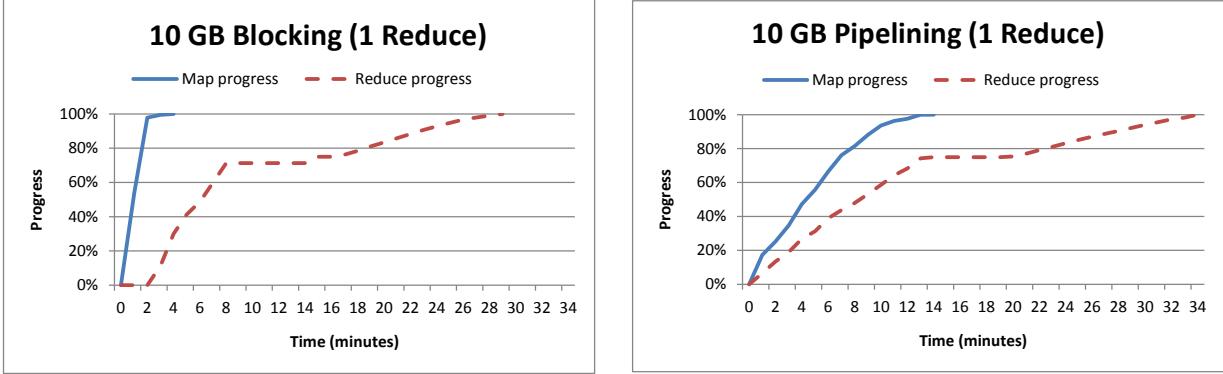


Figure 9: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 1 reduce task (512MB block size). The total job runtimes were 29 minutes for blocking and 34 minutes for pipelining.

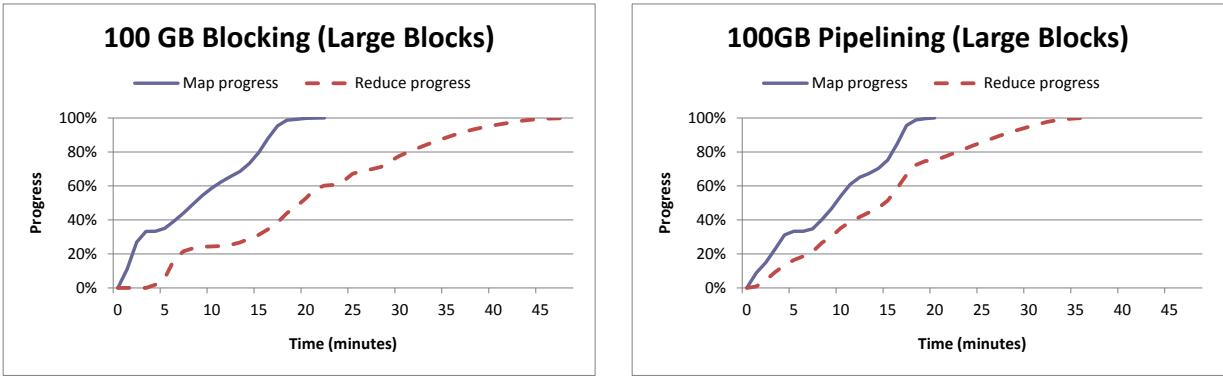


Figure 10: CDF of map and reduce task completion times for a 100GB wordcount job using 240 map tasks and 60 reduce tasks (512MB block size). The total job runtimes were 48 minutes for blocking and 36 minutes for pipelining.

longer. In this case, the increase in map duration is outweighed by the increase in cluster utilization, resulting in shorter job completion times: pipelining reduced completion time by 17.7% with 5 reducers and by 19.7% with 20 reducers.

Figure 9 describes an experiment in which we ran a 10GB wordcount job using a single reduce task. This caused job completion times to increase dramatically for both pipelining and blocking, because of the extreme load placed on the reduce node. Pipelining delayed job completion by  $\sim 17\%$ , which suggests that our simple adaptive flow control scheme (Section 3.1.3) was unable to move load back to the map tasks aggressively enough.

### 6.3 Large Job Results

Our second set of experiments focused on the performance of somewhat larger jobs. We increased the input size to 100GB (from 10GB) and the number of worker nodes to 20 (from 10). Each worker was configured to execute at most four map and three reduce tasks, which meant that at most 80 map and 60 reduce tasks could

execute at once. We conducted two sets of experimental runs, each run comparing blocking to pipelining using either large (512MB) or small (32MB) block sizes. We were interested in blocking performance with small block sizes because blocking can effectively emulate pipelining if the block size is small enough.

Figure 10 reports the performance of a 100GB wordcount job with 512MB blocks, which resulted in 240 map tasks, scheduled in three waves of 80 tasks each. The 60 reduce tasks were coscheduled with the first wave of map tasks. In the blocking case, the reduce tasks began working as soon as they received the output of the first wave, which is why the reduce progress begins to climb around four minutes (well before the completion of all maps). Pipelining was able to achieve significantly better cluster utilization, and hence reduced job completion time by  $\sim 25\%$ .

Figure 11 reports the performance of blocking and pipelining using 32MB blocks. While the performance of pipelining remained similar, the performance of blocking improved considerably, but still trailed somewhat behind pipelining. Using block sizes smaller than 32MB did

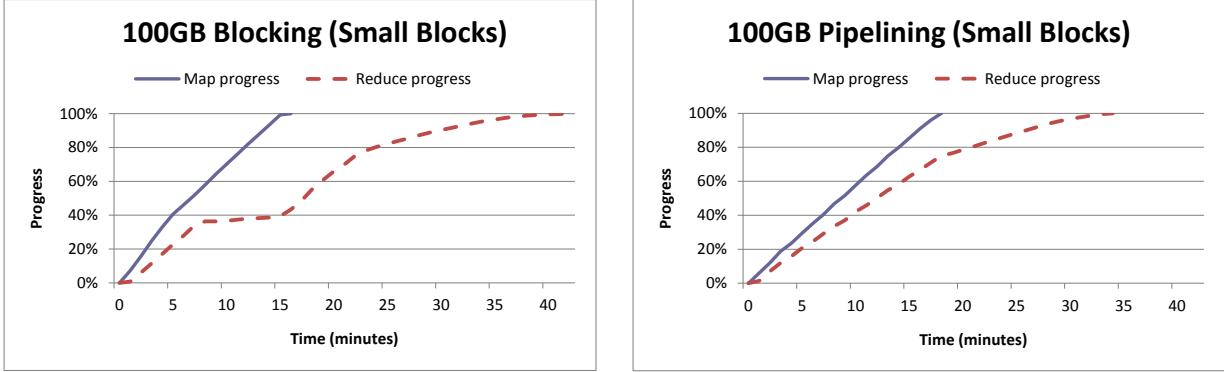


Figure 11: CDF of map and reduce task completion times for a 100GB wordcount job using 3120 map tasks and 60 reduce tasks (32MB block size). The total job runtimes were 42 minutes for blocking and 34 minutes for pipelining.

not yield a significant performance improvement in our experiments.

## 7 Related Work

The work in this paper relates to literature on parallel dataflow frameworks, online aggregation, and continuous query processing.

### 7.1 Parallel Dataflow

Dean and Ghemawat’s paper on Google’s MapReduce [6] has become a standard reference, and forms the basis of the open-source Hadoop implementation. As noted in Section 1, the Google MapReduce design targets very large clusters where the probability of worker failure or slowdown is high. This led to their elegant checkpoint/restart approach to fault tolerance, and their lack of pipelining. Our work extends the Google design to accommodate pipelining without significant modification to their core programming model or fault tolerance mechanisms.

*Dryad* [13] is a data-parallel programming model and runtime that is often compared to MapReduce, supporting a more general model of acyclic dataflow graphs. Like MapReduce, Dryad puts disk materialization steps between dataflow stages by default, breaking pipelines. The Dryad paper describes support for optionally “encapsulating” multiple asynchronous stages into a single process so they can pipeline, but this requires a more complicated programming interface. The Dryad paper explicitly mentions that the system is targeted at batch processing, and not at scenarios like continuous queries.

It has been noted that parallel database systems have long provided partitioned dataflow frameworks [21], and recent commercial databases have begun to offer MapReduce programming models on top of those frameworks [5, 10]. Most parallel database systems can pro-

vide pipelined execution akin to our work here, but they use a more tightly coupled iterator and *Exchange* model that keeps producers and consumers rate-matched via queues, spreading the work of each dataflow stage across all nodes in the cluster [8]. This provides less scheduling flexibility than MapReduce and typically offers no tolerance to mid-query worker faults. Yang et al. recently proposed a scheme to add support for mid-query fault tolerance to traditional parallel databases, using a middleware-based approach that shares some similarities with MapReduce [31].

Logothetis and Yocum describe a MapReduce interface over a continuous query system called *Mortar* that is similar in some ways to our work [16]. Like HOP, their mappers push data to reducers in a pipelined fashion. They focus on specific issues in efficient stream query processing, including minimization of work for aggregates in overlapping windows via special reducer APIs. They are not built on Hadoop, and explicitly sidestep issues in fault tolerance.

*Hadoop Streaming* is part of the Hadoop distribution, and allows map and reduce functions to be expressed as UNIX shell command lines. It does not stream data through map and reduce phases in a pipelined fashion.

### 7.2 Online Aggregation

Online aggregation was originally proposed in the context of simple single-table SQL queries involving “Group By” aggregations, a workload quite similar to MapReduce [12]. The focus of the initial work was on providing not only “early returns” to these SQL queries, but also statistically robust estimators and confidence interval metrics for the final result based on random sampling. These statistical matters do not generalize to arbitrary MapReduce jobs, though our framework can support those that have been developed. Subsequently, online aggregation was extended to handle join queries (via the *Ripple Join* method),

and the *CONTROL* project generalized the idea of online query processing to provide interactivity for data cleaning, data mining, and data visualization tasks [11]. That work was targeted at single-processor systems. Luo et al. developed a partitioned-parallel variant of Ripple Join, without statistical guarantees on approximate answers [17].

In recent years, this topic has seen renewed interest, starting with Jermaine et al.’s work on the *DBO* system [15]. That effort includes more disk-conscious online join algorithms, as well as techniques for maintaining randomly-shuffled files to remove any potential for statistical bias in scans [14]. Wu et al. describe a system for peer-to-peer online aggregation in a distributed hash table context [29]. The open programmability and fault-tolerance of MapReduce are not addressed significantly in prior work on online aggregation.

An alternative to online aggregation combines precomputation with sampling, storing fixed samples and summaries to provide small storage footprints and interactive performance [7]. An advantage of these techniques is that they are compatible with both pipelining and blocking models of MapReduce. The downside of these techniques is that they do not allow users to choose the query stopping points or time/accuracy trade-offs dynamically [11].

### 7.3 Continuous Queries

In the last decade there was a great deal of work in the database research community on the topic of continuous queries over data streams, including systems such as Borealis [1], STREAM [18], and Telegraph [4]. Of these, Borealis and Telegraph [24] studied fault tolerance and load balancing across machines. In the Borealis context this was done for pipelined dataflows, but without partitioned parallelism: each stage (“operator”) of the pipeline runs serially on a different machine in the wide area, and fault tolerance deals with failures of entire operators [3]. SBON [22] is an overlay network that can be integrated with Borealis, which handles “operator placement” optimizations for these wide-area pipelined dataflows.

Telegraph’s *FLuX* operator [24, 25] is the only work to our knowledge that addresses mid-stream fault-tolerance for dataflows that are both pipelined and partitioned in the style of HOP. *FLuX* (“Fault-tolerant, Load-balanced eXchange”) is a dataflow operator that encapsulates the shuffling done between stages such as map and reduce. It provides load-balancing interfaces that can migrate operator state (e.g., reducer state) between nodes, while handling scheduling policy and changes to data-routing policies [25]. For fault tolerance, *FLuX* develops a solution based on process pairs [24], which work redundantly to ensure that operator state is always being maintained live on multiple nodes. This removes any burden on the continuous query programmer of the sort we describe in Sec-

tion 5. On the other hand, the *FLuX* protocol is far more complex and resource-intensive than our pipelined adaptation of Google’s checkpoint/restart tolerance model.

## 8 Conclusion and Future Work

MapReduce has proven to be a popular model for large-scale parallel programming. Our Hadoop Online Prototype extends the applicability of the model to pipelining behaviors, while preserving the simple programming model and fault tolerance of a full-featured MapReduce framework. This provides significant new functionality, including “early returns” on long-running jobs via online aggregation, and continuous queries over streaming data. We also demonstrate benefits for batch processing: by pipelining both within and across jobs, HOP can reduce the time to job completion.

In considering future work, scheduling is a topic that arises immediately. Stock Hadoop already has many degrees of freedom in scheduling batch tasks across machines and time, and the introduction of pipelining in HOP only increases this design space. First, pipeline parallelism is a new option for improving performance of MapReduce jobs, but needs to be integrated intelligently with both intra-task partition parallelism and speculative redundant execution for “straggler” handling. Second, the ability to schedule deep pipelines with direct communication between reduces and maps (bypassing the distributed file system) opens up new opportunities and challenges in carefully co-locating tasks from different jobs, to avoid communication when possible.

Olston and colleagues have noted that MapReduce systems—unlike traditional databases—employ “model-light” optimization approaches that gather and react to performance information during runtime [19]. The continuous query facilities of HOP enable powerful introspective programming interfaces for this: a full-featured MapReduce interface can be used to script performance monitoring tasks that gather system-wide information in near-real-time, enabling tight feedback loops for scheduling and dataflow optimization. This is a topic we plan to explore, including opportunistic methods to do monitoring work with minimal interference to outstanding jobs, as well as dynamic approaches to continuous optimization in the spirit of earlier work like Eddies [2] and *FLuX* [25].

As a more long-term agenda, we want to explore using MapReduce-style programming for even more interactive applications. As a first step, we hope to revisit interactive data processing in the spirit of the *CONTROL* work [11], with an eye toward improved scalability via parallelism. More aggressively, we are considering the idea of bridging the gap between MapReduce dataflow programming and lightweight event-flow programming models like SEDA [28]. Our HOP implementation’s roots

in Hadoop make it unlikely to compete with something like SEDA in terms of raw performance. However, it would be interesting to translate ideas across these two traditionally separate programming models, perhaps with an eye toward building a new and more general-purpose framework for programming in architectures like cloud computing and many-core.

## Acknowledgments

We would like to thank Daniel Abadi, Kuang Chen, Mosharaf Chowdhury, Akshay Krishnamurthy, Andrew Pavlo, Hong Tang , and our shepherd Jeff Dean for their helpful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant Nos. 0713661, 0722077 and 0803690, the Air Force Office of Scientific Research under Grant No. FA95500810352, the Natural Sciences and Engineering Research Council of Canada, and gifts from IBM, Microsoft, and Yahoo!.

## References

- [1] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The design of the Borealis stream processing engine. In *CIDR* (2005).
- [2] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *SIGMOD* (2000).
- [3] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD* (2005).
- [4] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR* (2003).
- [5] CIESLEWICZ, J., FRIEDMAN, E., AND PAWLOWSKI, P. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In *VLDB* (2009).
- [6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).
- [7] GIBBONS, P. B., AND MATIAS, Y. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD* (1998).
- [8] GRAEFE, G. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD* (1990).
- [9] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.* 1, 1 (1997), 29–53.
- [10] Greenplum: A unified engine for RDBMS and MapReduce, Oct. 2008. Downloaded from <http://www.greenplum.com/download.php?alias=register-map-reduce&file=Greenplum-MapReduce-Whitepaper.pdf>.
- [11] HELLERSTEIN, J. M., AVNUR, R., CHOU, A., HIDBER, C., OLSTON, C., RAMAN, V., ROTH, T., AND HAAS, P. J. Interactive data analysis with CONTROL. *IEEE Computer* 32, 8 (Aug. 1999).
- [12] HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. Online aggregation. In *SIGMOD* (1997).
- [13] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).
- [14] JERMAINE, C. Online random shuffling of large database tables. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 73–84.
- [15] JERMAINE, C., ARUMUGAM, S., POL, A., AND DOBRA, A. Scalable approximate query processing with the DBO engine. In *SIGMOD* (2007).
- [16] LOGOTHETIS, D., AND YOCUM, K. Ad-hoc data processing in the cloud (demonstration). *Proc. VLDB Endow.* 1, 2 (2008).
- [17] LUO, G., ELLMANN, C. J., HAAS, P. J., AND NAUGHTON, J. F. A scalable hash ripple join algorithm. In *SIGMOD* (2002).
- [18] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. In *CIDR* (2003).
- [19] OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA, U. Automatic optimization of parallel dataflow programs. In *USENIX Technical Conference* (2008).
- [20] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008).
- [21] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD* (2009).
- [22] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. Network-aware operator placement for stream-processing systems. In *ICDE* (2006).
- [23] PIKE, R., DORWARD, S., GRIESMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* 13, 4 (2005), 277–298.
- [24] SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. A. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD* (2004).
- [25] SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE* (2003).
- [26] SKOMOROCH, P. N. Wikipedia page traffic statistics, 2009. Downloaded from <http://developer.amazonwebservices.com/connect/entry.jspa?externalID=2596>.
- [27] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive — a warehousing solution over a Map-Reduce framework. In *VLDB* (2009).
- [28] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP* (2001).
- [29] WU, S., JIANG, S., OOI, B. C., AND TAN, K.-L. Distributed online aggregation. In *VLDB* (2009).
- [30] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009).
- [31] YANG, C., YEN, C., TAN, C., AND MADDEN, S. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE* (2010).

# Frustratingly Easy Domain Adaptation

**Hal Daumé III**

School of Computing

University of Utah

Salt Lake City, Utah 84112

`me@hal3.name`

## Abstract

We describe an approach to domain adaptation that is appropriate exactly in the case when one has enough “target” data to do slightly better than just using only “source” data. Our approach is incredibly simple, easy to implement as a preprocessing step (10 lines of Perl!) and outperforms state-of-the-art approaches on a range of datasets. Moreover, it is trivially extended to a multi-domain adaptation problem, where one has data from a variety of different domains.

## 1 Introduction

The task of domain adaptation is to develop learning algorithms that can be easily ported from one domain to another—say, from newswire to biomedical documents. This problem is particularly interesting in NLP because we are often in the situation that we have a large collection of labeled data in one “source” domain (say, newswire) but truly desire a model that performs well in a second “target” domain. The approach we present in this paper is based on the idea of transforming the domain adaptation learning problem into a standard supervised learning problem to which any standard algorithm may be applied (eg., maxent, SVMs, etc.). Our transformation is incredibly simple: we augment the feature space of both the source and target data and use the result as input to a standard learning algorithm.

There are roughly two varieties of the domain adaptation problem that have been addressed in the literature: the fully supervised case and the semi-

supervised case. The fully supervised case models the following scenario. We have access to a large, annotated corpus of data from a source domain. In addition, we spend a little money to annotate a small corpus in the target domain. We want to leverage both annotated datasets to obtain a model that performs well on the target domain. The semi-supervised case is similar, but instead of having a small annotated target corpus, we have a large but *unannotated* target corpus. In this paper, we focus exclusively on the fully supervised case.

One particularly nice property of our approach is that it is incredibly easy to implement: the Appendix provides a 10 line, 194 character Perl script for performing the complete transformation (available at <http://hal3.name/easyadapt.pl.gz>). In addition to this simplicity, our algorithm performs as well as (or, in some cases, better than) current state of the art techniques.

## 2 Problem Formalization and Prior Work

To facilitate discussion, we first introduce some notation. Denote by  $\mathcal{X}$  the input space (typically either a real vector or a binary vector), and by  $\mathcal{Y}$  the output space. We will write  $\mathcal{D}^s$  to denote the distribution over source examples and  $\mathcal{D}^t$  to denote the distribution over target examples. We assume access to a samples  $D^s \sim \mathcal{D}^s$  of source examples from the source domain, and samples  $D^t \sim \mathcal{D}^t$  of target examples from the target domain. We will assume that  $D^s$  is a collection of  $N$  examples and  $D^t$  is a collection of  $M$  examples (where, typically,  $N \gg M$ ). Our goal is to learn a function  $h : \mathcal{X} \rightarrow \mathcal{Y}$  with low expected loss with respect to the target domain.

For the purposes of discussion, we will suppose that  $\mathcal{X} = \mathbb{R}^F$  and that  $\mathcal{Y} = \{-1, +1\}$ . However, most of the techniques described in this section (as well as our own technique) are more general.

There are several “obvious” ways to attack the domain adaptation problem without developing new algorithms. Many of these are presented and evaluated by Daumé III and Marcu (2006).

The SRCONLY baseline ignores the target data and trains a single model, only on the source data.

The TGTONLY baseline trains a single model only on the target data.

The ALL baseline simply trains a standard learning algorithm on the union of the two datasets.

A potential problem with the ALL baseline is that if  $N \gg M$ , then  $D^s$  may “wash out” any effect  $D^t$  might have. We will discuss this problem in more detail later, but one potential solution is to re-weight examples from  $D^s$ . For instance, if  $N = 10 \times M$ , we may weight each example from the source domain by 0.1. The next baseline, WEIGHTED, is exactly this approach, with the weight chosen by cross-validation.

The PRED baseline is based on the idea of using the output of the source classifier as a feature in the target classifier. Specifically, we first train a SRCONLY model. Then we run the SRCONLY model on the target data (training, development and test). We use the predictions made by the SRCONLY model as additional features and train a second model on the target data, augmented with this new feature.

In the LININT baseline, we linearly interpolate the predictions of the SRCONLY and the TGTONLY models. The interpolation parameter is adjusted based on target development data.

These baselines are actually surprisingly difficult to beat. To date, there are two models that have successfully defeated them on a handful of datasets. The first model, which we shall refer to as the PRIOR model, was first introduced by Chelba and Acero (2004). The idea of this model is to use the SRCONLY model as a *prior* on the weights for a second model, trained on the target data. Chelba and Acero (2004) describe this approach within the context of a maximum entropy classifier, but the idea

is more general. In particular, for many learning algorithms (maxent, SVMs, averaged perceptron, naive Bayes, etc.), one *regularizes* the weight vector toward zero. In other words, all of these algorithms contain a regularization term on the weights  $w$  of the form  $\lambda \|w\|_2^2$ . In the generalized PRIOR model, we simply replace this regularization term with  $\lambda \|w - w^s\|_2^2$ , where  $w^s$  is the weight vector learned in the SRCONLY model.<sup>1</sup> In this way, the model trained on the target data “prefers” to have weights that are similar to the weights from the SRCONLY model, unless the data demands otherwise. Daumé III and Marcu (2006) provide empirical evidence on four datasets that the PRIOR model outperforms the baseline approaches.

More recently, Daumé III and Marcu (2006) presented an algorithm for domain adaptation for maximum entropy classifiers. The key idea of their approach is to learn *three* separate models. One model captures “source specific” information, one captures “target specific” information and one captures “general” information. The distinction between these three sorts of information is made on a *per-example* basis. In this way, each source example is considered either source specific or general, while each target example is considered either target specific or general. Daumé III and Marcu (2006) present an EM algorithm for training their model. This model consistently outperformed all the baseline approaches as well as the PRIOR model. Unfortunately, despite the empirical success of this algorithm, it is quite complex to implement and is roughly 10 to 15 times slower than training the PRIOR model.

### 3 Adaptation by Feature Augmentation

In this section, we describe our approach to the domain adaptation problem. Essentially, all we are going to do is take each feature in the original problem and make three versions of it: a general version, a source-specific version and a target-specific version. The augmented source data will contain only general and source-specific versions. The augmented target

---

<sup>1</sup>For the maximum entropy, SVM and naive Bayes learning algorithms, modifying the regularization term is simple because it appears explicitly. For the perceptron algorithm, one can obtain an equivalent regularization by performing standard perceptron updates, but using  $(w + w^s)^\top x$  for making predictions rather than simply  $w^\top x$ .

data contains general and target-specific versions.

To state this more formally, first recall the notation from Section 2:  $\mathcal{X}$  and  $\mathcal{Y}$  are the input and output spaces, respectively;  $D^s$  is the source domain data set and  $D^t$  is the target domain data set. Suppose for simplicity that  $\mathcal{X} = \mathbb{R}^F$  for some  $F > 0$ . We will define our augmented input space by  $\check{\mathcal{X}} = \mathbb{R}^{3F}$ . Then, define mappings  $\Phi^s, \Phi^t : \mathcal{X} \rightarrow \check{\mathcal{X}}$  for mapping the source and target data respectively. These are defined by Eq (1), where  $\mathbf{0} = \langle 0, 0, \dots, 0 \rangle \in \mathbb{R}^F$  is the zero vector.

$$\Phi^s(\mathbf{x}) = \langle \mathbf{x}, \mathbf{x}, \mathbf{0} \rangle, \quad \Phi^t(\mathbf{x}) = \langle \mathbf{x}, \mathbf{0}, \mathbf{x} \rangle \quad (1)$$

Before we proceed with a formal analysis of this transformation, let us consider why it might be expected to work. Suppose our task is part of speech tagging, our source domain is the Wall Street Journal and our target domain is a collection of reviews of computer hardware. Here, a word like “the” should be tagged as a determiner in both cases. However, a word like “monitor” is more likely to be a verb in the WSJ and more likely to be a noun in the hardware corpus. Consider a simple case where  $\mathcal{X} = \mathbb{R}^2$ , where  $x_1$  indicates if the word is “the” and  $x_2$  indicates if the word is “monitor.” Then, in  $\check{\mathcal{X}}$ ,  $\check{x}_1$  and  $\check{x}_2$  will be “general” versions of the two indicator functions,  $\check{x}_3$  and  $\check{x}_4$  will be source-specific versions, and  $\check{x}_5$  and  $\check{x}_6$  will be target-specific versions.

Now, consider what a learning algorithm could do to capture the fact that the appropriate tag for “the” remains constant across the domains, and the tag for “monitor” changes. In this case, the model can set the “determiner” weight vector to something like  $\langle 1, 0, 0, 0, 0, 0 \rangle$ . This places high weight on the common version of “the” and indicates that “the” is most likely a determiner, regardless of the domain. On the other hand, the weight vector for “noun” might look something like  $\langle 0, 0, 0, 0, 0, 1 \rangle$ , indicating that the word “monitor” is a noun *only* in the target domain. Similar, the weight vector for “verb” might look like  $\langle 0, 0, 0, 1, 0, 0 \rangle$ , indicating the “monitor” is a verb *only* in the source domain.

Note that this expansion is actually redundant. We could equally well use  $\Phi^s(\mathbf{x}) = \langle \mathbf{x}, \mathbf{x} \rangle$  and  $\Phi^t(\mathbf{x}) = \langle \mathbf{x}, \mathbf{0} \rangle$ . However, it turns out that it is easier to analyze the first case, so we will stick with

that. Moreover, the first case has the nice property that it is straightforward to generalize it to the multi-domain adaptation problem: when there are more than two domains. In general, for  $K$  domains, the augmented feature space will consist of  $K+1$  copies of the original feature space.

### 3.1 A Kernelized Version

It is straightforward to derive a kernelized version of the above approach. We do not exploit this property in our experiments—all are conducted with a simple linear kernel. However, by deriving the kernelized version, we gain some insight into the method. For this reason, we sketch the derivation here.

Suppose that the data points  $x$  are drawn from a reproducing kernel Hilbert space  $\mathcal{X}$  with kernel  $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{R}$ , with  $K$  positive semi-definite. Then,  $K$  can be written as the dot product (in  $\mathcal{X}$ ) of two (perhaps infinite-dimensional) vectors:  $K(x, x') = \langle \Phi(x), \Phi(x') \rangle_{\mathcal{X}}$ . Define  $\Phi^s$  and  $\Phi^t$  in terms of  $\Phi$ , as:

$$\begin{aligned} \Phi^s(x) &= \langle \Phi(x), \Phi(x), \mathbf{0} \rangle \\ \Phi^t(x) &= \langle \Phi(x), \mathbf{0}, \Phi(x) \rangle \end{aligned} \quad (2)$$

Now, we can compute the kernel product between  $\Phi^s$  and  $\Phi^t$  in the expanded RKHS by making use of the original kernel  $K$ . We denote the expanded kernel by  $\check{K}(x, x')$ . It is simplest to first describe  $\check{K}(x, x')$  when  $x$  and  $x'$  are from the same domain, then analyze the case when the domain differs. When the domain is the same, we get:  $\check{K}(x, x') = \langle \Phi(x), \Phi(x') \rangle_{\mathcal{X}} + \langle \Phi(x), \Phi(x') \rangle_{\mathcal{X}} = 2K(x, x')$ . When they are from different domains, we get:  $\check{K}(x, x') = \langle \Phi(x), \Phi(x') \rangle_{\mathcal{X}} = K(x, x')$ . Putting this together, we have:

$$\check{K}(x, x') = \begin{cases} 2K(x, x') & \text{same domain} \\ K(x, x') & \text{diff. domain} \end{cases} \quad (3)$$

This is an intuitively pleasing result. What it says is that—considering the kernel as a measure of similarity—data points from the same domain are “by default” twice as similar as those from different domains. Loosely speaking, this means that data points from the target domain have twice as much influence as source points when making predictions about test target data.

### 3.2 Analysis

We first note an obvious property of the feature-augmentation approach. Namely, it does not make learning harder, in a minimum Bayes error sense. A more interesting statement would be that it makes learning *easier*, along the lines of the result of (Ben-David et al., 2006) — note, however, that their results are for the “semi-supervised” domain adaptation problem and so do not apply directly. As yet, we do not know a proper formalism in which to analyze the fully supervised case.

It turns out that the feature-augmentation method is remarkably similar to the PRIOR model<sup>2</sup>. Suppose we learn feature-augmented weights in a classifier regularized by an  $\ell_2$  norm (eg., SVMs, maximum entropy). We can denote by  $w_s$  the sum of the “source” and “general” components of the learned weight vector, and by  $w_t$  the sum of the “target” and “general” components, so that  $w_s$  and  $w_t$  are the predictive weights for each task. Then, the regularization condition on the entire weight vector is approximately  $\|w_g\|^2 + \|w_s - w_g\|^2 + \|w_t - w_g\|^2$ , with free parameter  $w_g$  which can be chosen to minimize this sum. This leads to a regularizer proportional to  $\|w_s - w_t\|^2$ , akin to the PRIOR model.

Given this similarity between the feature-augmentation method and the PRIOR model, one might wonder why we expect our approach to do better. Our belief is that this occurs because we optimize  $w_s$  and  $w_t$  *jointly*, not sequentially. First, this means that we do not need to cross-validate to estimate good hyperparameters for each task (though in our experiments, we do not use any hyperparameters). Second, and more importantly, this means that the single supervised learning algorithm that is run is allowed to regulate the trade-off between source/target and general weights. In the PRIOR model, we are forced to use the prior variance on in the target learning scenario to do this ourselves.

### 3.3 Multi-domain adaptation

Our formulation is agnostic to the number of “source” domains. In particular, it may be the case that the source data actually falls into a variety of more specific domains. This is simple to account for in our model. In the two-domain case, we ex-

panded the feature space from  $\mathbb{R}^F$  to  $\mathbb{R}^{3F}$ . For a  $K$ -domain problem, we simply expand the feature space to  $\mathbb{R}^{(K+1)F}$  in the obvious way (the “+1” corresponds to the “general domain” while each of the other  $1 \dots K$  correspond to a single task).

## 4 Results

In this section we describe experimental results on a wide variety of domains. First we describe the tasks, then we present experimental results, and finally we look more closely at a few of the experiments.

### 4.1 Tasks

All tasks we consider are sequence labeling tasks (either named-entity recognition, shallow parsing or part-of-speech tagging) on the following datasets:

**ACE-NER.** We use data from the 2005 Automatic Content Extraction task, restricting ourselves to the named-entity recognition task. The 2005 ACE data comes from 5 domains: Broadcast News (bn), Broadcast Conversations (bc), Newswire (nw), Weblog (wl), Usenet (un) and Conversational Telephone Speech (cts).

**CoNLL-NE.** Similar to ACE-NER, a named-entity recognition task. The difference is: we use the 2006 ACE data as the source domain and the CoNLL 2003 NER data as the target domain.

**PubMed-POS.** A part-of-speech tagging problem on PubMed abstracts introduced by Blitzer et al. (2006). There are two domains: the source domain is the WSJ portion of the Penn Treebank and the target domain is PubMed.

**CNN-Recap.** This is a recapitalization task introduced by Chelba and Acero (2004) and also used by Daumé III and Marcu (2006). The source domain is newswire and the target domain is the output of an ASR system.

**Treebank-Chunk.** This is a shallow parsing task based on the data from the Penn Treebank. This data comes from a variety of domains: the standard WSJ domain (we use the same data as for CoNLL 2000), the ATIS switchboard domain, and the Brown corpus (which is, itself, assembled from six subdomains).

**Treebank-Brown.** This is identical to the Treebank-Chunk task, except that we consider all of the Brown corpus to be a single domain.

<sup>2</sup>Thanks an anonymous reviewer for pointing this out!

Task	Dom	# Tr	# De	# Te	# Ft
ACE-NER	bn	52,998	6,625	6,626	80k
	bc	38,073	4,759	4,761	109k
	nw	44,364	5,546	5,547	113k
	wl	35,883	4,485	4,487	109k
	un	35,083	4,385	4,387	96k
	cts	39,677	4,960	4,961	54k
CoNLL-NER	src	256,145	-	-	368k
	tgt	29,791	5,258	8,806	88k
PubMed-POS	src	950,028	-	-	571k
	tgt	11,264	1,987	14,554	39k
CNN-Recap	src	2,000,000	-	-	368k
	tgt	39,684	7,003	8,075	88k
Treebank-Chunk	wsj	191,209	29,455	38,440	94k
	swbd3	45,282	5,596	41,840	55k
	br-cf	58,201	8,307	7,607	144k
	br-cg	67,429	9,444	6,897	149k
	br-ck	51,379	6,061	9,451	121k
	br-cl	47,382	5,101	5,880	95k
	br-cm	11,696	1,324	1,594	51k
	br-cn	56,057	6,751	7,847	115k
	br-cp	55,318	7,477	5,977	112k
	br-cr	16,742	2,522	2,712	65k

Table 1: Task statistics; columns are task, domain, size of the training, development and test sets, and the number of unique features in the training set.

In all cases (except for CNN-Recap), we use roughly the same feature set, which has become somewhat standardized: lexical information (words, stems, capitalization, prefixes and suffixes), membership on gazetteers, etc. For the CNN-Recap task, we use identical feature to those used by both Chelba and Acero (2004) and Daumé III and Marcu (2006): the current, previous and next word, and 1-3 letter prefixes and suffixes.

Statistics on the tasks and datasets are in Table 1.

In all cases, we use the SEARN algorithm for solving the sequence labeling problem (Daumé III et al., 2007) with an underlying averaged perceptron classifier; implementation due to (Daumé III, 2004). For structural features, we make a second-order Markov assumption and only place a bias feature on the transitions. For simplicity, we optimize and report only on label accuracy (but require that our outputs be parsimonious: we do not allow “I-NP” to follow “B-PP,” for instance). We do this for three reasons. First, our focus in this work is on building better learning algorithms and introducing a more complicated measure only serves to mask these effects. Second, it is arguable that a measure like  $F_1$  is inappropriate for chunking tasks (Manning, 2006).

Third, we can easily compute statistical significance over accuracies using McNemar’s test.

## 4.2 Experimental Results

The full—somewhat daunting—table of results is presented in Table 2. The first two columns specify the task and domain. For the tasks with only a single source and target, we simply report results on the target. For the multi-domain adaptation tasks, we report results for each setting of the target (where all other data-sets are used as different “source” domains). The next set of eight columns are the *error rates* for the task, using one of the different techniques (“AUGMENT” is our proposed technique). For each row, the error rate of the best performing technique is bolded (as are all techniques whose performance is not statistically significantly different at the 95% level). The “T<S” column is contains a “+” whenever TGTONLY outperforms SRCONLY (this will become important shortly). The final column indicates when AUGMENT comes in first.<sup>3</sup>

There are several trends to note in the results. Excluding for a moment the “br-\*” domains on the Treebank-Chunk task, our technique always performs best. Still excluding “br-\*”, the clear second-place contestant is the PRIOR model, a finding consistent with prior research. When we repeat the Treebank-Chunk task, but lumping all of the “br-\*” data together into a single “brown” domain, the story reverts to what we expected before: our algorithm performs best, followed by the PRIOR method.

Importantly, this simple story breaks down on the Treebank-Chunk task for the eight sections of the Brown corpus. For these, our AUGMENT technique performs rather poorly. Moreover, there is no clear winning approach on this task. Our hypothesis is that the common feature of these examples is that these are exactly the tasks for which SRCONLY outperforms TGTONLY (with one exception: CoNLL). This seems like a plausible explanation, since it implies that the source and target domains may not be that different. If the domains are so similar that a large amount of source data outperforms a small amount of target data, then it is unlikely that blow-

---

<sup>3</sup>One advantage of using the averaged perceptron for all experiments is that the only tunable hyperparameter is the number of iterations. In all cases, we run 20 iterations and choose the one with the lowest error on development data.

Task	Dom	SRCONLY	TGTONLY	ALL	WEIGHT	PRED	LININT	PRIOR	AUGMENT	T<S Win
ACE-NER	bn	4.98	2.37	2.29	2.23	2.11	2.21	2.06	<b>1.98</b>	+
	bc	4.54	4.07	3.55	3.53	3.89	4.01	<b>3.47</b>	<b>3.47</b>	+
	nw	4.78	3.71	3.86	3.65	3.56	3.79	3.68	<b>3.39</b>	+
	wl	2.45	2.45	<b>2.12</b>	<b>2.12</b>	2.45	2.33	2.41	<b>2.12</b>	=
	un	3.67	2.46	2.48	2.40	2.18	2.10	2.03	<b>1.91</b>	+
	cts	2.08	0.46	0.40	0.40	0.46	0.44	<b>0.34</b>	<b>0.32</b>	+
	CoNLL	tgt	2.49	2.95	1.80	<b>1.75</b>	2.13	<b>1.77</b>	1.89	<b>1.76</b>
PubMed	tgt	12.02	4.15	5.43	4.15	4.14	3.95	3.99	<b>3.61</b>	+
CNN	tgt	10.29	3.82	3.67	3.45	3.46	3.44	<b>3.35</b>	<b>3.37</b>	+
Treebank-chunk	wsj	6.63	4.35	4.33	4.30	4.32	4.32	4.27	<b>4.11</b>	+
	swbd3	15.90	4.15	4.50	4.10	4.13	4.09	3.60	<b>3.51</b>	+
	br-cf	5.16	6.27	4.85	4.80	4.78	<b>4.72</b>	5.22	5.15	
	br-cg	4.32	5.36	<b>4.16</b>	<b>4.15</b>	4.27	4.30	4.25	4.90	
	br-ck	5.05	6.32	5.05	4.98	<b>5.01</b>	<b>5.05</b>	5.27	5.41	
	br-cl	5.66	6.60	5.42	<b>5.39</b>	<b>5.39</b>	5.53	5.99	5.73	
	br-cm	3.57	6.59	<b>3.14</b>	<b>3.11</b>	3.15	3.31	4.08	4.89	
	br-cn	4.60	5.56	4.27	4.22	<b>4.20</b>	<b>4.19</b>	4.48	4.42	
	br-cp	4.82	5.62	4.63	<b>4.57</b>	<b>4.55</b>	<b>4.55</b>	4.87	4.78	
	br-cr	5.78	9.13	5.71	5.19	5.20	<b>5.15</b>	6.71	6.30	
Treebank-brown		6.35	5.75	4.80	4.75	4.81	4.72	4.72	<b>4.65</b>	+

Table 2: Task results.

ing up the feature space will help.

We additionally ran the MEGAM model (Daumé III and Marcu, 2006) on these data (though not in the multi-conditional case; for this, we considered the single source as the union of all sources). The results are not displayed in Table 2 to save space. For the majority of results, MEGAM performed roughly comparably to the best of the systems in the table. In particular, it was not statistically significantly different that AUGMENT on: ACE-NER, CoNLL, PubMed, Treebank-chunk-wsj, Treebank-chunk-swbd3, CNN and Treebank-brown. It did outperform AUGMENT on the Treebank-chunk on the Treebank-chunk-br-\* data sets, but only outperformed the best other model on these data sets for br-cg, br-cm and br-cp. However, despite its advantages on these data sets, it was quite significantly slower to train: a single run required about ten times longer than any of the other models (including AUGMENT), and also required five-to-ten iterations of cross-validation to tune its hyperparameters so as to achieve these results.

### 4.3 Model Introspection

One explanation of our model’s improved performance is simply that by augmenting the feature space, we are creating a more powerful model. While this may be a partial explanation, here we show that what the model learns about the various

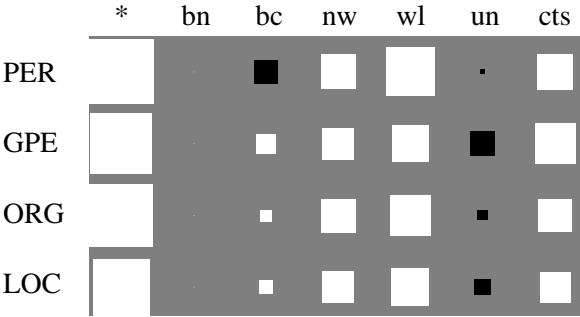


Figure 1: Hinton diagram for feature /Aa+/ at current position.

domains actually makes some plausible sense.

We perform this analysis only on the ACE-NER data by looking specifically at the learned weights. That is, for any given feature  $f$ , there will be seven versions of  $f$ : one corresponding to the “cross-domain”  $f$  and seven corresponding to each domain. We visualize these weights, using Hinton diagrams, to see how the weights vary across domains.

For example, consider the feature “current word has an initial capital letter and is then followed by one or more lower-case letters.” This feature is presumably useless for data that lacks capitalization information, but potentially quite useful for other domains. In Figure 1 we show a Hinton diagram for this figure. Each column in this figure corresponds to a domain (the top row is the “general domain”).

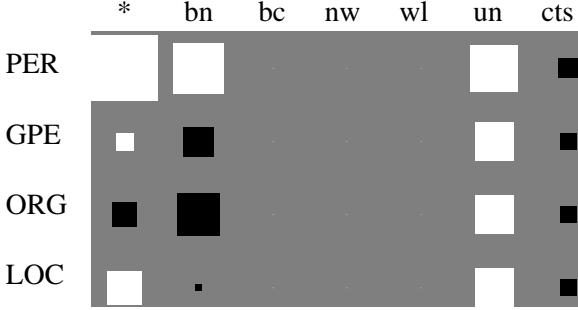


Figure 2: Hinton diagram for feature /bush/ at current position.

Each row corresponds to a class.<sup>4</sup> Black boxes correspond to negative weights and white boxes correspond to positive weights. The size of the box depicts the absolute value of the weight.

As we can see from Figure 1, the /Aa+/ feature is a very good indicator of entity-hood (it’s value is strongly positive for all four entity classes), regardless of domain (i.e., for the “\*” domain). The lack of boxes in the “bn” column means that, beyond the settings in “\*”, the broadcast news is agnostic with respect to this feature. This makes sense: there is no capitalization in broadcast news domain, so there would be no sense in setting these weights to anything by zero. The usenet column is filled with negative weights. While this may seem strange, it is due to the fact that many email addresses and URLs match this pattern, but are not entities.

Figure 2 depicts a similar figure for the feature “word is ‘bush’ at the current position” (this figure is case sensitive).<sup>5</sup> These weights are somewhat harder to interpret. What is happening is that “by default” the word “bush” is going to be a person—this is because it rarely appears referring to a plant and so even in the capitalized domains like broadcast conversations, if it appears at all, it is a person. The exception is that in the conversations data, people *do* actually talk about bushes as plants, and so the weights are set accordingly. The weights are high in the usenet domain because people tend to talk about the president without capitalizing his name.

<sup>4</sup>Technically there are many more classes than are shown here. We do not depict the smallest classes, and have merged the ‘Begin-\*’ and ‘In-\*’ weights for each entity type.

<sup>5</sup>The scale of weights across features is *not* comparable, so do not try to compare Figure 1 with Figure 2.

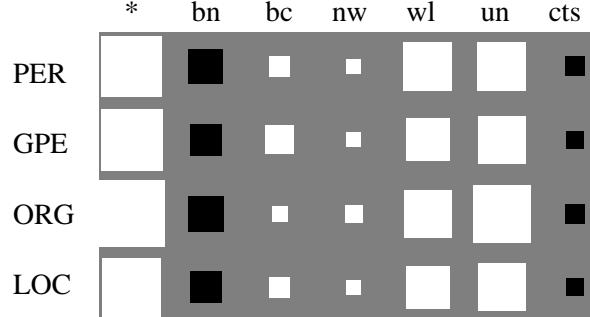


Figure 3: Hinton diagram for feature /the/ at current position.

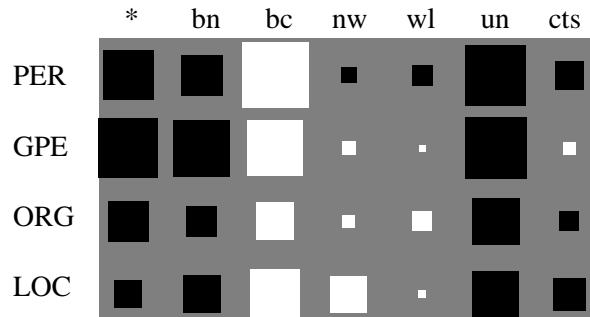


Figure 4: Hinton diagram for feature /the/ at previous position.

Figure 3 presents the Hinton diagram for the feature “word at the current position is ‘the’” (again, case-sensitive). In general, it appears, “the” is a common word in entities in all domain except for broadcast news and conversations. These exceptions crop up because of the capitalization issue.

In Figure 4, we show the diagram for the feature “previous word is ‘the’.” The only domain for which this is a good feature of entity-hood is broadcast conversations (to a much lesser extent, newswire). This occurs because of four phrases very common in the broadcast conversations and rare elsewhere: “the Iraqi people” (“Iraqi” is a GPE), “the Pentagon” (an ORG), “the Bush (cabinet|advisors|...)” (PER), and “the South” (LOC).

Finally, Figure 5 shows the Hinton diagram for the feature “the current word is on a list of common names” (this feature is case-insensitive). All around, this is a good feature for picking out people and nothing else. The two exceptions are: it is also a good feature for other entity types for broadcast

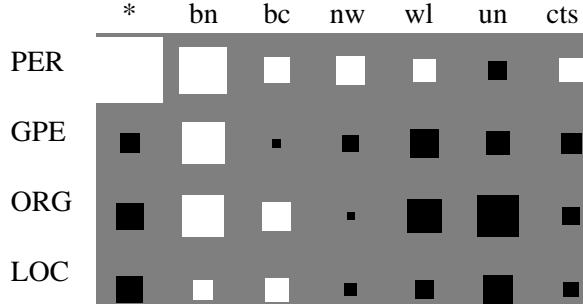


Figure 5: Hinton diagram for membership on a list of names at current position.

news and it is not quite so good for people in usenet. The first is easily explained: in broadcast news, it is very common to refer to countries and organizations by the name of their respective leaders. This is essentially a metonymy issue, but as the data is annotated, these are marked by their true referent. For usenet, it is because the list of names comes from news data, but usenet names are more diverse.

In general, the weights depict for these features make some intuitive sense (in as much as weights for any learned algorithm make intuitive sense). It is particularly interesting to note that while there are some regularities to the patterns in the five diagrams, it is definitely *not* the case that there are, eg., two domains that behave identically across all features. This supports the hypothesis that the reason our algorithm works so well on this data is because the domains are actually quite well separated.

## 5 Discussion

In this paper we have described an *incredibly* simple approach to domain adaptation that—under a common and easy-to-verify condition—outperforms previous approaches. While it is somewhat frustrating that something so simple does so well, it is perhaps not surprising. By augmenting the feature space, we are essentially forcing the learning algorithm to do the adaptation for us. Good supervised learning algorithms have been developed over decades, and so we are essentially just leveraging all that previous work. Our hope is that this approach is so simple that it can be used for many more real-world tasks than we have presented here with little effort. Finally, it is very interesting to note that using our method, shallow parsing error rate on the

CoNLL section of the treebank improves from 5.35 to 5.11. While this improvement is small, it is real, and may carry over to full parsing. The most important avenue of future work is to develop a formal framework under which we can analyze this (and other supervised domain adaptation models) theoretically. Currently our results only state that this augmentation procedure doesn’t make the learning harder — we would like to know that it actually makes it easier. An additional future direction is to explore the kernelization interpretation further: why should we use 2 as the “similarity” between domains—we could introduce a hyperparameter  $\alpha$  that indicates the similarity between domains and could be tuned via cross-validation.

**Acknowledgments.** We thank the three anonymous reviewers, as well as Ryan McDonald and John Blitzer for very helpful comments and insights.

## References

- Shai Ben-David, John Blitzer, Koby Crammer, and Fernando Pereira. 2006. Analysis of representations for domain adaptation. In *Advances in Neural Information Processing Systems (NIPS)*.
- John Blitzer, Ryan McDonald, and Fernando Pereira. 2006. Domain adaptation with structural correspondence learning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- Ciprian Chelba and Alex Acero. 2004. Adaptation of maximum entropy classifier: Little data can help a lot. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Barcelona, Spain.
- Hal Daumé III and Daniel Marcu. 2006. Domain adaptation for statistical classifiers. *Journal of Artificial Intelligence Research*, 26.
- Hal Daumé III, John Langford, and Daniel Marcu. 2007. Search-based structured prediction. *Machine Learning Journal (submitted)*.
- Hal Daumé III. 2004. Notes on CG and LM-BFGS optimization of logistic regression. Paper available at <http://pub.hal3.name/#daume04cg-bfgs>, implementation available at <http://hal3.name/megam/>, August.
- Christopher Manning. 2006. Doing named entity recognition? Don’t optimize for  $F_1$ . Post on the NLPers Blog, 25 August. <http://nlpers.blogspot.com/2006/08/doing-named-entity-recognition-dont.html>.

# The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google\*

## ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

## Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

## General Terms

Design, reliability, performance, measurement

## Keywords

Fault tolerance, scalability, data storage, clustered storage

---

\*The authors can be reached at the following addresses:  
{sanjay,hgobioff,shuntak}@google.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

Copyright 2003 ACM 1-58113-757-5/03/0010 ...\$5.00.

## 1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

For example, we have relaxed GFS’s consistency model to vastly simplify the file system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

## 2. DESIGN OVERVIEW

### 2.1 Assumptions

In designing a file system for our needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

- The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.
- The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Our files are often used as producer-consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.
- High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

### 2.2 Interface

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* files.

Moreover, GFS has *snapshot* and *record append* operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client’s append. It is useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications. Snapshot and record append are discussed further in Sections 3.4 and 3.3 respectively.

### 2.3 Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserv in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserv caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux’s buffer cache already keeps frequently accessed data in memory.

### 2.4 Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placement

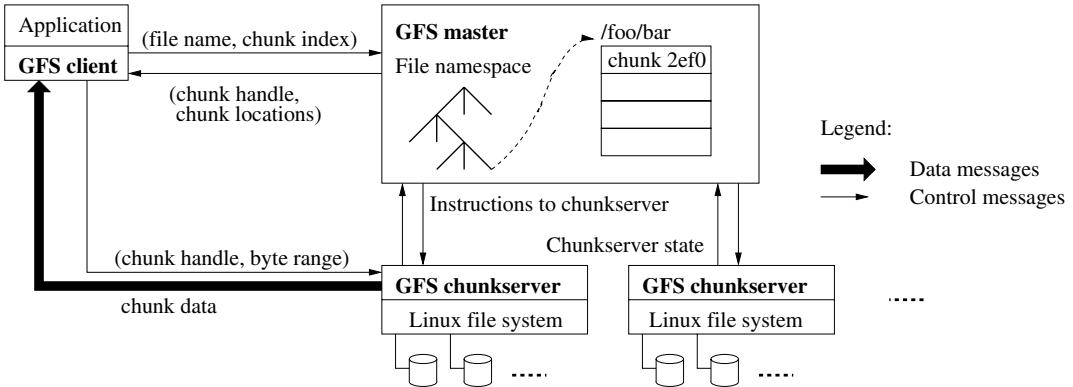


Figure 1: GFS Architecture

and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with reference to Figure 1. First, using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file. Then, it sends the master a request containing the file name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key.

The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost.

## 2.5 Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserv and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persis-

tent TCP connection to the chunkserv over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservs storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few chunkservs storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batch-queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

## 2.6 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the master's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserv about its chunks at master startup and whenever a chunkserv joins the cluster.

### 2.6.1 In-Memory Data Structures

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserv failures, and chunk migration to balance load and disk space

usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less than 64 bytes per file because it stores file names compactly using prefix compression.

If necessary to support even larger file systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

### 2.6.2 Chunk Locations

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to realize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

### 2.6.3 Operation Log

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the

	Write	Record Append
Serial success	<i>defined</i>	<i>defined</i> interspersed with <i>inconsistent</i>
Concurrent successes	<i>consistent</i> but <i>undefined</i>	
Failure		<i>inconsistent</i>

Table 1: File Region State After Mutation

limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without extra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the master’s internal state is structured in such a way that a new checkpoint can be created without delaying incoming mutations. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

## 2.7 Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS’s guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

### 2.7.1 Guarantees by GFS

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master’s operation log defines a global total order of these operations (Section 2.6.3).

The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. We describe below how our applications can distinguish defined regions from undefined

regions. The applications do not need to further distinguish between different kinds of undefined regions.

Data mutations may be *writes* or *record appends*. A write causes data to be written at an application-specified file offset. A record append causes data (the “record”) to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS’s choosing (Section 3.3). (In contrast, a “regular” append is merely a write at an offset that the client believes to be the current end of file.) The offset is returned to the client and marks the beginning of a defined region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This window is limited by the cache entry’s timeout and the next open of the file, which purges from the cache all chunk information for that file. Moreover, as most of our files are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

### 2.7.2 Implications for Applications

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. In one typical use, a writer generates a file from beginning to end. It atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. Checkpoints may also include application-level checksums. Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more efficient and more resilient to application failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written

file data that is still incomplete from the application’s perspective.

In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append’s append-at-least-once semantics preserves each writer’s output. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other file interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

## 3. SYSTEM INTERACTIONS

We designed the system to minimize the master’s involvement in all operations. With that background, we now describe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot.

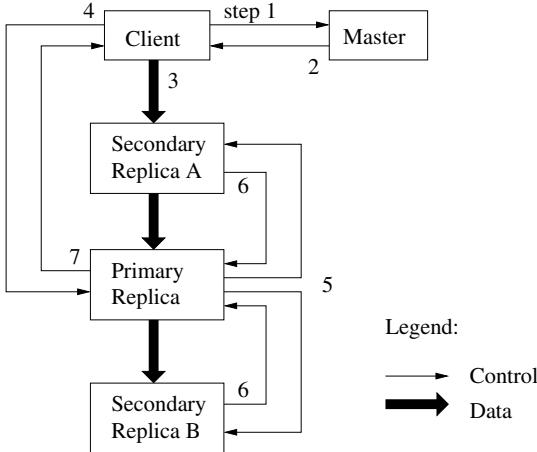
### 3.1 Leases and Mutation Order

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk’s replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary*. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the *HeartBeat* messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

In Figure 2, we illustrate this process by following the control flow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary



**Figure 2: Write Control and Data Flow**

- becomes unreachable or replies that it no longer holds a lease.
3. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
  4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
  5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
  6. The secondaries all reply to the primary indicating that they have completed the operation.
  7. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control flow described above but may be interleaved with and overwritten by concurrent operations from other clients. Therefore, the shared

file region may end up containing fragments from different clients, although the replicas will be identical because the individual operations are completed successfully in the same order on all replicas. This leaves the file region in consistent but undefined state as noted in Section 2.7.

### 3.2 Data Flow

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine’s network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine’s network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine’s full outbound bandwidth is used to transfer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the “closest” machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 forwards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that “distances” can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data transfer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is especially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring  $B$  bytes to  $R$  replicas is  $B/T + RL$  where  $T$  is the network throughput and  $L$  is latency to transfer bytes between two machines. Our network links are typically 100 Mbps ( $T$ ), and  $L$  is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms.

### 3.3 Atomic Record Appends

GFS provides an atomic append operation called *record append*. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS’s choosing and returns that offset to the client. This is similar to writing to a file opened in `O_APPEND` mode in Unix without the race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applications in which many clients on different machines append to the same file concurrently. Clients would need additional complicated and expensive synchronization, for example through a distributed lock manager, if they do so with traditional writes. In our workloads, such files often

serve as multiple-producer/single-consumer queues or contain merged results from many different clients.

Record append is a kind of mutation and follows the control flow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file. Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst-case fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary. In terms of our consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

### 3.4 Snapshot

The snapshot operation makes a copy of a file or a directory tree (the “source”) almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder. This will give the master an opportunity to create a new copy of the chunk first.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

The first time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to find the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk

handle C’. It then asks each chunkserver that has a current replica of C to create a new chunk called C’. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the network (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no different from that for any chunk: the master grants one of the replicas a lease on the new chunk C’ and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

## 4. MASTER OPERATION

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now discuss each of these topics.

### 4.1 Namespace Management and Locking

Many master operations can take a long time: for example, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e., hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Each master operation acquires a set of locks before it runs. Typically, if it involves /d1/d2/.../dn/leaf, it will acquire read-locks on the directory names /d1, /d1/d2, ..., /d1/d2/.../dn, and either a read lock or a write lock on the full pathname /d1/d2/.../dn/leaf. Note that leaf may be a file or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a file /home/user/foo from being created while /home/user is being snapshotted to /save/user. The snapshot operation acquires read locks on /home and /save, and write locks on /home/user and /save/user. The file creation acquires read locks on /home and /home/user, and a write lock on /home/user/foo. The two operations will be serialized properly because they try to obtain conflicting locks on /home/user. File creation does not require a write lock on the parent directory because there is no “directory”, or *inode*-like, data structure to be protected from modification. The read lock on the name is sufficient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snapshotted. The write locks on

file names serialize attempts to create a file with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level.

## 4.2 Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.

## 4.3 Creation, Re-replication, Rebalancing

Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of “recent” creations on each chunkserver. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one. In addition, we prefer to first re-replicate chunks for live files as opposed to chunks that belong to recently deleted files (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and “clones” it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebala*nces replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

## 4.4 Garbage Collection

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. We find that this approach makes the system much simpler and more reliable.

### 4.4.1 Mechanism

When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master’s regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its in-memory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master’s metadata. The chunkserver is free to delete its replicas of such chunks.

### 4.4.2 Discussion

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the file-to-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux files under designated directories on each chunkserver. Any such replica not known to the master is “garbage.”

The garbage collection approach to storage reclamation offers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may succeed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion messages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver's. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of namespaces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can respond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user effort to fine tune usage when storage is tight. Applications that repeatedly create and delete temporary files may not be able to reuse the storage right away. We address these issues by expediting storage reclamation if a deleted file is explicitly deleted again. We also allow users to apply different replication and reclamation policies to different parts of the namespace. For example, users can specify that all the chunks in the files within some directory tree are to be stored without replication, and any deleted files are immediately and irrevocably removed from the file system state.

## 4.5 Stale Replica Detection

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notified and therefore before it can start writing to the chunk. If another replica is currently unavailable, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the master assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver verifies the version number when it performs the operation so that it is always accessing up-to-date data.

## 5. FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and

quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose problems when they inevitably occur.

## 5.1 High Availability

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet effective strategies: fast recovery and replication.

### 5.1.1 Fast Recovery

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Section 6.2.2 reports observed startup times.

### 5.1.2 Chunk Replication

As discussed earlier, each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redundancy such as parity or erasure codes for our increasing read-only storage requirements. We expect that it is challenging but manageable to implement these more complicated redundancy schemes in our very loosely coupled system because our traffic is dominated by appends and reads rather than small random writes.

### 5.1.3 Master Replication

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master (e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, “shadow” masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since file content is read from chunkservers, applications do not observe stale file content. What could be

stale within short windows is file metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary's decisions to create and delete replicas.

## 5.2 Data Integrity

Each chunkserv uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guarantee identical replicas. Therefore, each chunkserv must independently verify the integrity of its own copy by maintaining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserv verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserv. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserv returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserv that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserv are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and

finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

## 5.3 Diagnostic Tools

Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate diagnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

# 6 MEASUREMENTS

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

## 6.1 Micro-benchmarks

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

### 6.1.1 Reads

$N$  clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buffer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for  $N$  clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The efficiency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

### 6.1.2 Writes

$N$  clients write simultaneously to  $N$  distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or 2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three different replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered by the system to a large number of clients.

### 6.1.3 Record Appends

Figure 3(c) shows record append performance.  $N$  clients append simultaneously to a single file. Performance is limited by the network bandwidth of the chunkservers that store the last chunk of the file, independent of the number of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by different clients.

Our applications tend to produce multiple such files concurrently. In other words,  $N$  clients append to  $M$  shared files simultaneously where both  $N$  and  $M$  are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a significant issue in practice because a client can make progress on writing one file while the chunkservers for another file are busy.

## 6.2 Real World Clusters

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

**Table 2: Characteristics of two GFS clusters**

longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single “task” consists of many processes on many machines reading and writing many files simultaneously.

### 6.2.1 Storage

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. “Used space” includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

### 6.2.2 Metadata

The chunkservers in aggregate store tens of GBs of metadata, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.

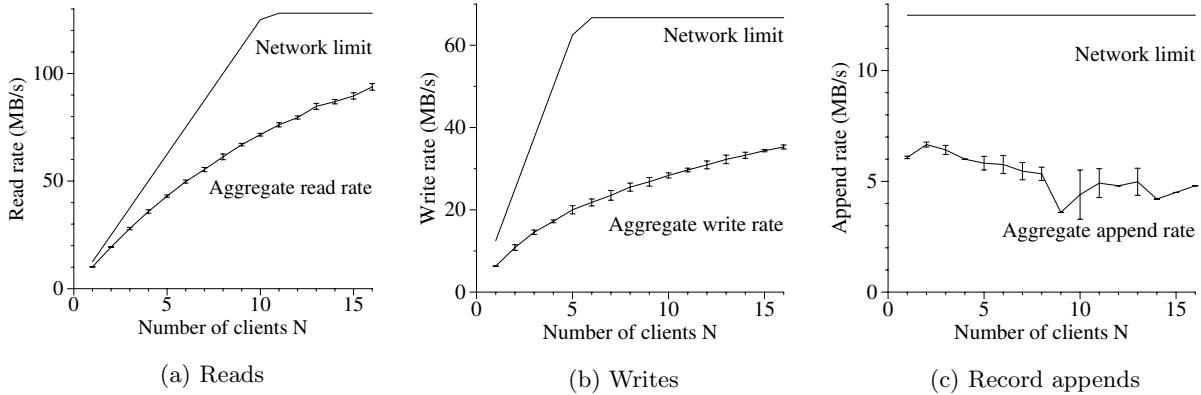
The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master’s memory does not limit the system’s capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file ownership and permissions, mapping from files to chunks, and each chunk’s current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

### 6.2.3 Read and Write Rates

Table 3 shows read and write rates for various time periods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.



**Figure 3: Aggregate Throughputs.** Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

**Table 3: Performance Metrics for Two GFS Clusters**

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network configuration can support 750 MB/s, so it was using its resources efficiently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

#### 6.2.4 Master Load

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which contained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.

#### 6.2.5 Recovery Time

After a chunkserver fails, some chunks will become under-replicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about

15,000 chunks containing 600 GB of data. To limit the impact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent cloning (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps). All chunks were restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

### 6.3 Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identical to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

#### 6.3.1 Methodology and Caveats

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include inter-server requests to carry out client requests or internal background activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Explicit logging by applications might have provided slightly more accurate data, but it is logically impossible to recompile and restart thousands of running clients to do so and cumbersome to collect the results from as many machines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual influence may also exist between general applications

Operation	Read		Write		Record Append		
	Cluster	X	Y	X	Y	X	Y
0K		0.4	2.6	0	0	0	0
1B..1K		0.1	4.1	6.6	4.9	0.2	9.2
1K..8K		65.2	38.5	0.4	1.0	18.9	15.2
8K..64K		29.9	45.1	17.8	43.0	78.0	2.8
64K..128K		0.1	0.7	2.3	1.9	< .1	4.3
128K..256K		0.2	0.3	31.6	0.4	< .1	10.6
256K..512K		0.1	0.1	4.2	7.7	< .1	31.2
512K..1M		3.9	6.9	35.5	28.7	2.2	25.5
1M..inf		0.1	1.8	1.5	12.3	0.7	2.2

**Table 4: Operations Breakdown by Size (%)**. For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

and file systems, but the effect is likely more pronounced in our case.

### 6.3.2 Chunkserver Workload

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (under 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads (over 512 KB) come from long sequential reads through entire files.

A significant number of reads return no data at all in cluster Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Producers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buffering within the writers. Writers that buffer less data, checkpoint or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does because our production systems, which use cluster Y, are more aggressively tuned for GFS.

Table 5 shows the total amount of data transferred in operations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but significant portion of the read data because of the random seek workload.

### 6.3.3 Appends versus Writes

Record appends are heavily used especially in our production systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ratios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two applications with particular buffer size choices.

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This ap-

Operation	Read		Write		Record Append		
	Cluster	X	Y	X	Y	X	Y
1B..1K		< .1	< .1	< .1	< .1	< .1	< .1
1K..8K		13.8	3.9	< .1	< .1	< .1	0.1
8K..64K		11.4	9.3	2.4	5.9	2.3	0.3
64K..128K		0.3	0.7	0.3	0.3	22.7	1.2
128K..256K		0.8	0.6	16.5	0.2	< .1	5.8
256K..512K		1.4	0.3	3.4	7.7	< .1	38.4
512K..1M		65.9	55.1	74.1	58.0	.1	46.8
1M..inf		6.4	30.1	3.3	28.0	53.9	7.4

**Table 5: Bytes Transferred Breakdown by Operation Size (%)**. For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

**Table 6: Master Requests Breakdown by Type (%)**

proximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

### 6.3.4 Master Workload

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLocation*) for reads and lease holder information (*FindLeaseLocker*) for data mutations.

Clusters X and Y see significantly different numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this difference is further hidden in the difference in *Open* requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode “w” in Unix open terminology).

*FindMatchingFiles* is a pattern matching request that supports “ls” and similar file system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global application state. In contrast, cluster X’s applications are under more explicit user control and usually know the names of all needed files in advance.

## 7. EXPERIENCES

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of `fsync()`. Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an `mmap()` call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing `mmap()` with `pread()` at the cost of an extra copy.

Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

## 8. RELATED WORK

Like other large distributed file systems such as AFS [5], GFS provides a location independent namespace which enables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a file's data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS currently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minnesota's GFS[11] and GPFS [10] remove the centralized server

and rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to provide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota's GFS and NASD, we do not seek to alter the model of the storage device. We focus on addressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues distributed across machines and careful data flow control, GFS uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n distributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues efficiently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

## 9. CONCLUSIONS

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness.

We started by reexamining traditional file system assumptions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system.

Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserv-

failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

## ACKNOWLEDGMENTS

We wish to thank the following people for their contributions to the system or the paper. Brian Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David des-Jardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

## REFERENCES

- [1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.
- [2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.
- [3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.
- [4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.
- [5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [6] InterMezzo. <http://www.inter-mezzo.org>, 2003.
- [7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.
- [8] Lustre. <http://www.lustreorg>, 2003.
- [9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.
- [10] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.
- [11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.
- [12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237, Saint-Malo, France, October 1997.

# Cassandra - A Decentralized Structured Storage System

Avinash Lakshman  
Facebook

Prashant Malik  
Facebook

## ABSTRACT

Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different data centers). At this scale, small and large components fail continuously. The way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

## 1. INTRODUCTION

Facebook runs the largest social networking platform that serves hundreds of millions users at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Facebook's platform in terms of performance, reliability and efficiency, and to support *continuous growth* the platform needs to be highly scalable. Dealing with failures in an infrastructure comprised of thousands of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such, the software systems need to be constructed in a manner that treats failures as the norm rather than the exception. To meet the reliability and scalability needs described above Facebook has developed Cassandra.

Cassandra uses a synthesis of well known techniques to achieve scalability and availability. Cassandra was designed to fulfill the storage needs of the Inbox Search problem. In-

box Search is a feature that enables users to search through their Facebook Inbox. At Facebook this meant the system was required to handle a very high write throughput, billions of writes per day, and also scale with the number of users. Since users are served from data centers that are geographically distributed, being able to replicate data across data centers was key to keep search latencies down. Inbox Search was launched in June of 2008 for around 100 million users and today we are at over 250 million users and Cassandra has kept up the promise so far. Cassandra is now deployed as the backend storage system for multiple services within Facebook.

This paper is structured as follows. Section 2 talks about related work, some of which has been very influential on our design. Section 3 presents the data model in more detail. Section 4 presents the overview of the client API. Section 5 presents the system design and the distributed algorithms that make Cassandra work. Section 6 details the experiences of making Cassandra work and refinements to improve performance. In Section 6.1 we describe how one of the applications in the Facebook platform uses Cassandra. Finally Section 7 concludes with future work on Cassandra.

## 2. RELATED WORK

Distributing data for performance, availability and durability has been widely studied in the file system and database communities. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus[14] and Coda[16] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. Farsite[2] is a distributed file system that does not use any centralized server. Farsite achieves high availability and scalability using replication. The Google File System (GFS)[9] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunk servers. However the GFS master is now made fault tolerant using the Chubby[3] abstraction. Bayou[18] is a distributed relational database system that allows disconnected operations and provides eventual data consistency. Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXXX-XX-X/XX/XX ...\$10.00.

resolution. All of them however, guarantee eventual consistency. Similar to these systems, Dynamo[6] allows read and write operations to continue even during network partitions and resolves update conflicts using different conflict resolution mechanisms, some client driven. Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency of replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [10]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

Dynamo[6] is a storage system that is used by Amazon to store and retrieve user shopping carts. Dynamo's Gossip based membership algorithm helps every node maintain information about every other node. Dynamo can be defined as a structured overlay with at most one-hop request routing. Dynamo detects updated conflicts using a vector clock scheme, but prefers a client side conflict resolution mechanism. A write operation in Dynamo also requires a read to be performed for managing the vector timestamps. This is can be very limiting in environments where systems need to handle a very high write throughput. Bigtable[4] provides both structure and data distribution but relies on a distributed file system for its durability.

### 3. DATA MODEL

A table in Cassandra is a distributed multi dimensional map indexed by a key. The value is an object which is highly structured. The row key in a table is a string with no size restrictions, although typically 16 to 36 bytes long. Every operation under a single row key is atomic per replica no matter how many columns are being read or written into. Columns are grouped together into sets called column families very much similar to what happens in the Bigtable[4] system. Cassandra exposes two kinds of columns families, Simple and Super column families. Super column families can be visualized as a column family within a column family.

Furthermore, applications can specify the sort order of columns within a Super Column or Simple Column family. The system allows columns to be sorted either by time or by name. Time sorting of columns is exploited by application like Inbox Search where the results are always displayed in time sorted order. Any column within a column family is accessed using the convention *column\_family : column* and any column within a column family that is of type super is accessed using the convention *column\_family : super\_column : column*. A very good example of the super column family abstraction power is given in Section 6.1. Typically applications use a dedicated Cassandra cluster and manage them as part of their service. Although the system supports the notion of multiple tables all deployments have only one table in their schema.

### 4. API

The Cassandra API consists of the following three simple methods.

- *insert(table, key, rowMutation)*
- *get(table, key, columnName)*
- *delete(table, key, columnName)*

*columnName* can refer to a specific column within a column family, a column family, a super column family, or a column within a super column.

## 5. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have the following characteristics; scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is beyond the scope of this paper, so we will focus on the core distributed systems techniques used in Cassandra: partitioning, replication, membership, failure handling and scaling. All these modules work in synchrony to handle read/write requests. Typically a read/write request for a key gets routed to any node in the Cassandra cluster. The node then determines the replicas for this particular key. For writes, the system routes the requests to the replicas and waits for a quorum of replicas to acknowledge the completion of the writes. For reads, based on the consistency guarantees required by the client, the system either routes the requests to the closest replica or routes the requests to all replicas and waits for a quorum of responses.

### 5.1 Partitioning

One of the key design features for Cassandra is the ability to scale incrementally. This requires, the ability to dynamically partition the data over the set of nodes (i.e., storage hosts) in the cluster. Cassandra partitions data across the cluster using consistent hashing [11] but uses an order preserving hash function to do so. In consistent hashing the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its *position* on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. This node is deemed the coordinator for this key. The application specifies this key and the Cassandra uses it to route requests. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principal advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected. The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. Typically there exist two ways to address this issue: One is for nodes to get assigned to multiple positions in the circle (like in Dynamo), and the second is to analyze load information on the ring and have lightly loaded nodes move on the ring to alleviate heavily loaded nodes as described in [17]. Cassandra opts for the latter as it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing.

## 5.2 Replication

Cassandra uses replication to achieve high availability and durability. Each data item is replicated at N hosts, where N is the replication factor configured “per-instance”. Each key,  $k$ , is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring. Cassandra provides the client with various options for how data needs to be replicated. Cassandra provides various replication policies such as “Rack Unaware”, “Rack Aware” (within a data-center) and “Datacenter Aware”. Replicas are chosen based on the replication policy chosen by the application. If certain application chooses “Rack Unaware” replication strategy then the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring. For “Rack Aware” and “Datacenter Aware” strategies the algorithm is slightly more involved. Cassandra system elects a leader amongst its nodes using a system called Zookeeper[13]. All nodes on joining the cluster contact the leader who tells them for what ranges they are replicas for and leader makes a concerted effort to maintain the invariant that no node is responsible for more than N-1 ranges in the ring. The metadata about the ranges a node is responsible is cached locally at each node and in a fault-tolerant manner inside Zookeeper - this way a node that crashes and comes back up knows what ranges it was responsible for. We borrow from Dynamo parlance and deem the nodes that are responsible for a given range the “preference list” for the range.

As is explained in Section 5.1 every node is aware of every other node in the system and hence the range they are responsible for. Cassandra provides durability guarantees in the presence of node failures and network partitions by relaxing the quorum requirements as described in Section 5.2. Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Cassandra is configured such that each row is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple datacenters. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without any outage.

## 5.3 Membership

Cluster membership in Cassandra is based on Scuttlebutt[19], a very efficient anti-entropy Gossip based mechanism. The salient feature of Scuttlebutt is that it has very efficient CPU utilization and very efficient utilization of the gossip channel. Within the Cassandra system Gossip is not only used for membership but also to disseminate other system related control state.

### 5.3.1 Failure Detection

Failure detection is a mechanism by which a node can locally determine if any other node in the system is up or down. In Cassandra failure detection is also used to avoid attempts to communicate with unreachable nodes during various operations. Cassandra uses a modified version of the  $\Phi$  Accrual Failure Detector[8]. The idea of an Accrual Failure Detection is that the failure detection module doesn’t emit a Boolean value stating a node is up or down. Instead the

failure detection module emits a value which represents a suspicion level for each of monitored nodes. This value is defined as  $\Phi$ . The basic idea is to express the value of  $\Phi$  on a scale that is dynamically adjusted to reflect network and load conditions at the monitored nodes.

$\Phi$  has the following meaning: Given some threshold  $\Phi$ , and assuming that we decide to suspect a node A when  $\Phi = 1$ , then the likelihood that we will make a mistake (i.e., the decision will be contradicted in the future by the reception of a late heartbeat) is about 10%. The likelihood is about 1% with  $\Phi = 2$ , 0.1% with  $\Phi = 3$ , and so on. Every node in the system maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. The distribution of these inter-arrival times is determined and  $\Phi$  is calculated. Although the original paper suggests that the distribution is approximated by the Gaussian distribution we found the Exponential Distribution to be a better approximation, because of the nature of the gossip channel and its impact on latency. To our knowledge our implementation of the Accrual Failure Detection in a Gossip based setting is the first of its kind. Accrual Failure Detectors are very good in both their accuracy and their speed and they also adjust well to network conditions and server load conditions.

## 5.4 Bootstrapping

When a node starts for the first time, it chooses a random token for its position in the ring. For fault tolerance, the mapping is persisted to disk locally and also in Zookeeper. The token information is then gossiped around the cluster. This is how we know about all nodes and their respective positions in the ring. This enables any node to route a request for a key to the correct node in the cluster. In the bootstrap case, when a node needs to join a cluster, it reads its configuration file which contains a list of a few contact points within the cluster. We call these initial contact points, seeds of the cluster. Seeds can also come from a configuration service like Zookeeper.

In Facebook’s environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. Failures can be of various forms such as disk failures, bad CPU etc. A node outage rarely signifies a permanent departure and therefore should not result in re-balancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Cassandra nodes. To that effect every message contains the cluster name of each Cassandra instance. If a manual error in configuration led to a node trying to join a wrong Cassandra instance it can be thwarted based on the cluster name. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Cassandra instance. An administrator uses a command line tool or a browser to connect to a Cassandra node and issue a membership change to join or leave the cluster.

## 5.5 Scaling the Cluster

When a new node is added into the system, it gets assigned a token such that it can alleviate a heavily loaded node. This results in the new node splitting a range that some other node was previously responsible for. The Cassandra bootstrap algorithm is initiated from any other node in the system by an operator using either a command line utility

or the Cassandra web dashboard. The node giving up the data streams the data over to the new node using kernel copy techniques. Operational experience has shown that data can be transferred at the rate of 40 MB/sec from a single node. We are working on improving this by having multiple replicas take part in the bootstrap transfer thereby parallelizing the effort, similar to BitTorrent.

## 5.6 Local Persistence

The Cassandra system relies on the local file system for data persistence. The data is represented on disk using a format that lends itself to efficient data retrieval. Typical write operation involves a write into a commit log for durability and recoverability and an update into an in-memory data structure. The write into the in-memory data structure is performed only after a successful write into the commit log. We have a dedicated disk on each machine for the commit log since all writes into the commit log are sequential and so we can maximize disk throughput. When the in-memory data structure crosses a certain threshold, calculated based on data size and number of objects, it dumps itself to disk. This write is performed on one of many commodity disks that machines are equipped with. All writes are sequential to disk and also generate an index for efficient lookup based on row key. These indices are also persisted along with the data file. Over time many such files could exist on disk and a merge process runs in the background to collate the different files into one file. This process is very similar to the compaction process that happens in the Bigtable system.

A typical read operation first queries the in-memory data structure before looking into the files on disk. The files are looked at in the order of newest to oldest. When a disk lookup occurs we could be looking up a key in multiple files on disk. In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory. This bloom filter is first consulted to check if the key being looked up does indeed exist in the given file. A key in a column family could have many columns. Some special indexing is required to retrieve columns which are further away from the key. In order to prevent scanning of every column on disk we maintain column indices which allow us to jump to the right chunk on disk for column retrieval. As the columns for a given key are being serialized and written out to disk we generate indices at every 256K chunk boundary. This boundary is configurable, but we have found 256K to work well for us in our production workloads.

## 5.7 Implementation Details

The Cassandra process on a single machine is primarily consists of the following abstractions: partitioning module, the cluster membership and failure detection module and the storage engine module. Each of these modules rely on an event driven substrate where the message processing pipeline and the task pipeline are split into multiple stages along the line of the SEDA[20] architecture. Each of these modules has been implemented from the ground up using Java. The cluster membership and failure detection module, is built on top of a network layer which uses non-blocking I/O. All system control messages rely on UDP based messaging while the application related messages for replication and request routing relies on TCP. The request routing modules are implemented using a certain state machine. When a read/write

request arrives at any node in the cluster the state machine morphs through the following states (i) identify the node(s) that own the data for the key (ii) route the requests to the nodes and wait on the responses to arrive (iii) if the replies do not arrive within a configured timeout value fail the request and return to the client (iv) figure out the latest response based on timestamp (v) schedule a repair of the data at any replica if they do not have the latest piece of data. For sake of exposition we do not talk about failure scenarios here. The system can be configured to perform either synchronous or asynchronous writes. For certain systems that require high throughput we rely on asynchronous replication. Here the writes far exceed the reads that come into the system. During the synchronous case we wait for a quorum of responses before we return a result to the client.

In any journaled system there needs to exist a mechanism for purging commit log entries. In Cassandra we use a rolling a commit log where a new commit log is rolled out after an older one exceeds a particular, configurable, size. We have found that rolling commit logs after 128MB size seems to work very well in our production workloads. Every commit log has a header which is basically a bit vector whose size is fixed and typically more than the number of column families that a particular system will ever handle. In our implementation we have an in-memory data structure and a data file that is generated per column family. Every time the in-memory data structure for a particular column family is dumped to disk we set its bit in the commit log stating that this column family has been successfully persisted to disk. This is an indication that this piece of information is already committed. These bit vectors are per commit log and also maintained in memory. Every time a commit log is rolled its bit vector and all the bit vectors of commit logs rolled prior to it are checked. If it is deemed that all the data has been successfully persisted to disk then these commit logs are deleted. The write operation into the commit log can either be in normal mode or in *fast sync* mode. In the fast sync mode the writes to the commit log are buffered. This implies that there is a potential of data loss on machine crash. In this mode we also dump the in-memory data structure to disk in a buffered fashion. Traditional databases are not designed to handle particularly high write throughput. Cassandra morphs all writes to disk into sequential writes thus maximizing disk write throughput. Since the files dumped to disk are never mutated no locks need to be taken while reading them. The server instance of Cassandra is practically lockless for read/write operations. Hence we do not need to deal with or handle the concurrency issues that exist in B-Tree based database implementations.

The Cassandra system indexes all data based on primary key. The data file on disk is broken down into a sequence of blocks. Each block contains at most 128 keys and is demarcated by a block index. The block index captures the relative offset of a key within the block and the size of its data. When an in-memory data structure is dumped to disk a block index is generated and their offsets written out to disk as indices. This index is also maintained in memory for fast access. A typical read operation always looks up data first in the in-memory data structure. If found the data is returned to the application since the in-memory data structure contains the latest data for any key. If not found then we perform disk I/O against all the data files on disk in reverse time order. Since we are always looking for the latest

data we look into the latest file first and return if we find the data. Over time the number of data files will increase on disk. We perform a compaction process, very much like the Bigtable system, which merges multiple files into one; essentially merge sort on a bunch of sorted data files. The system will always compact files that are close to each other with respect to size i.e there will never be a situation where a 100GB file is compacted with a file which is less than 50GB. Periodically a major compaction process is run to compact all related data files into one big file. This compaction process is a disk I/O intensive operation. Many optimizations can be put in place to not affect in coming read requests.

## 6. PRACTICAL EXPERIENCES

In the process of designing, implementing and maintaining Cassandra we gained a lot of useful experience and learned numerous lessons. One very fundamental lesson learned was not to add any new feature without understanding the effects of its usage by applications. Most problematic scenarios do not stem from just node crashes and network partitions. We share just a few interesting scenarios here.

- Before launching the Inbox Search application we had to index 7TB of inbox data for over 100M users, then stored in our MySQL[1] infrastructure, and load it into the Cassandra system. The whole process involved running Map/Reduce[7] jobs against the MySQL data files, indexing them and then storing the reverse-index in Cassandra. The M/R process actually behaves as the client of Cassandra. We exposed some background channels for the M/R process to aggregate the reverse index per user and send over the serialized data over to the Cassandra instance, to avoid the serialization/deserialization overhead. This way the Cassandra instance is only bottlenecked by network bandwidth.
- Most applications only require atomic operation per key per replica. However there have been some applications that have asked for transactional mainly for the purpose of maintaining secondary indices. Most developers with years of development experience working with RDBMS's find this a very useful feature to have. We are working on a mechanism to expose such atomic operations.
- We experimented with various implementations of Failure Detectors such as the ones described in [15] and [5]. Our experience had been that the time to detect failures increased beyond an acceptable limit as the size of the cluster grew. In one particular experiment in a cluster of 100 nodes time to taken to detect a failed node was in the order of two minutes. This is practically unworkable in our environments. With the accrual failure detector with a slightly conservative value of PHI, set to 5, the average time to detect failures in the above experiment was about 15 seconds.
- Monitoring is not to be taken for granted. The Cassandra system is well integrated with Ganglia[12], a distributed performance monitoring tool. We expose various system level metrics to Ganglia and this has helped us understand the behavior of the system when subject to our production workload. Disks fail for no apparent reasons. The bootstrap algorithm has some

hooks to repair nodes when disk fail. This is however an administrative operation.

- Although Cassandra is a completely decentralized system we have learned that having some amount of co-ordination is essential to making the implementation of some distributed features tractable. For example Cassandra is integrated with Zookeeper, which can be used for various coordination tasks in large scale distributed systems. We intend to use the Zookeeper abstraction for some key features which actually do not come in the way of applications that use Cassandra as the storage engine.

### 6.1 Facebook Inbox Search

For Inbox Search we maintain a per user index of all messages that have been exchanged between the sender and the recipients of the message. There are two kinds of search features that are enabled today (a) term search (b) interactions - given the name of a person return all messages that the user might have ever sent or received from that person. The schema consists of two column families. For query (a) the user id is the key and the words that make up the message become the super column. Individual message identifiers of the messages that contain the word become the columns within the super column. For query (b) again the user id is the key and the recipients id's are the super columns. For each of these super columns the individual message identifiers are the columns. In order to make the searches fast Cassandra provides certain hooks for intelligent caching of data. For instance when a user clicks into the search bar an asynchronous message is sent to the Cassandra cluster to prime the buffer cache with that user's index. This way when the actual search query is executed the search results are likely to already be in memory. The system currently stores about 50+TB of data on a 150 node cluster, which is spread out between east and west coast data centers. We show some production measured numbers for read performance.

Latency Stat	Search Interactions	Term Search
Min	7.69ms	7.78ms
Median	15.69ms	18.27ms
Max	26.13ms	44.41ms

## 7. CONCLUSION

We have built, implemented, and operated a storage system providing scalability, high performance, and wide applicability. We have empirically demonstrated that Cassandra can support a very high update throughput while delivering low latency. Future works involves adding compression, ability to support atomicity across keys and secondary index support.

## 8. ACKNOWLEDGEMENTS

Cassandra system has benefitted greatly from feedback from many individuals within Facebook. In addition we thank Karthik Ranganathan who indexed all the existing data in MySQL and moved it into Cassandra for our first production deployment. We would also like to thank Dan Dumitriu from EPFL for his valuable suggestions about [19] and [8].

## 9. REFERENCES

- [1] MySQL AB. Mysql.
- [2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2002.
- [3] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
- [4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 205–218, 2006.
- [5] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303–312, Washington, DC, USA, 2002. IEEE Computer Society.
- [6] Giuseppe de Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon’s highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220. ACM, 2007.
- [7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
- [8] Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. The  $\phi$  accrual failure detector. In *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology*, pages 66–78, 2004.
- [9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.
- [10] Jim Gray and Pat Helland. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.
- [11] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In ACM Symposium on Theory of Computing*, pages 654–663, 1997.
- [12] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30:2004, 2004.
- [13] Benjamin Reed and Flavio Junqueira. Zookeeper.
- [14] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *USTR’94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 12–12, Berkeley, CA, USA, 1994. USENIX Association.
- [15] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Service, Proc. Conf. Middleware*, pages 55–70, 1996.
- [16] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990.
- [17] Ion Stoica, Robert Morris, David Liben-nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, 2003.
- [18] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP ’95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.
- [19] Robbert van Renesse, Dan Mihai Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Large Scale Distributed Systems and Middleware Workshop (LADIS ’08)*, New York, NY, USA, 2008. ACM.
- [20] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP ’01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM.

# MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

*Google, Inc.*

## Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google’s clusters every day.

## 1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

for a rewrite of our production indexing system. Section 7 discusses related and future work.

## 2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

*Map*, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key  $I$  and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key  $I$  and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

### 2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

The map function emits each word plus an associated count of occurrences (just '1' in this simple example). The reduce function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

## 2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

$$\begin{array}{lll} \text{map} & (k_1, v_1) & \rightarrow \text{list}(k_2, v_2) \\ \text{reduce} & (k_2, \text{list}(v_2)) & \rightarrow \text{list}(v_2) \end{array}$$

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

### 2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

**Distributed Grep:** The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

**Count of URL Access Frequency:** The map function processes logs of web page requests and outputs  $\langle \text{URL}, 1 \rangle$ . The reduce function adds together all values for the same URL and emits a  $\langle \text{URL}, \text{total count} \rangle$  pair.

**Reverse Web-Link Graph:** The map function outputs  $\langle \text{target}, \text{source} \rangle$  pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair:  $\langle \text{target}, \text{list}(\text{source}) \rangle$

**Term-Vector per Host:** A term vector summarizes the most important words that occur in a document or a set of documents as a list of  $\langle \text{word}, \text{frequency} \rangle$  pairs. The map function emits a  $\langle \text{hostname}, \text{term vector} \rangle$  pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final  $\langle \text{hostname}, \text{term vector} \rangle$  pair.

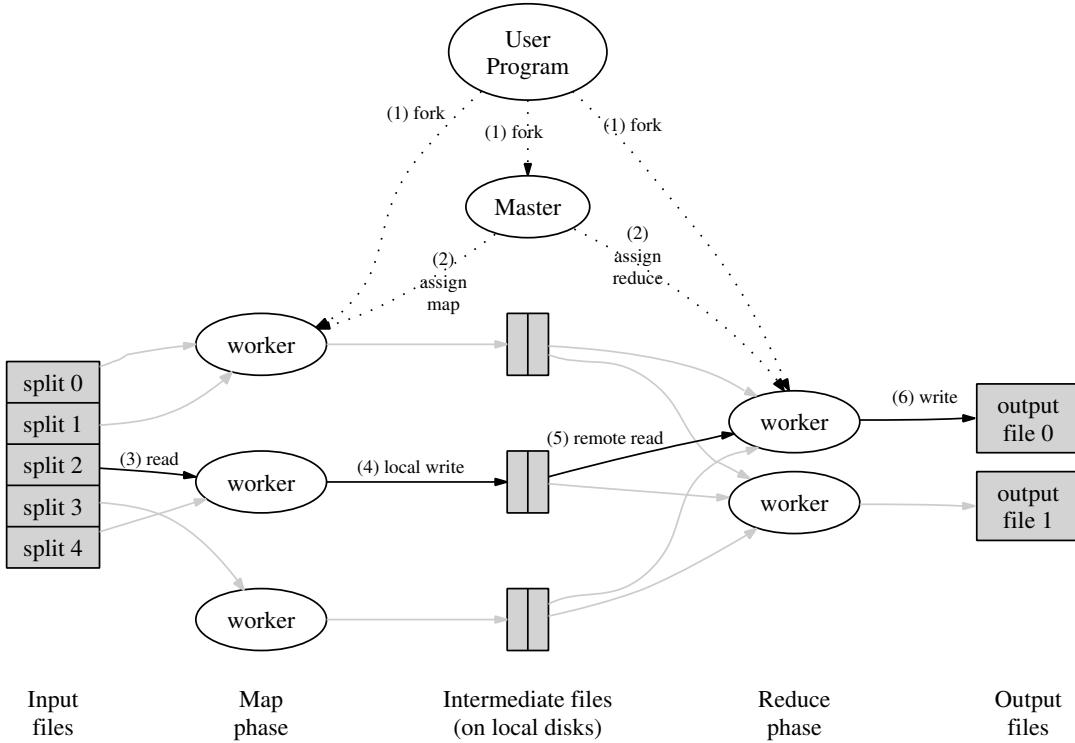


Figure 1: Execution overview

**Inverted Index:** The map function parses each document, and emits a sequence of  $\langle \text{word}, \text{document ID} \rangle$  pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a  $\langle \text{word}, \text{list(document ID)} \rangle$  pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**Distributed Sort:** The map function extracts the key from each record, and emits a  $\langle \text{key}, \text{record} \rangle$  pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

### 3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google:

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

- (1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.
- (2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
- (3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
- (4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.
- (5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

#### 3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of  $M$  *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into  $R$  pieces using a partitioning function (e.g.,  $\text{hash}(\text{key}) \bmod R$ ). The number of partitions ( $R$ ) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into  $M$  pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are  $M$  map tasks and  $R$  reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into  $R$  regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the  $R$  output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these  $R$  output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

## 3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the  $R$  intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

## 3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

### Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker  $A$  and then later executed by worker  $B$  (because  $A$  failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker  $A$  will read the data from worker  $B$ .

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

### Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

### Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces  $R$  such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the  $R$  temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of  $R$  files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very

easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task  $R_1$  is equivalent to the output for  $R_1$  produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task  $R_2$  may correspond to the output for  $R_2$  produced by a different sequential execution of the non-deterministic program.

Consider map task  $M$  and reduce tasks  $R_1$  and  $R_2$ . Let  $e(R_i)$  be the execution of  $R_i$  that committed (there is exactly one such execution). The weaker semantics arise because  $e(R_1)$  may have read the output produced by one execution of  $M$  and  $e(R_2)$  may have read the output produced by a different execution of  $M$ .

### 3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

### 3.5 Task Granularity

We subdivide the map phase into  $M$  pieces and the reduce phase into  $R$  pieces, as described above. Ideally,  $M$  and  $R$  should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large  $M$  and  $R$  can be in our implementation, since the master must make  $O(M + R)$  scheduling decisions and keeps  $O(M * R)$  state in memory as described above. (The constant factors for memory usage are small however: the  $O(M * R)$  piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore,  $R$  is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose  $M$  so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make  $R$  a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with  $M = 200,000$  and  $R = 5,000$ , using 2,000 worker machines.

### 3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a “straggler”: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

## 4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

### 4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire ( $R$ ). Data gets partitioned across these tasks using a partitioning function on

the intermediate key. A default partitioning function is provided that uses hashing (e.g. “ $\text{hash}(\text{key}) \bmod R$ ”). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using “ $\text{hash}(\text{Hostname}(\text{urlkey})) \bmod R$ ” as the partitioning function causes all URLs from the same host to end up in the same output file.

### 4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

### 4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

### 4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, “text”

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode’s range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

## 4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

## 4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal,

the signal handler sends a “last gasp” UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

## 4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

## 4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

## 4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```

Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents) :
    for each word w in contents:
        if (IsCapitalized(w)):
            uppercase->Increment();
        EmitIntermediate(w, "1");

```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

## 5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

### 5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE

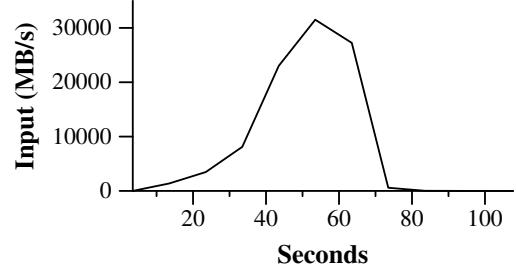


Figure 2: Data transfer rate over time

disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

### 5.2 Grep

The *grep* program scans through  $10^{10}$  100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ( $M = 15000$ ), and the entire output is placed in one file ( $R = 1$ ).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

### 5.3 Sort

The *sort* program sorts  $10^{10}$  100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the

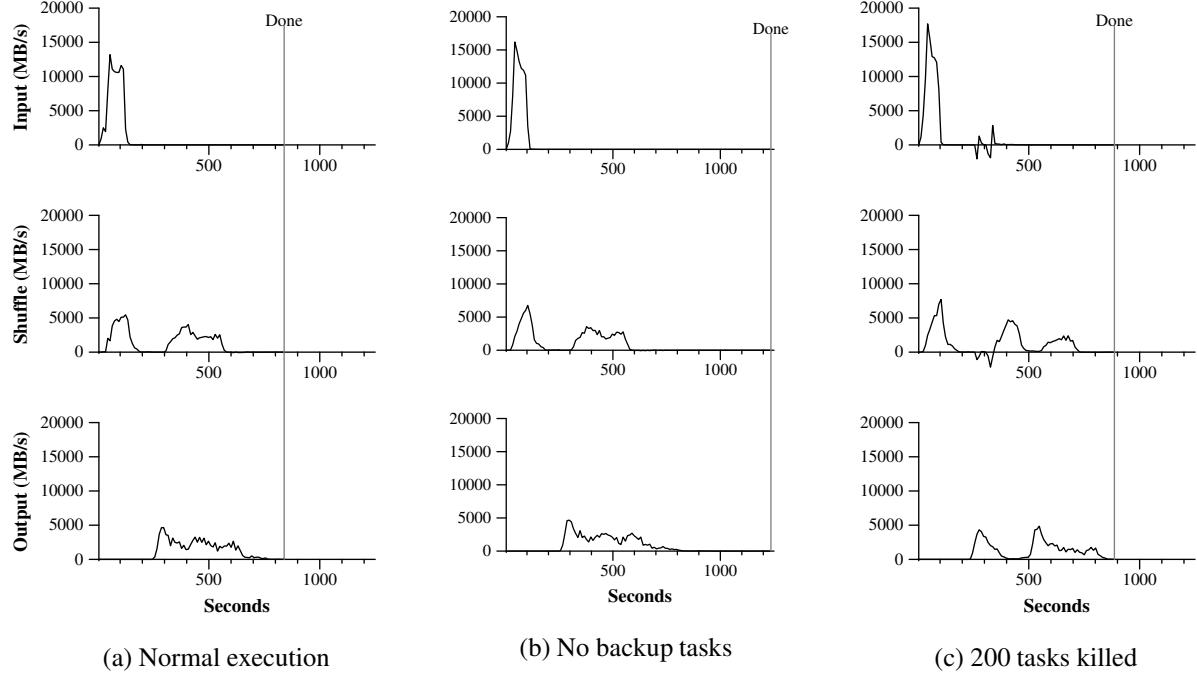


Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This function passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ( $M = 15000$ ). We partition the sorted output into 4000 files ( $R = 4000$ ). The partitioning function uses the initial bytes of the key to segregate it into one of  $R$  pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for *grep* had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for

the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

## 5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

## 5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

## 6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

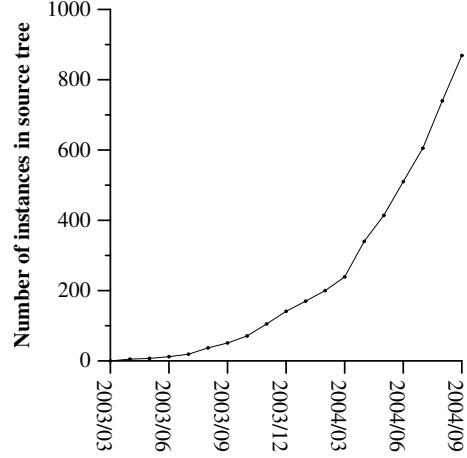


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique <i>map</i> implementations	395
Unique <i>reduce</i> implementations	269
Unique <i>map/reduce</i> combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

### 6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

## 7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an  $N$  element array in  $\log N$  time on  $N$  processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that

make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of  $R$  reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

## 8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

## Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke,

David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

## References

- [1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.
- [2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.
- [3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
- [4] Luiz A. Barroso, Jeffrey Dean, and Urs Hözle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.
- [5] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.
- [6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.
- [7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.
- [8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

- [9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraignaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.
- [10] Jim Gray. Sort benchmark home page. <http://research.microsoft.com/barc/SortBenchmark/>.
- [11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
- [12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.
- [13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.
- [14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
- [15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.
- [16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.
- [17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.
- [18] Jim Wyllie. Spsort: How to sort a terabyte quickly. <http://almel.almaden.ibm.com/cs/spsort.pdf>.

## A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
                i++;

            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
        }
    }

    virtual void Reduce(const ReduceInput& input) {
        if (start < i)
            Emit(text.substr(start,i-start),"1");
    }
};

REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};

REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Specify the output files:
    //   /gfs/test/freq-00000-of-00100
    //   /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2000
    // machines and 100 MB of memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();

    // Done: 'result' structure contains info
    // about counters, time taken, number of
    // machines used, etc.

    return 0;
}
```

# NoSQL Databases

Christof Strauch  
(cs134@hdm-stuttgart.de)

**Lecture**  
Selected Topics on Software-Technology  
Ultra-Large Scale Sites

**Lecturer**  
Prof. Walter Kriha

**Course of Studies**  
Computer Science and Media (CSM)

**University**  
Hochschule der Medien, Stuttgart  
(Stuttgart Media University)

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Introduction and Overview . . . . .	1
1.2	Uncovered Topics . . . . .	1
<b>2</b>	<b>The NoSQL-Movement</b>	<b>2</b>
2.1	Motives and Main Drivers . . . . .	2
2.2	Criticism . . . . .	15
2.3	Classifications and Comparisons of NoSQL Databases . . . . .	23
<b>3</b>	<b>Basic Concepts, Techniques and Patterns</b>	<b>30</b>
3.1	Consistency . . . . .	30
3.2	Partitioning . . . . .	37
3.3	Storage Layout . . . . .	44
3.4	Query Models . . . . .	47
3.5	Distributed Data Processing via MapReduce . . . . .	50
<b>4</b>	<b>Key-/Value-Stores</b>	<b>52</b>
4.1	Amazon's Dynamo . . . . .	52
4.2	Project Voldemort . . . . .	62
4.3	Other Key-/Value-Stores . . . . .	67
<b>5</b>	<b>Document Databases</b>	<b>69</b>
5.1	Apache CouchDB . . . . .	69
5.2	MongoDB . . . . .	76
<b>6</b>	<b>Column-Oriented Databases</b>	<b>104</b>
6.1	Google's Bigtable . . . . .	104
6.2	Bigtable Derivatives . . . . .	113
6.3	Cassandra . . . . .	114
<b>7</b>	<b>Conclusion</b>	<b>121</b>
<b>A</b>	<b>Further Reading, Listening and Watching</b>	<b>iv</b>
<b>B</b>	<b>List of abbreviations</b>	<b>ix</b>
<b>C</b>	<b>Bibliography</b>	<b>xii</b>

# List of Figures

3.1	Vector Clocks . . . . .	34
3.2	Vector Clocks – Exchange via Gossip in State Transfer Mode . . . . .	36
3.3	Vector Clocks – Exchange via Gossip in Operation Transfer Mode . . . . .	37
3.4	Consistent Hashing – Initial Situation . . . . .	40
3.5	Consistent Hashing – Situation after Node Joining and Departure . . . . .	40
3.6	Consistent Hashing – Virtual Nodes Example . . . . .	41
3.7	Consistent Hashing – Example with Virtual Nodes and Replicated Data . . . . .	41
3.8	Membership Changes – Node X joins the System . . . . .	43
3.9	Membership Changes – Node B leaves the System . . . . .	43
3.10	Storage Layout – Row-based, Columnar with/out Locality Groups . . . . .	45
3.11	Storage Layout – Log Structured Merge Trees . . . . .	45
3.12	Storage Layout – MemTables and SSTables in Bigtable . . . . .	46
3.13	Storage Layout – Copy-on-modify in CouchDB . . . . .	47
3.14	Query Models – Companion SQL-Database . . . . .	48
3.15	Query Models – Scatter/Gather Local Search . . . . .	49
3.16	Query Models – Distributed B+Tree . . . . .	49
3.17	Query Models – Prefix Hash Table / Distributed Trie . . . . .	49
3.18	MapReduce – Execution Overview . . . . .	50
3.19	MapReduce – Execution on Distributed Storage Nodes . . . . .	51
4.1	Amazon's Dynamo – Consistent Hashing with Replication . . . . .	55
4.2	Amazon's Dynamo – Concurrent Updates on a Data Item . . . . .	57
4.3	Project Voldemort – Logical Architecture . . . . .	63
4.4	Project Voldemort – Physical Architecture Options . . . . .	64
5.1	MongoDB – Replication Approaches . . . . .	94
5.2	MongoDB – Sharding Components . . . . .	97
5.3	MongoDB – Sharding Metadata Example . . . . .	98
6.1	Google's Bigtable – Example of Web Crawler Results . . . . .	105
6.2	Google's Bigtable – Tablet Location Hierarchy . . . . .	108
6.3	Google's Bigtable – Tablet Representation at Runtime . . . . .	110

# List of Tables

2.1	Classifications – NoSQL Taxonomy by Stephen Yen . . . . .	24
2.2	Classifications – Categorization by Ken North . . . . .	25
2.3	Classifications – Categorization by Rick Cattell . . . . .	25
2.4	Classifications – Categorization and Comparison by Scofield and Popescu . . . . .	26
2.5	Classifications – Comparison of Scalability Features . . . . .	26
2.6	Classifications – Comparison of Data Model and Query API . . . . .	27
2.7	Classifications – Comparison of Persistence Design . . . . .	28
3.1	CAP-Theorem – Alternatives, Traits, Examples . . . . .	31
3.2	ACID vs. BASE . . . . .	32
4.1	Amazon's Dynamo – Summary of Techniques . . . . .	54
4.2	Amazon's Dynamo – Evaluation by Ippolito . . . . .	62
4.3	Project Voldemort – JSON Serialization Format Data Types . . . . .	66
5.1	MongoDB – Referencing vs. Embedding Objects . . . . .	79
5.2	MongoDB - Parameters of the group operation . . . . .	89

# 1. Introduction

## 1.1. Introduction and Overview

Relational database management systems (RDBMSs) today are the predominant technology for storing structured data in web and business applications. Since Codds paper “A relational model of data for large shared data banks” [Cod70] from 1970 these datastores relying on the relational calculus and providing comprehensive ad hoc querying facilities by SQL (cf. [CB74]) have been widely adopted and are often thought of as the only alternative for data storage accessible by multiple clients in a consistent way. Although there have been different approaches over the years such as object databases or XML stores these technologies have never gained the same adoption and market share as RDBMSs. Rather, these alternatives have either been absorbed by relational database management systems that e.g. allow to store XML and use it for purposes like text indexing or they have become niche products for e.g. OLAP or stream processing.

In the past few years, the "one size fits all"-thinking concerning datastores has been questioned by both, science and web affine companies, which has lead to the emergence of a great variety of alternative databases. The movement as well as the new datastores are commonly subsumed under the term *NoSQL*, "used to describe the increasing usage of non-relational databases among Web developers" (cf. [Oba09a]).

This paper's aims at giving a systematic overview of the motives and rationales directing this movement (chapter 2), common concepts, techniques and patterns (chapter 3) as well as several classes of NoSQL databases (key-/value-stores, document databases, column-oriented databases) and individual products (chapters 4–6).

## 1.2. Uncovered Topics

This paper excludes the discussion of datastores existing before and are not referred to as part of the NoSQL movement, such as object-databases, pure XML databases and DBMSs for special purposes (such as analytics or stream processing). The class of graph databases is also left out of this paper but some resources are provided in the appendix A. It is out of the scope of this paper to suggest individual NoSQL datastores in general as this would be a total misunderstanding of both, the movement and the approach of the NoSQL datastores, as not being “one size fits all”-solutions. In-depth comparisons between all available NoSQL databases would also exceed the scope of this paper.

## 2. The NoSQL-Movement

In this chapter, motives and main drivers of the NoSQL movement will be discussed along with remarks passed by critics and reactions from NoSQL advocates. The chapter will conclude by different attempts to classify and characterize NoSQL databases. One of them will be treated in the subsequent chapters.

### 2.1. Motives and Main Drivers

The term NoSQL was first used in 1998 for a relational database that omitted the use of SQL (see [Str10]). The term was picked up again in 2009 and used for conferences of advocates of non-relational databases such as Last.fm developer Jon Oskarsson, who organized the NoSQL meetup in San Francisco (cf. [Eva09a]). A blogger, often referred to as having made the term popular is Rackspace employee Eric Evans who later described the ambition of the NoSQL movement as “the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for” (cf. [Eva09b]).

This section will discuss rationales of practitioners for developing and using nonrelational databases and display theoretical work in this field. Furthermore, it will treat the origins and main drivers of the NoSQL movement.

#### 2.1.1. Motives of NoSQL practitioners

The Computerworld magazine reports in an article about the NoSQL meet-up in San Francisco that “NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favor of more efficient and cheaper ways of managing data.” (cf. [Com09a]). It states that especially Web 2.0 startups have begun their business without Oracle and even without MySQL which formerly was popular among startups. Instead, they built their own datastores influenced by Amazon’s Dynamo ([DHJ<sup>+</sup>07]) and Google’s Bigtable ([CDG<sup>+</sup>06]) in order to store and process huge amounts of data like they appear e.g. in social community or cloud computing applications; meanwhile, most of these datastores became open source software. For example, Cassandra originally developed for a new search feature by Facebook is now part of the Apache Software Project. According to engineer Avinash Lakshman, it is able to write 2500 times faster into a 50 gigabytes large database than MySQL (cf. [LM09]).

The Computerworld article summarizes reasons commonly given to develop and use NoSQL datastores:

**Avoidance of Unneeded Complexity** Relational databases provide a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases.

As an example, Adobe’s ConnectNow holds three copies of user session data; these replicas do not neither have to undergo all consistency checks of a relational database management systems nor do they have to be persisted. Hence, it is fully sufficient to hold them in memory (cf. [Com09b]).

**High Throughput** Some NoSQL databases provide a significantly higher data throughput than traditional RDBMSs. For instance, the column-store Hypertable which pursues Google's Bigtable approach allows the local search engine Zevent to store one billion data cells per day [Jud09]. To give another example, Google is able to process 20 petabyte a day stored in Bigtable via its MapReduce approach [Com09b].

**Horizontal Scalability and Running on Commodity Hardware** "Definitely, the volume of data is getting so huge that people are looking at other technologies", says Jon Travis, an engineer at SpringSource (cited in [Com09a]). Blogger Jonathan Ellis agrees with this notion by mentioning three problem areas of current relational databases that NoSQL databases are trying to address (cf. [Ell09a]):

1. Scale out data (e.g. 3 TB for the *green badges* feature at Digg, 50 GB for the inbox search at Facebook or 2 PB in total at eBay)
2. Performance of single servers
3. Rigid schema design

In contrast to relational database management systems most NoSQL databases are designed to scale well in the horizontal direction and not rely on highly available hardware. Machines can be added and removed (or crash) without causing the same operational efforts to perform sharding in RDBMS cluster-solutions; some NoSQL datastores even provide automatic sharding (such as MongoDB as of March 2010, cf. [Mer10g]). Javier Soltero, CTO of SpringSource puts it this way: "Oracle would tell you that with the right degree of hardware and the right configuration of Oracle RAC (Real Application Clusters) and other associated magic software, you can achieve the same scalability. But at what cost?" (cited in [Com09a]). Especially for Web 2.0 companies the scalability aspect is considered crucial for their business, as Johan Oskarsson of Last.fm states: "Web 2.0 companies can take chances and they need scalability. When you have these two things in combination, it makes [NoSQL] very compelling." (Johan Oskarsson, Organizer of the meet-up and web developer at Last.fm, cf. [Com09a]). Blogger Nati Shalom agrees with that: "cost pressure also forced many organizations to look at more cost-effective alternatives, and with that came research that showed that distributed storage based on commodity hardware can be even more reliable than[sic!] many of the existing high end databases" (cf. [Sha09a] and for further reading [Sha09c]). He concludes: "All of this led to a demand for a cost effective "scale-first database"".

**Avoidance of Expensive Object-Relational Mapping** Most of the NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented programming languages compared to relational data structures. They do not make expensive object-relational mapping necessary (such as Key/Value-Stores or Document-Stores). This is particularly important for applications with data structures of low complexity that can hardly benefit from the features of a relational database. Dare Obasanjo claims a little provokingly that "all you really need [as a web developer] is a key<->value or tuple store that supports some level of query functionality and has decent persistence semantics." (cf. [Oba09a]). The blogger and database-analyst Curt Monash iterates on this aspect: "SQL is an awkward fit for procedural code, and almost all code is procedural. [For data upon which users expect to do heavy, repeated manipulations, the cost of mapping data into SQL is] well worth paying [...] But when your database structure is very, very simple, SQL may not seem that beneficial." Jon Travis, an engineer at SpringSource agrees with that: "Relational databases give you too much. They force you to twist your object data to fit a RDBMS." (cited in [Com09a]).

In a blog post on the Computerworld article Nati Shalom, CTO and founder of GigaSpaces, identifies the following further drivers of the NoSQL movement (cf. [Sha09b]):

**Complexity and Cost of Setting up Database Clusters** He states that NoSQL databases are designed in a way that “PC clusters can be easily and cheaply expanded without the complexity and cost of ‘sharding,’ which involves cutting up databases into multiple tables to run on large clusters or grids”.

**Compromising Reliability for Better Performance** Shalom argues that there are “different scenarios where applications would be willing to compromise reliability for better performance.” As an example of such a scenario favoring performance over reliability, he mentions HTTP session data which “needs to be shared between various web servers but since the data is transient in nature (it goes away when the user logs off) there is no need to store it in persistent storage.”

**The Current “One size fit’s it all” Databases Thinking Was and Is Wrong** Shalom states that “a growing number of application scenarios cannot be addressed with a traditional database approach”. He argues that “this realization is actually not that new” as the studies of Michael Stonebraker (see below) have been around for years but the old ‘news’ has spread to a larger community in the last years. Shalom thinks that this realization and the search for alternatives towards traditional RDBMSs can be explained by two major trends:

1. The continuous growth of data volumes (to be stored)
2. The growing need to process larger amounts of data in shorter time

Some companies, especially web-affine ones have already adopted NoSQL databases and Shalom expects that they will find their way into mainstream development as these datastores mature. Blogger Dennis Forbes agrees with this issue by underlining that the requirements of a bank are not universal and especially social media sites have different characteristics: “unrelated islands of data”, a “very low [...] user/transaction value” and no strong need for data integrity. Considering these characteristics he states the following with regard to social media sites and big web applications:

“The truth is that you don’t need ACID for Facebook status updates or tweets or Slashdots comments. So long as your business and presentation layers can robustly deal with inconsistent data, it doesn’t really matter. It isn’t ideal, obviously, and preferably [sic!] you see zero data loss, inconsistency, or service interruption, however accepting data loss or inconsistency (even just temporary) as a possibility, breaking free of by far the biggest scaling “hindrance” of the RDBMS world, can yield dramatic flexibility. [...]”

This is the case for many social media sites: data integrity is largely optional, and the expense to guarantee it is an unnecessary expenditure. When you yield pennies for ad clicks after thousands of users and hundreds of thousands of transactions, you start to look to optimize.” (cf. [For10])

Shalom suggests caution when moving towards NoSQL solutions and to get familiar with their specific strengths and weaknesses (e.g. the ability of the business logic to deal with inconsistency). Others, like David Merriman of 10gen (the company behind MongoDB) also stress that there is no single tool or technology for the purpose of data storage but that there is a segmentation currently underway in the database field bringing forth new and different data stores for e.g. business intelligence vs. online transaction processing vs. persisting large amounts of binary data (cf. [Tec09]).

**The Myth of Effortless Distribution and Partitioning of Centralized Data Models** Shalom further addresses the myth surrounding the perception that data models originally designed with a single database in mind (centralized datamodels, as he puts it) often cannot easily be partitioned and distributed among database servers. This signifies that without further effort, the application will neither necessarily scale and nor work correct any longer. The professionals of Ajatus agree with this in a blog post stating that if a database grows, at first, replication is configured. In addition, as the amount of data grows further, the database is sharded by expensive system admins requiring large financial sums or a fortune worth of money for commercial DBMS-vendors are needed to operate the sharded database (cf. [Aja09]). Shalom reports from an architecture summit at eBay in the summer of 2009. Participants agreed on the fact that although typically, abstractions involved trying to hide distribution and partitioning issues away from applications (e.g. by proxy layers routing requests to certain servers) "this abstraction cannot insulate the application from the reality that [...] partitioning and distribution is involved. The spectrum of failures within a network is entirely different from failures within a single machine. The application needs to be made aware of latency, distributed failures, etc., so that it has enough information to make the correct context-specific decision about what to do. The fact that the system is distributed leaks through the abstraction." ([Sha09b]). Therefore he suggests designing datamodels to fit into a partitioned environment even if there will be only one centralized database server initially. This approach offers the advantage to avoid exceedingly late and expensive changes of application code.

Shalom concludes that in his opinion relational database management systems will not disappear soon. However, there is definitely a place for more specialized solutions as a "one size fits all" thinking was and is wrong with regards to databases.

**Movements in Programming Languages and Development Frameworks** The blogger David Intersimone additionally observes movements in programming languages and development frameworks that provide abstractions for database access trying to hide the use of SQL (cf. []) and relational databases ([Int10]). Examples for this trend in the last couple of years include:

- Object-relational mappers in the Java and .NET world like the Java Persistence API (JPA, part of the EJB 3 specification, cf. [DKE06], [BO06],), implemented by e.g. Hibernate ([JBo10a], or the LINQ-Framework (cf. [BH05]) with its code generator SQLMetal and the ADO.NET Entity Framework (cf. [Mic10])) since .NET version 4.
- Likewise, the popular Ruby on Rails (RoR, [HR10]) framework and others try to hide away the usage of a relational database (e.g. by implementing the active record pattern as of RoR).
- NoSQL datastores as well as some databases offered by cloud computing providers completely omit a relational database. One example of such a cloud datastore is Amazon's SimpleDB, a schema-free, Erlang-based eventually consistent datastore which is characterized as an Entity-Attribute-Value (EAV). It can store large collections of items which themselves are hashables containing attributes that consist of key-value-pairs (cf. [Nor09]).

The NoSQL databases react on this trend and try to provide data structures in their APIs that are closer to the ones of programming languages (e.g. key/value-structures, documents, graphs).

**Requirements of Cloud Computing** In an interview Dwight Merriman of 10gen (the company behind MongoDB) mentions two major requirements of datastores in cloud computing environments ([Tec09]):

1. High until almost ultimate scalability—especially in the horizontal direction
2. Low administration overhead

In his view, the following classes of databases work well in the cloud:

- Data warehousing specific databases for batch data processing and map/reduce operations.
- Simple, scalable and fast key/value-stores.
- Databases containing a richer feature set than key/value-stores fitting the gap with traditional RDBMSs while offering good performance and scalability properties (such as document databases).

Blogger Nati Shalom agrees with Merriman in the fact that application areas like cloud-computing boosted NoSQL databases: “what used to be a niche problem that only a few fairly high-end organizations faced, became much more common with the introduction of social networking and cloud computing” (cf. [Sha09a]).

**The RDBMS plus Caching-Layer Pattern/Workaround vs. Systems Built from Scratch with Scalability in Mind** In his article “MySQL and memcached: End of an era?” Todd Hoff states that in a “pre-cloud, relational database dominated world” scalability was an issue of “leveraging MySQL and memcached”:

“Shard MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together. That was state of the art, that was how it was done. The architecture of many major sites still follow[sic!] this pattern today, largely because with enough elbow grease, it works.” (cf. [Hof10c])

But as scalability requirements grow and these technologies are less and less capable to suit with them. In addition, as NoSQL datastores are arising Hoff comes to the conclusion that “[with] a little perspective, it’s clear the MySQL + memcached era is passing. It will stick around for a while. Old technologies seldom fade away completely.” (cf. [Hof10c]). As examples, he cites big websites and players that have moved towards non-relational datastores including LinkedIn, Amazon, Digg and Twitter. Hoff mentions the following reasons for using NoSQL solutions which have been explained earlier in this paper:

- Relational databases place computation on reads, which is considered wrong for large-scale web applications such as Digg. NoSQL databases therefore do not offer or avoid complex read operations.
- The serial nature of applications<sup>1</sup> often waiting for I/O from the data store which does no good to scalability and low response times.
- Huge amounts of data and a high growth factor lead Twitter towards facilitating Cassandra, which is designed to operate with large scale data.
- Furthermore, operational costs of running and maintaining systems like Twitter escalate. Web applications of this size therefore “need a system that can grow in a more automated fashion and be highly available.” (cited in [Hof10c]).

For these reasons and the “clunkiness” of the MySQL and memcached era (as Hoff calls it) large scale (web) applications nowadays can utilize systems built from scratch with scalability, non-blocking and asynchronous database I/O, handling of huge amounts of data and automation of maintainance and operational tasks in mind. He regards these systems to be far better alternatives compared to relational DBMSs with additional object-caching. Amazon’s James Hamilton agrees with this by stating that for many large-scale web sites scalability from scratch is crucial and even outweighs the lack of features compared to traditional RDBMSs:

---

<sup>1</sup>Hoff does not give more detail on the types of applications meant here, but—in the authors opinion—a synchronous mindset and implementation of database I/O can be seen in database connectivity APIs (such as ODBC or JDBC) as well as in object-relational mappers and it spreads into many applications from these base technologies.

"Scale-first applications are those that absolutely must scale without bound and being able to do this without restriction is much more important than more features. These applications are exemplified by very high scale web sites such as Facebook, MySpace, Gmail, Yahoo, and Amazon.com. Some of these sites actually do make use of relational databases but many do not. The common theme across all of these services is that scale is more important than features and none of them could possibly run on a single RDBMS." (cf. [Ham09] cited in [Sha09a])

**Yesterday's vs. Today's Needs** In a discussion on CouchDB Lehnhardt and Lang point out that needs regarding data storage have considerably changed over time (cf. [PLL09]; this argument is iterated further by Stonebraker, see below). In the 1960s and 1970s databases have been designed for single, large high-end machines. In contrast to this, today, many large (web) companies use commodity hardware which will predictably fail. Applications are consequently designed to handle such failures which are considered the "standard mode of operation", as Amazon refers to it (cf. [DHJ<sup>+</sup>07, p. 205]). Furthermore, relational databases fit well for data that is rigidly structured with relations and allows for dynamic queries expressed in a sophisticated language. Lehnhardt and Lang point out that today, particularly in the web sector, data is neither rigidly structured nor are dynamic queries needed as most applications already use prepared statements or stored procedures. Therefore, it is sufficient to predefine queries within the database and assign values to their variables dynamically (cf. [PLL09]).

Furthermore, relational databases were initially designed for centralized deployments and not for distribution. Although enhancements for clustering have been added on top of them it still leaks through that traditional were not designed having distribution concepts in mind at the beginning (like the issues adverted by the "fallacies of network computing" quoted below). As an example, synchronization is often not implemented efficiently but requires expensive protocols like two or three phase commit. Another difficulty Lehnhardt and Lang see is that clusters of relational databases try to be "transparent" towards applications. This means that the application should not contain any notion if talking to a singly machine or a cluster since all distribution aspects are tried to be hidden from the application. They question this approach to keep the application unaware of all consequences of distribution that are e.g. stated in the famous eight fallacies of distributed computing (cf. [Gos07]<sup>2</sup>):

"Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero

---

<sup>2</sup>The fallacies are cited according to James Gosling's here. There is some discussion about who came up with the list: the first seven fallacies are commonly credited to Peter Deutsch, a Sun Fellow, having published them in 1994. Fallacy number eight was added by James Gosling around 1997. Though—according to the English Wikipedia—"Bill Joy and Tom Lyon had already identified the first four as "The Fallacies of Networked Computing"" (cf. [Wik10]). More details on the eight fallacies can be found in an article of Rotem-Gal-Oz (cf. [RGO06])

### 8. The network is homogeneous"

While typical business application using relational database management systems try, in most cases, to hide distribution aspects from the application (e.g. by clusters, persistence layers doing object-relational mapping) many large web companies as well as most of the NoSQL databases do not pursue this approach. Instead, they let the application know and leverage them. This is considered a paradigm shift in the eyes of Lehnardt and Lang (cf. [PLL09]).

**Further Motives** In addition to the aspects mentioned above David Intersimone sees the following three goals of the NoSQL movement (cf. [Int10]):

- Reach less overhead and memory-footprint of relational databases
- Usage of Web technologies and RPC calls for access
- Optional forms of data query

#### 2.1.2. Theoretical work

In their widely adopted paper “The End of an Architectural Era” (cf. [SMA<sup>+</sup>07]) Michael Stonebraker<sup>3</sup> et al. come to the conclusion “that the current RDBMS code lines, while attempting to be a “one size fits all” solution, in fact, excel at nothing”. *Nothing* in this context means that they can neither compete with “specialized engines in the data warehouse, stream processing, text, and scientific database markets” which outperform them “by 1–2 orders of magnitude” (as shown in previous papers, cf. [Sc05], [SBc<sup>+</sup>07]) nor do they perform well in their home market of business data processing / online transaction processing (OLTP), where a prototype named H-Store developed at the M.I.T. beats up RDBMSs by nearly two orders of magnitude in the TPC-C benchmark. Because of these results they conclude that RDBMSs“ are 25 year old legacy code lines that should be retired in favor of a collection of “from scratch” specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow’s requirements, not continue to push code lines and architectures designed for yesterday’s needs”. But how do Stonebraker et al. come to this conclusion? Which inherent flaws do they find in relational database management systems and which suggestions do they provide for the “complete rewrite” they are requiring?

At first, Stonebraker et al. argue that RDBMSs have been architected more than 25 years ago when the hardware characteristics, user requirements and database markets were different from those today. They point out that “popular relational DBMSs all trace their roots to System R from the 1970s”: IBM’s DB2 is a direct descendant of System R, Microsoft’s SQL Server has evolved from Sybase System 5 (another direct System R descendant) and Oracle implemented System R’s user interface in its first release. Now, the architecture of System R has been influenced by the hardware characteristics of the 1970s. Since then, processor speed, memory and disk sizes have increased enormously and today do not limit programs in the way they did formerly. However, the bandwidth between hard disks and memory has not increased as fast as the CPU speed, memory and disk size. Stonebraker et al. criticize that this development in the field of hardware has not impacted the architecture of relational DBMSs. They especially see the following architectural characteristics of System R shine through in today’s RDBMSs:

- “Disk oriented storage and indexing structures”
- “Multithreading to hide latency”

---

<sup>3</sup>When reading and evaluating Stonebraker’s writings it has to be in mind that he is commercially involved into multiple DBMS products such Vertica, a column-store providing data warehousing and business analytics.

- “Locking-based concurrency control mechanisms”
- “Log-based recovery”

In this regard, they underline that although “there have been some extensions over the years, including support for compression, shared-disk architectures, bitmap indexes, support for user-defined data types and operators [...] no system has had a complete redesign since its inception”.

Secondly, Stonebraker et al. point out that new markets and use cases have evolved since the 1970s when there was only business data processing. Examples of these new markets include “data warehouses, text management, and stream processing” which “have very different requirements than business data processing”. In a previous paper (cf. [Sc05]) they have shown that RDBMSs “could be beaten by specialized architectures by an order of magnitude or more in several application areas, including:

- Text (specialized engines from Google, Yahoo, etc.)
- Data Warehouses (column stores such as Vertica, Monet, etc.)
- Stream Processing (stream processing engines such as StreamBase and Coral8)
- Scientific and intelligence databases (array storage engines such as MATLAB and ASAP)”

They go on noticing that user interfaces and usage model also changed over the past decades from terminals where “operators [were] inputting queries” to rich client and web applications today where interactive transactions and direct SQL interfaces are rare.

Stonebraker et al. now present “evidence that the current architecture of RDBMSs is not even appropriate for business data processing”. They have designed a DBMS engine for OLTP called H-Store that is functionally equipped to run the TPC-C benchmark and does so 82 times faster than a popular commercial DBMS. Based on this evidence, they conclude that “there is no market where they are competitive. As such, they should be considered as legacy technology more than a quarter of a century in age, for which a complete redesign and re-architecting is the appropriate next step”.

## Design Considerations

In a section about design considerations Stonebraker et al. explain why relational DBMSs can be outperformed even in their home market of business data processing and how their own DBMS prototype H-Store can “achieve dramatically better performance than current RDBMSs”. Their considerations especially reflect the hardware development over the past decades and how it could or should have changed the architecture of RDBMSs in order to gain benefit from faster and bigger hardware, which also gives hints for “the complete rewrite” they insist on.

Stonebraker et al. see five particularly significant areas in database design:

**Main Memory** As—compared to the 1970s—enormous amounts of main memory have become cheap and available and as “The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing [...] quite slowly” they conclude that such databases are “capable of main memory deployment now or in near future”. Stonebraker et al. therefore consider the OLTP market a main memory market even today or in near future. They criticize therefore that “the current RDBMS vendors have disk-oriented solutions for a main memory problem. In summary, 30 years of Moore’s law has antiquated the disk-oriented relational architecture for OLTP applications”. Although there are relational databases operating in memory (e.g. TimesTen, SolidDB) these systems also inherit “baggage”—as Stonebraker et al. call it—from System R, e.g. disk-based recovery logs or dynamic locking, which have a negative impact on the performance of these systems.

**Multi-Threading and Resource Control** As discussed before, Stonebraker et al. consider databases a main-memory market. They now argue that transactions typically affect only a few data sets that have to be read and/or written (at most 200 read in the TPC-C benchmark for example) which is very cheap if all of this data is kept in memory and no disk I/O or user stalls are present. As a consequence, they do not see any need for multithreaded execution models in such main-memory databases which makes a considerable amount of “elaborate code” of conventional relational databases irrelevant, namely multi-threading systems to maximize CPU- and disk-usage, resource governors limiting load to avoid resource exhausting and multi-threaded datastructures like concurrent B-trees. “This results in a more reliable system, and one with higher performance”, they argue. To avoid long running transactions in such a single-threaded system they require either the application to break up such transactions into smaller ones or—in the case of analytical purposes—to run these transactions in data warehouses optimized for this job.

**Grid Computing and Fork-Lift Upgrades** Stonebraker et al. furthermore outline the development from shared-memory architectures of the 1970s over shared-disk architectures of the 1980s towards shared-nothing approaches of today and the near future, which are “often called grid computing or blade computing”. As a consequence, Stonebraker et al. insist that databases have to reflect this development, for example (and most obviously) by horizontal partitioning of data over several nodes of a DBMS grid. Furthermore, they advocate for incremental horizontal expansion of such grids without the need to reload any or all data by an administrator and also without downtimes. They point out that these requirements have significant impact on the architecture of DBMSs—e.g. the ability to transfer parts of the data between nodes without impacting running transactions—which probably cannot be easily added to existing RDBMSs but can be considered in the design of new systems (as younger databases like Vertica show).

**High Availability** The next topics Stonebraker et al. address are high availability and failover. Again, they outline the historical development of these issues from log-tapes that have been sent off site by organizations and were run on newly delivered hardware in the case of disaster over disaster recovery services installing log-tapes on remote hardware towards hot standby or multiple-site solutions that are common today. Stonebraker et al. regard high availability and built-in disaster recovery as a crucial feature for DBMSs which—like the other design issues they mention—has to be considered in the architecture and design of these systems. They particularly require DBMSs in the OLTP field to

1. “keep multiple replicas consistent, requiring the ability to run seamlessly on a grid of geographically dispersed systems”
2. “start with shared-nothing support at the bottom of the system” instead of gluing “multi-machine support onto [...] SMP architectures.”
3. support a shared-nothing architecture in the best way by using “multiple machines in a peer-to-peer configuration” so that “load can be dispersed across multiple machines, and inter-machine replication can be utilized for fault tolerance”. In such a configuration all machine resources can be utilized during normal operation and failures only cause a degraded operation as fewer resources are available. In contrast, todays HA solutions having a hot standby only utilize part of the hardware resources in normal operation as standby machines only wait for the live machines to go down. They conclude that “[these] points argue for a complete redesign of RDBMS engines so they can implement peer-to-peer HA in the guts of a new architecture” they conclude this aspect.

In such a highly available system that Stonebraker et al. require they do not see any need for a redo log as in the case of failure a dead site resuming activity “can be refreshed from the data on an operational site”. Thus, there is only a need for an undo log allowing to rollback transactions. Such an undo log does not have to be persisted beyond a transaction and therefore “can be a main memory data structure that

is discarded on transaction commit". As "In an HA world, one is led to having no persistent redo log, just a transient undo one" Stonebraker et al. see another potential to remove complex code that is needed for recovery from a redo log; but they also admit that the recovery logic only changes "to new functionality to bring failed sites up to date from operational sites when they resume operation".

**No Knobs** Finally, Stonebraker et al. point out that current RDBMSs were designed in an "era, [when] computers were expensive and people were cheap. Today we have the reverse. Personnel costs are the dominant expense in an IT shop". They especially criticize that "RDBMSs have a vast array of complex tuning knobs, which are legacy features from a bygone era" but still used as automatic tuning aids of RDBMSs "do not produce systems with anywhere near the performance that a skilled DBA can produce". Instead of providing such features that only try to figure out a better configuration for a number of knobs Stonebraker et al. require a database to have no such knobs at all but to be "self-everything" (self-healing, self-maintaining, self-tuning, etc.)".

### Considerations Concerning Transactions, Processing and Environment

Having discussed the historical development of the IT business since the 1970s when RDBMSs were designed and the consequences this development should have had on their architecture Stonebraker et al. now turn towards other issues that impact the performance of these systems negatively:

- Persistent redo-logs have to be avoided since they are "almost guaranteed to be a significant performance bottleneck". In the HA/failover system discussed above they can be omitted totally.
- Communication between client and DBMS-server via JDBC/ODBC-like interfaces is the next performance-degrading issue they address. Instead of such an interface they "advocate running application logic – in the form of stored procedures – "in process" inside the database system" to avoid "the inter-process overheads implied by the traditional database client / server model."
- They suggest furthermore to eliminate an undo-log "wherever practical, since it will also be a significant bottleneck".
- The next performance bottleneck addressed is dynamic locking to allow concurrent access. The cost of dynamic locking should also be reduced or eliminated.
- Multi-threaded datastructures lead to latching of transactions. If transaction runtimes are short, a single-threaded execution model can eliminate this latching and the overhead associated with multi-threaded data structures "at little loss in performance".
- Finally, two-phase-commit (2PC) transactions should be avoided whenever possible as the network round trips caused by this protocol degrade performance since they "often take the order of milliseconds".

If these suggestions can be pursued depends on characteristics of OLTP transactions and schemes as Stonbraker et al. point out subsequently.

### Transaction and Schema Characteristics

In addition to hardware characteristics, threading model, distribution or availability requirements discussed above, Stonebraker et al. also point out that the characteristics of database schemes as well as transaction properties also significantly influence the performance of a DBMS. Regarding database schemes and transactions they state that the following characteristics should be exploited by a DBMS:

**Tree Schemes** are database schemes in which “every table except a single one called *root*, has exactly one join term which is a 1-n relationship with its ancestor. Hence, the schema is a tree of 1-n relationships”. Schemes with this property are especially easy to distribute among nodes of a grid “such that all equi-joins in the tree span only a single site”. The root table of such a schema may be typically partitioned by its primary key and moved to the nodes of a grid, so that on each node has the partition of the root table together with the data of all other tables referencing the primary keys in that root table partition.

**Constrained Tree Application (CTA)** in the notion of Stonebraker et al. is an application that has a tree schema and only runs transactions with the following characteristics:

1. “every command in every transaction class has equality predicates on the primary key(s) of the root node”
2. “every SQL command in every transaction class is local to one site”

Transaction classes are collections of “the same SQL statements and program logic, differing in the run-time constants used by individual transactions” which Stonebraker et al. require to be defined in advance in their prototype H-Store. They furthermore argue that current OLTP applications are often designed to be CTAs or that it is at least possible to decompose them in that way and suggest schema transformations to be applied systematically in order to make an application CTA (cf. [SMA<sup>+</sup>07, page 1153]. The profit of these efforts is that “CTAs [...] can be executed very efficiently”.

**Single-Sited Transactions** can be executed to completion on only one node without having to communicate with other sites of a DBMS grid. Constrained tree application e.g. have that property.

**One-Shot Applications** entirely consist of “transactions that can be executed in parallel without requiring intermediate results to be communicated among sites”. In addition, queries in one-shot applications never use results of earlier queries. These properties allow the DBMS to decompose transactions “into a collection of single-site plans which can be dispatched to the appropriate sites for execution”. A common technique to make applications one-shot is to partition tables vertically among sites.

**Two-Phase Transactions** are transactions that contain a first phase of read operations, which can—depending on its results—lead to an abortion of the transaction, and a second phase of write operations which are guaranteed to cause no integrity violations. Stonebraker et al. argue that a lot of OLTP transactions have that property and therefore exploit it in their H-Store prototype to get rid of the undo-log.

**Strongly Two-Phase Transactions** in addition to two-phase transactions have the property that in the second phase all sites either rollback or complete the transaction.

**Transaction Commutativity** is defined by Stonebraker et al. as follows: “Two concurrent transactions from the same or different classes *commute* when any interleaving of their single-site sub-plans produces the same final database state as any other interleaving (assuming both transactions commit)”.

**Sterile Transactions Classes** are those that commute “with all transaction classes (including itself)”.

## H-Store Overview

Having discussed the parameters that should be considered when designing a database management system today, Stonebraker et al. sketch their prototype H-Store that performs significantly better than a commercial DBMS in the TPC-C benchmark. Their system sketch shall not need to be repeated in this paper (details can be found in [SMA<sup>+</sup>07, p. 154ff], but a few properties shall be mentioned:

- H-Store runs on a grid

- On each rows of tables are placed contiguously in main memory
- B-tree indexing is used
- Sites are partitioned into logical sites which are dedicated to one CPU core
- Logical sites are completely independent having their own indexes, tuple storage and partition of main memory of the machine they run on
- H-Store works single-threaded and runs transactions uninterrupted
- H-Store allows to run only predefined transactions implemented as stored procedures
- H-Store omits the redo-log and tries to avoid writing an undo-log whenever possible; if an undo-log cannot be avoided it is discarded on transaction commit
- If possible, query execution plans exploit the single-sited and one-shot properties discussed above
- H-Store tries to achieve the no-knobs and high availability requirements as well as transformation of transactions to be single-sited by “an automatical physical database designer which will specify horizontal partitioning, replication locations, indexed fields”
- Since H-Store keeps replicas of each table these have to be updated transactionally. Read commands can go to any copy of a table while updates are directed to all replicas.
- H-Store leverages the above mentioned schema and transaction characteristics for optimizations, e.g. omitting the undo-log in two-phase transactions.

### TPC-C Benchmark

When comparing their H-Store prototype with a commercial relational DBMS Stonebraker et al. apply some important tricks to the benchmarks implementation. First, they partition the database scheme and replicate parts of it in such a way that “the schema is decomposed such that each site has a subset of the records rooted at a distinct partition of the warehouses”. Secondly, they discuss how to profit of transaction characteristics discussed above. If the benchmark would run “on a single core, single CPU machine” then “every transaction class would be single-sited, and each transaction can be run to completion in a single-threaded environment”. In a “paired-HA site [...] all transaction classes can be made strongly two-phase, meaning that all transactions will either succeed or abort at both sites. Hence, on a single site with a paired HA site, ACID properties are achieved with no overhead whatsoever.“ By applying some further tricks, they achieve that ”with the basic strategy [of schema partitioning and replication (the author of this paper)] augmented with the tricks described above, all transaction classes become one-shot and strongly two-phase. As long as we add a short delay [...], ACID properties are achieved with no concurrency control overhead whatsoever.”

Based on this setting, they achieved a 82 times better performance compared to a commercial DBMS. They also analyzed the overhead of performance in the commercial DBMS and examined that it was mainly caused by logging and concurrency control.

It has to be said that Stonebraker et al. implemented only part of the TPC-C benchmark and do not seem to have adapted the TPC-C benchmark to perfectly fit with the commercial DBMS as they did for their own H-Store prototype although they hired a professional DBA to tune the DBMS they compared to H-Store and also tried to optimize the logging of this system to allow it to perform better.

## Consequences

As a result of their analysis Stonebraker et al. conclude that “we are heading toward a world with at least 5 (and probably more) specialized engines and the death of the “one size fits all” legacy systems”. This applies to the relational model as well as its query language SQL.

Stonebraker et al. state that in contrast to the 1970s when “the DBMS world contained only business data processing applications” nowadays there are at least the following markets which need specialized DBMSs<sup>4</sup>:

1. **Data warehouses** which typically have star or snowflake schemes, i.e. “a central fact table with 1-n joins to surrounding dimension tables, which may in turn participate in further 1-n joins to second level dimension tables, and so forth”. These datastructures could be easily modeled using the relational model but Stonebraker et al. suggest an entity-relationship model in this case which would be simpler and more natural to model and to query.
2. The **stream processing** market has different requirements, namely to “Process streams of messages at high speed [and to] Correlate such streams with stored data”. An SQL generalization called StreamSQL which allows to mix streams and relational data in SQL FROM-clauses has caused some enthusiasm and been suggested for standardization in this field. Stonebraker et al. also mention a problem in stream processing often requiring stream data to be flat (as some news agencies deliver it) but there is also a need for hierarchically structured data. Therefore, they “expect the stream processing vendors to move aggressively to hierarchical data models” and that “they will assuredly deviate from Ted Codd’s principles”.
3. **Text processing** is a field where relational databases have never been used.
4. **Scientific-oriented databases** will likely supply arrays rather than tables as their basic data structure.
5. **Semi-structured data** is a field where useful data models are still being discussed. Suggestions include e.g. XML schema (fiercely debated because of its complexity) and RDF.

While “the relational model was developed for a “one size fits all” world, the various specialized systems which we envision can each rethink what data model would work best for their particular needs” Stonebraker et al. conclude.

Regarding query languages for DBMSs they argue against a “one size fits all language” like SQL as, in their opinion, it has no use case at all: in the OLTP market ad-hoc queries are seldom or not needed, applications query in a prepared statement fashion or the query logic is deployed into the DBMS as stored procedures. In opposition to this, other DBMS markets like data warehouses need abilities for complex ad-hoc queries which are not fulfilled by SQL. For this reason, Stonebraker et al. do not see a need for this language any more.

For specialized DBMSs in the above mentioned markets they furthermore discuss how these languages should integrate with programming languages and argue against data sublanguages “interfaced to any programming language” as it is done in JDBC and ODBC. “[this] has led to high overhead interfaces”. Stonebraker et al. therefore suggest an “embedding of database capabilities in programming languages”. Examples of such language embeddings include Pascal R and Rigel in the 1970s or Microsoft’s LINQ-approach in the .NET platform nowadays. Regarding the languages to integrate such capabilities they are in favor of what they call “little languages” like Python, Perl, Ruby and PHP which are “open source, and can be altered by the community” and are additionally “less daunting to modify than the current general

---

<sup>4</sup>Further details on these DBMS markets can be found in the earlier released paper “One Size Fits All”: An Idea Whose Time Has Come and Gone (cf. [Sc05])

purpose languages" (that appear to them as a "one size fits all approach in the programming languages world"). For their H-Store prototype they plan to move from C++ to Ruby for stored procedures.

### 2.1.3. Main Drivers

The NoSQL movement has attracted a great number of companies and projects over the last couple of years. Especially big web companies or businesses running large and highly-frequented web sites have switched from relational databases (often with a caching layer typically realized with memcached<sup>5</sup>, cf. [Hof10c], [Hof10b]) towards non-relational datastores. Examples include Cassandra ([Apa10d]) originally developed at Facebook and also used by Twitter and Digg today (cf. [Pop10a], [Eur09]), Project Voldemort developed and used at LinkedIn, cloud services like the NoSQL store Amazon SimpleDB (cf. [Ama10b]) as well as Ubuntu One, a cloud storage and synchronization service based on CouchDB (cf. [Can10c], [Hof10c]). These users of NoSQL datastores are naturally highly interested in the further development of the non-relational solutions they use. However, most of the popular NoSQL datastores have adopted ideas of either Google's Bigtable (cf. [CDG<sup>+</sup>06]) or Amazon's Dynamo (cf. [DHJ<sup>+</sup>07]); Bigtable-inspired NoSQL stores are commonly referred to as column-stores (e.g. HyperTable, HBase) whereas the Dynamo influenced most of the key-/values-stores (e.g. Cassandra [Apa10d], Redis [S<sup>+</sup>10], Project Voldemort [K<sup>+</sup>10a]). Other projects pursue different attempts like graph-databases that form an own class of data stores and document-stores which can be seen as key/value-stores with additional features (at least allowing different, often hierarchical namespaces for key/value-pairs which provides the abstraction of "documents"); at least one of the latter (CouchDB) is a reimplementation of existing document stores such as Lotus Notes, which has been around since the 1990s and is consequently not designed with the awareness of current web technologies (which is criticized by CouchDB-developers who consequently implemented their document-store from scratch, cf. [Apa10c], [PLL09]). In summary, it can be concluded that the pioneers of the NoSQL movement are mainly big web companies or companies running large-scale web sites like Facebook, Google and Amazon (cf. [Oba09a]) and others in this field have adopted their ideas and modified them to meet their own requirements and needs.

## 2.2. Criticism

### 2.2.1. Scepticism on the Business Side

In an article about the NoSQL meet-up in San Francisco Computerworld mentions some business related issues concerning NoSQL databases. As most of them are open-source software they are well appreciated by developers who do not have to care about licensing and commercial support issues. However, this can scare business people in particular in the case of failures with nobody to blame for. Even at Adobe the developers of ConnectNow which uses a Terracotta cluster instead of a relational database were only able to convince their managers when they saw the system up and running (cf. [Com09a]).

### 2.2.2. NoSQL as a Hype

Some businesses appear to be cautious towards NoSQL as the movement seems like a hype potentially lacking the fulfillment of its promises. This is a general skepticism towards new technologies provoking considerable enthusiasm and has been expressed e.g. by James Bezdek in an IEEE editorial as follows:

---

<sup>5</sup>Examples include Friendfeed, Wikipedia, XING, StudiVz/SchülerVz/MeinVz

"Every new technology begins with naïve euphoria – its inventor(s) are usually submerged in the ideas themselves; it is their immediate colleagues that experience most of the wild enthusiasm. Most technologies are overpromised, more often than not simply to generate funds to continue the work, for funding is an integral part of scientific development; without it, only the most imaginative and revolutionary ideas make it beyond the embryonic stage. Hype is a natural handmaiden to overpromise, and most technologies build rapidly to a peak of hype. Following this, there is almost always an overreaction to ideas that are not fully developed, and this inevitably leads to a crash of sorts, followed by a period of wallowing in the depths of cynicism. Many new technologies evolve to this point, and then fade away. The ones that survive do so because someone finds a good use (= true user benefit) for the basic ideas." (cf. [Bez93] cited in [For10])

The participants of the NoSQL meet-up in San Francisco gave pragmatic advice for such remarks: companies do not miss anything if they do not switch to NoSQL databases and if a relational DBMS does its job, there is no reason to replace it. Even the organizer of the meet-up, Johan Oskarsson of Last.fm, admitted that Last.fm did not yet use a NoSQL database in production as of June 2009. He furthermore states that NoSQL databases "aren't relevant right now to mainstream enterprises, but that might change one to two years down the line" (Johan Oskarsson, Last.fm, cf. [Com09a]). Nonetheless, the participants of the NoSQL meet-up suggest to take a look at NoSQL alternatives if it is possible and it makes sense (e.g. in the development of new software) (cf. [Com09a]). Blogger Dennis Forbes does not see any overenthusiasm among the inventors and developers of non-relational datastores ("most of them are quite brilliant, pragmatic devs") but rather among developers using these technologies and hoping that "this movement invalidates their weaknesses". He—coming from traditional relational database development for the financial, insurance, telecommunication and power generation industry—however states that "there is indisputably a lot of fantastic work happening among the NoSQL camp, with a very strong focus on scalability". On the other hand, he criticizes in a postnote to his blog post that "the discussion is, by nature of the venue, hijacked by people building or hoping to build very large scale web properties (all hoping to be the next Facebook), and the values and judgments of that arena are then cast across the entire database industry—which comprises a set of solutions that absolutely dwarf the edge cases of social media—which is really...extraordinary" (cf. [For10]).

### 2.2.3. NoSQL as Being Nothing New

Similar to the hype argument are common remarks by NoSQL critics that NoSQL databases are nothing new since other attempts like object databases have been around for decades. As an example for this argument blogger David Intersimone mentions Lotus Notes which can be subsumed as an early document store supporting distribution and replication while favoring performance over concurrency control (unless otherwise indicated, [Int10]). This example is especially interesting as the main developer of CouchDB, Damien Katz, worked for Lotus Notes for several years. NoSQL advocates comment that CouchDB is 'Notes done right' as Notes' distribution features were not aware of current web technologies and the software also got bloated with business relevant features instead of being just a slim datastore ([PLL09]).

Examples like Lotus Notes, business analytic or stream processing oriented datastores show that these alternatives to relational databases have existed for a long time and blogger Dennis Forbes criticizes therefore that the "one size fitting it all" argument is nothing more but a strawman since few people ever held "Rdbms' as the only tool for all of your structured and unstructured data storage needs" (cf. [For10]).

### 2.2.4. NoSQL Meant as a Total “No to SQL”

At first, many NoSQL advocates especially in the blogosphere understood the term and the movement as a total denial of RDBMSs and proclaimed the death of these systems. Eric Evans to whom the term “NoSQL” is often credited though he was not the first who used it (see section 2.1 and e.g. [Eli09a]) suggested in a blog post of 2009 that the term now should mean “Not only SQL” instead of “No to SQL” (cf. [Eva09b]). This term has been adopted by many bloggers as it stresses that persistence in databases does not automatically mean to use a relational DBMS but that alternatives exist. Blogger Nati shalom comments this shift in the following way: “I think that what we are seeing is more of a realization that existing SQL database alternatives are probably not going away any time soon, but at the same time they can't solve all the problems of the world. Interestingly enough the term NOSQL has now been changed to Not Only SQL, to represent that line of thought” (cf. [Sha09a]). Some bloggers stress that the term is imprecise such as Dare Obasanjo saying that “there is a[sic!] yet to be a solid technical definition of what it means for a product to be a “NoSQL” database aside from the fact that it isn't a relational database” (cf. [Oba09a]). Others, such as Michael Stonebraker claim that it has nothing to do with SQL at all<sup>6</sup> and should be named something like “NoACID” as suggested by Dennis Forbes, who refers to Stonebraker in his suggestion ([For10]). Still others such as Adam Keys criticize the term “NoSQL” as defining the movement by what it does not stand for ([Key09]): “The problem with that name is that it only defines what it is not. That makes it confrontational and not amazingly particular to what it includes or excludes. [...] What we're seeing its [sic!] the end of the assumption that valuable data should go in some kind of relational database. The end of the assumption that SQL and ACID are the only tools for solving our problems. The end of the viability of master/slave scaling. The end of weaving the relational model through our application code”. He suggests to subsume the movement and the datastores under the term “post-relational” instead of “NoSQL”: “We're seeing an explosion in the ideas about how one should store important data. We're looking at data to see if it's even worth persisting. We're experimenting with new semantics around structure, consistency and concurrency. [...] In the same way that post-modernism is about reconsidering the ways of the past in art and architecture, post-relational is a chance for software developers to reconsider our own ways. Just as post-modernism didn't invalidate the entire history of art, post-relational won't invalidate the usefulness of relational databases.” (cf. [Key09]). However, as the readers of his blog post comment this term is not that much better than “NoSQL” as it still defines the movement and the databases by what they do not reflect (or better: by what they have omitted) instead of what they stand for.

The irritation about the term and its first notion as a total neglect of relational databases has lead to many provoking statements by NoSQL advocates<sup>7</sup> and caused a number of unfruitful discussions and some flamewars (see e.g. [Dzi10] and as a response to it [Sch10]).

### 2.2.5. Stonebraker's Critical Reception of NoSQL Databases

In his blog post “The “NoSQL” Discussion has Nothing to Do With SQL” ([Sto09]) Michael Stonebraker states that there has been a lot of buzz around NoSQL databases lately. In his reception the main drivers behind NoSQL conferences in the US are advocates of document stores and key/values-stores which provide a “a low-level record-at-a-time DBMS interface, instead of SQL” in his sight. Stonebraker sees two reasons for moving towards non-relational datastores—flexibility and performance.

<sup>6</sup>See the next section on Stonebraker's reception of the NoSQL-movement which goes far beyond criticizing only the term “NoSQL”.

<sup>7</sup>E.g. “The ACIDy, Transactional, RDBMS doesn't scale, and it needs to be relegated to the proper dustbin before it does any more damage to engineers trying to write scalable software”. The opinion expressed here has been softened by the author in a postnote to his blog post: “This isn't about a complete death of the RDBMS. Just the death of the idea that it's a tool meant for all your structured data storage needs.” (cf. [Ste09] cited in [For10])

**The Flexibility Argument** is not further examined by Stonebraker but it contains the following view: there might be data that does not fit into a rigid relational model and which is bound too much by the structure of a RDBMS. For this kind of data something more flexible is needed.

**The Performance Argument** is described as follows Stonebraker: one starts with MySQL to store data and performance drops over the time. This leads to the following options: either to shard/partition data among several sites causing “a serious headache managing distributed data” in the application; or to move from MySQL towards a commercial RDMBS which can provoke large licensing fees; or to even totally abandon a relational DBMS.

Stonebraker subsequently examines the latter argument in his blog post. He focuses on “workloads for which NoSQL databases are most often considered: update- and lookup-intensive OLTP workloads, not query-intensive data warehousing workloads” or specialized workflows like document-repositories.

Stonebraker sees two options to improve the performance of OLTP transactions:

1. Horizontal scaling achieved by automatic sharding “over a shared-nothing processing environment”. In this scenario performance gets improved by adding new nodes. In his point of view, RDMBSs written in the last ten years provide such a “shared nothing architecture” and “nobody should ever run a DBMS that does not provide” this.
2. Improvement of the OLTP performance of a single node.

Stonebraker focuses on the second option and analyzes the sources of overhead which decrease performance of OLTP transactions on a single node; as indicated by the title of his blog post, these have no relation to the the query language (SQL). He states only a small percentage of total transaction cost is incurred by useful work and sees five sources of performance overhead that have to be addressed when single node performance should be optimized:

**Communication** between the application and the DBMS via ODBC or JDBC. This is considered the main source of overhead in OLTP transactions which is typically addressed as follows: “Essentially **all** applications that are performance sensitive use a stored-procedure interface to run application logic inside the DBMS and avoid the crippling overhead of back-and-forth communication between the application and the DBMS”. The other option to reduce communication overhead is using an embeddable DBMS which means that the application and the DBMS run in the same address space; because of strong coupling, security and access control issues this is no viable alternative “for mainstream OLTP, where security is a big deal” in Stonebrakers sight.

**Logging** is done by traditional DBMSs in addition to modifications of relational data on each transaction. As log files are persisted to disk to ensure durability logging is expensive and decreases transaction performance.

**Locking** of datasets to be manipulated causes overhead as write operations in the lock-table have to occur before and after the modifications of the transaction.

**Latching** because of shared data structures (e. g. B-trees, the lock-table, resource-tables) inside an RDBMS incurs further transaction costs. As these datastructures have to be accessed by multiple threads short-term locks (aka latches) are often used to provide parallel but careful access to them.

**Buffer Management** finally also plays its part when it comes to transaction overhead. As data in traditional RDBMSs is organized in fixed pages work has to be done to manage the disk-pages cached in memory (done by the buffer-pool) and also to resolve database entries to disk pages (and back) and identify field boundaries.

As stated before, communication is considered the main source of overhead according to Stonebraker and by far outweighs the other ones (take from this survey: [HAMS08]) which almost equally increase total transaction costs. Besides avoiding communication between the application and the database all four other sources of performance overhead have to be eliminated in order to considerably improve single node performance.

Now, datastores whether relational or not have specific themes in common and NoSQL databases also have to address the components of performance overhead mentioned above. In this context, Stonebraker raises the following examples:

- Distribution of data among multiple sites and a shared-nothing approach is provided by relational as well as non-relational datastores. “Obviously, a well-designed multi-site system, whether based on SQL or something else, is way more scalable than a single-site system” according to Stonebraker.
- Many NoSQL databases are disk-based, implement a buffer pool and are multi-threaded. When providing these features and properties, two of the four sources of performance overhead still remain (Locking, Buffer management) and cannot be eliminated.
- Transaction-wise many NoSQL datastores provide only single-record transactions with BASE properties (see chapter 3 on that). In contrast to relational DBMSs ACID properties are sacrificed in favor of performance.

Stonebraker consequently summarizes his considerations as follows: “However, the net-net is that the single-node performance of a NoSQL, disk-based, non-ACID, multithreaded system is limited to be a modest factor faster than a well-designed stored-procedure SQL OLTP engine. In essence, ACID transactions are jettisoned for a modest performance boost, and this performance boost has nothing to do with SQL”. In his point of view, the real tasks to speed up a DBMS focus on the elimination of locking, latching, logging and buffer management as well as support for stored procedures which compile a high level language (such as SQL) into low level code. How such a system can look like is described in the paper “The end of an architectural era: (it’s time for a complete rewrite)” ([SMA<sup>+</sup>07]) that has already been discussed above.

Stonebraker also does not expect SQL datastores to die but rather states: “I fully expect very high speed, open-source SQL engines in the near future that provide automatic sharding. [...] Moreover, they will continue to provide ACID transactions along with the increased programmer productivity, lower maintenance, and better data independence afforded by SQL.” Hence “high performance does not require jettisoning either SQL or ACID transactions”. It rather “depends on removing overhead” caused by traditional implementations of ACID transactions, multi-threading and disk management. The removal of these sources of overhead “is possible in either a SQL context or some other context”, Stonebraker concludes.

## 2.2.6. Requirements of Administrators and Operators

In his blog post “The dark side of NoSQL” (cf. [Sch09]) Stephan Schmidt argues that the NoSQL debate is dominated by a developer’s view on the topic which usually iterates on properties and capabilities developers like (e.g. performance, ease of use, schemalessness, nice APIs) whereas the needs of operations people and system administrators are often forgotten in his sight. He reports that companies<sup>8</sup> encounter difficulties especially in the following fields:

**Ad Hoc Data Fixing** To allow for ad hoc data fixing there first has to be some kind of query and manipulation language. Secondly, it is more difficult to fix data in distributed databases (like Project Voldemort or Cassandra) compared to datastores that run on a single node or have dedicated shards.

---

<sup>8</sup>He cites an Infinispan director and the vice president of engineering at a company called Loop.

**Ad Hoc Data Querying** Similarly to data fixing a query and manipulation for the particular datastore is required when it comes to ad hoc queries and querying distributed datastores is harder than querying centralized ones. Schmidt states that for some reporting tasks the MapReduce approach (cf. [DG04]) is the right one, but not for every ad hoc query. Furthermore, he sees the rather cultural than technical problem that customers have become trained and “addicted” to ad hoc reporting and therefore dislike the absence of these means. For exhaustive reporting requirements Schmidt suggests to use a relational database that mirrors the data of live databases for which a NoSQL store might be used due to performance and scalability requirements.

**Data Export** Schmidt states that there are huge differences among the NoSQL databases regarding this aspect. Some provide a useful API to access all data and in some it is absent. He also points out that it is more easy to export data from non-distributed NoSQL stores like CouchDB, MongoDB or Tokyo Tyrant as from distributed ones like Projekt Voldemort or Cassandra.

Schmidt's points are humorously and extensively iterated in the talk “Your Guide to NoSQL” (cf. [Ake09]) which especially parodies the NoSQL advocates' argument of treating every querying need in a MapReduce fashion.

### 2.2.7. Performance vs. Scalability

BJ Clark presents an examination of various NoSQL databases and MySQL regarding performance and scalability in his blog post “NoSQL: If only it was that easy”. At first, he defines scalability as “to change the size while maintaining proportions and in CS this usually means to increase throughput.” Blogger Dennis Forbes agrees with this notion of scalability as “pragmatically the measure of a solution's ability to grow to the highest realistic level of usage in an achievable fashion, while maintaining acceptable service levels” (cf. [For10]). BJ Clark continues: “What scaling isn't: performance. [...] In reality, scaling doesn't have anything to do with being fast. It has only to do with size. [...] Now, scaling and performance do relate in that typically, if something is performant, it may not actually need to scale.” (cf. [Cla09]).

Regarding relational databases Clark states that “The problem with RDBMS isn't that they don't scale, it's that they are incredibly hard to scale. Sharding[sic!] is the most obvious way to scale things, and sharding multiple tables which can be accessed by any column pretty quickly gets insane.” Blogger Dennis Forbes agrees with him that “There are some real scalability concerns with old school relational database systems” (cf. [For10]) but that it is still possible to make them scale using e.g. the techniques described by Adam Wiggins (cf. [Wig09]).

Besides sharding of relational databases and the avoidance of typical mistakes (as expensive joins caused by rigid normalization or poor indexing) Forbes sees vertical scaling as still an option that can be easy, computationally effective and which can lead far with “armies of powerful cores, hundreds of GBs of memory, operating against SAN arrays with ranks and ranks of SSDs”. On the downside, vertical scaling can relatively costly as Forbes also admits. But Forbes also argues for horizontal scaling of relational databases by partitioning data and adding each machine to a failover cluster in order to achieve redundancy and availability. Having deployments in large companies in mind where constraints are few and money is often not that critical he states from his own experience<sup>9</sup> that “This sort of scaling that is at the heart of virtually every bank, trading system, energy platform, retailing system, and so on. [...] To claim that SQL systems don't scale, in defiance of such *obvious* and *overwhelming* evidence, defies all reason” (cf. [For10]). Forbes argues for the use of own servers as he sees some artificial limits in cloud computing environments such as limitations of IO and relatively high expenditures for single instances in Amazon's EC2; he states that “These financial and artificial limits explain the strong interest in technologies that allows you to spin up and cycle down as needed” (cf. [For10]).

---

<sup>9</sup>Forbes has worked in the financial, assurance, telecommunication and power supply industry.

BJ Clark continues his blog post by an evaluation of some NoSQL datastores with a focus on automatic scalability (such as via auto-sharding) as he updates millions of objects (primary objects as he calls them) on which other objects depend in 1:1 and 1:n relationships (he calls them secondary objects); secondary objects are mostly inserted at the end of tables.

His evaluation can be summarized in the following way:

- The key/value-stores **Tokyo Tyrant/Cabinet** and **Redis** do not provide means for automatic, horizontal scalability. If a machine is added—similar to memcached—it has to be made known to the application which then (re-)hashes the identifiers of database entries against the collection of database servers to benefit from the additional resources. On the other hand he states that Tokyo Tyrant/Cabinet and Redis perform extremely well so that the need for scaling horizontally will not appear very early.
- The distributed key/value-store **Project Voldemort** utilizes additional servers added to a cluster automatically and also provides for fault tolerance<sup>10</sup>. As it concentrates on sharding and fault-tolerance and has a pluggable storage architecture, Tokyo Tyrant/Cabinet or Redis can be used as a storage backend for Voldemort. This might also be a migration path from these systems if they need to be scaled.
- The document-database **MongoDB** showed good performance characteristics but did not scale automatically at the time of the evaluation as it did not provide automatic sharding (which has changed in version 1.6 released in August 2010 cf. [Mon10] and [MHC<sup>+</sup>10b]).
- Regarding the column-database **Cassandra** Clark thinks that it is “definitely supposed to scale, and probably does at Facebook, by simply adding another machine (they will hook up with each other using a gossip protocol), but the OSS version doesn’t seem to support some key things, like losing a machine all together”. This conclusion reflects the development state of Cassandra at the time of the evaluation.
- The key/value store **S3** of Amazon scaled very well, although, due to Clark, it is not as performant as other candidates.
- **MySQL**, in comparison, does not provide automatic horizontal scalability by e.g. automatic sharding, but Clark states that for most applications (and even web applications like Friendfeed, cf. [Tay09]) MySQL is fast enough and—in addition—is “familiar and ubiquitous”. In comparison to Tokyo Tyrant/Cabinet and Redis Clark concludes that “It can do everything that Tokyo and Redis can do, and it really isn’t that much slower. In fact, for some data sets, I’ve seen MySQL perform ALOT[sic!] faster than Tokyo Tyrant” and “it’s just as easy or easier to shard MySQL as it is Tokyo or Redis, and it’s hard to argue that they can win on many other points.”

The systems mentioned in the above evaluation will be discussed in more detail later on in this paper. Besides, it has to be mentioned that the evaluation took place in summer of 2009 (the blog post is of August), therefore results reflect the development state of the evaluated systems at that time.

Clark summarizes his results by indicating that RDBMSs are not harder to scale than “lots of other things” in his perspective, and that only a couple of NoSQL databases provide means for automatic, horizontal scalability allowing to add machines while not requiring operators to interact. Therefore, it could be even argued that “it’s just as easy to scale mysql (with sharding via mysql proxy) as it is to shard some of these NoSQL dbs.” Therefore he does not proclaim an early death of RDBMSs and reminds that MySQL is still in use at big web sites like Facebook, Wikipedia and Friendfeed. For new applications he suggests to use the tool fitting the job best, reflecting that non-relational databases—just like relational ones—are no “one size fits all” solutions either:

---

<sup>10</sup>Redis in particular does not offer fault-tolerance and as data is held in memory it will be lost if a server crashes.

"If I need reporting, I won't be using any NoSQL. If I need caching, I'll probably use Tokyo Tyrant. If I need ACIDity, I won't use NoSQL. If I need a ton of counters, I'll use Redis. If I need transactions, I'll use Postgres. If I have a ton of a single type of documents, I'll probably use Mongo. If I need to write 1 billion objects a day, I'd probably use Voldemort. If I need full text search, I'd probably use Solr. If I need full text search of volatile data, I'd probably use Sphinx."

### 2.2.8. Not All RDBMSs Behave like MySQL

An argument often found in the NoSQL debate is that RDBMSs do not scale very well and are difficult to shard. This is often pointed out by the example of MySQL. Furthermore, NoSQL databases are sometimes seen as the successor of a MySQL plus memcached solution (cf. e.g. [Hof10c]) where the latter is taking load away from the database to decrease and defer the need to distribute it. Concerning these typical arguments blogger Dennis Forbes reminds that RDBMSs in general cannot be easily identified with MySQL but others may be easier or better scalable: "MySQL isn't the vanguard of the RDBMS world. Issues and concerns with it on high load sites have remarkably little relevance to other database systems" (cf. [For10]).

### 2.2.9. Misconceptions of Critics

NoSQL advocate Ben Scofield responds to some of the criticism mentioned above which he perceives to be misconceived. He does so by expressing incisive arguments from the NoSQL debate responding to them (cf. [Sco09]):

**"NoSQL is just about scalability and/or performance."** Scofield argues that this could be an attractive claim for those "traditionalists" (as he calls them) who think that NoSQL data stores can be made obsolete by making RDBMSs faster and more scalable. He claims that "there's a lot more to NoSQL than just performance and scaling" and that for example "NoSQL DBs often provide better substrates for modeling business domains".

**"NoSQL is just document databases, or key-value stores, or . . ."** Scofield notes that many NoSQL-articles address only document-oriented databases or key-value stores, sometimes column-stores. He states that "Good arguments can be made against each of those solutions for specific circumstances, but those are arguments against a specific type of storage engine." Scofield therefore criticizes that narrowing the discussion to only one sort of NoSQL databases allows traditionalists to argue easily against the whole movement or whole set of different NoSQL approaches.

**"I can do NoSQL just as well in a relational database."** With the frequent argument of Friendfeed's usage of MySQL (cf. [Tay09]) in mind, Scofield notes that although it is possible to tweak and tune a relational database this does not make sense for all types of data. "Different applications are good for different things; relational databases are great for relational data, but why would you want to use them for non-relational data?" he asks.

**"NoSQL is a wholesale rejection of relational databases."** Some time ago this claim was often heard in the NoSQL community but it became less common which Scofield appreciates that: "It seems that we're moving towards a pluralistic approach to storing our data, and that's a good thing. I've suggested 'polyglot persistence' for this approach (though I didn't coin the term), but I also like Ezra Zygmuntowicz's 'LessSQL' as a label, too."

## 2.3. Classifications and Comparisons of NoSQL Databases

In the last years a variety of NoSQL databases has been developed mainly by practitioners and web companies to fit their specific requirements regarding scalability performance, maintainance and feature-set. As it has been revealed some of these databases have taken up ideas from either Amazon's Dynamo (cf. [DHJ<sup>+</sup>07]) or Google's Bigtable (cf. [CDG<sup>+</sup>06]) or a combination of both. Others have ported ideas in existing databases towards modern web technologies such as CouchDB. Still others have pursued totally different approaches like Neo4j or HypergraphDB.

Because of the variety of these approaches and overlappings regarding the nonfunctional requirements and the feature-set it could be difficult to get and maintain an overview of the nonrelational database scene. So there have been various approaches to classify and subsume NoSQL databases, each with different categories and subcategories. Some classification approaches shall be presented here out of which one possibility to classify NoSQL datastores will be pursued in the subsequent chapters.

### 2.3.1. Taxonomies by Data Model

Concerning the classification of NoSQL stores Highscalability author Todd Hoff cites a presentation by Stephen Yen in his blog post "A yes for a NoSQL taxonomy" (cf. [Hof09c]). In the presentation "NoSQL is a Horseless Carriage" (cf. [Yen09]) Yen suggests a taxonomy that can be found in table 2.1.

Term	Matching Databases
Key-Value-Cache	Memcached Repcached Coherence Infinispan EXtreme Scale JBoss Cache Velocity Terracotta
Key-Value-Store	keyspace Flare Schema Free RAMCloud
Eventually-Consistent Key-Value-Store	Dynamo Voldemort Dynomite SubRecord Mo8onDb Dovetaildb
Ordered-Key-Value-Store	Tokyo Tyrant Lightcloud NMDB Luxio MemcacheDB Actord

Term	Matching Databases
Data-Structures Server	Redis
Tuple Store	Gigaspaces Coord Apache River
Object Database	ZopeDB DB4O Shoal
Document Store	CouchDB Mongo Jackrabbit XML Databases ThruDB CloudKit Perservere Riak Basho Scalarmis
Wide Columnar Store	Bigtable Hbase Cassandra Hypertable KAI OpenNeptune Qbase KDI

**Table 2.1.:** Classifications – NoSQL Taxonomy by Stephen Yen (cf. [Yen09])

A similar taxonomy which is less fine-grained and comprehensive than the classification above can be found in the article “Databases in the cloud” by Ken North (cf. [Nor09]). Table 2.2 summarizes his classification of datastores that additionally include some datastores available in cloud-computing environments only.

Category	Matching databases
Distributed Hash Table, Key-Value Data Stores	memcached MemcacheDB Project Voldemort Scalarmis Tokyo Cabinet
Entity-Attribute-Value Datastores	Amazon SimpleDB Google AppEngine datastore Microsoft SQL Data Services Google Bigtable Hadoop HyperTable HBase

Category	Matching databases
Amazon Platform	Amazon SimpleDB
Document Stores, Column Stores	Sybase IQ Vertica Analytic Database Apache CouchDB

**Table 2.2.:** Classifications – Categorization by Ken North (cf. [Nor09])

Similarly to the classifications mentioned above Rick Cattell subsumes different NoSQL databases primarily by their data model (cf. [Cat10]) as shown in table 2.3.

Category	Matching databases
Key-value Stores	Redis Scalairis Tokyo Tyrant Voldemort Riak
Document Stores	SimpleDB CouchDB MongoDB Terrastore
Extensible Record Stores	Bigtable HBase HyperTable Cassandra

**Table 2.3.:** Classifications – Categorization by Rick Cattell (cf. [Cat10])

### 2.3.2. Categorization by Ben Scofield

Blogger Alex Popescu summarizes a presentation by Ben Scofield who gave a generic introduction to NoSQL databases along with a categorization and some ruby examples of different NoSQL databases (cf. [Sco10]). The categorization is in fact a short comparison of classes of NoSQL databases by some nonfunctional categories (“(il)ities”) plus a rating of their feature coverage. Popescu summarizes Scofield’s ideas as presented in table 2.4.

	<b>Performance</b>	<b>Scalability</b>	<b>Flexibility</b>	<b>Complexity</b>	<b>Functionality</b>
Key-Value Stores	high	high	high	none	variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

**Table 2.4.:** Classifications – Categorization and Comparison by Scofield and Popescu (cf. [Pop10b], [Sco10])

### 2.3.3. Comparison by Scalability, Data and Query Model, Persistence-Design

In his blog post “NoSQL ecosystem” Jonathan Ellis discusses NoSQL datastores by three important aspects:

1. Scalability
2. Data and query model
3. Persistence design

#### Scalability

Ellis argues that it is easy to scale read operations by replication of data and load distribution among those replicas. Therefore, he only investigates the scaling of write operations in databases that are really distributed and offer automatic data partitioning. When data size exceeds the capabilities of a single machine the latter systems seem to be the only option to him if provided no will to partition data manually (which is not a good idea according to [Oba09b]). For the relevant distributed systems which provide auto-sharding Ellis sees two important features:

- Support for multiple datacenters
- Possibility to add machines live to an existing cluster, transparent for applications using this cluster

By these features Ellis compares a selection of NoSQL datastores fulfilling the requirements of real distribution and auto-sharding (cf. table 2.5).

<b>Datastore</b>	<b>Add Machines Live</b>	<b>Multi-Datacenter Support</b>
Cassandra	x	x
HBase	x	
Riak	x	
Scalaris	x	
Voldemort		Some code required

**Table 2.5.:** Classifications – Comparison of Scalability Features (cf. [Ell09a])

The following NoSQL stores have been excluded in this comparison as they are not distributed in the way Ellis requires (at the time of his blog post in November 2009): CouchDB, MongoDB, Neo4j, Redis and Tokyo Cabinet. Nonetheless, these systems can find use as a persistence layer for distributed systems, according to him. The document databases MongoDB and CouchDB supported limited support for auto-sharding at the time of his investigation (MongoDB out of the box, CouchDB by partitioning/clustering framework Lounge). Regarding Tokyo Cabinet, Ellis notices that it can be used as a storage backend for Project Voldemort.

### Data and Query Model

The second area Ellis examines is the data model and the query API offered by different NoSQL stores. Table 2.6 shows the results of his investigation pointing out a great variety of data models and query APIs.

Datastore	Data Model	Query API
Cassandra	Columnfamily	Thrift
CouchDB	Document	map/reduce views
HBase	Columnfamily	Thrift, REST
MongoDB	Document	Cursor
Neo4j	Graph	Graph
Redis	Collection	Collection
Riak	Document	Nested hashes
Scalairis	Key/value	get/put
Tokyo Cabinet	Key/value	get/put
Voldemort	Key/value	get/put

**Table 2.6.:** Classifications – Comparison of Data Model and Query API (cf. [Ell09a])

Ellis makes the following remarks concerning the data models and query APIs of these systems:

- The **columnfamily model**, implemented by Cassandra and HBase is inspired by the corresponding paragraph in the second section of Google's Bigtable paper (cf. [CDG<sup>+</sup>06, Page 2]). Cassandra omits historical versions in contrast to Bigtable and introduces the further concept of supercolumns. In Cassandra as well as HBase rows are sparse<sup>11</sup>, which means that they may have different numbers of cells and columns do not have to be defined in advance.
- The **key/value model** is easiest to implement but may be inefficient if one is only interested in requesting or updating part of the value associated with a certain key. Furthermore, it is difficult to build complex data structures on top of key/value stores (as Ellis describes in another blog post, cf. [Ell09b]).

---

<sup>11</sup>See e.g. the Wikipedia-article on Sparse arrays

- Ellis sees **document databases** being the next step from key/value stores as they allow nested values. They permit to query data structures more efficiently than key/value stores as they do not necessarily reply whole BLOBs when requesting a key.
- The **graph database** Neo4j has a unique data model in Ellis' selection as objects and their relationships are modelled and persisted as nodes and edges of a graph. Queries fitting this model well might be three orders faster than corresponding queries in the other stores discussed here (due to Emil Eifrem, CEO of the company behind Neo4j, cf. [Eif09]).
- **Scalaris** is unique among the selected key/value stores as it allows distributed transactions over multiple keys.

### Persistence Design

The third aspect by which Ellis compares his selection of NoSQL datastores is the way they store data (see table 2.7).

Datastore	Persistence Design
Cassandra	Memtable / SSTable
CouchDB	Append-only B-tree
HBase	Memtable / SSTable on HDFS
MongoDB	B-tree
Neo4j	On-disk linked lists
Redis	In-memory with background snapshots
Riak	?
Scalaris	In-memory only
Tokyo Cabinet	Hash or B-tree
Voldemort	Pluggable (primarily BDB MySQL)

**Table 2.7.:** Classifications – Comparison of Persistence Design (cf. [Ell09a])

Ellis considers persistence design as particularly important to estimate under which workloads these databases will perform well:

**In-Memory Databases** are very fast (e.g. Redis can reach over 100.000 operations per second on a single machine) but the data size is inherently limited by the size of RAM. Another downside is that durability may become a problem as the amount of data which can get lost between subsequent disk flushes (e.g. due to server crashes or by losing power supply) is potentially large. Scalaris addresses this issue by replication but—as it does not support multiple datacenters—threats like power supply failures remain.

**Memtables and SSTables** work in the following way: write operations are buffered in memory (in a Memtable) after they have been written to an append-only commit log to ensure durability. After a certain amount of writes the Memtable gets flushed to disk as a whole (and is called SSTable then; these ideas are taken from Google's Bigtable paper, cf. [CDG<sup>+</sup>06, Sections 5.3 and 5.4]). This

persistence strategy has performance characteristics comparable to those of in-memory-databases (as—compared to disk-based strategies—disk seeks are reduced due to the append-only log and the flushing of whole Memtables to disk) but avoids the durability difficulties of pure in-memory-databases.

**B-trees** have been used in databases since their beginning to provide a robust indexing-support. Their performance characteristics on rotating disks are not very positive due to the amount of disk seeks needed for read and write operations. The document-database CouchDB uses B-trees internally but tries to avoid the overhead of disk seeks by only appending to B-trees, which implies the downside that only one write operation at a time can happen as concurrent reads to different sections of a B-tree are not allowed in this case.

### Categorization Based on Customer Needs

Todd Hoff (cf. [Hof09b]) cites blogger James Hamilton who presents a different approach to classify NoSQL datastores (cf. [Ham09]) subsuming databases by customer requirements:

**Features-First** This class of databases provides a (large) number of high level features that make the programmer's job easier. On the downside, they are difficult to scale. Examples include: *Oracle*, *Microsoft SQL Server*, *IBM DB2*, *MySQL*, *PostgreSQL*, *Amazon RDS*<sup>12</sup>.

**Scale-First** This sort of databases has to scale from the start. On the downside, they lack particular features and put responsibility back to the programmer. Examples include: *Project Voldemort*, *Ringo*, *Amazon SimpleDB*, *Kai*, *Dynomite*, *Yahoo PNUTS*, *ThruDB*, *Hypertable*, *CouchDB*, *Cassandra*, *MemcacheDB*.

**Simple Structure Storage** This class subsumes key/value-stores with an emphasis on storing and retrieving sets of arbitrary structure. The downside according to Hamilton is that “they generally don't have the features or the scalability of other systems”. Examples include: *file systems*, *Cassandra*, *BerkelyDB*, *Amazon SimpleDB*.

**Purpose-Optimized Storage** These are databases which are designed and built to be good at one thing, e.g. data warehousing or stream processing. Examples of such databases are: *StreamBase*, *Vertica*, *VoltDB*, *Aster Data*, *Netezza*, *Greenplum*.

Hamilton and Hoff consider this categorization useful to match a given use case to a class of databases. Though, the categorization is not complete as e.g. graph-databases are missing and it is not clear how they would fit into the four categories mentioned above. Furthermore, it is important to notice that some databases can be found in different classes (*Cassandra*, *SimpleDB*) so the categorization does not provide a sharp distinction where each database only fits into one class (which is—in the authors opinion—at least difficult if not impossible in the field of NoSQL databases).

#### 2.3.4. Classification in this Paper

After considering basic concepts and techniques of NoSQL databases in the next chapter, the following chapters will study various classes of nonrelational datastores and also take a look on particular products. These products are classified by their datastructure, following the taxonomies of Yen, North and Cattell mentioned above (see 2.3.1 on page 23) which is also shared by most other sources of this paper. The classes used in this writing will be: key/value stores, document databases, and column-oriented databases.

---

<sup>12</sup>Relational Database Service providing cocooned MySQL instances in Amazon's cloud services.

# 3. Basic Concepts, Techniques and Patterns

This chapter outlines some fundamental concepts, techniques and patterns that are common among NoSQL datastores and not unique to only one class of nonrelational databases or a single NoSQL store. Specific concepts and techniques of the various classes of NoSQL datastores and individual products will be discussed in the subsequent chapters of this paper.

## 3.1. Consistency

### 3.1.1. The CAP-Theorem

In a keynote titled “Towards Robust Distributed Systems” at ACM’s PODC<sup>1</sup> symposium in 2000 Eric Brewer came up with the so called CAP-theorem (cf. [Bre00]) which is widely adopted today by large web companies (e.g. Amazon, cf. [Vog07], [Vog08]) as well as in the NoSQL community. The CAP acronym stands for (as summarized by Gray in [Gra09]):

**Consistency** meaning if and how a system is in a consistent state after the execution of an operation. A distributed system is typically considered to be consistent if after an update operation of some writer all readers see his updates in some shared data source. (Nevertheless there are several alternatives towards this strict notion of consistency as we will see below.)

**Availability** and especially high availability meaning that a system is designed and implemented in a way that allows it to continue operation (i.e. allowing read and write operations) if e.g. nodes in a cluster crash or some hardware or software parts are down due to upgrades.

**Partition Tolerance** understood as the ability of the system to continue operation in the presence of network partitions. These occur if two or more “islands” of network nodes arise which (temporarily or permanently) cannot connect to each other. Some people also understand partition tolerance as the ability of a system to cope with the dynamic addition and removal of nodes (e.g. for maintenance purposes; removed and again added nodes are considered an own network partition in this notion; cf. [Ipp09]).

Now, Brewer alleges that one can at most choose two of these three characteristics in a “shared-data system” (cf. [Bre00, slide 14]). In his talk, he referred to trade-offs between ACID and BASE systems (see next subsection) and proposed as a decision criteria to select one or the other for individual use-cases: if a system or parts of a system have to be consistent and partition-tolerant, ACID properties are required and if availability and partition-tolerance are favored over consistency, the resulting system can be characterized by the BASE properties. The latter is the case for Amazon’s Dynamo (cf. [DHJ<sup>+</sup>07]), which is available and partition-tolerant but not strictly consistent, i.e. writes of one client are not seen immediately after being committed to all readers. Google’s Bigtable chooses neither ACID nor BASE but the third CAP-alternative being a consistent and available system and consequently not able to fully operate in the presence of network partitions. In his keynote Brewer points out traits and examples of the three different choices that can be made according to his CAP-theorem (see table 3.1).

---

<sup>1</sup>Principles of Distributed Computing

Choice	Traits	Examples
Consistence + Availability (Forfeit Partitions)	2-phase-commit cache-validation protocols	Single-site databases Cluster databases LDAP xFS file system
Consistency + Partition tolerance (Forfeit Availability)	Pessimistic locking Make minority partitions unavailable	Distributed databases Distributed locking Majority protocols
Availability + Partition tolerance (Forfeit Consistency)	expirations/leases conflict resolution optimistic	Coda Web caching [sic!] DNS

**Table 3.1.:** CAP-Theorem – Alternatives, Traits, Examples (cf. [Bre00, slides 14–16])

With regards to databases, Brewer concludes that current “Databases [are] better at C[onsistency] than Availability” and that “Wide-area databases can’t have both” (cf. [Bre00, slide 17])—a notion that is widely adopted in the NoSQL community and has influenced the design of nonrelational datastores.

### 3.1.2. ACID vs. BASE

The internet with its wikis, blogs, social networks etc. creates an enormous and constantly growing amount of data needing to be processed, analyzed and delivered. Companies, organizations and individuals offering applications or services in this field have to determine their individual requirements regarding performance, reliability, availability, consistency and durability (cf. [Gra09]). As discussed above, the CAP-theorem states that a choice can only be made for two options out of consistency, availability and partition tolerance. For a growing number of applications and use-cases (including web applications, especially in large and ultra-large scale, and even in the e-commerce sector, see [Vog07], [Vog08]) availability and partition tolerance are more important than strict consistency. These applications have to be reliable which implicates availability and redundancy (consequently distribution among two or more nodes, which is necessary as many systems run on “cheap, commoditized and unreliable” machines [Ho09a] and also provides scalability). These properties are difficult to achieve with ACID properties therefore approaches like BASE are applied (cf. [Ipp09]).

The BASE approach according to Brewer forfeits the ACID properties of consistency and isolation in favor of “availability, graceful degradation, and performance” (cf. [Bre00, slide 12]). The acronym BASE is composed of the following characteristics:

- Basically available
- Soft-state
- Eventual consistency

Brewer contrasts ACID with BASE as illustrated in table 3.2, nonetheless considering the two concepts as a spectrum instead of alternatives excluding each other. Ippolito summarizes the BASE properties in the following way: an application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known-state state eventually (eventual consistency, cf. [Ipp09]).

ACID	BASE
Strong consistency Isolation Focus on “commit” Nested transactions Availability? Conservative (pessimistic) Difficult evolution (e.g. schema)	Weak consistency – stale data OK Availability first Best effort Approximate answers OK Aggressive (optimistic) Simpler! Faster Easier evolution

**Table 3.2.: ACID vs. BASE (cf. [Bre00, slide 13])**

In his talk “Design Patterns for Distributed Non-Relational Databases” Todd Lipcon contrasts strict and eventual consistency. He defines that a consistency model—such as strict or eventual consistency—“determines rules for **visibility** and **apparent order** of updates”. Lipcon, like Brewer, refers to consistency as “a continuum with tradeoffs” and describes strict and eventual consistency as follows (cf. [Lip09, slides 14–16]):

**Strict Consistency** according to Lipcon means that “All read operations must return data from the latest completed write operation, regardless of which replica the operations went to”. This implies that either read and write operations for a given dataset have to be executed on the same node<sup>2</sup> or that strict consistency is assured by a distributed transaction protocol (like two-phase-commit or Paxos). As we have seen above, such a strict consistency cannot be achieved together with availability and partition tolerance according to the CAP-theorem.

**Eventual Consistency** means that readers will see writes, as time goes on: “In a steady state, the system will eventually return the last written value”. Clients therefore may face an inconsistent state of data as updates are in progress. For instance, in a replicated database updates may go to one node which replicates the latest version to all other nodes that contain a replica of the modified dataset so that the replica nodes eventually will have the latest version.

Lipcon and Ho point out that an eventually consistent system may provide more differentiated, additional guarantees to its clients (cf. [Lip09, slide 16], [Ho09a]):

**Read Your Own Writes (RYOW) Consistency** signifies that a client sees his updates immediately after they have been issued and completed, regardless if he wrote to one server and in the following reads from different servers. Updates by other clients are not visible to him instantly.

**Session Consistency** means read your own writes consistency which is limited to a session scope (usually bound to one server), so a client sees his updates immediately only if read requests after an update are issued in the same session scope.

**Casual Consistency** expresses that if one client reads version x and subsequently writes version y, any client reading version y will also see version x.

**Monotonic Read Consistency** provides the time monotonicity guarantee that clients will only see more updated versions of the data in future requests.

---

<sup>2</sup>In such a scenario there can be further (slave) nodes to which datasets are replicated for availability purposes. But this replications cannot be done asynchronously as data on the replica nodes has to be up to date instantaneously.

Ho comments that eventual consistency is useful if concurrent updates of the same partitions of data are unlikely and if clients do not immediately depend on reading updates issued by themselves or by other clients (cf. [Ho09a]). He furthermore notes that the consistency model chosen for a system (or parts of the system) implicates how client requests are dispatched to replicas as well as how replicas propagate and apply updates.

### 3.1.3. Versioning of Datasets in Distributed Scenarios

If datasets are distributed among nodes, they can be read and altered on each node and no strict consistency is ensured by distributed transaction protocols, questions arise on how “concurrent” modifications and versions are processed and to which values a dataset will eventually converge to. There are several options to handle these issues:

**Timestamps** seem to be an obvious solution for developing a chronological order. However, timestamps “rely on synchronized clocks and don't capture causality” as Lipcon points out (cf. [Lip09, slide 17]; for a more fundamental and thorough discussion on these issues cf. [Mat89]).

**Optimistic Locking** implies that a unique counter or clock value is saved for each piece of data. When a client tries to update a dataset it has to provide the counter/clock-value of the revision it likes to update (cf. [K<sup>+</sup>10b]). As a downside of this procedure, the Project Voldemort development team notes that it does not work well in a distributed and dynamic scenario where servers show up and go away often and without prior notice. To allow causality reasoning on versions (e.g. which revision is considered the most recent by a node on which an update was issued) a lot of history has to be saved, maintained and processed as the optimistic locking scheme needs a total order of version numbers to make causality reasoning possible. Such a total order easily gets disrupted in a distributed and dynamic setting of nodes, the Project Voldemort team argues.

**Vector Clocks** are an alternative approach to capture order and allow reasoning between updates in a distributed system. They are explained in more detail below.

**Multiversion Storage** means to store a timestamp for each table cell. These timestamps “don't necessarily need to correspond to real life”, but can also be some artificial values that can be brought into a definite order. For a given row multiple versions can exist concurrently. Besides the most recent version a reader may also request the “most recent before T” version. This provides “optimistic concurrency control with compare-and-swap on timestamps” and also allows to take snapshots of datasets (cf. [Lip09, slide 20]).

#### Vector Clocks

A vector clock is defined as a tuple  $V[0], V[1], \dots, V[n]$  of clock values from each node (cf. [Lip09, slide 18]). In a distributed scenario node  $i$  maintains such a tuple of clock values, which represent the state of itself and the other (replica) nodes' state as it is aware about at a given time ( $V_i[0]$  for the clock value of the first node,  $V_i[1]$  for the clock value of the second node, …  $V_i[i]$  for itself, …  $V_i[n]$  for the clock value of the last node). Clock values may be real timestamps derived from a node's local clock, version/revision numbers or some other ordinal values.

As an example, the vector clock on node number 2 may take on the following values:

$$V_2[0] = 45, V_2[1] = 3, V_2[2] = 55$$

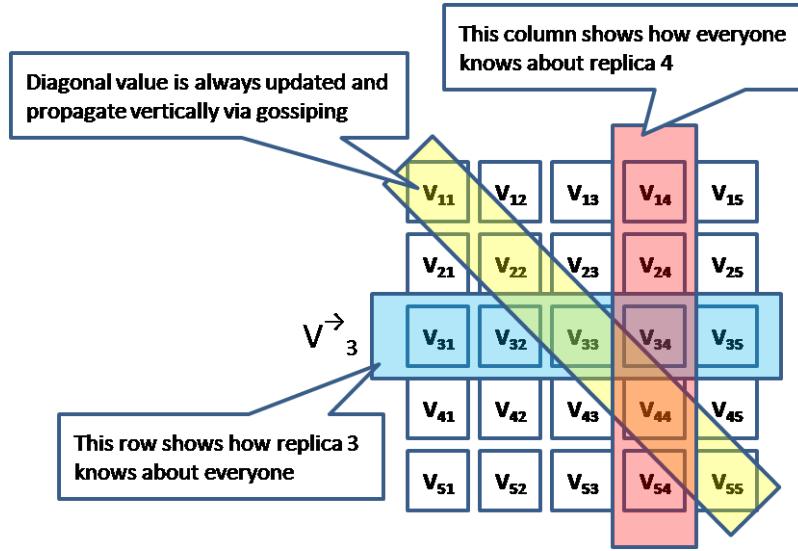


Figure 3.1.: Vector Clocks (taken from [Ho09a])

This reflects that from the perspective of the second node, the following updates occurred to the dataset the vector clock refers to: an update on node 1 produced revision 3, an update on node 0 lead to revision 45 and the most recent update is encountered on node 2 itself which produced revision 55.

Vector clocks are updated in a way defined by the following rules (cf. [Ho09a], [K<sup>+</sup>10b]):

- If an internal operation happens at node  $i$ , this node will increment its clock  $V_i[i]$ . This means that internal updates are seen immediately by the executing node.
- If node  $i$  sends a message to node  $k$ , it first advances its own clock value  $V_i[i]$  and attaches the vector clock  $V_i$  to the message to node  $k$ . Thereby, he tells the receiving node about his internal state and his view of the other nodes at the time the message is sent.
- If node  $i$  receives a message from node  $j$ , it first advances its vector clock  $V_i[i]$  and then merges its own vector clock with the vector clock  $V_{message}$  attached to the message from node  $j$  so that:

$$V_i = \max(V_i, V_{message})$$

To compare two vector clocks  $V_i$  and  $V_j$  in order to derive a partial ordering, the following rule is applied:

$$V_i > V_j, \text{ if } \forall k \ V_i[k] > V_j[k]$$

If neither  $V_i > V_j$  nor  $V_i < V_j$  applies, a conflict caused by concurrent updates has occurred and needs to be resolved by e.g. a client application.

As seen, vector clocks can be utilized “to resolve consistency between writes on multiple replicas” (cf. [Lip09, slide 18]) because they allow causal reasoning between updates. Replica nodes do typically not maintain a vector clock for clients but clients participate in the vector clock scenario in such a way that they keep a vector clock of the last replica node they have talked to and use this vector clock depending on the client consistency model that is required; e.g. for monotonic read consistency a client attaches this last vector clock it received to requests and the contacted replica node makes sure that the vector clock of its response is greater than the vector clock the client submitted. This means that the client can be sure to see only newer versions of some piece of data (“newer” compared to the versions it has already seen; cf. [Ho09a]).

Compared to the alternative approaches mentioned above (timestamps, optimistic locking with revision numbers, multiversion storage) the advantages of vector clocks are:

- No dependence on synchronized clocks
- No total ordering of revision numbers required for causal reasoning
- No need to store and maintain multiple revisions of a piece of data on all nodes

### Vector Clocks Utilized to Propagate State via Gossip

Blogger Ricky Ho gives an overview about how vector clocks can be utilized to handle consistency, conflicting versions and also transfer state between replicas in a partitioned database (as it will be discussed in the next section). He describes the transfer of vector clocks between clients and database nodes as well as among the latter over the Gossip protocol which can be operated in either a *state* or an *operation transfer model* to handle read and update operations as well as replication among database nodes (cf. [Ho09a]). Regarding the internode propagation of state via Gossip Todd Lipcon points out that this method provides scalability and avoids a single point of failure (SPOF). However, as the information about state in a cluster of  $n$  nodes needs  $O(\log n)$  rounds to spread, only eventual consistency can be achieved (cf. [Lip09, slide 34]).

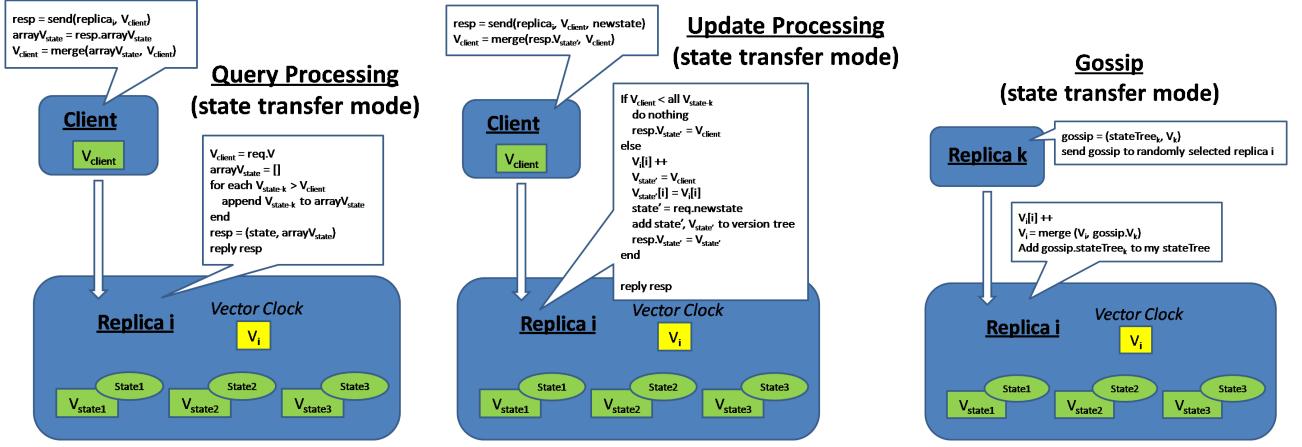
**State Transfer Model** In a state transfer model data or deltas of data are exchanged between clients and servers as well as among servers. In this model database server nodes maintain vector clocks for their data and also state version trees for conflicting versions (i.e. versions where the corresponding vector clocks cannot be brought into a  $V_A < V_B$  or  $V_A > V_B$  relation); clients also maintain vector clocks for pieces of data they have already requested or updated. These are exchanged and processed in the following manner:

**Query Processing** When a client queries for data it sends its vector clock of the requested data along with the request. The database server node responds with part of his state tree for the piece of data that precedes the vector clock attached to the client request (in order to guarantee monotonic read consistency) and the server's vector clock. Next, the client advances his vector clock by merging it with the server node's vector clock that had been attached to the servers response. For this step, the client also has to resolve potential version conflicts; Ho notes that this is necessary because if the client did not resolve conflicts at read time, it may be the case that he operates on and maybe submits updates for an outdated revision of data compared to the revision the contacted replica node maintains.

**Update Processing** Like in the case of read requests clients also have to attach their vector clock of the data to be updated along with the update request. The contacted replica server then checks, if the client's state according to the transmitted vector clock precedes the current server state and if so omits the update request (as the client has to acquire the latest version and also fix version conflicts at read time—as described in the former paragraph); if the client's transmitted vector clock is greater than its own the server executes the update request.

**Internode Gossiping** Replicas responsible for the same partitions of data exchange their vector clocks and version trees in the background and try to merge them in order to keep synchronized.

Figure 3.2 depicts the messages exchanged via Gossip in the state transfer model and how they are processed by receivers in the cases just described.



**Figure 3.2.: Vector Clocks – Exchange via Gossip in State Transfer Mode (taken from [Ho09a])**

**Operation Transfer Model** In contrast to the state transfer model discussed above, operations applicable to locally maintained data are communicated among nodes in the operation transfer model. An obvious advantage of this procedure is that lesser bandwidth is consumed to interchange operations in contrast to actual data or deltas of data. Within the operation transfer model, it is of particular importance to apply the operations in correct order on each node. Hence, a replica node first has to determine a causal relationship between operations (by vector clock comparison) before applying them to its data. Secondly, it has to defer the application of operations until all preceding operations have been executed; this implies to maintain a queue of deferred operations which has to be exchanged and consolidated with those of other replicas (as we will see below). In this model a replica node maintains the following vector clocks (cf. [Ho09a]):

- $V_{state}$ : The vector clock corresponding to the last updated state of its data.
- $V_i$ : A vector clock for itself where—compared to  $V_{state}$ —merges with received vector clocks may already have happened.
- $V_j$ : A vector clock received by the last gossip message of replica node j (for each of those replica nodes)

In the operation transfer model the exchange and processing vector clocks in read-, update- and internode-messages is as follows:

**Query Processing** In a read request a client again attaches his vector clock. The contacted replica node determines whether it has a view causally following the one submitted in the client request. It responds with the latest view of the state i.e. either the state corresponding to the vector clock attached by the client or the one corresponding to a causally succeeding vector clock of the node itself.

**Update Processing** When a client submits an update request the contacted replica node buffers this update operation until it can be applied to its local state taking into account the three vector clocks it maintains (see above) and the queue of already buffered operations. The buffered update operation gets tagged with two vector clocks:  $V_{client}$  representing the clients view when submitting the update, and  $V_{received}$  representing the replica node's view when it received the update (i.e. the vector clock  $V_i$  mentioned above). When all other operations causally preceding the received update operation have arrived and been applied the update requested by the client can be executed.

**Internode Gossiping** In the operation transfer model replica nodes exchange their queues of pending operations and update each other's vector clocks. Whenever nodes have exchanged their operation-queues they decide whether certain operations in these queues can be applied (as all dependent operations have been received). If more than one operation can be executed at a time, the replica node brings these operations into the correct (casual) order reasoned by the vector clocks attached to these operations and applies them to its local state of data. In case of concurrent updates on different replicas—resulting in multiple sequences of operations valid at a time—there has to be a distinction between the following cases:

1. The concurrent updates are commutative, so their order of application does not matter.
2. The concurrent updates are not commutative. In this case partial ordering reasoned from vector clock comparison is not sufficient but a total order has to be determined. To achieve this, Ho mentions the introduction of a global sequence number obtained from a central counter: “whoever do[sic!] the update first acquire a monotonic sequence number and late comers follow the sequence” (cf. [Ho09a]).

When operations have been applied locally on a replica they can only be removed from the node's operation queue after all other replica nodes have been informed about the execution of the update.

Figure 3.3 shows the messages exchanged and processed in operational transfer model via Gossip in the case of queries, updates and internode communication.

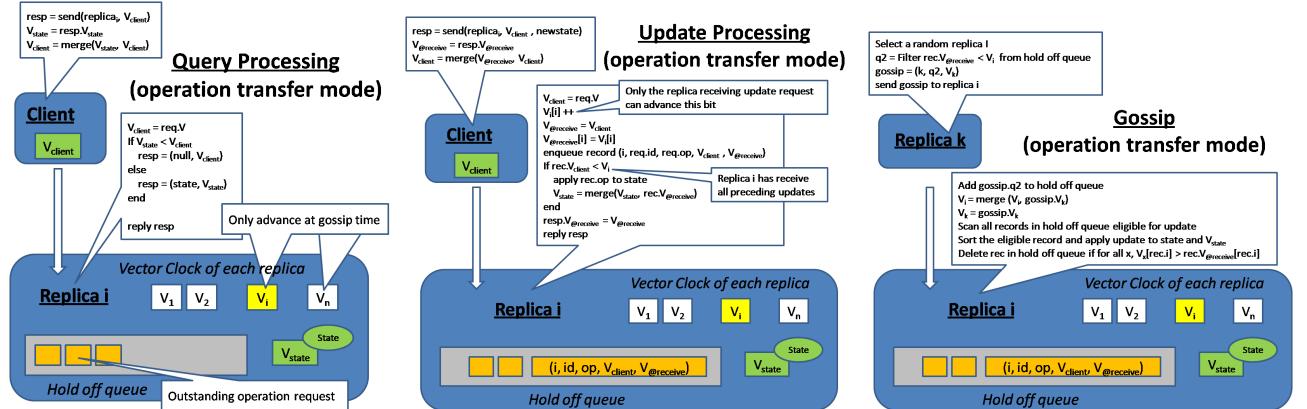


Figure 3.3.: Vector Clocks – Exchange via Gossip in Operation Transfer Mode (taken from [Ho09a])

## 3.2. Partitioning

Assuming that data in large scale systems exceeds the capacity of a single machine and should also be replicated to ensure reliability and allow scaling measures such as load-balancing, ways of partitioning the data of such a system have to be thought about. Depending on the size of the system and the other factors like dynamism (e.g. how often and dynamically storage nodes may join and leave) there are different approaches to this issue:

**Memory Caches** like memcached (cf. [F+10a], [F+10b]) can be seen as partitioned—though transient—in-memory databases as they replicate most frequently requested parts of a database to main memory, rapidly deliver this data to clients and therefore disburden database servers significantly. In the case of memcached the memory cache consists of an array of processes with an assigned amount of memory that can be launched on several machines in a network and are made known to an application via configuration. The memcached protocol (cf. [F+10c]) whose implementation is available in

different programming languages (cf. [F<sup>+</sup>09]) to be used in client applications provides a simple key-/value-store API<sup>3</sup>. It stores objects placed under a key into the cache by hashing that key against the configured memcached-instances. If a memcached process does not respond, most API-implementations ignore the non-answering node and use the responding nodes instead which leads to an implicit rehashing of cache-objects as part of them gets hashed to a different memcached-server after a cache-miss; when the formerly non-answering node joins the memcached server array again, keys for part of the data get hashed to it again after a cache miss and objects now dedicated to that node will implicitly leave the memory of some other memcached server they have been hashed to while the node was down (as memcached applies a LRU<sup>4</sup> strategy for cache cleaning and additionally allows it to specify timeouts for cache-objects). Other implementations of memory caches are available e.g. for application servers like JBoss (cf. [JB010b]).

**Clustering** of database servers is another approach to partition data which strives for transparency towards clients who should not notice talking to a cluster of database servers instead of a single server. While this approach can help to scale the persistence layer of a system to a certain degree many criticize that clustering features have only been added on top of DBMSs that were not originally designed for distribution (cf. the comments of Stonebraker et al. in subsection 2.1.2).

**Separating Reads from Writes** means to specify one or more dedicated servers, write-operations for all or parts of the data are routed to (master(s)), as well as a number of replica-servers satisfying read-requests (slaves). If the master replicates to its clients asynchronously there are no write lags but if the master crashes before completing replication to at least one client the write-operation is lost; if the master replicates writes synchronously to one slave lags the update does not get lost, but read request cannot go to any slave if strict consistency is required and furthermore write lags cannot be avoided (cf. [Ho09a] and regarding write lags StudiVz's Dennis Bemann's comments [Bem10]). In the latter case, if the master crashes the slave with the most recent version of data can be elected as the new master. The master-/slave-model works well if the read/write ratio is high. The replication of data can happen either by transfer of state (i.e. copying of the recent version of data or delta towards the former version) or by transfer of operations which are applied to the state on the slaves nodes and have to arrive in the correct order (cf. [Ho09a]).

**Sharding** means to partition the data in such a way that data typically requested and updated together resides on the same node and that load and storage volume is roughly even distributed among the servers (in relation to their storage volume and processing power; as an example confer the experiences of Bemann with a large German social-network on that topic [Bem10]). Data shards may also be replicated for reasons of reliability and load-balancing and it may be either allowed to write to a dedicated replica only or to all replicas maintaining a partition of the data. To allow such a sharding scenario there has to be a mapping between data partitions (shards) and storage nodes that are responsible for these shards. This mapping can be static or dynamic, determined by a client application, by some dedicated "mapping-service/component" or by some network infrastructure between the client application and the storage nodes. The downside of sharding scenarios is that joins between data shards are not possible, so that the client application or proxy layer inside or outside the database has to issue several requests and postprocess (e.g. filter, aggregate) results instead. Lipcon therefore comments that with sharding "you lose all the features that make a RDBMS useful" and that sharding "is operationally obnoxious" (cf. [Lip09]). This valuation refers to the fact that sharding originally was not designed within current RDBMSs but rather added on top. In contrast many NoSQL databases have embraced sharding as a key feature and some even provide automatic

---

<sup>3</sup>The memcached API provides the operations `get(key)`, `put(key, value)` and—for reasons of completeness—`remove(key)`.

<sup>4</sup>Least recently used

partitioning and balancing of data among nodes—as e.g. MongoDB of version 1.6 (cf. [MHC<sup>+</sup>10b], [Mon10]).

In a partitioned scenario knowing how to map database objects to servers is key. An obvious approach may be a simple hashing of database-object primary keys against the set of available database nodes in the following manner:

$$\text{partition} = \text{hash}(o) \bmod n \quad \text{with } o = \text{object to hash}, n = \text{number of nodes}$$

As mentioned above the downside of this procedure is that at least parts of the data have to be redistributed whenever nodes leave and join. In a memory caching scenario data redistribution may happen implicitly by observing cache misses, reading data again from a database or backend system, hashing it against the currently available cache servers and let stale cache data be purged from the cache servers by some cleaning policy like LRU. But for persistent data stores this implicit redistribution process is not acceptable as data not present on the available nodes cannot be reconstructed (cf. [Ho09a], [Whi07]). Hence, in a setting where nodes may join and leave at runtime (e.g. due to node crashes, temporal unattainability, maintenance work) a different approach such as consistent hashing has to be found which shall be discussed hereafter.

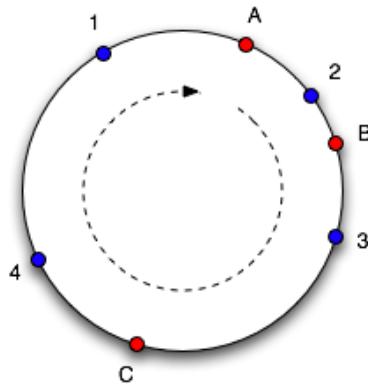
### 3.2.1. Consistent Hashing

The idea of consistent hashing was introduced by David Karger et al. in 1997 (cf. [KLL<sup>+</sup>97]) in the context of a paper about “a family of caching protocols for distributed[sic!] networks that can be used to decrease or eliminate the occurrence of hot spots in the networks”. This family of caching protocols grounds on consistent hashing which has been adopted in other fields than caching network protocols since 1997, e.g. in distributed hash-table implementations like Chord ([Par09]) and some memcached clients as well as in the NoSQL scene where it has been integrated into databases like Amazon’s Dynamo and Project Voldemort.

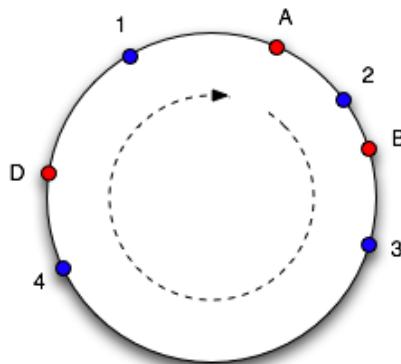
“The basic idea behind the consistent hashing algorithm is to hash both objects and caches using the same hash function” blogger Tom White points out. Not only hashing objects but also machines has the advantage, that machines get an interval of the hash-function’s range and adjacent machines can take over parts of the interval of their neighbors if those leave and can give parts of their own interval away if a new node joins and gets mapped to an adjacent interval. The consistent hashing approach furthermore has the advantage that client applications can calculate which node to contact in order to request or write a piece of data and there is no metadata server necessary as in systems like e.g. the Google File System (GFS) which has such a central (though clustered) metadata server that contains the mappings between storage servers and data partitions (called *chunks* in GFS; cf. [GL03, p. 2f]).

Figures 3.4 and 3.5 illustrate the idea behind the consistent hashing approach. In figure 3.4 there are three red colored nodes A, B and C and four blue colored objects 1–4 that are mapped to a hash-function’s result range which is imagined and pictured as a ring. Which object is mapped to which node is determined by moving clockwise around the ring. So, objects 4 and 1 are mapped to node A, object 2 to node B and object 3 to node C. When a node leaves the system, cache objects will get mapped to their adjacent node (in clockwise direction) and when a node enters the system it will get hashed onto the ring and will overtake objects. An example is depicted in figure 3.5 where compared to figure 3.4 node C left and node D entered the system, so that now objects 3 and 4 will get mapped to node D. This shows that by changing the number of nodes not all objects have to be remapped to the new set of nodes but only part of the objects.

In this raw form there are still issues with this procedure: at first, the distribution of nodes on the ring is actually random as their positions are determined by a hash function and the intervals between nodes may



**Figure 3.4.:** Consistent Hashing – Initial Situation (taken from [Whi07])

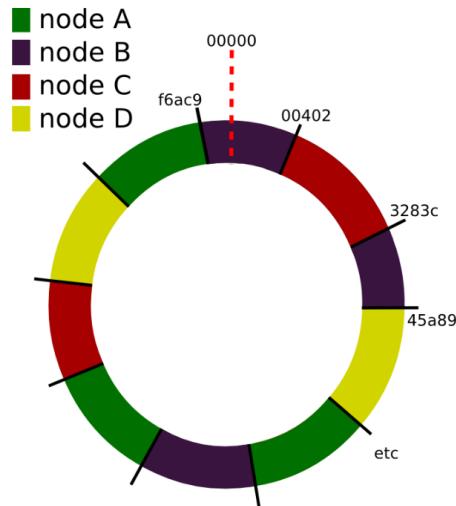


**Figure 3.5.:** Consistent Hashing – Situation after Node Joining and Departure (taken from [Whi07])

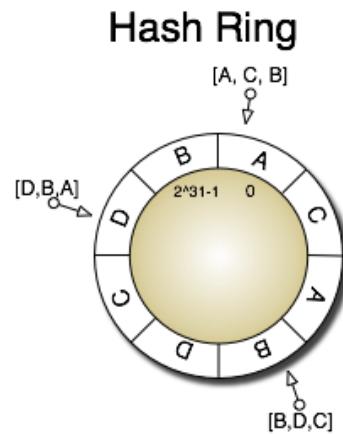
be “unbalanced” which in turn results in an unbalanced distribution of cache objects on these nodes (as it can already be seen in the small scenario of figure 3.5 where node D has to take cache objects from a greater interval than node A and especially node B). An approach to solve this issue is to hash a number of representatives/replicas—also called virtual nodes—for each physical node onto the ring (cf. [Whi07], as an example see figure 3.6). The number of virtual nodes for a physical can be defined individually according to its hardware capacity (cpu, memory, disk capacity) and does not have to be the same for all physical nodes. By appending e.g. a replica counter to a node’s id which then gets hashed, these virtual nodes should distribute points for this node all over the ring.

In his blog post on consistent hashing Tom White has simulated the effect of adding virtual nodes in a setting where he distributes 10,000 objects across ten physical nodes. As a result, the standard deviation of the objects distribution can be dropped from 100% without virtual nodes to 50% with only 2–5 virtual nodes and to 5–10% with 500 virtual nodes per physical node (cf. [Whi07]).

When applied to persistent storages, further issues arise: if a node has left the scene, data stored on this node becomes unavailable, unless it has been replicated to other nodes before; in the opposite case of a new node joining the others, adjacent nodes are no longer responsible for some pieces of data which they still store but not get asked for anymore as the corresponding objects are no longer hashed to them by requesting clients. In order to address this issue, a replication factor ( $r$ ) can be introduced. By doing so, not only the next node but the next  $r$  (physical!) nodes in clockwise direction become responsible for an object (cf. [K<sup>+</sup>10b], [Ho09a]). Figure 3.7 depicts such a scenario with replicated data: the uppercase letters again represent storage nodes that are—according to the idea of virtual nodes—mapped multiple times onto



**Figure 3.6.:** Consistent Hashing – Virtual Nodes Example (taken from [Lip09, slide 12])



**Figure 3.7.:** Consistent Hashing – Example with Virtual Nodes and Replicated Data (taken from [K<sup>+</sup>10b])

the ring, and the circles with arrows represent data objects which are mapped onto the ring at the depicted positions. In this example, the replication factor is three, so for every data object three physical nodes are responsible which are listed in square brackets in the figure. Introducing replicas implicates read and write operations on data partitions which will be considered in the next subsection. The above mentioned issues causing membership changes shall be discussed thereafter.

### 3.2.2. Read- and Write Operations on Partitioned Data

Introducing replicas in a partitioning scheme—besides reliability benefits—also makes it possible to spread workload for read requests that can go to any physical node responsible for a requested piece of data. In qualification, it should be stated that the possibility of load-balancing read operations does not apply to scenarios in which clients have to decide between multiple versions of a dataset and therefore have to read from a quorum of servers which in turn reduces the ability to load-balance read requests. The Project Voldemort team points out that three parameters are specifically important regarding read as well as write operations (cf. [K<sup>+</sup>10b]):

**N** The number of replicas for the data or the piece of data to be read or written.

**R** The number of machines contacted in read operations.

**W** The number of machines that have to be blocked in write operations<sup>5</sup>.

In the interest to provide e.g. the read-your-own-writes consistency model the following relation between the above parameters becomes necessary:

$$R + W > N$$

Regarding write operations clients have to be clear that these are neither immediately consistent nor isolated (cf. [K<sup>+</sup>10b]):

- If a write operation completes without errors or exceptions a client can be sure that at least W nodes have executed the operation.
- If the write operation fails as e.g. less than W nodes have executed it, the state of the dataset is unspecified. If at least one node has successfully written the value it will eventually become the new value on all replica nodes, given that the replicas exchange values and agree on the most recent versions of their datasets in background. If no server has been capable to write the new value, it gets lost. Therefore, if the write operation fails, the client can only achieve a consistent state by re-issuing the write operation.

### 3.2.3. Membership Changes

In a partitioned database where nodes may join and leave the system at any time without impacting its operation all nodes have to communicate with each other, especially when membership changes.

When a new node joins the system the following actions have to happen (cf. [Ho09a]):

1. The newly arriving node announces its presence and its identifier to adjacent nodes or to all nodes via broadcast.
2. The neighbors of the joining node react by adjusting their object and replica ownerships.
3. The joining node copies datasets it is now responsible for from its neighbours. This can be done in bulk and also asynchronously.
4. If, in step 1, the membership change has not been broadcasted to all nodes, the joining node is now announcing its arrival.

In figure 3.8, this process is illustrated. Node X joins a system for which a replication factor of three is configured. It is hashed between A and B, so that the nodes H, A and B transfer data to the new node X and after that the nodes B, C and D can drop parts of their data for which node X is now responsible as a third replica (in addition to nodes H, A and B).

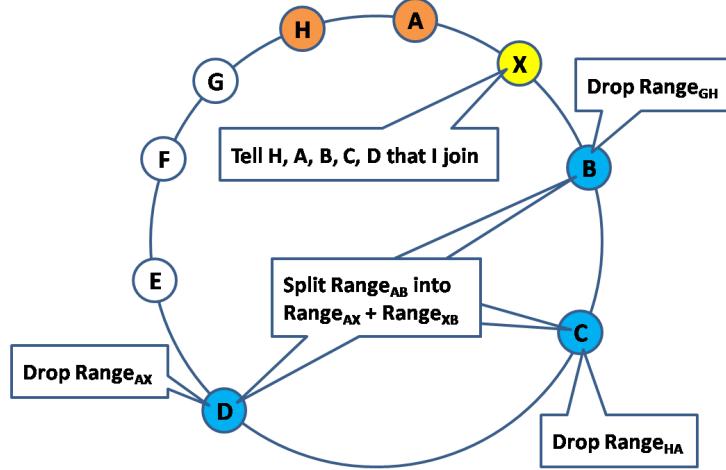
Ho notes that while data transfer and range adjustments happen in steps 2 and 3 the adjacent nodes of the new node may still be requested and can forward them to the new node. This is the case if it e.g. has already received the requested piece of data or can send vector clocks to clients to let them determine the most recent version of a dataset after having contacted multiple replicas.

When a node leaves the system the following actions have to occur (cf. [Ho09a]):

---

<sup>5</sup>W can be set to zero if a client wishes to write in a non-blocking fashion which does not assure the client of the operation's success.

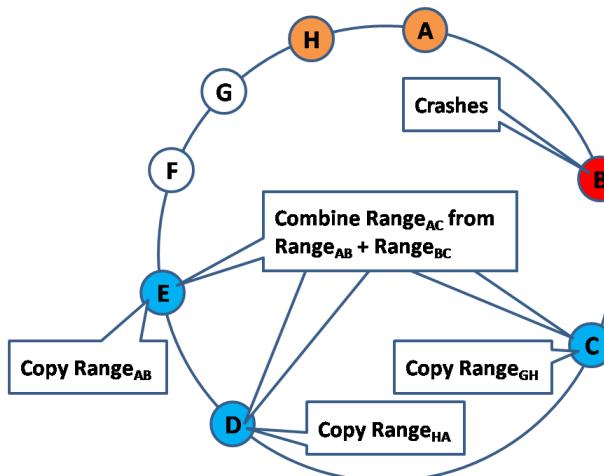
H, A, X, B, C, D will update the membership synchronously  
And then asynchronously propagate the membership changes to other nodes



**Figure 3.8.: Membership Changes – Node X joins the System (taken from [Ho09a])**

1. Nodes within the system need to detect whether a node has left as it might have crashed and not been able to notify the other nodes of its departure. It is also common in many systems that no notifications get exchanged when a node leaves. If the nodes of the system communicate regularly e.g. via the Gossip protocol they are able to detect a node's departure because it no longer responds.
2. If a node's departure has been detected, the neighbors of the node have to react by exchanging data with each other and adjusting their object and replica ownerships.

Asynchronously propagate the membership changes to other nodes



**Figure 3.9.: Membership Changes – Node B leaves the System (taken from [Ho09a])**

Figure 3.9 shows the actions to be taken when node B—due to a crash—leaves the system. Nodes C, D and E become responsible for new intervals of hashed objects and therefore have to copy data from nodes in counterclockwise direction and also reorganize their internal representation of the intervals as the  $\text{Range}_{AB}$  and  $\text{Range}_{BC}$  now have collapsed to  $\text{Range}_{AC}$ .

### 3.3. Storage Layout

In his talk on design patterns for nonrelational databases Todd Lipcon presents an overview of storage layouts, which determine how the disk is accessed and therefore directly implicate performance. Furthermore, the storage layout defines which kind of data (e.g. whole rows, whole columns, subset of columns) can be read en bloc (cf. [Lip09, slide 21–31]).

**Row-Based Storage Layout** means that a table of a relational model gets serialized as its lines are appended and flushed to disk (see figure 3.10a). The advantages of this storage layout are that at first whole datasets can be read and written in a single IO operation and that secondly one has a “[g]ood locality of access (on disk and in cache) of different columns”. On the downside, operating on columns is expensive as a considerable amount data (in a naïve implementation all of the data) has to be read.

**Columnar Storage Layout** serializes tables by appending their columns and flushing them to disk (see figure 3.10b). Therefore operations on columns are fast and cheap while operations on rows are costly and can lead to seeks in a lot or all of the columns. A typical application field for this type of storage layout is analytics where an efficient examination of columns for statistical purposes is important.

**Columnar Storage Layout with Locality Groups** is similar to column-based storage but adds the feature of defining so called locality groups that are groups of columns expected to be accessed together by clients. The columns of such a group may therefore be stored together and physically separated from other columns and column groups (see figure 3.10c). The idea of locality groups was introduced in Google’s Bigtable paper. It firstly describes the logical model of column-families for semantically affine or interrelated columns (cf. [CDG<sup>+</sup>06, section 2]) and later on presents the locality groups idea as a refinement where column-families can be grouped or segregated for physical storage (cf. [CDG<sup>+</sup>06, section 6]).

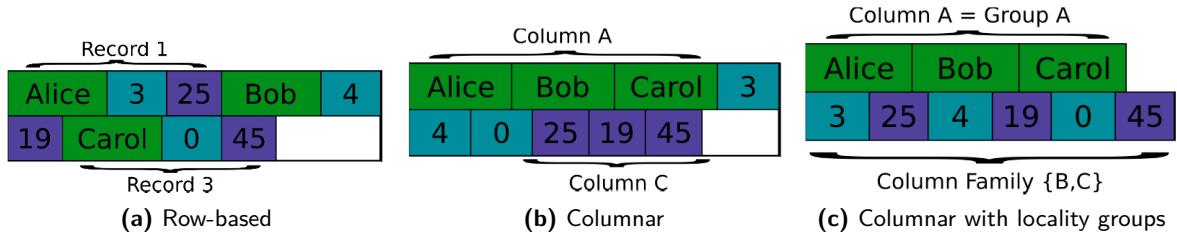
**Log Structured Merge Trees** (LSM-trees) in contrast to the storage layouts explained before do not describe how to serialize logical datastructures (like tables, documents etc.) but how to efficiently use memory and disk storage in order to satisfy read and write requests in an efficient, performant and still safely manner. The idea, brought up by O’Neil et al. in 1996 (cf. [OCGO96]), is to hold chunks of data in memory (in so called Memtables), maintaining on-disk commit-logs for these in-memory data structures and flushing the memtables to disk from time to time into so called SSTables (see figure 3.11a and 3.11e). These are immutable and get compacted over time by copying the compacted SSTable to another area of the disk while preserving the original SSTable and removing the latter after the compaction process has happened (see figure 3.11f). The compaction is necessary as data stored in these SSTables may have changed or been deleted by clients. These data modifications are first reflected in a Memtable that later gets flushed to disk as a whole into a SSTable which may be compacted together with other SSTables already on disk. Read-requests go to the Memtable as well as the SSTables containing the requested data and return a merged view of it (see figure 3.11b). To optimize read-requests and only read the relevant SSTables bloom filters<sup>6</sup> can be used (see figure 3.11c). Write requests go to the Memtable as well as an on-disk commit-log synchronously (see figure 3.11d).

The advantages of log structured merge trees are that memory can be utilized to satisfy read requests quickly and disk I/O gets faster as SSTables can be read sequentially because their data is not randomly distributed over the disk. LSM-trees also tolerate machine crashes as write operations not only go to memory but also (synchronously) to a commit-log by which a machine can recover from a

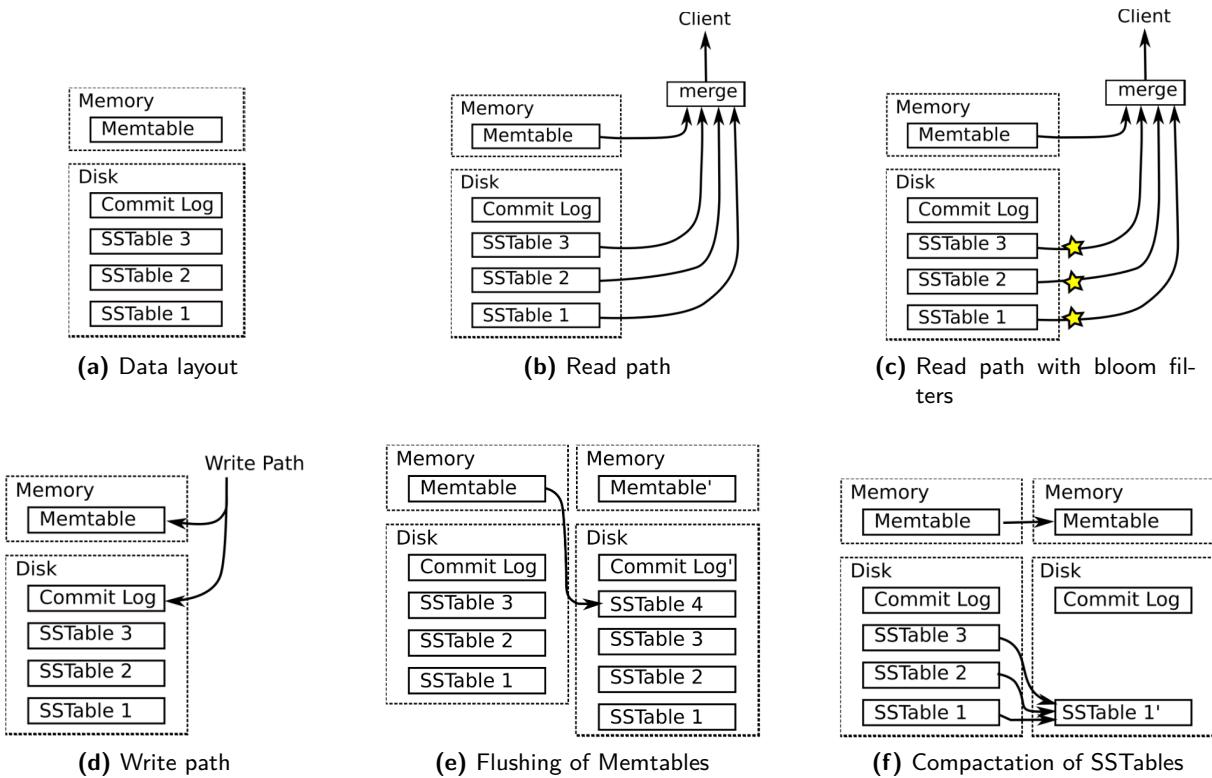
---

<sup>6</sup>Bloom filters, as described by Burton Bloom in 1970 (cf. [Blo70]), are probabilistic data structures used to efficiently determine if elements are member of a set in a way that avoids false-negatives (while false-positives are possible).

crash. Log structured merge trees are implemented in e.g. Google's Bigtable (cf. [CDG<sup>+</sup>06, sections 4, 5.3, 5.4, 6]) as blogger Ricky Ho summarizes graphically in figure 3.12.



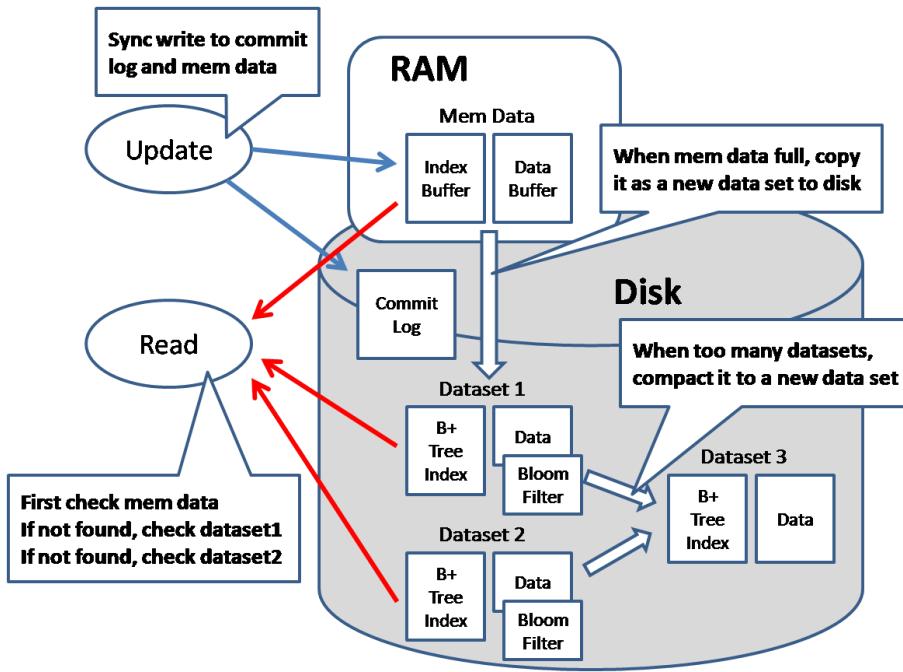
**Figure 3.10.:** Storage Layout – Row-based, Columnar with/out Locality Groups (taken from [Lip09, slides 22–24])



**Figure 3.11.:** Storage Layout – Log Structured Merge Trees (taken from [Lip09, slides 26–31])

Regarding the discussion of on-disk or in-memory storage or a combination of both (as in LSM-trees) blogger Nati Shalom notes that it “comes down mostly to cost/GB of data and read/write performance”. He quotes from an analysis by the Stanford University (entitled “The Case for RAMClouds”, cf. [OAE<sup>+</sup>10]) which comes to the conclusion that “cost is also a function of performance”. He cites from the analysis that “if an application needs to store a large amount of data inexpensively and has a relatively low access rate, RAMCloud is not the best solution” but “for systems with high throughput requirements a RAM-Cloud[sic!] can provide not just high performance but also energy efficiency” (cf. [Sha09a]).

Blogger Ricky Ho adds to the categorization of storage layouts above that some NoSQL databases leave the storage implementation open and allow to plug-in different kinds of it—such as Project Voldemort which allows to use e.g. a relational database, the key/value database BerkleyDB, the filesystem or a memory



**Figure 3.12.: Storage Layout – MemTables and SSTables in Bigtable (taken from [Ho09a])**

hashtable as storage implementations. In contrast, most NoSQL databases implement a storage system optimized to their specific characteristics.

In addition to these general storage layouts and the distinction between pluggable and proprietary storage implementations most databases carry out optimizations according to their specific data model, query processing means, indexing structures etc..

As an example, the document database CouchDB provides a copy-on-modified semantic in combination with an append only file for database contents (cf. [Apa10b, Section “ACID Properties”]). The copy-on-modified semantic means that a private copy for the user issuing the modification request is made from the data as well as the index (a B-Tree in case of CouchDB); this allows CouchDB to provide read-your-own-writes consistency for the issuer while modifications are only eventually seen by other clients. The private copy is propagated to replication nodes and—as CouchDB also supports multi-version concurrency control (MVCC)—maybe clients have to merge conflicting versions before a new version becomes valid for all clients (which is done by swapping the root pointer of storage metadata that is organized in a B-Tree as shown in figure 3.13). CouchDB synchronously persists all updates to disk in an append-only fashion. A garbage-collection compacts the persisted data from time to time by copying the compacted file contents into a new file and not touching the old file until the new one is written correctly. This treatment allows continuous operation of the system while the garbage-collection is executed.

Ho depicts how the CouchDB index and files are affected by updates as shown in figure 3.13.

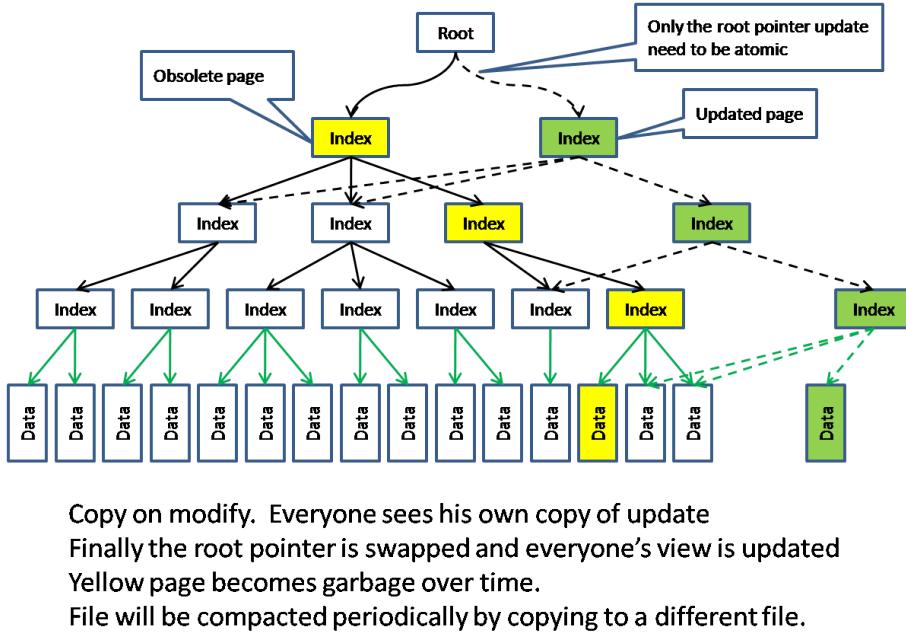


Figure 3.13.: Storage Layout – Copy-on-modify in CouchDB (taken from [Ho09a])

### 3.4. Query Models

As blogger Nati Shalom notes there are substantial differences in the querying capabilities the different NoSQL datastores offer (cf. [Sha09a]): whereas key/value stores by design often only provide a lookup by primary key or some id field and lack capabilities to query any further fields, other datastores like the document databases CouchDB and MongoDB allow for complex queries—at least static ones predefined on the database nodes (as in CouchDB). This is not surprising as in the design of many NoSQL databases rich dynamic querying features have been omitted in favor of performance and scalability. On the other hand, also when using NoSQL databases, there are use-cases requiring at least some querying features for non-primary key attributes. In his blog post “Query processing for NOSQL DB” blogger Ricky Ho addresses this issue and presents several ways to implement query features that do not come out of the box with some NoSQL datastores:

**Companion SQL-database** is an approach in which searchable attributes are copied to a SQL or text database. The querying capabilities of this database are used to retrieve the primary keys of matching datasets by which the NoSQL database will subsequently be accessed (see figure 3.14).

**Scatter/Gather Local Search** can be used if the NoSQL store allows querying and indexing within database server nodes. If this is the case a query processor can dispatch queries to the database nodes where the query is executed locally. The results from all database servers are sent back to the query processor postprocessing them to e.g. do some aggregation and returning the results to a client that issued the query (see figure 3.15).

**Distributed B+Trees** are another alternative to implement querying features ((see figure 3.16)). The basic idea is to hash the searchable attribute to locate the root node of a distributed B+tree (further information on scalable, distributed B+Trees can be found in a paper by Microsoft, HP and the University of Toronto, cf. [AGS08]). The “value” of this root node then contains an id for a child node in the B+tree which can again be looked up. This process is repeated until a leaf node is reached which contains the primary-key or id of a NoSQL database entry matching search criteria.

Ho notes that node-updates in distributed B+trees (due to splits and merges) have to be handled cautiously and should be handled in an atomic fashion.

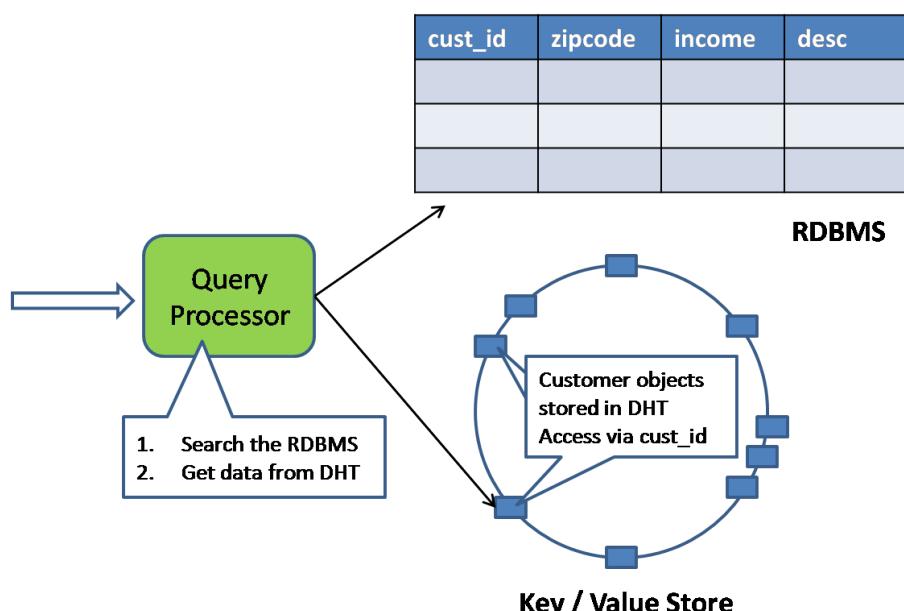
**Prefix Hash Table (aka Distributed Trie)** is a tree-datastructure where every path from the root-node to the leafs contains the prefix of the key and every node in the trie contains all the data whose key is prefixed by it (for further information cf. a Berkley-paper on this datastructure [RRHS04]). Besides an illustration (see figure 3.17) Ho provides some code-snippets in his blog post that describe how to operate on prefix hash tables / distributed tries and how to use them for querying purposes (cf. [Ho09b]).

Ho furthermore points out that junctions in the search criteria have to be addressed explicitly in querying approaches involving distribution (scatter/gather local search, distributed B+trees):

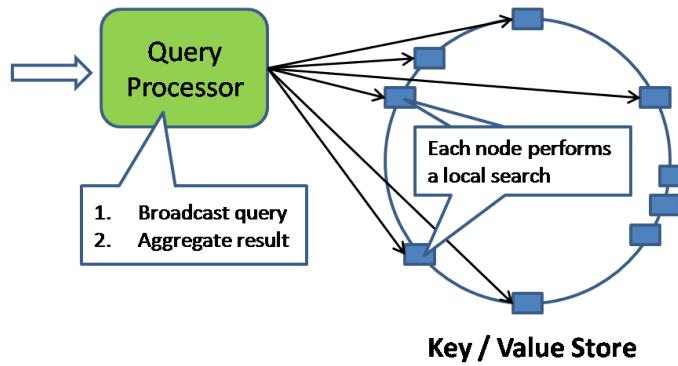
**OR-Junctions** are simple since the search results from different database nodes can be put together by a union-operation.

**AND-Junctions** are more difficult as the intersection of individually matching criteria is wanted and therefore efficient ways to intersect potentially large sets are needed. A naïve implementation might send all matching objects to a server that performs set intersection (e.g. a query processor initiating the distributed query as in figures 3.14–3.16). However, this involves large bandwidth consumption, if some or all datasets are large. A number of more efficient approaches are described in a paper by Reynolds and Vahdat (cf. [RV03]):

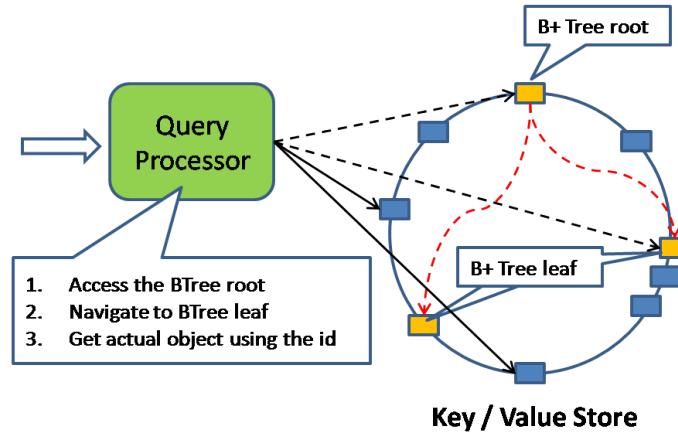
- Bloom filters can be used to test if elements are definitely not contained in a (result) set, which can help in intersecting sets as the number of elements having to be transferred over the network and/or compared by more expensive computation than bloom filters can be drastically reduced.
- Result sets of certain popular searches or bloom filters of popular search criteria can be cached.
- If client applications do not require the full result set at once an incremental fetch strategy can be applied where result set data is streamed to the client using a cursor mode. This moves computation into the client application which could perhaps also become responsible for filtering operations like (sub-)set intersection.



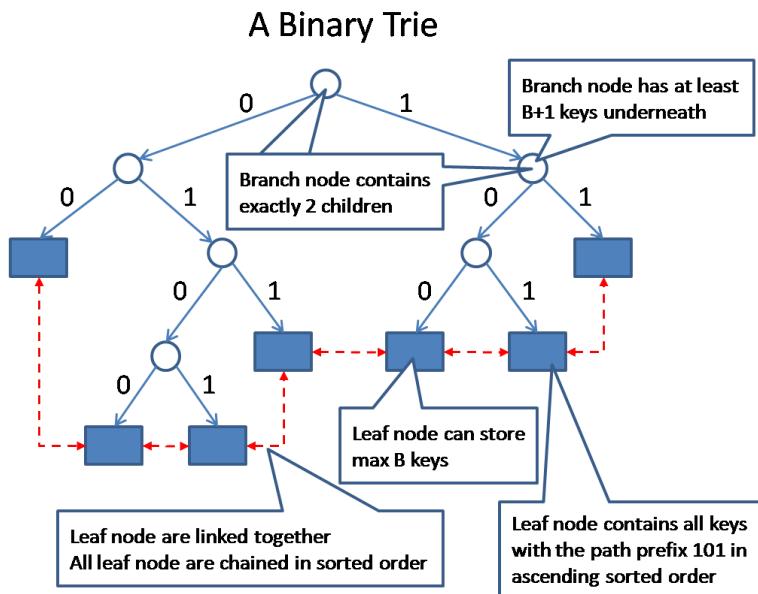
**Figure 3.14.: Query Models – Companion SQL-Database (taken from [Ho09b])**



**Figure 3.15.:** Query Models – Scatter/Gather Local Search (taken from [Ho09b])



**Figure 3.16.:** Query Models – Distributed B+Tree (taken from [Ho09b])

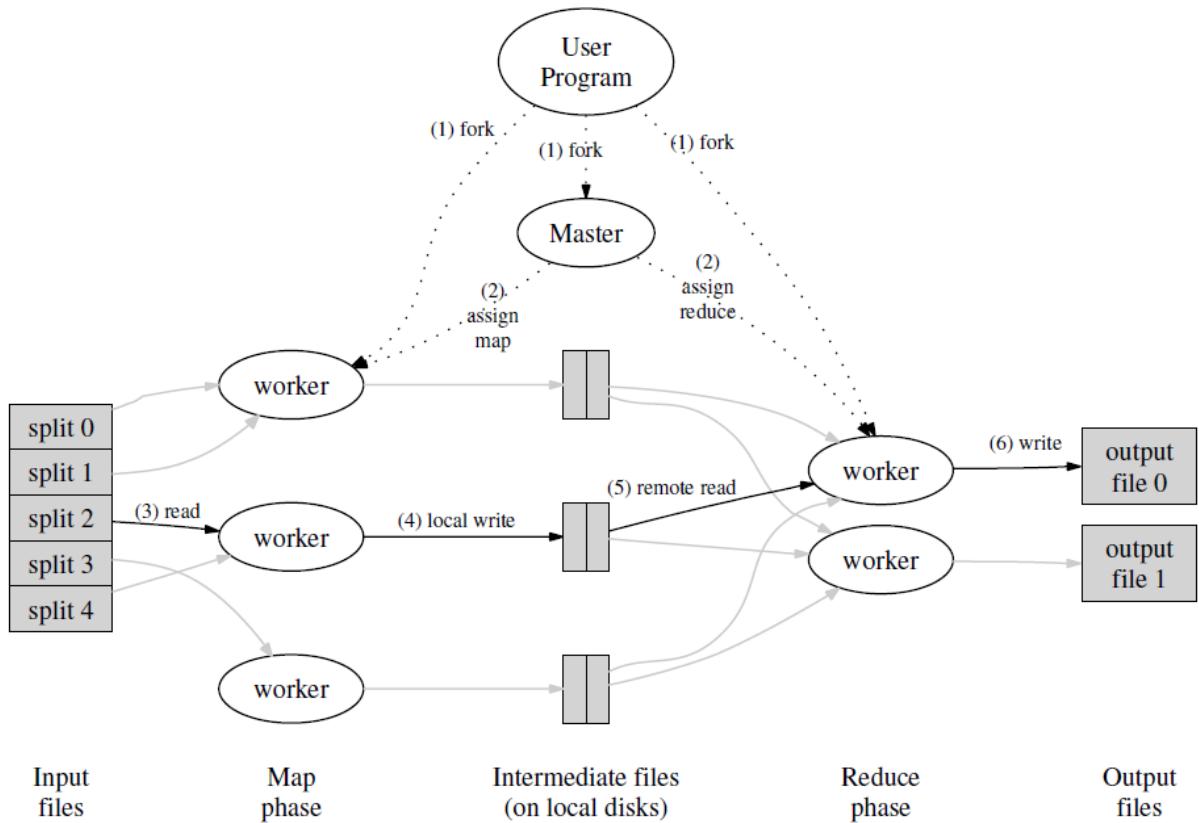


**Figure 3.17.:** Query Models – Prefix Hash Table / Distributed Trie (taken from [Ho09b])

### 3.5. Distributed Data Processing via MapReduce

The last section focused on how to operate on distributed environments by concentrating on dynamic querying mechanisms. Although no substitution for lacking query and maintenance capabilities (as Brian Aker correctly humorously argues in his talk “Your Guide To NoSQL”), but somehow related is the possibility for operating on distributed database nodes in a MapReduce fashion. This approach, brought up by Google employees in 2004 (cf. [DG04]), splits a task into two stages, described by the functions `map` and `reduce`. In the first stage a coordinator designates pieces of data to process a number of nodes which execute a given `map` function and produce intermediate output. Next, the intermediate output is processed by a number of machines executing a given `reduce` function whose purpose it is to create the final output from the intermediate results, e.g. by some aggregation. Both the `map` and `reduce` functions have to be understood in a real functional manner, so they do not depend on some state on the machine they are executed on and therefore produce identical output on each execution environment given the identical input data. The partitioning of the original data as well as the assignment of intermediate data is done by the coordinator according to Google’s original paper.

Figure 3.18 depicts and summarizes the MapReduce fashion of distributed data processing.

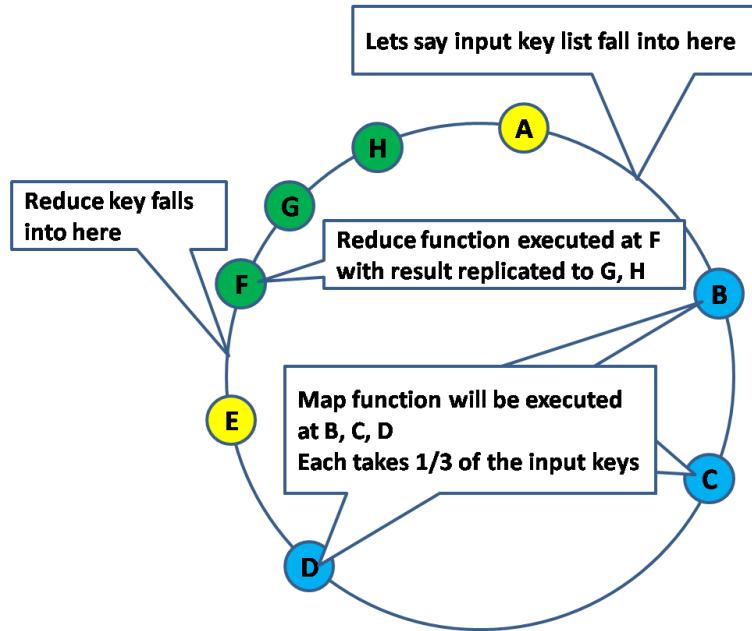


**Figure 3.18.: MapReduce – Execution Overview (taken from [DG04, p. 3])**

The MapReduce paradigm has been adopted by many programming languages (e.g. Python), frameworks (e.g. Apache Hadoop), even JavaScript toolkits (e.g. Dojo) and also NoSQL databases (e.g. CouchDB). This is the case because it fits well into distributed processing as blogger Ricky Ho notes (cf. [Ho09a]), especially for analytical purposes or precalculation tasks (e.g. in CouchDB to generate views of the data cf. [Apa10b]). When applied to databases, MapReduce means to process a set of keys by submitting the process logic (`map`- and `reduce`-function code) to the storage nodes which locally apply the `map` function

to keys that should be processed and that they own. The intermediate results can be consistently hashed just as regular data and processed by the following nodes in clockwise direction, which apply the reduce function to the intermediate results and produce the final results. It should be noted that by consistently hashing the intermediate results there is no coordinator needed to tell processing nodes where to find them.

The idea of applying MapReduce to a distributed datastore is illustrated in figure 3.19.



**Figure 3.19.: MapReduce – Execution on Distributed Storage Nodes (taken from [Ho09a])**

# 4. Key-/Value-Stores

Having discussed common concepts, techniques and patterns the first category of NoSQL datastores will be investigated in this chapter. Key-/value-stores have a simple data model in common: a map/dictionary, allowing clients to put and request values per key. Besides the data-model and the API, modern key-value stores favor high scalability over consistency and therefore most of them also omit rich ad-hoc querying and analytics features (especially joins and aggregate operations are set aside). Often, the length of keys to be stored is limited to a certain number of bytes while there is less limitation on values (cf. [Ipp09], [Nor09]).

Key-/value-stores have existed for a long time (e.g. Berkeley DB [Ora10d]) but a large number of this class of NoSQL stores that has emerged in the last couple of years has been heavily influenced by Amazon's Dynamo which will be investigated thoroughly in this chapter. Among the great variety of free and open-source key-/value-stores, Project Voldemort will be further examined. At the end of the chapter some other notable key-/value-stores will be briefly looked at.

## 4.1. Amazon's Dynamo

Amazon Dynamo is one of several databases used at Amazon for different purposes (others are e.g. SimpleDB or S3, the Simple Storage Service, cf. [Ama10b], [Ama10a]). Because of its influence on a number of NoSQL databases, especially key-/value-stores, the following section will investigate in more detail Dynamo's influencing factors, applied concepts, system design and implementation.

### 4.1.1. Context and Requirements at Amazon

The technological context these storage services operate upon shall be outlined as follows (according to Amazon's Dynamo Paper by DeCandia et al. cf. [DHJ<sup>+</sup>07, p. 205]):

- The infrastructure is made up by tens of thousands of servers and network components located in many datacenters around the world.
- Commodity hardware is used.
- Component failure is the "standard mode of operation".
- "Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services."

Apart from these technological factors, the design of Dynamo is also influenced by business considerations (cf. [DHJ<sup>+</sup>07, p. 205]):

- Strict, internal service level agreements (SLAs) regarding “performance, reliability and efficiency” have to be met in the 99.9<sup>th</sup> percentile of the distribution<sup>1</sup>. DeCandia et al. consider “state management” as offered by Dynamo and the other databases as being crucial to meet these SLAs in a service whose business logic is in most cases rather lightweight at Amazon (cf. [DHJ<sup>+</sup>07, p. 207–208]).
- One of the most important requirements at Amazon is reliability “because even the slightest outage has significant financial consequences and impacts customer trust”.
- “[To] support continuous growth, the platform needs to be highly scalable”.

At Amazon “Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance”. DeCandia et al. furthermore argue that a lot of services only need access via primary key (such as “best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog”) and that the usage of a common relational database “would lead to inefficiencies and limit scale and availability” (cf. [DHJ<sup>+</sup>07, p. 205]).

#### 4.1.2. Concepts Applied in Dynamo

Out of the concepts presented in chapter 3, Dynamo uses consistent hashing along with replication as a partitioning scheme. Objects stored in partitions among nodes are versioned (multi-version storage). To maintain consistency during updates Dynamo uses a quorum-like technique and a (not further specified) protocol for decentralized replica synchronization. To manage membership and detect machine failures it employs a gossip-based protocol which allows to add and remove servers with “a minimal need for manual administration” (cf. [DHJ<sup>+</sup>07, p. 205–206]).

DeCandia et al. note their contribution to the research community is that Dynamo as an “eventually-consistent storage system can be used in production with demanding applications”.

#### 4.1.3. System Design

##### Cosiderations and Overview

DeCandia et al. militate against RDBMSs at Amazon as most services there “only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS”. Furthermore they consider the “available replication technologies” for RDBMSs as “limited and typically choose[ing] consistency over availability”. They admit that advances have been made to scale and partition RDBMSs but state that such setups remain difficult to configure and operate (cf. [DHJ<sup>+</sup>07, p. 206]).

Therefore, they took the decision to build Dynamo with a simple key/value interface storing values as BLOBs. Operations are limited to one key/value-pair at a time, so update operations are limited to single keys allowing no cross-references to other key-/value-pairs or operations spanning more than one key-/value-pair. In addition, hierarchical namespaces (like in directory services or filesystems) are not supported by Dynamo (cf. [DHJ<sup>+</sup>07, p. 206 and 209]).

Dynamo is implemented as a partitioned system with replication and defined consistency windows. Therefore, Dynamo “targets applications that operate with weaker consistency [...] if this results in high availability”. It does not provide any isolation guarantees (cf. [DHJ<sup>+</sup>07, p. 206]). DeCandia et al. argue

---

<sup>1</sup>To meet requirements in the average or median case plus some variance is good enough to satisfy more than the just majority of users according to experience made at Amazon.

Problem	Technique	Advantages
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information

**Table 4.1.:** Amazon's Dynamo – Summary of Techniques (taken from [DHJ<sup>+</sup>07, p. 209])

that a synchronous replication scheme is not achievable given the context and requirements at Amazon (especially high-availability and scalability). Instead, they have selected an optimistic replication scheme. This features background replication and the possibility for write operations even in the presence of disconnected replicas (due to e.g. network or machine failures). So, as Dynamo is designed to be “always writeable (i.e. a datastore that is highly available for writes)” conflict resolution has to happen during reads. Furthermore DeCandia et al. argue that a datastore can only perform simple policies of conflict resolution and therefore a client application is better equipped for this task as it is “aware of the data schema” and “can decide on a conflict resolution method that is best suited for the client’s experience”. If application developers do not want to implement such a business logic specific reconciliation strategy Dynamo also provides simple strategies they can just use, such as “last write wins”, a timestamp-based reconciliation (cf. [DHJ<sup>+</sup>07, p. 207–208 and 214]).

To provide simple scalability Dynamo features “a simple scale-out scheme to address growth in data set size or request rates” which allows adding of “one storage host [...] at a time” (cf. [DHJ<sup>+</sup>07, p. 206 and 208]).

In Dynamo, all nodes have equal responsibilities; there are no distinguished nodes having special roles. In addition, its design favors “decentralized peer-to-peer techniques over centralized control” as the latter has “resulted in outages” in the past at Amazon. Storage hosts added to the system can have heterogeneous hardware which Dynamo has to consider to distribute work proportionally “to the capabilities of the individual servers” (cf. [DHJ<sup>+</sup>07, p. 208]).

As Dynamo is operated in Amazon’s own administrative domain, the environment and all nodes are considered non-hostile and therefore no security related features such as authorization and authentication are implemented in Dynamo (cf. [DHJ<sup>+</sup>07, p. 206 and 209]).

Table 4.1 summarizes difficulties faced in the design of Dynamo along with techniques applied to address them and respective advantages.

## System Interface

The interface Dynamo provides to client applications consists of only two operations:

- `get(key)`, returning a list of objects and a context
- `put(key, context, object)`, with no return value

The `get`-operation may return more than one object if there are version conflicts for objects stored under the given key. It also returns a context, in which system metadata such as the object version is stored, and clients have to provide this context object as a parameter in addition to key and object in the `put`-operation.

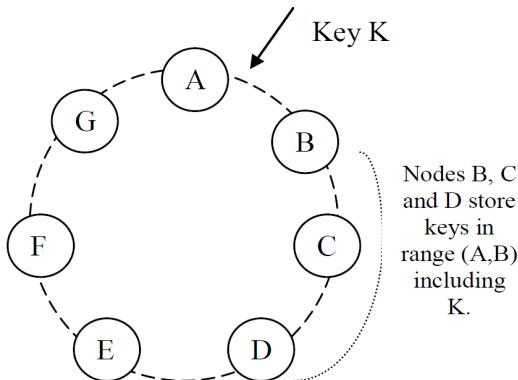
Key and object values are not interpreted by Dynamo but handled as “an opaque array of bytes”. The key is hashed by the MD5 algorithm to determine the storage nodes responsible for this key-/value-pair.

### Partitioning Algorithm

To provide incremental scalability, Dynamo uses consistent hashing to dynamically partition data across the storage hosts that are present in the system at a given time. DeCandia et al. argue that using consistent hashing in its raw form has two downsides: first, the random mapping of storage hosts and data items onto the ring leads to unbalanced distribution of data and load; second, the consistent hashing algorithm treats each storage node equally and does not take into account its hardware resources. To overcome both difficulties, Dynamo applies the concepts of virtual nodes, i. e. for each storage host multiple virtual nodes get hashed onto the ring (as described in subsection 3.2.1) and the number of virtual nodes per physical node is used to take the hardware capabilities of storage nodes into account (i. e. the more resources the more virtual nodes per physical node, cf. [DHJ<sup>+</sup>07, p. 209–210]).

### Replication

To ensure availability and durability in an infrastructure where machine-crashes are the “standard mode of operation” Dynamo uses replication of data among nodes. Each data item is replicated N-times where N can be configured “per-instance” of Dynamo (a typical for N at Amazon is 3, cf. [DHJ<sup>+</sup>07, p. 214]). The storage node in charge of storing a tuple with key  $k^2$  also becomes responsible for replicating updated versions of the tuple with key  $k$  to its  $N-1$  successors in clockwise direction. There is a list of nodes – called *preference list* – determined for each key  $k$  that has to be stored in Dynamo. This list consists of more than N nodes as N successive virtual nodes may map to less than N distinct physical nodes (as also discussed in subsection 3.2.1 on consistent hashing).



**Figure 4.1.:** Amazon’s Dynamo – Consistent Hashing with Replication (taken from [DHJ<sup>+</sup>07, p. 209])

<sup>2</sup>i. e. the next node on the ring in clockwise direction from  $\text{hash}(k)$

## Data Versioning

Dynamo is designed to be an eventually consistent system. This means that update operations return before all replica nodes have received and applied the update. Subsequent read operations therefore may return different versions from different replica nodes. The update propagation time between replicas is limited in Amazon's platform if no errors are present; under certain failure scenarios however “updates may not arrive at all replicas for an extend period of time” (cf. [DHJ<sup>+</sup>07, p. 210]).

Such inconsistencies need to be taken into consideration by applications. As an example, the shopping cart application never rejects add-to-cart-operations. Even when evident that the replica does not feature the latest version of a shopping cart (indicated by a vector clock delivered with update requests, see below), it applies the add-operation to its local shopping cart.

As a consequence of an update operation, Dynamo always creates a new and immutable version of the updated data item. In Amazon's production systems most of these versions subsume one another linearly and the system can determine the latest version by syntactic reconciliation. However, because of failures (like network partitions) and concurrent updates multiple, conflicting versions of the same data item may be present in the system at the same time. As the datastore cannot reconcile these concurrent versions only the client application that contains knowledge about its data structures and semantics is able to resolve version conflicts and conciliate a valid version out of two or more conflicting versions (semantic reconciliation). So, client applications using Dynamo have to be aware of this and must “explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates)” [DHJ<sup>+</sup>07, p. 210]).

To determine conflicting versions, perform syntactic reconciliation and support client application to resolve conflicting versions Dynamo uses the concept of vector clocks (introduced in subsection 3.1.3). As mentioned in the subsection on Dynamo's system interface (see 4.1.3) clients have to deliver a context when issuing update requests. This context includes a vector clock of the data they have read earlier, so that Dynamo knows which version to update and can set the vector clock of the updated data item properly. If concurrent versions of a data item occur, Dynamo will deliver them to clients within the context-information replied by the read-request. Before client issues an update to such a data item, it has to reconcile the concurrent versions into one valid version.

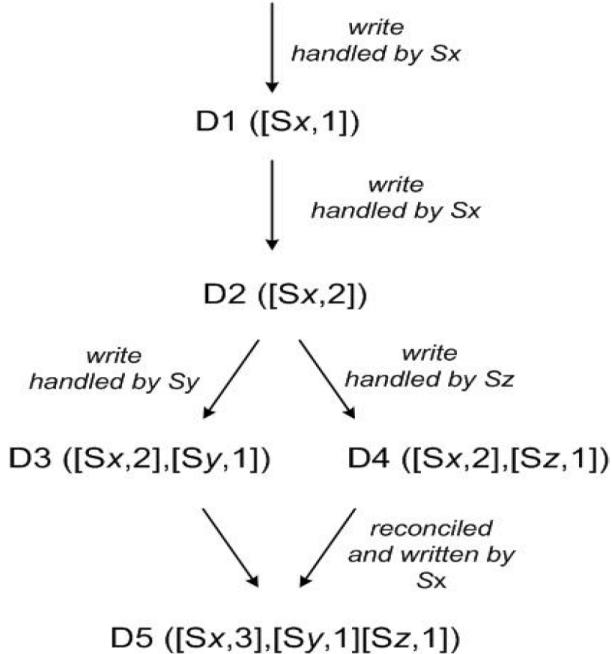
Figure 4.2 illustrates the usage of vector clocks as well as the detection and reconciliation of concurrent versions.

In this illustration a client first creates a data item and the update request is handled by a storage host Sx, so that the vector clock ([Sx,1]) will be associated with it. Next, a client updates this data item and this update request leading to version 2 is also executed by node Sx. The resulting vector clock will be ([Sx,2]), as a causal ordering between [Sx,1] and [Sx,2] can be determined as the latter clearly succeeds the former (same storage host, ascending version numbers). As [Sx,1] has only one successor ([Sx,2]) it is no longer needed for causal reasoning and can be removed from the vector clock.

Next, one client issues an update for version D2 that gets handled by storage host Sy and results in the vector clock ([Sx,2],[Sy,1]). Here, the element [Sx,2] of the vector clock cannot get omitted. This is because the example assumes<sup>3</sup> that a client issuing it has read version D2 from a node (say Sx) while storage host Sy has not yet received this version from its companion nodes. So the client wants to update a more recent version than node Sy knows at that time which is accepted due to the “always writeable”-property of Dynamo. The same happens with another client that has read D2 and issued an update handled by storage host Sz which is unaware of the version D2 at that time. This results in version D4 with the vector clock ([Sx,2],[Sz,1]).

---

<sup>3</sup>These assumptions cannot be seen in the illustration of figure 4.2 but are only contained in the textual description by DeCandia et al. (cf. [DHJ<sup>+</sup>07, p. 211]).



**Figure 4.2.:** Amazon’s Dynamo – Concurrent Updates on a Data Item (taken from [DHJ<sup>+</sup>07, p. 211])

In the next read request both D3 and D4 get delivered to a client along with a summary of their vector clocks—in particular:  $([Sx,2],[Sy,1],[Sz,1])$ . The client can detect that versions D3 and D4 are in conflict, as the combined vector clock submitted in the read context does not reflect a linear, subsequent ordering. Before a client can issue another update to the system it has to reconcile a version D5 from the concurrent versions D3 and D4. If this update request is handled by node Sx again, it will advance its version number in the vector clock resulting in  $([Sx,3],[Sy,1],[Sz,1])$  as depicted in figure 4.2.

### Execution of `get()` and `put()` Operations

Dynamo allows any storage node of a Dynamo instance to receive get and put requests for any key. In order to contact a node, clients applications may use either a routing through a generic load balancer or a client-library that reflects Dynamo’s partitioning scheme and can determine the storage host to contact by calculation. The advantage of the first node-selection approach is that it can remain unaware of Dynamo specific code. The second approach reduces latency as the load balancer does not have to be contacted, demanding one less network hop (cf. [DHJ<sup>+</sup>07, p. 211]).

As any node can receive requests for any key in the ring the requests may have to be forwarded between storage hosts. This is taking place based on the preference list containing prioritized storage hosts to contact for a given key. When a node receives a client request, it will forward it to storage hosts according to their order in the preference list (if the node itself is placed first in the preference list, forwarding will naturally become dispensable). Once a read or update request occurs, the first  $N$  healthy nodes will be taken into account (inaccessible and down nodes are just skipped over; cf. [DHJ<sup>+</sup>07, p. 211]).

To provide a consistent view to clients, Dynamo applies a quorum-like consistency protocol containing two configurable values:  $R$  and  $W$  serving as the minimum number of storage hosts having to take part in successful read or write operations, respectively. To get a quorum-like system, these values have to be set to  $R + W > N$ . DeCandia et al. remark that the slowest replica dictates the latency of an operation and therefore  $R$  and  $W$  are often chosen so that  $R + W < N$  (in order to reduce the probability of slow hosts getting involved into read and write requests; cf. [DHJ<sup>+</sup>07, p. 211]). In other use cases such

as Dynamo setups that “function as the authoritative persistence cache for data stored in more heavy weight backing stores”  $R$  is typically set to 1 and  $W$  to  $N$  to allow high performance for read operations. On the other hand “low values of  $W$  and  $R$  can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas”. In addition, durability remains vulnerable for a certain amount of time when the write request has been completed but the updated version has been propagated to a few nodes only. DeCandia et al. comment that “[the] main advantage of Dynamo is that its client applications can tune the values of  $N$ ,  $R$  and  $W$  to achieve their desired levels of performance, availability and durability” (cf. [DHJ<sup>+</sup>07, p. 214]). (A typical configuration that fits Amazon’s SLAs concerning performance, durability, consistency and availability is  $N = 3, R = 2, W = 2$ , cf. [DHJ<sup>+</sup>07, p. 215]).

Upon write requests (such as the put operation of Dynamo’s interface), the first answering node on the preference list (called *coordinator* in Dynamo) creates a new vector clock for the new version and writes it locally. Following this procedure, the same node replicates the new version to other storage hosts responsible for the written data item (the next top  $N$  storage hosts on the preference list). The write request is considered successful if at least  $W - 1$  of these hosts respond to this update (cf. [DHJ<sup>+</sup>07, p. 211–212]).

Once a storage host receives a read request it will ask the  $N$  top storage hosts on the preference list for the requested data item and waits for  $R - 1$  to respond. If these nodes reply with different versions that are in conflict (observable by vector-clock reasoning), it will return concurrent versions to the requesting client (cf. [DHJ<sup>+</sup>07, p. 212]).

## Membership

Dynamo implements an explicit mechanism of adding and removing nodes to the system. There is no implicit membership detection implemented in Dynamo as temporary outages or flapping servers would cause the system to recalculate ranges on the consistent hash ring and associated data structures like Merkle-trees (see below). Additionally, Dynamo already provides means to handle temporary unavailable nodes, as will be discussed in the next subsection. Therefore, Dynamo administrators have to explicitly add and remove nodes via a command-line or a browser interface. This results in a membership-change request for a randomly chosen node which persists the change along with a timestamp (to keep a membership history) and propagates it via a Gossip-based protocol to the other nodes of the system. Every second, the membership protocol requires the storage hosts to randomly contact a peer in order to bilaterally reconcile the “persisted membership change histories” (cf. [DHJ<sup>+</sup>07, p. 212]). Doing so, membership changes are spread and an eventually consistent membership view is being established.

In the process described above logical partitions can occur temporarily when adding nodes to a Dynamo system. To prevent these, certain nodes can take the special role of seeds which can be discovered by an external mechanism such as static configuration or some configuration service. Seeds are consequently known to all nodes and will thus be contacted in the process of membership change gossiping so that all nodes will eventually reconcile their view of membership and “logical partitions are highly unlikely” (cf. [DHJ<sup>+</sup>07, p. 213]).

Once a node is joining the system, the number of virtual nodes getting mapped to the consistent hash ring has to be determined. For this purpose, a token is chosen for each virtual node, i.e. a value that determines the virtual node’s position on the consistent hash ring. The set of tokens is persisted to disk on the physical node and spread via the same Gossip-based protocol that is used to spread to membership changes, as seen in the previous paragraph (cf. [DHJ<sup>+</sup>07, p. 212f]). The number of virtual nodes per physical node is chosen proportionally to the hardware resources of the physical node (cf. [DHJ<sup>+</sup>07, p. 210]).

By adding nodes to the system the ownership of key ranges on the consistent hash ring changes. When some node gets to know a recently added node via the membership propagation protocol and determines that it is no longer in charge of some portion of keys it transfers these to the node added. When a node is being removed, keys are being reallocated in a reverse process (cf. [DHJ<sup>+</sup>07, p. 213]).

With regards to the mapping of nodes on the consistent hash ring and its impact on load distribution, partition, balance, data placement, archival and bootstrapping, DeCandia et al. discuss three strategies implemented in Dynamo since its introduction:

1. T random tokens per node and partition by token value (the initial strategy)
2. T random tokens per node and equal sized partitions (an interim strategy as a migration path from strategy 1 to 3)
3. Q/S tokens per node and equal sized partitions (the current strategy; Q is the number of equal sized partitions on the consistent hash ring, S is the number of storage nodes)

For details on these strategies see [DHJ<sup>+</sup>07, p. 215–217]. Experiments and practice at Amazon show that “strategy 3 is advantageous and simpler to deploy” as well as more efficient and requiring the least amount of space to maintain membership information. According to DeCandia et al., the major benefits of this strategy are (cf. [DHJ<sup>+</sup>07, p. 217]):

- “Faster bootstrapping/recovery: Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery.”
- “Ease of archival: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow.”

However, a disadvantage of this strategy is that “changing the node membership requires coordination in order to preserve the properties required of the assignment” (cf. [DHJ<sup>+</sup>07, p. 217]).

## Handling of Failures

For tolerating failures provoked by temporary unavailability storage hosts, Dynamo is not employing any strict quorum approach but a *sloopy* one. This approach implies that in the execution of read and write operations the first  $N$  *healthy* nodes of a data item’s preference list are taken into account. These are not necessarily the first  $N$  nodes walking clockwise around the consistent hashing ring (cf. [DHJ<sup>+</sup>07, 212]).

A second measure to handle temporary unavailable storage hosts are so called *hinted handoffs*. They come into play if a node is not accessible during a write operation of a data item it is responsible for. In this case, the write coordinator will replicate the update to a different node, usually carrying no responsibility for this data item (to ensure durability on  $N$  nodes). In this replication request, the identifier of the node the update request was originally destined to is contained as a hint. As this node is recovering and becoming available again, it will receive the update; the node having received the update as a substitute can then delete it from its local database (cf. [DHJ<sup>+</sup>07, 212]).

DeCandia et al. remark that it is sufficient to address temporary node unavailability as permanent removal and addition of nodes is done explicitly as discussed in the prior subsection. In earlier versions of Dynamo a global view of node failures was established by an external failure detector that informed all nodes of a Dynamo ring. “Later it was determined that the explicit node join and leave methods obviates the need for

a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests)", DeCandia et al. conclude (cf. [DHJ<sup>+07</sup>, p. 213]).

In order to address the issue of entire datacenter outage, Dynamo is configured in a way to ensure storage in more than one data center. "In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers", DeCandia et al. remark (cf. [DHJ<sup>+07</sup>, 212]). Data centers of Amazon's infrastructure are connected via high speed network links, so that latency due to replication between datacenters appears to be no issue.

In addition to the failure scenarios described above there may be threats to durability. These are addressed by implementing an anti-entropy protocol for replica synchronization. Dynamo uses Merkle-trees to efficiently detect inconsistencies between replicas and determine data to be synchronized. "A Merkle-tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children", DeCandia et al. explicate. This allows efficient checking for inconsistencies, as two nodes determine differences by hierarchically comparing hash-values of their Merkle-trees: firstly, by examining the tree's root node, secondly—if inconsistencies have been detected—by inspecting its child nodes, and so forth. The replica synchronization protocol is requiring little network bandwidth as only hash-values have to be transferred over the network. It can also operate fast as data is being compared in a divide-and conquer-manner by tree traversal. In Dynamo a storage node maintains a Merkle-tree for each key-range (i. e. range between two storage nodes on the consistent hash ring) it is responsible for and can compare such a Merkle-tree with those of other replicas responsible for the same key-range. DeCandia et al. consider a downside of this approach that whenever a node joins or leaves the system some Merkle-tree will get invalid due to changed key-ranges (cf. [DHJ<sup>+07</sup>, p. 212]).

#### 4.1.4. Implementation and Optimizations

In addition to the system design, DeCandia et al. make concrete remarks on the implementation of Dynamo. Based on experience at Amazon, they also provide suggestions for its optimization (cf. [DHJ<sup>+07</sup>, p. 213f]):

- The code running on a Dynamo node consists of a request coordination, a membership and a failure detection component—each of them implemented in Java.
- Dynamo provides pluggable persistence components (e.g. Berkeley Database Transactional Data Store and BDB Java Edition [Ora10d], MySQL [Ora10c], an in-memory buffer with a persistent backing store). Experience from Amazon shows that "[applications] choose Dynamo's local persistence engine based on their object size distribution".
- The component in charge of coordinating requests is implemented "on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages". Internode communication employs Java NIO channels (see [Ora10b]).
- When a client issues a read or write request, the contacted node will become its coordinator. It then creates a state machine that "contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request."
- If a request coordinator receives a stale version of data from a node, it will update it with the latest version. This is called read-repair "because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it".

- To achieve even load-distribution write requests can address to “any of the top N nodes in the preference list”.
- As an optimization reducing latency for both, read and write requests, Dynamo allows client applications to be coordinator of these operations. In this case, the state machine created for a request is held locally at a client. To gain information on the current state of storage host membership the client periodically contacts a random node and downloads its view on membership. This allows clients to determine which nodes are in charge of any given key so that it can coordinate read requests. A client can forward write requests to nodes in the preference list of the key to become written. Alternatively, clients can coordinate write requests locally if the versioning of the Dynamo instance is based on physical timestamps (and not on vector clocks). Both, the 99.9<sup>th</sup> percentile as well as the average latency can be dropped significantly by using client-controlled request coordination. This “improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node”, DeCandia et al. conclude (cf. [DHJ<sup>+</sup>07, p. 217f]).
- As write requests are usually succeeding read requests, Dynamo is pursuing a further optimization: in a read response, the storage node replying the fastest to the read coordinator is transmitted to the client; in a subsequent write request, the client will contact this node. “This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting “read-your-writes” consistency”.
- A further optimization to reach Amazon’s desired performance in the 99.9<sup>th</sup> percentile is the introduction of an object buffer in the main memory of storage nodes. In this buffer write requests are queued and persisted to disk periodically by a writer thread. In read operations both the object buffer and the persisted data of storage nodes have to be examined. Using an in-memory buffer that is being persisted asynchronously can result in data loss when a server crashes. To reduce this risk, the coordinator of the write request will choose one particular replica node to perform a *durable write* for the data item. This durable write will not impact the performance of the request as the coordinator only waits for  $W - 1$  nodes to answer before responding to the client (cf. [DHJ<sup>+</sup>07, p. 215]).
- Dynamo nodes do not only serve client requests but also perform a number of background tasks. For resource division between request processing and background tasks a component named *admission controller* is in charge that “constantly monitors the behavior of resource accesses while executing a “foreground” put/get operation”. Monitoring includes disk I/O latencies, lock-contention and transaction timeouts resulting in failed database access as well as waiting periods in the request queue. Via monitoring feedback, the admission controller will decide on the number of time-slices for resource access or consumption to be given to background tasks. It also coordinates the execution of background tasks which have to explicitly apply for resources with the admission controller (cf. [DHJ<sup>+</sup>07, p. 218]).

#### 4.1.5. Evaluation

To conclude the discussions on Amazon’s Dynamo, a brief evaluation of Bob Ippolito’s talk “Drop ACID and think about Data” shall be presented in table 4.2.

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>• “No master”</li> <li>• “Highly available for write” operations</li> <li>• “Knobs for tuning reads” (as well as writes)</li> <li>• “Simple”</li> </ul>	<ul style="list-style-type: none"> <li>• “Proprietary to Amazon”</li> <li>• “Clients need to be smart” (support vector clocks and conflict resolution, balance clusters)</li> <li>• “No compression” (client applications may compress values themselves, keys cannot be compressed)</li> <li>• “Not suitable for column-like workloads”</li> <li>• “Just a Key/Value store” (e.g. range queries or batch operations are not possible)</li> </ul>

**Table 4.2.:** Amazon’s Dynamo – Evaluation by Ippolito (cf. [Ipp09])

## 4.2. Project Voldemort

Project Voldemort is a key-/value-store initially developed for and still used at LinkedIn. It provides an API consisting of the following functions: (cf. [K<sup>+10b</sup>]<sup>4</sup>):

- `get(key)`, returning a value object
- `put(key, value)`
- `delete(key)`

Both, keys and values can be complex, compound objects as well consisting of lists and maps. In the Project Voldemort design documentation it is discussed that—compared to relational databases—the simple data structure and API of a key-value store does not provide complex querying capabilities: joins have to be implemented in client applications while constraints on foreign-keys are impossible; besides, no triggers and views may be set up. Nevertheless, a simple concept like the key-/value store offers a number of advantages (cf. [K<sup>+10b</sup>]):

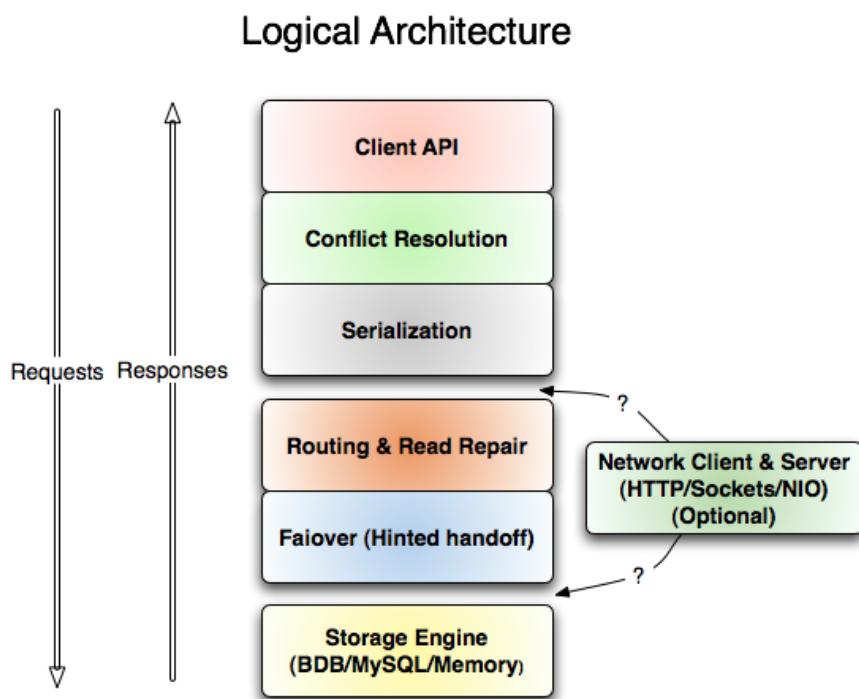
- Only efficient queries are allowed.
- The performance of queries can be predicted quite well.
- Data can be easily distributed to a cluster or a collection of nodes.
- In service oriented architectures it is not uncommon to have no foreign key constraints and to do joins in the application code as data is retrieved and stored in more than one service or datasource.
- Gaining performance in a relational database often leads to denormalized datastructures or storing more complex objects as BLOBs or XML-documents.

<sup>4</sup> Unless indicated otherwise, the information provided on Project Voldemort within this section has been taken from its official design documentation (cf. [K<sup>+10b</sup>]).

- Application logic and storage can be separated nicely (in contrast to relational databases where application developers might get encouraged to mix business logic with storage operation or to implement business logic in the database as stored procedures to optimize performance).
- There is no such impedance mismatch between the object-oriented paradigm in applications and paradigm of the datastore as it is present with relational databases.

#### 4.2.1. System Architecture

Project Voldemort specifies a logical architecture consisting of a number of layers as depicted in figure 4.3.

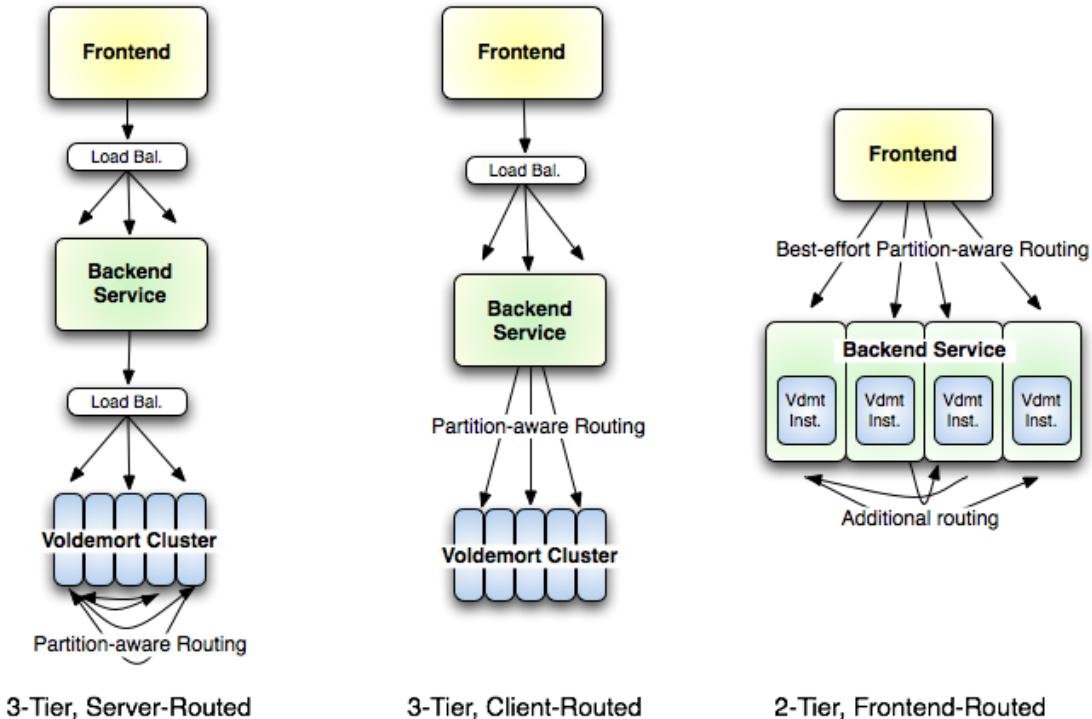


**Figure 4.3.: Project Voldemort – Logical Architecture (taken from [K<sup>+</sup>10b])**

Each layer of the logical architecture has its own responsibility (e.g. TCP/IP network communication, serialization, version recovery, routing between nodes) and also implements an interface consisting of the operations `get`, `put` and `delete`. These are the operations exposed by a Project Voldemort instance as a whole. If e.g. the `put` operation is invoked on the routing layer it is responsible for distributing this operation to all nodes in parallel and for handling possible errors.

The layered logical architecture provides certain flexibility for deployments of Project Voldemort as layers can be mixed and matched to meet the requirements of an application. For example, a compression layer may be introduced beneath the serialization layer in order to compress all exchanged data. Likewise, intelligent routing (i.e. determining the node which manages the partition containing the requested data) can be provided transparently by the datastore if the network layer is placed on top of the routing layer; if these layers are twisted, the application can do the routing itself reducing latency caused by network hops.

## Physical Architecture Options



**Figure 4.4.: Project Voldemort – Physical Architecture Options (taken from [K<sup>+</sup>10b])**

Figure 4.4 depicts options for the physical deployment of Project Voldemort with a focus on routing and load-balancing.

- The left side illustrates a common three-tier architecture with two load-balancing components (either realized in hardware or software, e.g. in a round-robin process).
- The centre illustration avoids the load-balancer between the backend-service and the Voldemort cluster. While reducing latency, this approach tightens coupling. This is the case because the backend service has to determine the partitions of data and the node(s) they are residing on in order to route requests from the frontend (“partition-aware routing”).
- In the right subfigure backend services even wrap Voldemort instances. The frontend is already trying to carry out partition-aware routing to enable high performant setups. This is the case as frontend requests are immediately directed to a backend-service containing a partition of the datastore (which might go wrong but then gets corrected by routing between the backend-services).

The trade-off depicted in figure 4.4 is reducing latency by minimizing network hops vs. strong coupling by routing-logic that moves up the software-stack towards the frontend<sup>5</sup>.

Further examinations could be conducted about reducing the impact of disk I/O, the biggest performance killer in storage systems. The Project Voldemort design documentation suggests data partitioning as well as heavy caching to reduce performance bottlenecks caused by disk I/O, especially disk seek times and a low disk-cache efficiency. Therefore Project Voldemort—like Dynamo—uses consistent hashing with replication of data to tolerate downtimes of storage nodes.

<sup>5</sup>Voldemort provides a Java-library that can be used to do routing.

### 4.2.2. Data Format and Queries

Project Voldemort allows namespaces for key-/value-pairs called “stores”, in which keys are unique. While each key is associated with exactly one value, values are allowed to contain lists and maps as well as scalar values.

Operations in Project Voldemort are atomic to exactly one key-/value-pair. Once a get operation is executed, the value is streamed from the server via a cursor. Documentation of Project Voldemort considers this approach to not work very well in combination with values consisting of large lists “which must be kept on the server and streamed lazily via a cursor”; in this case, breaking the query into subqueries is seen as more efficient.

### 4.2.3. Versioning and Consistency

Like Amazon’s Dynamo Project Voldemort is designed to be highly available for write operations, allows concurrent modifications of data and uses vector clocks to allow casual reasoning about different versions (see sections 3.1.3 and 4.1.3). If the datastore itself cannot resolve version conflicts, client applications are requested for conflict resolution at read time. This read reconciliation approach is being favored over the strongly consistent but inefficient two-phase commit (2PC) approach as well as Paxos-style consensus protocols. This is the case because it requires little coordination and provides high availability and efficiency<sup>6</sup> as well as failure tolerance. On the downside, client applications have to implement conflict resolution logic that is not necessary in 2PC and Paxos-style consensus protocols.

### 4.2.4. Persistence Layer and Storage Engines

As indicated in figure 4.3, page 63, Project Voldemort provides pluggable persistency as the lowest layer of the logical architecture allows for different storage engines. Out of the box Berkeley DB (default storage engine), MySQL as well as in-memory storage are delivered with Project Voldemort. To use another existing or self-written storage engine, the operations get, put and delete besides an iterator for values will have to be implemented.

### 4.2.5. Data Model and Serialization

On the lowest level keys and values in Project Voldemort are simply byte-arrays. In order to allow applications a more sophisticated notion of keys and values, data models can be configured for each Voldemort store. These data models define serializers that are responsible to convert byte-arrays into the desired data structures and formats. Project Voldemort already contains serializers for the following data structures and formats:

**JSON (JavaScript Object Notation)** is a binary and typed data model which supports the data types list, map, date, boolean as well as numbers of different precision (cf. [Cro06]). JSON can be serialized to and deserialized from bytes as well as strings. By using the JSON data type it is possible to communicate with Project Voldemort instances in a human-readable format via administration tools like the Voldemort command line client.

---

<sup>6</sup>As an example, the Project Voldemort design documentation indicates the number of roundtrips needed for write-operations:  $W$  in case of read-repair compared to  $2 * N$  for 2PC (with  $N$  as the number of nodes and  $W$  as the number of nodes that are required to successfully write an update); the number of roundtrips in Paxos-style consensus protocols varies for different implementations but is considered to be in the magnitude of 2PC.

**String** to store uninterpreted strings, which can also be used for XML blobs.

**Java Serialization** provided by Java classes implementing the `java.io.Serializable` interface (cf. [Ora10a], [Blo01, p. 213ff]).

**Protocol Buffers** are “Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data” which also contains an interface description language to generate code for custom data interchange. Protocol Buffers are widely used at Google “for almost all of its internal RPC protocols and file formats” (cf. [Goo10a], [Goo10b]).

**Identity** does no serialization or deserialization at all but simply hands over byte-arrays.

Project Voldemort can be extended by further custom serializers. In order to allow correct data interchange, they have to be made known to both, the data store logic and client applications.

The Project Voldemort design documentation makes further remarks with regards to the JSON format. It is considered to map well with numerous programming languages as it provides common data types (strings, numbers, lists, maps, objects) and does not have the object relational mapping problem of impedance mismatch. On the downside, JSON is schema-less internally which causes some issues for applications processing JSON data (e.g. data stores wishing to pursue fundamental checks of JSON-values). Each JSON document could contain an individual schema definition but this would be very wasteful considering that in a datastore a large amount of data shares the same structure. Project Voldemort therefore offers the possibility to specify data formats for keys and values, as listed in table 4.3.

Type	Storable Sub-types	Bytes used	Java-Type	JSON Example	Definition Example
number	int8, int16, int32, int64, float32, float64, date	8, 16, 32, 64, 32, 64, 32	Byte, Short, Integer, Long Float, Double, Date	1	"int32"
string	string, bytes	2 + length of string or bytes	String, byte[]	"hello"	"string"
boolean	boolean	1	Boolean	true	"boolean"
object	object	1 + size of contents	Map<String, Object>	{"key1":1, "key2":"2", "key3":false}	{"name":"string", "height":"int16"}
array	array	size * sizeof(type)	List<?>	[1, 2, 3]	["int32"]

**Table 4.3.:** Project Voldemort – JSON Serialization Format Data Types (taken from [K+10b])

The data type definition for the JSON serialization format allows Project Voldemort to check values and store them efficiently, albeit the data types for values cannot be leveraged for data queries and requests. To prevent invalidated data caused by redefinition of value data types, Project Voldemort is storing a version along with the data allowing schema migrations.

#### 4.2.6. Index Precalculation

Project Voldemort allows to prebuild indexes offline (e.g. on Hadoop), upload them to the datastore and transparently swap to them. This is especially useful for batch operations inserting or updating large amounts of data causing index rebuilds if the data is uploaded as a whole or index fragmentation if it is inefficiently inserted in small portions. In Project Voldemort large amounts of data can be inserted as a whole and the offline index building feature disburdens live systems from full index rebuilding or index fragmentation.

### 4.3. Other Key-/Value-Stores

#### 4.3.1. Tokyo Cabinet and Tokyo Tyrant

This datastore builds on a collection of software components out of which the Tokyo Cabinet and Tokyo Tyrant are the most important in this context. Tokyo Cabinet ([FAL10a]) is the core library of this datastore persisting data and exposing a key-/value-interface to clients and thereby abstracting from internal data structures such as hash-tables or B+tree-indexes. Tokyo Tyrant ([FAL10b]) provides access to this database library via network which is possible via a proprietary binary protocol, HTTP as well as through the memcached protocol. Tokyo Cabinet manages data storage on disk and in memory in a fashion similar to paging / swapping. Memory pages are flushed to disk periodically “which leaves an open data loss hole”, as Hoff comments (cf. [Hof09a]). On the other side, Tokyo Cabinet allows to compress pages by the LZW-algorithm which can achieve good compression ratios (see e.g. [Lin09]). Tokyo Cabinet does partition data automatically and therefore has to be replicated with a strategy similar to MySQL. In addition to lookups by key it can match prefixes and ranges if keys are ordered. Regarding transactional features worth mentioning Tokyo Cabinet provides write-ahead logging and shadow paging. The Toyko suite is developed actively, well documented and widely regarded as high-performant: 1 million records can be stored in 0.7 seconds using the hash-table engine and in 1.6 seconds using the b-tree according to North (cf. [Nor09], [Ipp09], [See09], [Lin09], [Hof09a]).

#### 4.3.2. Redis

Redis is a relatively new datastore which its developers unspecifically refer to as a “data structure store”; it is commonly subsumed under key-/value-stores because of its map/dictionary-API. Special about Redis is that it allows matching for key-ranges e.g. matching of numeric ranges or regular expressions. In contrast to other key-/value-stores, Redis does not only store bytes as values but also allows lists and sets in values by supporting them directly. A major disadvantage about Redis is that the amount of main memory limits the amount of data that is possible to store. This cannot be expanded by the usage of hard-disks. Ippolito comments that for this reason Redis is probably a good fit for a caching-layer (cf. [Ipp09]).

#### 4.3.3. Memcached and MemcacheDB

Memcached, the popular and fast memory-caching solution widely used among large and ultra-large scale web sites to reduce database-load, has already been mentioned in section 3.2 on partitioning. Memcached servers—like the key-/value-stores discussed in this chapter—provide a map/dictionary API consisting of the operations `get`, `put` and `remove`. Though not intended for persistent storage (cf. [F<sup>+</sup>10b]) there is an existing solution named MemcacheDB (cf. [Chu09]) that conforms to the memcached protocol (cf. [F<sup>+</sup>10c]) and adds persistence based on Berkeley DB to it (cf. [Nor09], [Ipp09]). As memcached does not

provide any replication between nodes and is also not tolerant towards machine failures, simply adding a persistent storage to it is not enough, as blogger Richard Jones of Last.fm remarks. He notes that solutions like repcached can replicate whole memcached servers (in a master slave setup) but without fault-tolerant partitioning they will cause management and maintenance efforts (cf. [Jon09]).

#### 4.3.4. Scalaris

Scalarmis is a key-/value-store written in Erlang taking profit of this language's approach e.g. in implementing a non-blocking commit protocol for atomic transactions. Scalaris uses an adapted version of the chord service (cf. [SMK<sup>+</sup>01]) to expose a distributed hash table to clients. As it stores keys in lexicographic order, range queries on prefixes are possible. In contrast to other key-/value-stores Scalaris has a strict consistency model, provides symmetric replication and allows for complex queries (via programming language libraries). It guarantees ACID properties also for concurrent transactions by implementing an adapted version of the Paxos consensus protocol (cf. [Lam98]). Like memcached, Scalaris is a pure in-memory key-/value-store (cf. [Nor09], [Jon09]).

# 5. Document Databases

In this chapter another class of NoSQL databases will be discussed. Document databases are considered by many as the next logical step from simple key-/value-stores to slightly more complex and meaningful data structures as they at least allow to encapsulate key-/value-pairs in documents. On the other hand there is no strict schema documents have to conform to which eliminates the need schema migration efforts (cf. [Ipp09]). In this chapter Apache CouchDB and MongoDB as the two major representatives for the class of document databases will be investigated.

## 5.1. Apache CouchDB

### 5.1.1. Overview

CouchDB is a document database written in Erlang. The name CouchDB is nowadays sometimes referred to as “Cluster of unreliable commodity hardware” database, which is in fact a backronym according to one of its main developers (cf. [PLL09]).

CouchDB can be regarded as a descendant of Lotus Notes for which CouchDB’s main developer Damien Katz worked at IBM before he later initiated the CouchDB project on his own<sup>1</sup>. A lot of concepts from Lotus Notes can be found in CouchDB: documents, views, distribution, and replication between servers and clients. The approach of CouchDB is to build such a document database from scratch with technologies of the web area like Representational State Transfer (REST; cf. [Fie00]), JavaScript Object Notation (JSON) as a data interchange format, and the ability to integrate with infrastructure components such as load balancers and caching proxies etc. (cf. [PLL09]).

CouchDB can be briefly characterized as a document database which is accessible via a RESTful HTTP-interface, containing schema-free documents in a flat address space. For these documents JavaScript functions select and aggregate documents and representations of them in a MapReduce manner to build views of the database which also get indexed. CouchDB is distributed and able to replicate between server nodes as well as clients and servers incrementally. Multiple concurrent versions of the same document (MVCC) are allowed in CouchDB and the database is able to detect conflicts and manage their resolution which is delegated to client applications (cf. [Apa10c], [Apa10a]).

The most notable use of CouchDB in production is *ubuntu one* ([Can10a]) the cloud storage and replication service for Ubuntu Linux ([Can10b]). CouchDB is also part of the BBC’s new web application platform (cf. [Far09]). Furthermore some (less prominent) blogs, wikis, social networks, Facebook apps and smaller web sites use CouchDB as their datastore (cf. [C<sup>+</sup>10]).

---

<sup>1</sup>Advocates therefore call it “Notes done right” sometimes.

### 5.1.2. Data Model and Key Abstractions

#### Documents

The main abstraction and data structure in CouchDB is a document. Documents consist of named fields that have a key/name and a value. A fieldname has to be unique within a document and its assigned value may a string (of arbitrary length), number, boolean, date, an ordered list or an associative map (cf. [Apa10a]). Documents may contain references to other documents (URLs, URLs) but these do not get checked or held consistent by the database (cf. [PLL09]). A further limitation is that documents in CouchDB cannot be nested (cf. [Ipp09]).

A wiki article may be an example of such a document:

```
"Title" : "CouchDB",
"Last editor" : "172.5.123.91",
"Last modified": "9/23/2010",
"Categories": ["Database", "NoSQL", "Document Database"],
"Body": "CouchDB is a ...",
"Reviewed": false
```

Besides fields, documents may also have attachments and CouchDB maintains some metadata such as a unique identifier and a sequence id<sup>2</sup>) for each document (cf. [Apa10b]). The document id is a 128 bit value (so a CouchDB database can store  $3.4^{38}$  different documents) ; the revision number is a 32 bit value determined by a hash-function<sup>3</sup>.

CouchDB considers itself as a semi-structured database. While relational databases are designed for structured and interdependent data and key-/value-stores operate on uninterpreted, isolated key-/value-pairs document databases like CouchDB pursue a third path: data is contained in documents which do not correspond to a fixed schema (schema-free) but have some inner structure known to applications as well as the database itself. The advantages of this approach are that first there is no need for schema migrations which cause a lot of effort in the relational databases world; secondly compared to key-/value-stores data can be evaluated more sophisticatedly (e.g. in the calculation of views). In the web application field there are a lot of document-oriented applications which CouchDB addresses as its data model fits this class of applications and the possibility to iteratively extend or change documents can be done with a lot less effort compared to a relational database (cf. [Apa10a]).

Each CouchDB database consists of exactly one flat/non-hierarchical namespace that contains all the documents which have a unique identifier (consisting of a document id and a revision number aka sequence id) calculated by CouchDB. A CouchDB server can host more than one of these databases (cf. [Apa10c], [Apa10b]). Documents were formerly stored as XML documents but today they are serialized in a JSON-like format to disk (cf. [PLL09]).

Document indexing is done in B-Trees which are indexing the document's id and revision number (sequence id; cf. [Apa10b]).

<sup>2</sup>The sequence id is a hash value that represents a revision number of the document and it will be called revision number in the discussion of CouchDB here.

<sup>3</sup>CouchDB assumes that 32 bit are enough to not let hash values of revision numbers collide and therefore does in fact not implement any means to prevent or handle such collisions (cf. [PLL09]).

## Views

CouchDB's way to query, present, aggregate and report the semi-structured document data are views (cf. [Apa10a], [Apa10b]). A typical example for views is to separate different types of documents (such as blog posts, comments, authors in a blog system) which are not distinguished by the database itself as all of them are just documents to it ([PLL09]).

Views are defined by JavaScript functions which neither change nor save or cache the underlying documents but only present them to the requesting user or client application. Therefore documents as well as views (which are in fact special documents, called *design-documents*) can be replicated and views do not interfere with replication. Views are calculated on demand. There is no limitation regarding the number of views in one database or the number of representations of documents by views.

The JavaScript functions defining a view are called `map` and `reduce` which have similar responsibilities as in Google's MapReduce approach (cf. [DG04]). The `map` function gets a document as a parameter, can do any calculation and may emit arbitrary data for it if it matches the view's criteria; if the given document does not match these criteria the `map` function emits nothing. Examples of emitted data for a document are the document itself, extracts from it, references to or contents of other documents (e.g. semantically related ones like the comments of a user in a forum, blog or wiki).

The data structure emitted by the `map` function is a triple consisting of the document id, a key and a value which can be chosen by the `map` function. Documents get sorted by the key which does not have to be unique but can occur for more than one document; the key as a sorting criteria can be used to e.g. define a view that sorts blog posts descending by date for a blog's home page. The value emitted by the `map` function is optional and may contain arbitrary data. The document id is set by CouchDB implicitly and represents the document that was given to the emitting `map` function as an argument (cf. [PLL09]).

After the `map` function has been executed its results get passed to an optional `reduce` function which is optional but can do some aggregation on the view (cf. [PLL09]).

As all documents of the database are processed by a view's functions this can be time consuming and resource intensive for large databases. Therefore a view is not created and indexed when write operations occur<sup>4</sup> but on demand (at the first request directed to it) and updated incrementally when it is requested again<sup>5</sup>. To provide incremental view updates CouchDB holds indexes for views. As mentioned before views are defined and stored in special documents. These design documents can contain functions for more than one view if they are named uniquely. View indexes are maintained on based on these design documents and not single views contained in them. Hence, if a user requests a view its index and the indexes of all views defined in the same design document get updated (cf. [Apa10b], [PLL09]). Incremental view updates furthermore have the precondition that the `map` function is required to be referentially transparent which means that for the same document it has to emit the same key and value each time it is invoked (cf. [Apa10e]).

To update a view, the component responsible for it (called view-builder) compares the sequence id of the whole database and checks if it has changed since the last refresh of the view. If not, the view-builder determines the documents changed, deleted or created since that time; it passes new and updated documents to the view's `map` and `reduce` functions and removes deleted documents from the view. As changes to the database are written in an append-only fashion to disk (see subsection 5.1.7), the incremental updates of views can occur efficiently as the number of disk head seeks is minimal. A further advantage of the append-only index persistence is that system crashes during the update of indexes the previous state

---

<sup>4</sup>This means in CouchDB there is no “write-penalty”, as some call it (cf. [PLL09]).

<sup>5</sup>One might want to precalculate an index if e.g. bulk-updates or inserts have occurred in order not to punish the next client requesting it. In this case, one can simply request the view to be its next reader (cf. [PLL09]).

remains consistent, CouchDB omits the incompletely appended data when it starts up and can update an index when it is requested the next time.

While the view-builder is updating a view data from the view's old state can be read by clients. It is also possible to present the old state of the view to one client and the new one to another client as view indexes are also written in an append-only manner and the compactation of view data does not omit an old index state while a client is still reading from it (more on that in subsection 5.1.7).

### 5.1.3. Versioning

Documents are updated optimistically and update operations do not imply any locks (cf. [Apa10b]). If an update is issued by some client the contacted server creates a new document revisions in a copy-on-modify manner (see section 3.3) and a history of recent revisions is stored in CouchDB until the database gets compacted the next time. A document therefore is identified by a document id/key that sticks to it until it gets deleted and a revision number created by CouchDB when the document is created and each time it is updated (cf. [Apa10b]). If a document is updated, not only the current revision number is stored but also a list of revision numbers preceding it to allow the database (when replicating with another node or processing read requests) as well as client applications to reason on the revision history in the presence of conflicting versions (cf. [PLL09]).

CouchDB does not consider version conflicts as an exception but rather a normal case. They can not only occur by different clients operating on the same CouchDB node but also due to clients operating on different replicas of the same database. It is not prohibited by the database to have an unlimited number of concurrent versions. A CouchDB database can deterministically detect which versions of document succeed each other and which are in conflict and have to be resolved by the client application. Conflict resolution may occur on any replica node of a database as the node which receiving the resolved version transmits it to all replicas which have to accept this version as valid. It may occur that conflict resolution is issued on different nodes concurrently; the locally resolved versions on both nodes then are detected to be in conflict and get resolved just like all other version conflicts (cf. [Apa10b]).

Version conflicts are detected at read time and the conflicting versions are returned to the client which is responsible for conflict resolution. (cf. [Apa10b]).

A document which's most recent versions are in conflict is excluded from views (cf. [Apa10b]).

### 5.1.4. Distribution and Replication

CouchDB is designed for distributed setups that follows a peer-approach where each server has the same set of responsibilities and there are no distinguished roles (like in master/slave-setups, standby-clusters etc.). Different database nodes can by design operate completely independent and process read and write requests. Two database nodes can replicate databases (documents, document attachments, views) bilaterally if they reach each other via network. The replication process works incrementally and can detect conflicting versions in simple manner as each update of a document causes CouchDB to create a new revision of the updated document and a list of outdated revision numbers is stored. By the current revision number as well as the list of outdated revision number CouchDB can determine if are conflicting or not; if there are version conflicts both nodes have a notion of them and can escalate the conflicting versions to clients for conflict resolution; if there are no version conflicts the node not having the most recent version of the document updates it (cf. [Apa10a], [Apa10b], [PLL09])

Distribution scenarios for CouchDB include clusters, offline-usage on a notebook (e.g. for employees visiting customers) or at company locations distributed over the world where live-access to a company's or

organization's local network is slow or unstable. In the latter two scenarios one can work on a disconnected CouchDB instance and is not limited in its usage. If the network connection to replica nodes is established again the database nodes can synchronize their state again (cf. [Apa10b]).

The replication process operates incrementally and document-wise. Incrementally means that only data changed since the last replication gets transmitted to another node and that not even whole documents are transferred but only changed fields and attachment-blobs; document-wise means that each document successfully replicated does not have to be replicated again if a replication process crashes (cf. [Apa10b]).

Besides replicating whole databases CouchDB also allows for partial replicas. For these a JavaScript filter function can be defined which passes through the data for replication and rejects the rest of the database (cf. [Apa10b]).

This partial replication mechanism can be used to shard data manually by defining different filters for each CouchDB node. If this is not used and no extension like Lounge (cf. [FRL10]) CouchDB replicates all data to all nodes and does no sharding automatically and therefore by default behaves just like MySQL's replication, as Ippolito remarks (cf. [Ipp09]).

According to the CouchDB documentation the replication mechanisms are designed to distribute and replicate databases with little effort. On the other hand they should also be able to handle extended and more elaborated distribution scenarios e.g. partitioned databases or databases with full revision history. To achieve this “[the] CouchDB replication model can be modified for other distributed update models”. As an example the storage engine can be “enhanced to allow multi-document update transactions” to make it possible “to perform Subversion-like “all or nothing” atomic commits when replicating with an upstream server, such that any single document conflict or validation failure will cause the entire update to fail” (cf. [Apa10b]; more information on validation can be found below).

### 5.1.5. Interface

CouchDB databases are addressed via a RESTful HTTP interface that allows to read and update documents (cf. [Apa10b]). The CouchDB project also provides libraries providing convenient access from a number of programming languages as well as a web administration interface (cf. [Apa10c]).

CouchDB documents are requested by their URL according to the RESTful HTTP paradigm (read via HTTP GET, created and updated via HTTP PUT and deleted via HTTP DELETE method). A read operation has to go before an update to a document as for the update operation the revision number of the document that has been read and should be updated has to be provided as a parameter. To retrieve document urls—and maybe already their data needed in an application<sup>6</sup>—views can be requested by client applications (via HTTP GET). The values of view entries can be retrieved by requesting their document id and key (cf. [PLL09]). Documents as well as views are exchanged via the JSON format (cf. [Ipp09]).

Lehnhardt and Lang point out that that by providing RESTful HTTP interface many standard web infrastructure components like load-balancers, caches, SSL proxies and authentication proxies can be easily integrated with CouchDB deployments. An example of that is to use the revision number of a document as an ETag header (cf. [FGM<sup>+</sup>99, section 14.19]) on caching servers. Web infrastructure components may also hide distribution aspects from client applications—if this is required or chosen to keep applications simpler though offering performance and the possibility to leverage from a notion of the distributed database (cf. [PLL09]).

---

<sup>6</sup>As mentioned in the subsection on views they may contain arbitrary key-/value-pairs associated to a document and therefore can be designed to contain all data the client application needs for a certain use case.

### 5.1.6. ACID Properties

According to the technical documentation ACID properties can be attributed to CouchDB because of its commitment system and the way it operates on files (cf. [Apa10b]).

Atomicity is provided regarding single update operations which can either be executed to completion or fail and are rolled back so that the database never contains partly saved or updated documents (cf. [Apa10b]).

Consistency can be questioned as it cannot mean strong consistency in a distributed CouchDB setup as all replicas are always writable and do not replicate with each other by themselves. This leads to a MVCC system in which version conflicts have to be resolved at read time by client applications if no syntactic reconciliation is possible.

Consistency of databases on single CouchDB nodes as well as durability is ensured by the way CouchDB operates on database files (see below; cf. [Apa10b]).

### 5.1.7. Storage Implementation

Considering the storage implementation CouchDB applies some methods guaranteeing consistency and durability as well as optimizing performance (cf. [Apa10b]):

- CouchDB never overwrites committed data or associated structures so that a database file is in a consistent state at each time. Hence, the database also does not need a shutdown to terminate correctly but its process can simply be killed.
- Document updates are executed in two stages to provide transactions while maintaining consistency and durability of the database:
  1. The updated documents are serialized to disk synchronously. An exception to this rule is BLOB-data in document which gets written concurrently.
  2. The updated database header is written in two identical and subsequent chunks to disk.

Now, if the system crashes while step 1 is executed, the incompletely written data will be ignored when CouchDB restarts. If the system goes down in step 2 there is chance that one of the two identical database headers is already written; if this is not the case, the inconsistency between database headers and database contents is discovered as CouchDB checks the database headers for consistency when it starts up. Besides these check of database headers there are no further checks or purges needed.

- Read requests are never blocked, never have to wait for a reader or writer and are never interrupted by CouchDB. It is guaranteed for a reading client to see a consistent snapshot of the database from the beginning to the end of a read operation.
- As mentioned above documents of CouchDB databases are indexed in B-Trees. On update operations on documents new revision numbers (sequence-ids) for the updated documents are generated, indexed and the updated index is written in an append-only way to disk.
- To store documents efficiently a document and its metadata is combined in a so called buffer first then written to disk sequentially. Hence, documents can be read by clients and the database (for e.g. indexing purposes, view-calculation) efficiently in one go.
- As mentioned before view indexes are written in an append-only manner just like document indexes.

- CouchDB needs to compact databases from time to time to gain disk space back that is no longer needed. This is due to the append-only database and index-files as well as the document revision history. Compaction can either be scheduled or is implicitly executed when the “wasted” space exceeds a certain threshold. The compaction process clones all database contents that are still needed and copies it to a new database file. During the copying process the old database file remains so that copy and even update operations can still be executed. If the system crashes during copying the old database file is still present and integer. The copying process is considered successful as soon as all data has been transferred to the new database file and all requests have been redirected to it; then CouchDB omits the old database file.

### 5.1.8. Security

#### Access Control

CouchDB implements a simple access control model: a document may have a list of authorized roles (called “readers”) allowed to read it. A user may be associated to zero, one or more of these roles and it is determined which roles he has when he accesses the database. If documents have a reader list associated they can only be read if an accessing user owns the role of one of these readers. Documents with reader lists that are returned by views also get filtered dynamically when the view’s contents are returned to a user (cf. [Apa10b]).

#### Administrator Privileges

There is a special role of an administrator in CouchDB who is allowed to create and administer user accounts as well as to manipulate design documents (e.g. views; cf. [Apa10b]).

#### Update Validation

Documents can be validated dynamically before they get written to disk to ensure security and data validity. For this purpose JavaScript functions can be defined that take a document and the credentials of the logged in user as parameters and can return a negative value if they do not consider the document should be written to disk. This leads CouchDB to an abort of the update request and an error message gets returned to the client that issued the update. As the validation function can be considered quite generic custom security models that go beyond data validation can be implemented this way.

The update validation functions are executed for updates occurring in live-usage of a CouchDB instance as well as for updates due to replication with another node (cf. [Apa10b]).

### 5.1.9. Implementation

CouchDB was originally implemented in C++ but for concurrency reasons later ported to the Erlang OTP (Open Telecommunication Platform; cf. [Eri10]), a functional, concurrent language and platform originally developed with a focus on availability and reliability in the telecommunications sector (by Ericsson). The peculiarities of the Erlang language are lightweight processes, an asynchronous and message-based approach to concurrency, no shared-state threading as well as immutability of data. The CouchDB implementation profits from these characteristics as Erlang was chosen to implement completely lock-free concurrency for read and write requests as well as replication in order to reduce bottlenecks and keep the database working predictably under high load (cf. [Apa10a], [Apa10b]).

To furthermore provide for high availability and allow large scale concurrent access CouchDB uses a shared-noting clustering approach which lets all replica nodes of a database work independently even if they are disconnected (see the subsection on views above and [Apa10b]).

Although the database itself is implemented in Erlang two libraries written in C are used by CouchDB the *IBM Components for Unicode* as well as Mozilla's *Spidermonkey* JavaScript Engine (cf. [Apa10a]).

### 5.1.10. Futher Notes

CouchDB allows to create whole applications in the database and document attachments may consist of HTML, CSS and JavaScript (cf. [Apa10b], [PLL09]).

CouchDB has a mechanism to react on server side events. For that purpose one can register for events on the server and provide JavaScript code that processes these events (cf. [PLL09]).

CouchDB allows to provide JavaScript transformation functions for documents and views that can be used for example to create non-JSON representations of them (e.g. XML documents; cf. [PLL09]).

Some projects have been emerged around CouchDB which extend it with additional functionality, most notably a fulltext search and indexing integration with Apache Lucene (cf. [N<sup>+</sup>10], [Apa09]) and the clustering framework Lounge (cf. [FRLL10]).

## 5.2. MongoDB

### 5.2.1. Overview

MongoDB is a schema-free document database written in C++ and developed in an open-source project which is mainly driven by the company 10gen Inc that also offers professional services around MongoDB. According to its developers the main goal of MongoDB is to close the gap between the fast and highly scalable key-/value-stores and feature-rich traditional RDBMSs relational database management systems. MongoDB's name is derived from the adjective *humongous* (cf. [10g10]). Prominent users of MongoDB include SourceForge.net, foursquare, the New York Times, the URL-shortener bit.ly and the distributed social network DIASPORA\* (cf. [Cop10], [Hey10], [Mah10], [Rid10], [Ric10]).

### 5.2.2. Databases and Collections

MongoDB databases reside on a MongoDB server that can host more than one of such databases which are independent and stored separately by the MongoDB server. A database contains one or more collections consisting of documents. In order to control access to the database a set of security credentials may be defined for databases (cf. [CB10]).

Collections inside databases are referred to by the MongoDB manual as "named groupings of documents" (cf. [CBH10a]). As MongoDB is schema-free the documents within a collection may be heterogeneous although the MongoDB manual suggests to create "one database collection for each of your top level objects" (cf. [MMM<sup>+</sup>10b]). Once the first document is inserted into a database, a collection is created automatically and the inserted document is added to this collection. Such an implicitly created collection gets configured with default parameters by MongoDB—if individual values for options such as auto-indexing,

preallocated disk space or size-limits (cf. [MDM<sup>+</sup>10] and see the subsection on *capped collections* below) are demanded, collections may also be created explicitly by the `createCollection`-command<sup>7</sup>:

```
db.createCollection(<name>, {<configuration parameters>})
```

As an example, the following command creates a collection named `mycoll` with 10,000,000 bytes of preallocated disk space and no automatically generated and indexed document-field `_id`:

```
db.createCollection("mycoll", {size:10000000, autoIndexId:false});
```

MongoDB allows to organize collections in hierarchical namespaces using a dot-notation, e.g. the collections `wiki.articles`, `wiki.categories` and `wiki.authors` residing under the namespace `wiki`. The MongoDB manual notes that “this is simply an organizational mechanism for the user -- the collection namespace is flat from the database’s perspective” (cf. [CBH10a]).

### 5.2.3. Documents

The abstraction and unit of data storable in MongoDB is a document, a data structure comparable to an XML document, a Python dictionary, a Ruby hash or a JSON document. In fact, MongoDB persists documents by a format called BSON which is very similar to JSON but in a binary representation for reasons of efficiency and because of additional datatypes compared to JSON<sup>8</sup>. Nonetheless, “BSON maps readily to and from JSON and also to various data structures in many programming languages” (cf. [CBH<sup>+</sup>10b]).

As an example, a document representing a wiki article<sup>9</sup> may look like the following in JSON notation:

```
{
  title: "MongoDB",
  last_editor: "172.5.123.91",
  last_modified: new Date("9/23/2010"),
  body: "MongoDB is a...",
  categories: ["Database", "NoSQL", "Document Database"] ,
  reviewed: false
}
```

To add such a document into a MongoDB collection the `insert` function is used:

```
db.<collection>.insert( { title: "MongoDB", last_editor: ... } );
```

Once a document is inserted it can be retrieved by matching queries issued by the `find` operation and updated via the `save` operation:

```
db.<collection>.find( { categories: [ "NoSQL", "Document Databases" ] } );
db.<collection>.save( { ... } );
```

<sup>7</sup>All syntax examples in this section are given in JavaScript which is used by the interactive MongoDB shell.

<sup>8</sup>As documents are represented as BSON objects when transmitted to and persisted by a MongoDB server, the MongoDB manual also uses the term *object* for them.

<sup>9</sup>This example contains the same information as the wiki article exemplified in the CouchDB section.

Documents in MongoDB are limited in size by 4 megabytes (cf. [SM10]).

### Datatypes for Document Fields

MongoDB provides the following datatypes for document fields (cf. [CHM10a], [MC09], [MMM<sup>+</sup>10a]):

- scalar types: **boolean**, **integer**, **double**
- character sequence types: **string** (for character sequences encoded in UTF-8), **regular expression**, **code** (JavaScript)
- **object** (for BSON-objects)
- **object id** is a data type for 12 byte long binary values used by MongoDB and all officially supported programming language drivers for a field named `_id` that uniquely identifies documents within collections<sup>10</sup>. The object id datatype is composed of the following components:
  - timestamp in seconds since epoch (first 4 bytes)
  - id of the machine assigning the object id value (next 3 bytes)
  - id of the MongoDB process (next 2 bytes)
  - counter (last 3 bytes)

For documents whose `_id` field has an object id value the timestamp of their creation can be extracted by all officially supported programming drivers.

- **null**
- **array**
- **date**

### References between Documents

MongoDB does not provide a foreign key mechanism so that references between documents have to be resolved by additional queries issued from client applications. References may be set manually by assigning some reference field the value of the `_id` field of the referenced document. In addition, MongoDB provides a more formal way to specify references called DBRef (“Database Reference”). The advantages of using DBRefs are that documents in other collections can be referenced by them and that some programming language drivers dereference them automatically (cf. [MDC<sup>+</sup>10], [MMM<sup>+</sup>10d, Database References]). The syntax for a DBRef is:

```
{ $ref : <collectionname>, $id : <documentid>[, $db : <dbname>] }
```

---

<sup>10</sup>The `_id`-field is not required to be an object id as all other data types are also allowed for object id values—provided that the `_id`-value is unique within a collection. However, MongoDB itself and all officially supported programming language bindings generate and assign an instance of object id if no id value is explicitly defined on document creation.

The MongoDB manual points out that the field `$db` is optional and not supported by many programming language drivers at the moment (cf. [MDC<sup>+</sup>10]).

The MongoDB manual points out that although references between documents are possible there is the alternative to nest documents within documents. The embedding of documents is “much more efficient” according to the MongoDB manual as “[data] is then colocated on disk; client-server turnarounds to the database are eliminated”. Instead when using references, “each reference traversal is a query to the database” which results at least in an addition of latency between the web or application server and the database but typically more as the referenced data is typically not cached in RAM but has to be loaded from disk.

The MongoDB manual gives some guidance when to reference an object and when to embed it as presented in table 5.1 (cf. [MMM<sup>+</sup>10b]).

Criteria for Object References	Criteria for Object Embeddings
<ul style="list-style-type: none"> <li>First-class domain objects (typically residing in a separate collection)</li> <li>Many-to-many reference between objects</li> <li>Objects of this type are often queried in large numbers (request all / the first n objects of a certain type)</li> <li>The object is large (multiple megabytes)</li> </ul>	<ul style="list-style-type: none"> <li>Objects with “line-item detail” characteristic</li> <li>Aggregation relationship between object and host object</li> <li>Object is not referenced by another object (DBRefs for embedded objects are not possible as of MongoDB, version 1.7)</li> <li>Performance to request and operate on the object and its host-object is crucial</li> </ul>

**Table 5.1.:** MongoDB – Referencing vs. Embedding Objects (cf. [MMM<sup>+</sup>10b])

#### 5.2.4. Database Operations

##### Queries

**Selection** Queries in MongoDB are specified as *query objects*, BSON documents containing selection criteria<sup>11</sup>, and passed as a parameter to the `find` operation which is executed on the collection to be queried (cf. [CMH<sup>+</sup>10]):

```
db.<collection>.find( { title: "MongoDB" } );
```

The selection criteria given to the `find` operation can be seen as an equivalent to the `WHERE` clause in SQL statements<sup>12</sup> (cf. [Mer10f]). If the query object is empty, all documents of a collection are returned.

<sup>11</sup>The examples in this section are given in JavaScript as it can be used in the interactive MongoDB shell. Thus, the query objects are represented in JSON and transformed by the JavaScript language driver to BSON when sent to the database.

<sup>12</sup>The MongoDB manual provides an “SQL to Mongo Mapping Chart” which helps to get familiar with the document database approach and MongoDB’s query syntax, cf. [MKB<sup>+</sup>10].

In the selection criteria passed to the find operation a lot of operators are allowed—besides equality comparisons as in the example above. These have the following general form:

```
<fieldname>: {$<operator>: <value>}
<fieldname>: {$<operator>: <value>, $<operator>: value} // AND-junction
```

The following comparison operators are allowed (cf. [MMM<sup>+10c</sup>]):

- Non-equality: \$ne
- Numerical Relations: \$gt, \$gte, \$lt, \$lte (representing >, ≥, <, ≤)
- Modulo with divisor and the modulo compare value in a two-element array, e.g.

```
{ age: {$mod: [2, 1]} } // to retrieve documents with an uneven age
```

- Equality-comparison to (at least) one element of an array: \$in with an array of values as comparison operand, e.g.

```
{ categories: {$in: ["NoSQL", "Document Databases"]} }
```

- Non-equality-comparison to all elements of an array: \$nin with an array of values as comparison operand
- Equality-comparison to all elements of an array: \$all, e.g.

```
{ categories: {$all: ["NoSQL", "Document Databases"]} }
```

- Size of array comparison: \$size, e.g.

```
{ categories: {$size: 2} }
```

- (Non-)Existence of a field: \$exists with the parameter true or false, e.g.

```
{ categories: { $exists: false}, body: { $exists: true} }
```

- Field type: \$type with a numerical value for the BSON data typ (as specified in the MongoDB manual, cf. [MMM<sup>+10c</sup>, Conditional Operators – \$type])

Logical junctions can be specified in query objects as follows:

- Comparison expressions that are separated by comma specify an **AND-junction**.
- **OR-junctions** can be defined by the special \$or operator that is assigned to an array of booleans or expressions, each of which can satisfy the query. As an example, the following query object matches documents that either are reviewed or have exactly two categories assigned:

```
{ $or: [ {reviewed: {$exists: true} }, {categories: {$size: 2} } ] }
```

- To express **NOR-junctions** the `$nor` operator is supported by MongoDB. Like `$or` it is assigned to an array of expressions or boolean values.
- Via the `$not` operator a term can be **negated**, e.g.

```
{ $not: {categories: {$in: ["NoSQL"]}} } // category does not contain "NoSQL"
{ $not: {title: /^Mongo/i} }           // title does not start with "Mongo"
```

The following remarks are made by the MongoDB manual regarding certain field types (cf. [MMM<sup>+10c</sup>]):

- Besides the abovementioned comparison operators it is also possible to do (PCRE<sup>13</sup>) regular expression matching for string fields. If the regular expression only consists a prefix check (`/^prefix/[modifiers]` equivalent to SQL's LIKE '`prefix%`') MongoDB uses indexes that are possibly defined on that field.
- To search for a single value inside an array the array field can simply be assigned to the desired value in the query object, e.g.

```
{ categories : "NoSQL" }
```

It is also possible to specify the position of an element inside an array:

```
{<field>. <index>. <field>: <value>}
```

- If multiple selection criteria are specified for an array field, it has to be distinguished whether documents have to fulfill all or one of these criteria. If the criteria are separated by comma, each criterion can be matched by a different clause. In contrast, if each return document has to satisfy all of that criteria the special `$elemMatch` operator has to be used (see also [MMM<sup>+10c</sup>, Value in an Array – `$elemMatch`]):

```
{ x: {$elemMatch: {a: 1, b: {$gt: 2}} } } // documents with x.a==1 and x.b>2
{ "x.a": 1, "x.b": {$gt: 2} }           // documents with x.a==1 or x.b>2
```

- Fields inside objects are referenced by separating the fieldnames with a dot and placing the whole term in double quotes: "field.subfield". If all fields of matching documents in their precise order are relevant as selection criteria the following syntax has to be used: { `<field>: { <subfield_1>: <value>, <subfield_2>: <value> } }` (cf. [CMH<sup>+10</sup>], [Mer10f], [MHC<sup>+10a</sup>]).

Selection criteria may be specified independent of programming language bindings used applications via the `$where` operator that is assigned to a string containing the selection criteria in JavaScript syntax (cf. [MMC<sup>+10b</sup>]), e.g.

```
db.<collection>.find( { $where : "this.a==1" } );
```

As noted in the code snippet, the object for which a comparison happens is referenced by the `this` keyword. The `$where` operator may be specified in addition to other operators. The MongoDB recommends to prefer standard operators as mentioned above over the `$where` operator with a JavaScript criteria expression. This is because the first kind of statements can directly be used by the query optimizer and therefore evaluate

<sup>13</sup>Perl-compatible Regular Expressions (cf. [Haz10])

faster. To optimize query performance MongoDB first evaluates all other criteria before it interprets the JavaScript expression assigned to `$where` (cf. [MMC<sup>+10b</sup>]).

**Projection** A second parameter can be given to the `find` operation to limit the fields that shall be retrieved—analogous to the projection clause of a SQL statement (i.e. the field specification between the keywords `SELECT` and `FROM`). These fields are again specified by a BSON object consisting of their names assigned to the value 1 (cf. [MS10], [CMB<sup>+10</sup>, Optimizing A Simple Example]):

```
db.<collection>.find( {<selection criteria>} , {<field_1>:1, ...} );
```

The field specification is also applicable to fields of embedded objects using a dot-notation (`field.subfield`) and to ranges within arrays<sup>14</sup>.

If only certain fields shall be excluded from the documents returned by the `find` operation, they are assigned to the value 0:

```
db.<collection>.find( {<selection criteria>} , {<field_1>:0, <field_2>:0, ...} );
```

In both cases only partial documents are returned that cannot be used to update the documents they have been derived from. The MongoDB manual furthermore remarks that the primary key field `_id` is always returned in result documents and cannot be excluded.

**Result Processing** The results of the `find` operation may be processed further by arranging them using the `sort` operation, restricting the number of results by the `limit` operation and ignoring the first  $n$  results by the `skip` operation (cf. [CMH<sup>+10</sup>, Sorting, Skip and Limit], [MMM<sup>+10c</sup>, Cursor Methods], [CMB<sup>+10</sup>, Optimizing A Simple Example]):

```
db.<collection>.find( ... ).sort({<field>:<1|-1>}).limit(<number>).skip(<number>);
```

The `sort` operation—equivalently to the `ORDER BY` clause in SQL—takes a document consisting of pairs of field names and their desired order (1: ascending, -1: descending) as a parameter. It is recommended to specify an index on fields that are often used as a sort criteria or to limit the number of results in order to avoid in memory sorting of large result sets. If no `sort` operation is applied to selection the documents of the result set are returned in their *natural order* which is which is “is not particularly useful because, although the order is often close to insertion order, it is not *guaranteed* to be” for standard tables according to the MongoDB manual. However, for tables in so called *capped collections* (see below), the natural order of a result set can be leveraged and used to efficiently “store and retrieve data in insertion order”. On the contrary, using the `sort` operation for selections on non-capped collections is highly advisable ((cf. [MMD<sup>+10</sup>])).

If the `limit` operation is not used, MongoDB returns all matching documents which are returned to clients in groups of documents called *chunks* (i.e. not as a whole); the number of documents in such a chunk is

<sup>14</sup>The array ranges to be returned are specified by the special `$slice` operator:

```
db.<collection>.find( {...}, {<field>: {$slice: n}});           // first n elements
db.<collection>.find( {...}, {<field>: {$slice: -n}});          // last n elements
db.<collection>.find( {...}, {<field>: {$slice: [m, n]}});      // skip m from beginning, limit n
db.<collection>.find( {...}, {<field>: {$slice: [-m, n]}});    // skip m from end, limit n
```

not fixed and may vary from query to query. The values for the `limit` and `skip` operation can also be given to the `find` operation as its third and fourth argument:

```
db.<collection>.find( {<selection-criteria>} , {<field-inclusion/exclusion>} ,
    <limit-number> , <skip-number> );
```

To retrieve the number of query results fast and efficiently, the `count` operation can be invoked on result sets:

```
db.<collection>.find( ... ).count();
```

To aggregate results by predefined or arbitrary custom functions, the `group` operation is provided which can be seen as an equivalent to SQL's GROUP BY.

To limit the amount of documents to be evaluated for a query it is possible to specify minimum and / or maximum values for fields (cf. [MCB10a]):

```
db.<collection>.find( {...} ).min( {<field>: <value>, <...>: <...>, ....} ) .
    max( {<field>: <value>, <...>: <...>, ....} );
db.<collection>.find( $min: {...}, $max: {...}, $query: {...} );
```

The fields used to specify the upper and lower bounds have to be indexed. The values passed to `min` are inclusive while the values for `max` are exclusive when MongoDB evaluates ranges. For single fields this can be expressed also by the `$gte` and `$lt` operators which is strongly recommended by the MongoDB manual. On the other hand, it is difficult to specify ranges for compound fields—in these cases the minimum/maximum specification is preferred.

Besides these operations on queries or result sets, the query execution and its results can be configured by special operators that e.g. limit the number of documents to be scanned or explain the query (for further details cf. [MMM<sup>+10c</sup>, Special Operators]).

**Cursors** The return value of the `find` operation is a cursor by which the result documents of a query can be processed. The following JavaScript example shows how the result set of a query can be iterated (cf. [MMM<sup>+10c</sup>, Cursor Methods]):

```
var cursor = db.<collection>.find( {...} );
cursor.forEach( function(result) { ... } );
```

To request a single document—typically identified by its default primary key field `_id`—the `findOne` operation should be used instead of `find` (cf. [MMM<sup>+10h</sup>]):

```
db.<collection>.findOne({ _id: 921394});
```

**Query Optimizer** As seen above, MongoDB—unlike many NoSQL databases—supports ad-hoc queries. To answer these queries efficiently query plans are created by a MongoDB component called *query optimizer*. In contrast to similar components in relational databases it is not based on statistics and does not model the costs of multiple possible query plans. Instead, it just executes different query plans in parallel and stops all of them as soon as the first has returned thereby learning which query plan worked the best for a certain query. The MongoDB manual states that this approach “works particularly well given the system is non-relational, which makes the space of possible query plans much smaller (as there are no joins)” (cf. [MC10b]).

## Inserts

Documents are inserted into a MongoDB collection by executing the `insert` operation which simply takes the document to insert as an argument (cf. [CBH<sup>+</sup>10b, Document Orientation]):

```
db.<collection>.insert( <document> );
```

MongoDB appends the primary key field `_id` to the document passed to `insert`.

Alternatively, documents may also be inserted into a collection using the `save` operation (cf. [MMM<sup>+</sup>10d]):

```
db.<collection>.save( <document> );
```

The `save` operation comprises inserts as well as updates: if the `_id` field is not present in the document given to `save` it will be inserted; otherwise it updates the document with that `_id` value in the collection.

## Updates

As discussed in the last paragraph, the `save` operation can be used to update documents. However, there is also an explicit `update` operation with additional parameters and the following syntax (cf. [CHD<sup>+</sup>10]):

```
db.<collection>.update( <criteria>, <new document>, <upsert>, <multi> );
```

The first argument specifies the selection criteria by which the document(s) to update shall be selected; it has to be provided in the same syntax as for the `find` operation. A document by which the matching documents shall be replaced is given as a second argument. If the third argument is set to `true`, the document in the second argument is inserted even if no document of the collection matches the criteria in the first argument (`upsert` is short for *update or insert*). If the last argument is set to `true`, all documents matching the criteria are replaced; otherwise only the first matching document is updated. The last two arguments for `update` are optional and set to `false` by default.

In its strive for good update performance, MongoDB collects statistics and infers which documents tend to grow. For these documents, some padding is reserved to allow them to grow. The rationale behind this approach is that updates can be processed most efficiently if the modified documents do not grow in size (cf. [CHD<sup>+</sup>10, Notes, Object Padding]).

**Modifier Operations** If only certain fields of a document shall be modified, MongoDB provides so called *modifier operations* that can be used instead of a complete document as a second parameter for update (cf. [CHD<sup>+</sup>10, Modifier Operations]). Modifier operations include incrementation of numerical values (\$inc), setting and removing of fields (\$set, \$unset), adding of values to an array (\$push, \$pushall, \$addToSet), removing values from an array (\$pop, \$pull, \$pullAll), replacing of array values (<sup>15</sup>) and renaming of fields (\$rename).

As an example, the following command increments by 1 the revision number of the first document whose title is "MongoDB":

```
db.<collection>.update( { title: "MongoDB" }, { $inc: { revision: 1 } } );
```

The advantage of modifier operations in comparison to the replacement by whole documents is that they can be efficiently executed as the "latency involved in querying and returning the object" is avoided. In addition, modifier operations feature "operation atomicity and very little network data transfer" according to the MongoDB manual (cf. [CHD<sup>+</sup>10, Modifier Operations]).

If modifier operations are used with upsert set to true, new documents containing the fields defined in modifier operation expressions will be inserted.

**Atomic Updates** By default, update operations are non-blocking since MongoDB version 1.5.2. This is especially relevant as updates for multiple documents that are issued by the same command may be interleaved by other read and also write operations which can lead to undesired results. Therefore, if atomicity is required the \$atomic flag has to be set to true and added to the selection criteria in update and also remove operations (cf. [CHD<sup>+</sup>10, Notes, Blocking]).

That said, the following operations and approaches already provide or achieve atomicity without using the \$atomic flag explicitly:

- Updates using modifier operations are atomic by default (cf. [CHD<sup>+</sup>10, Modifier Operations], [MCS<sup>+</sup>10, Modifier operations]).
- Another way of executing updates atomically is called *Update if Current* by the MongoDB manual and comparable to the *Compare and Swap* strategy employed in operating systems (cf. [MCS<sup>+</sup>10, Update if Current]). It means to select an object (here: document), modify it locally and request an update for it only if the object has not changed in the datastore since it has been selected.

As an example for the *Update if Current* the following operations increment the revision number of a wiki article:

```
wikiArticle = db.wiki.findOne( { title: "MongoDB" } );           // select document
oldRevision = wikiArticle.revision; // save old revision number
wikiArticle.revision++;          // increment revision number in document
db.wiki.update( { title: "MongoDB",
                  revision = oldRevision}, wikiArticle ); // Update if Current
```

The last statement will only update the document if the revision number is the same as at the time the first statement was issued. If it has changed while the first tree lines have been executed, the selection criteria in first argument of the fourth line will no longer match any document<sup>16</sup>.

<sup>15</sup>\$ is a positional operator representing an array element matched by a selection.

<sup>16</sup>The MongoDB furthermore argues that the selection criteria from the first selection (in the example title: "MongoDB") might have changed as well before the update statement is processed. This can be avoided by using e.g. an object-id field

- Single documents can be updated atomically by the special operation `findAndModify` which selects, updates and returns a document (cf. [MSB<sup>+</sup>10]). Its syntax is as follows:

```
db.<collection>.findAndModify( query: {...},
    sort: {...},
    remove: <true|false>,
    update: {...},
    new: <true|false>,
    fields: {...},
    upsert: <true|false>);
```

The `query` argument is used to select documents of which the first one is modified by the operation. To change the order of the result set, it can be sorted by fields specified in the `sort` argument. If the returned document shall be removed before it is returned the `remove` argument has to be set to `true`. In the `update` argument it is specified how the selected document shall be modified—either by giving a whole document replacing it or by using modifier expressions. The `new` argument has to be set to `true` if the modified document shall be returned instead of the original. To restrict the number of fields returned as a result document, the `fields` argument has to be used which contains field names assigned to 1 or 0 to select or deselect them. The last argument `upsert` specifies if a document shall be created if the result set of the `query` is empty.

The `findAndModify` operation is also applicable in sharded setups. If the collection affected by the modification is sharded, the `query` must contain the shard key.

## Deletes

To delete documents from a collection, the `remove` operation has to be used which takes a document containing selection criteria as a parameter (cf. [CNM<sup>+</sup>10]):

```
db.<collection>.remove( { <criteria> } );
```

Selection criteria has to be specified in the same manner as for the `find` operation. If it is empty, all documents of the collection are removed. The `remove` operation does not eliminate references to the documents it deletes<sup>17</sup>. To remove a single document from a collection it is best to pass its `_id` field to the `remove` operation as using the whole document is inefficient according to the MongoDB manual.

Since MongoDB version 1.3 concurrent operations are allowed by default while a `remove` operation executes. This may result in documents being not removed although matching the criteria passed to `remove` if a concurrent update operation grows a document. If such a behaviour is not desired, the `remove` operation may be executed atomically, not allowing any concurrent operations, via the following syntax:

```
db.<collection>.remove( { <criteria> , $atomic: true} );
```

---

(by default `_id`) or the entire object as selection criteria, or by using either the modifier expression `$set` or the positional operator `$` to modify the desired fields of the document.

<sup>17</sup>However, such “orphaned” references in the database can be detected easily as they return `null` when being evaluated.

## Transaction Properties

Regarding transactions, MongoDB only provides atomicity for update and delete operations by setting the `$atomic` flag to `true` and adding it to the selection criteria. Transactional locking and complex transactions are not supported for the following reasons discussed in the MongoDB manual (cf. [MCS<sup>+</sup>10]):

- Performance as “in sharded environments, distributed locks could be expensive and slow”. MongoDB is intended to be “lightweight and fast”.
- Avoidance of deadlocks
- Keeping database operations simple and predictable
- MongoDB shall “work well for realtime problems”. Therefore, locking of large amounts of data which “might stop some small light queries for an extended period of time [...] would make it even harder” to achieve this goal.

## Server-Side Code Execution

MongoDB—like relational databases with their stored procedures—allows to execute code locally on database nodes. Server-side code execution comprises three different approaches in MongoDB:

1. Execution arbitrary code on a *single* database node via the `eval` operation (cf. [MMC<sup>+</sup>10b])
2. Aggregation via the operations `count`, `group` and `distinct` (cf. [MMM<sup>+</sup>10g])
3. MapReduce-fashioned code execution on multiple database nodes (cf. [HMC<sup>+</sup>10])

Each of these approaches shall be briefly discussed in the next paragraphs.

**The eval-operation** To execute arbitrary blocks of code locally on a database server, the code has to be enclosed by a anonymous JavaScript function and passed to MongoDB’s generic `eval` operation (cf. [MMC<sup>+</sup>10b]):

```
db.eval( function(<formal parameters>) { ... }, <actual parameters>);
```

If function passed to `eval` has formal parameters, these have to be bound to actual parameters by passing them as arguments to `eval`. Although `eval` may be helpful to process large amounts of data locally on a server, it has to be used carefully a write lock is held during execution. A another important downside the `eval` operation is not supported in sharded setups, so that the MapReduce-approach has to be used in these scenarios as described below.

Functions containing arbitrary code may also be saved under a key (with the fieldname `_id`) on the database server in a special collection named `system.js` and later be invoked via their key, e.g.

```
system.js.save( {_id: "myfunction", value: function(...) {...}} );
db.<collection>.eval(myfunction, ...); // invoke the formerly saved function
```

**Aggregation** To accomplish the aggregation of query results MongoDB provides the `count`, `distinct` and `group` operation that may be invoked via programming language libraries but executed on the database servers (cf. [MMM<sup>+10g</sup>]).

The `count` operation returning the number of documents matching a query is invoked on a collection and takes selection criteria that is specified in the same way as for the `find` operation:

```
db.<collection>.count( <criteria> );
```

If `count` is invoked with empty criteria, the number of documents in the collection is returned. If selection criteria is used, the document fields used in it should be indexed to accelerate the execution of `count` (and likewise `find`).

To retrieve distinct values for certain document fields, the `distinct` operation is provided. It is invoked by passing a document in the following syntax to the generic `runCommand` operation or—if provided by a programming language driver—using the `distinct` operation directly:

```
db.runCommand( {distinct: <collection>, key: <field> [, query: <criteria> ]} );
db.<collection>.distinct( <document field> [, { <criteria> } ]);
```

The query part of the document is optional but may be useful to consider only the distinct field values of relevant documents of a collection. Document fields specified as key may also be nested fields using a dot notation (`field.subfield`).

As an equivalent to the GROUP BY clause in SQL MongoDB provides the aggregation operation `group`. The `group` operation returns an array of grouped items and is parameterized by a document consisting of the following fields:

```
db.<collection>.group( {
    key: { <document field to group by> },
    reduce: function(doc, aggrcounter) { <aggregation logic> },
    initial: { <initialization of aggregation variable(s)> },
    keyf: { <for grouping by key-objects or nested fields> },
    cond: { <selection criteria> },
    finalize: function(value){ <return value calculation> }
});
```

The fields of the document passed to the `group` operation are explained in table 5.2.

Field	Description	Mandatory?
<code>key</code>	The document fields by which the grouping shall happen.	Yes (if <code>keyf</code> is not specified)
<code>reduce</code>	A function that “aggregates (reduces) the objects iterated. Typical operations of a reduce function include summing and counting. <code>reduce</code> takes two arguments: the current document being iterated over and the aggregation counter object.”	Yes
<code>initial</code>	Initial values for the aggregation variable(s).	No

Field	Description	Mandatory?
keyf	If the grouping key is a calculated document/BSON object a function returning the key object has to be specified here; in this case key has to be omitted and keyf has to be used instead.	Yes (if key is not specified)
cond	Selection criteria for documents that shall be considered by the group operation. The criteria syntax is the same as for the find and count operation. If no criteria is specified, all documents of the collection are processed by group.	No
finalize	"An optional function to be run on each item in the result set just before the item is returned. Can either modify the item (e.g., add an average field given a count and a total) or return a replacement object (returning a new object with just _id and average fields)."	No

**Table 5.2.:** MongoDB - Parameters of the group operation (cf. [MMM<sup>+10g</sup>, Group])

The MongoDB manual provides an example how to use the group operation (cf. [MMM<sup>+10g</sup>, Group]):

```
db.<collection>.group(
    {key: { a:true, b:true },
     cond: { active:1 },
     reduce: function(doc,acc) { acc.csum += obj.c; },
     initial: { csum: 0 }
   }
);
```

A corresponding SQL statement looks like this:

```
select a, b, sum(c) csum from <collection> where active=1 group by a, b;
```

The SQL function `sum` is reconstructed in the group function example by the fields `initial` and `reduce` of the parameter document given to `group`. The value of `initial` defines the variable `csum` and initializes it with 0; this code-block is executed once before the documents are selected and processed. For each document matching the criteria defined via `cond` the function assigned to the `reduce` key is executed. This function adds the attribute value `c` of the selected document (`doc`) to the field `csum` of the accumulator variable (`acc`) which is given to the function as a second argument (cf. [MMM<sup>+10g</sup>]).

A limitation of the group operation is that it cannot be used in sharded setups. In these cases the MapReduce approach discussed in the next paragraph has to be taken. MapReduce also allows to implement custom aggregation operations in addition to the predefined `count`, `distinct` and `group` operations discussed above.

**MapReduce** A third approach to server-side code execution especially suited for batch manipulation and aggregation of data in sharded setups is MapReduce (cf. [HMC<sup>+10</sup>]). MongoDB's MapReduce implementation is similar to the concepts described in Google's MapReduce paper (cf. [DG04]) and its open-source implementation Hadoop: there are two phases—map and the reduce—in which code written in JavaScript

is executed on the database servers and results in a temporary or permanent collection containing the outcome. The MongoDB shell—as well as most programming language drivers—provides a syntax to launch such a MapReduce-fashioned data processing:

```
db.<collection>.mapreduce( map: <map-function>,
    reduce: <reduce-function>,
    query: <selection criteria>,
    sort: <sorting specification>,
    limit: <number of objects to process>,
    out: <output-collection name>,
    outType: <"normal"|"merge"|"reduce">,
    keepTemp: <true|false>,
    finalize: <finalize function>,
    scope: <object with variables to put in global namespace>,
    verbose: <true|false>
) ;
```

The MongoDB manual makes the following remarks concerning the arguments of `mapreduce`:

- The mandatory fields of the `mapreduce` operation are `map` and `reduce` which have to be assigned to JavaScript functions with a signature discussed below.
- If `out` is specified or `keepTemp` is set to `true` is specified the outcome of the MapReduce data processing is saved in a permanent collection; otherwise the collection containing the results is temporary and removed if the client disconnects or explicitly drops it.
- The `outType` parameter controls how the collection specified by `out` is populated: if set to "normal" the collection will be cleared before the new results are written into it; if set to "merge" old and new results are merged so that values from the latest MapReduce job overwrite those of former jobs for keys present in the old and new results (i.e. keys not present in the latest result remain untouched); if set to "reduce" the `reduce` operation of the latest MapReduce job is executed for keys that have old and new values.
- The `finalize` function is applied to all results when the `map` and `reduce` functions have been executed. In contrast to `reduce` it is only invoked once for a given key and value while `reduce` is invoked iteratively and may be called multiple times for a given key (see constraint for the `reduce` function below).
- The `scope` argument is assigned to a javascript object that is put into the global namespace. Therefore, its variables are accessible in the functions `map`, `reduce` and `finalize`.
- If `verbose` is set to `true` statistics on the execution time of the MapReduce job are provided.

The functions `map`, `reduce` and `finalize` are specified as follows:

- In the `map` function a single document whose reference is accessible via the keyword `this` is considered. `map` is required to call the function `emit(key, value)` at least once in its implementation to add a key and a single value to the intermediate results which are processed by the `reduce` function in the second phase of the MapReduce job execution. The `map` function has the following signature:

```
function map(void) --> void
```

- The `reduce` is responsible to calculate a single value for a given key out of an array of values emitted by the `map` function. Therefore, its signature is as follows:

```
function reduce(key, value_array) --> value
```

The MongoDB manual notes that “[the] MapReduce engine may invoke reduce functions iteratively; thus, these functions must be idempotent”. This constraint for `reduce` can be formalized by the following predicate:

$$\forall k, \text{vals} : \text{reduce}(k, [\text{reduce}(k, \text{vals})]) \equiv \text{reduce}(k, \text{vals})$$

The value returned by `reduce` is not allowed to be an array as of MongoDB version 1.6. The MongoDB furthermore suggests, that the values emitted by `map` and `reduce` “should be the same format to make iterative reduce possible” and avoid “weird bugs that are hard to debug”.

- The optional `finalize` function is executed after the reduce-phase and processes a key and a value:

```
function finalize(key, value) --> final_value
```

In contrast to `reduce`, the `finalize` function is only called once per key.

The MongoDB manual points out that—in contrast to CouchDB—MapReduce is neither used for basic queries nor indexing in MongoDB; it is only be used if explicitly invoked by the `mapreduce` operation. As mentioned in the sections before, the MapReduce approach is especially useful or even has to be used (e.g. for server-side aggregation) in sharded setups. Only in these scenarios MapReduce jobs can be executed in parallel as “jobs on a single mongod process are single threaded [...] due to a design limitation in current JavaScript engines” (cf. [MMM<sup>+10g</sup>, Map/Reduce], [HMC<sup>+10</sup>]).

## Commands for Maintenance and Administration

MongoDB features commands which are sent to the database to gain information about its operational status or to perform maintenance or administration tasks. A command is executed via the special namespace `$cmd` and has the following general syntax:

```
db.$cmd.findOne( { <commandname>: <value> [, options] } );
```

When such a command is sent to a MongoDB server it will answer with a single document containing the command results (cf. [MMM<sup>+10e</sup>]).

Examples of MongoDB commands include cloning of databases, flushing of pending writes to datafiles on disk, locking and unlocking of datafiles to block and unblock write operations (for backup purposes), creating and dropping of indexes, obtaining information about recent errors, viewing and terminating running operations, validating collections and retrieving statistics as well as database system information (cf. [CHM<sup>+10b</sup>]).

### 5.2.5. Indexing

Like relational database systems, MongoDB allows to specify indexes on document fields of a collection. The information gathered about these fields is stored in B-Trees and utilized by the query optimizing component to “to quickly sort through and order the documents in a collection” thereby enhancing read performance.

As in relational databases, indexes accelerate select as well as update operations as documents can be found faster by the index than via a full collection scan; on the other side indexes add overhead to insert and delete operations as the B-tree index has to be updated in addition to the collection itself. Therefore the MongoDB manual concludes that “indexes are best for collections where the number of reads is much greater than the number of writes. For collections which are write-intensive, indexes, in some cases, may be counterproductive” (cf. [MMH<sup>+</sup>10]). As a general rule, the MongoDB manual suggests to index “fields upon which keys are looked up” as well as sort fields; fields that should get indexed can also be determined using the profiling facility provided by MongoDB or by retrieving an execution plan by invoking the `explain` operation on a query<sup>18</sup> (cf. [MMM<sup>+</sup>10b, Index Selection], [CMB<sup>+</sup>10, Optimizing A Simple Example]).

Indexes are created by an operation named `ensureIndex`:

```
db.<collection>.ensureIndex({<field1>:<sorting>, <field2>:<sorting>, ...});
```

Indexes may be defined on fields of any type—even on nested documents<sup>19</sup>. If only certain fields of nested documents shall get indexed, they can be specified using a dot-notation (i.e. `fieldname.subfieldname`). If an index is specified for an array field, each element of the array gets indexed by MongoDB. The sorting flag may be set to 1 for an ascending and -1 for a descending order. The sort order is relevant for sorts and range queries on compound indexes (i.e. indexes on multiple fields) according to the MongoDB manual (cf. [MMH<sup>+</sup>10, Compound Keys Indexes], [MMM<sup>+</sup>10f]). To query an array or object field for multiple values the `$all` operator has to be used:

```
db.<collection>.find( <field>: { $all: [ <value_1>, <value_2> ... ] } );
```

By default, MongoDB creates an index on the `_id` field as it uniquely identifies documents in a collection and is expected to be chosen as a selection criterion often; however, when creating a collection manually, the automatic indexing of this field can be neglected. Another important option regarding indexes is background index building which has to be explicitly chosen as by default the index building blocks all other database operations. When using background indexing, the index is built incrementally and which is slower than indexing as a foreground job. Indexes built in background are taken into account for queries only when the background indexing job has finished. The MongoDB manual mentions two limitations with regards to (background) indexing: first, only one index per collection can be built a time; second, certain administrative operations (like `repairDatabase`) are disabled while the index is built (cf. [MH10b]).

MongoDB also supports unique indexes specifying that no two documents of a collections have the same value for such a unique index field. To specify unique indexes, another parameter is passed to the aforementioned `ensureIndex` operation:

```
db.<collection>.ensureIndex({<field1>:<sorting>, <field2>:<sorting>, ...}, {unique: true});
```

<sup>18</sup>For example, the execution plan of a query can be retrieved by:

```
db.<collection>.find( ... ).explain();
```

<sup>19</sup>As discussed in the MongoDB manual the indexing of nested documents may be helpful if the set of fields to index is not known in advance and shall be easily extensible. In this case, a field containing a nested document can be defined and indexed as a whole. As there is no strict schema for documents, fields to be indexed are put into the nested document and thereby get indexed. The downside of this approach is that it limits index parameters like the sort order and the uniqueness as the index is defined on the whole nested document and therefore all contained fields of this documents get indexed in the same manner.

The MongoDB manual warns that index fields have to be present in all documents of a collection. As MongoDB automatically inserts all missing indexed fields with null values, no two documents can be inserted missing the same unique indexed key.

MongoDB can be forced to use a certain index, by the `hint` operation which takes the names of index fields assigned to 1 as a parameter:

```
<db.<collection>.find( ... ).hint( {<indexfield_1>:1, <indexfield_2>:1, ...} );
```

The MongoDB points out that the query optimizer component in MongoDB usually leverages indexes but may fail in doing so e.g. if a query involves indexed and non indexed fields. It is also possible to force the query optimizer to ignore all indexes and do a full collection scan by passing `$natural:1` as a parameter to the `hint` operation (cf. [CMB<sup>+10</sup>, Hint]).

To view all indexes specified for a certain collection, the `getIndexes` operation is to be called:

```
db.<collection>.getIndexes();
```

To remove all indexes or a specific index of a collection the following operations have to be used:

```
db.<collection>.dropIndexes(); // drops all indexes
db.<collection>.dropIndex({<fieldname>:<sortorder>, <fieldname>:<sortorder>, ...});
```

Indexes can also be rebuilt by invoking the `reIndex` operation:

```
db.<collection>.reIndex();
```

The MongoDB manual considers this manually executed index rebuilding only “unnecessary” in general but useful for administrators in situations when the “size of your collection has changed dramatically or the disk space used by indexes seems oddly large”.

### 5.2.6. Programming Language Libraries

As of December 2010 the MongoDB project provides client libraries for the following programming languages: C, C#, C++, Haskell, Java, JavaScript, Perl, PHP, Python, and Ruby. Besides these officially delivered libraries, there are a lot of additional, community ones for further languages, such as Clojure, F#, Groovy, Lua, Objective C, Scala, Schema, and Smalltalk. These libraries—called *drivers* by the MongoDB project—provide data conversion between the programming language’s datatypes and the BSON format as well an API to interact with MongoDB servers (cf. [HCS<sup>+10</sup>]).

### 5.2.7. Distribution Aspects

#### Concurrency and Locking

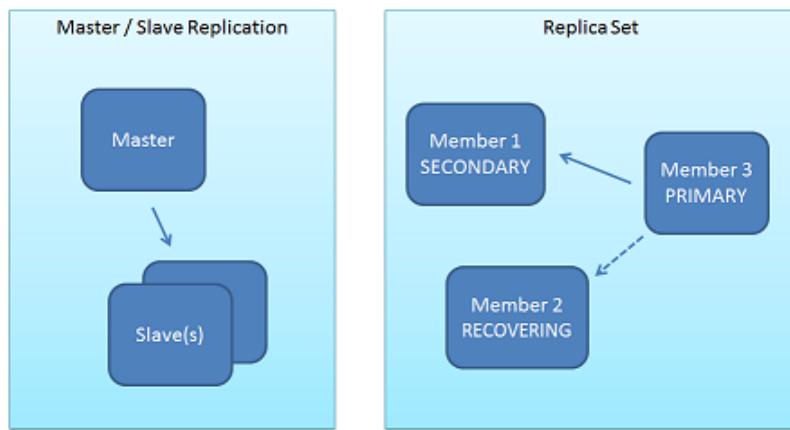
The MongoDB architecture is described to be “concurrency friendly” although “some work with respect to granular locking and latching is not yet done. This means that some operations can block others.”

MongoDB uses read/write locks for many operations with “[any] number of concurrent read operations allowed, but typically only one write operation”. The acquisition of write locks is greedy and, if pending, prevents subsequent read lock acquisitions (cf. [MHH10]).

## Replication

For redundancy and the failover of a MongoDB cluster in the presence of unavailable database nodes, MongoDB provides asynchronous replication. In such a setup only one database node is in charge of write operations at any given time (called *primary server/node*). Read operations may go to this same server for strong consistency semantics or to any of its replica peers if eventual consistency is sufficient (cf. [MMG<sup>+</sup>10]).

The MongoDB documentation discusses two approaches to replication—Master-Slave Replication and Replica Sets—that are depicted in figure 5.1.



**Figure 5.1.:** MongoDB – Replication Approaches (taken from [MMG<sup>+</sup>10])

**Master-Slave** is a setup consisting of two servers out of one which takes the role of a master handling write requests and replicating those operations to the second server, the slave (cf. [MJS<sup>+</sup>10]).

Starting up a master-slave setup is done via starting up a MongoDB process as a master and a second process in slave mode with a further parameter identifying its master:

```
mongod --master <further parameters>
mongod --slave --source <master hostname>[:<port>] <further parameters>
```

An important note about slave servers is that replication will stop if they get too far behind the write operations from their master or if they are restarted and the update operations during their downtime can no longer be retrieved completely from the master. In these cases “replication will terminate and operator intervention is required by default if replication is to be restarted” (cf. [MJS<sup>+</sup>10], also [BMH10]). As an alternative, slaves can start with the `-autoresync` parameter which causes them to restart the replication if they become out of sync.

A permanent failover from a master to a slave or an inversion of their roles is only possible with a shutdown and restart of both servers (cf. [MJS<sup>+</sup>10, Administrative Tasks]).

**Replica Sets** are groups of MongoDB nodes “that work together to provide automated failover” (cf. [BCM<sup>+10</sup>]). They are described as an “an elaboration on the existing master/slave replication, adding automatic failover and automatic recovery of member nodes” (cf. [MCD<sup>+10</sup>]). To set up a replica set, the following steps are required (cf. [BCM<sup>+10</sup>]):

1. Start up a number of n MongoDB nodes with the `-replSet` parameter:

```
mongod --replSet <name> --port <port> --dbpath <data directory>
```

2. Initialize the replica set by connecting to one of the started `mongod` processes and passing a configuration document<sup>20</sup> for the replica set to the `rs.initiate` operation via the MongoDB shell:

```
mongo <host>:<port>
> config = {_id: '<name>',
  members: [
    {_id: 0: host: '<host>:<port>'},
    {_id: 1: host: '<host>:<port>'},
    {_id: 2: host: '<host>:<port>'}
  ]}
> rs.initiate(config);
```

The MongoDB node receiving the `rs.initiate` command propagates the configuration object to its peers and a primary node is elected among the set members. The status of the replica set can be retrieved via the `rs.status` operation or via a administrative web interface if one of the replica set nodes has been started with the `-rest` option (cf. [Mer10c]).

To add further nodes to the replica set they are started up with the `-replSet` parameter and the name of the replica set the new node shall become part of (cf. [DBC10]):

```
mongo --replSet <name> <further parameters>
```

Nodes added to a replica set either have to have an empty data directory or a recent copy of the data directory from another node (to accelerate synchronization). This is due to the fact that MongoDB does not feature multi-version storage or any conflict resolution support (see the section on limitations below).

If a replica set is up and running correctly write operations to the primary node get replicated to the secondary nodes by propagating operations to them which are applied to the locally stored data (cf. [Mer10e]). This approach is similar to the *operation transfer model* described in section 3.1.3 in the context of state propagation via vector clocks. In MongoDB operations received by peer nodes of a replica set are written into a capped collection that has a limited size which should not be chosen to low. However, no data gets lost if operations are omitted as new operations arrive faster than the node can apply them to its local data store; it is still possible to fully resynchronize such a node [Mer10d]. The write operations propagated among the replica set nodes are identified by the id of the server receiving them (the primary server at that time) and a monolithically increasing ordinal number (cf. [MHD<sup>+10</sup>]). The MongoDB points out that an “initial replication is essential for failover; **the system won’t fail over to a new master until an initial sync between nodes is complete**”.

---

<sup>20</sup>In the example only the mandatory configuration parameters are set – a full reference of configuration options—such as priorities for servers to get priority, the number votes in consensus protocols or different roles of servers—can be found in the MongoDB documentation ([MCB<sup>+10b</sup>, The Replica Set Config Object]).

Replica Sets can consist of up to seven servers which can take the role of a standard server (stores data, can become primary server), passive server (stores data, cannot become primary server) or arbiter (does not store data but participates in the consensus process to elect new primary servers; cf. [MHD<sup>+</sup>10]). Replica sets take into account data center configuration which—according to the MongoDB documentation—is implemented rudimentary as of version 1.6 but already features options like primary and disaster recovery sites as well as local reads (cf. [Mer10b]). Replica sets have the following consistency and durability implications (cf. [MD10]):

- Write operations are committed only if they have been replicated to a quorum (majority) of database nodes.
- While write operations are propagated to the replica nodes they are already visible at the primary node so that a newer version not committed at all nodes can already be read from the master. This *read uncommitted* semantic<sup>21</sup> is employed as—in theory—a higher performance and availability can be achieved this way.
- In case of a failover due to a failure of the primary database node all data that has not been replicated to the other nodes is dropped. If the primary node gets available again, its data is available as a backup but not recovered automatically as manual intervention is required to merge its database contents with those of the replica nodes.
- During a failover, i. e. the window of time in which a primary node is declared down by the other nodes and a new primary node is elected, write and strongly consistent read operations are not possible. However, eventually consistent read operations can still be executed (cf. [MH10a, How long does failover take?]).
- To detect network partitions each node monitors the other nodes via heartbeats. If a primary server does not receive heartbeats from at least half of the nodes (including itself), it leaves its primary state and does not handle write operations any longer. “Otherwise in a network partition, a server might think it is still primary when it is not” (cf. [MHD<sup>+</sup>10, Design, Heartbeat Monitoring]).
- If a new primary server is elected some serious implications regarding consistency may result: as the new primary is assumed to have the latest state all newer data on the new secondary nodes will be discarded. This situation is detected by the secondary nodes via their operation logs. Operations that have already been executed are rolled back until their state corresponds to the state of the new primary. The data changed during this rollback is saved to a file and cannot be applied automatically to the replica set. Situations in which secondary nodes may have a later state than a newly elected primary can result from operations already committed at a secondary node but not at the new primary (which was secondary at the time they were issued; cf. [MHD<sup>+</sup>10, Design, Assumption of Primary; Design, Resync (Connecting to a New Primary)]).

In MongoDB version 1.6 authentication is not available for replica sets. A further limitation is that MapReduce jobs may only run on the primary node as they create new collections (cf. [MC10a]).

**Excluding Data from Replication** can be achieved by putting the data that shall not become subject to replication into the special database `local`. Besides user data MongoDB also stores administrative data such as replication configuration documents in this database (cf. [Mer10a]).

---

<sup>21</sup>This read semantic is comparable to the Read Your Own Writes (RYOW) and Session Consistency model discussed in 3.1.2 if the client issuing write operations at the primary node also reads from this node subsequently.

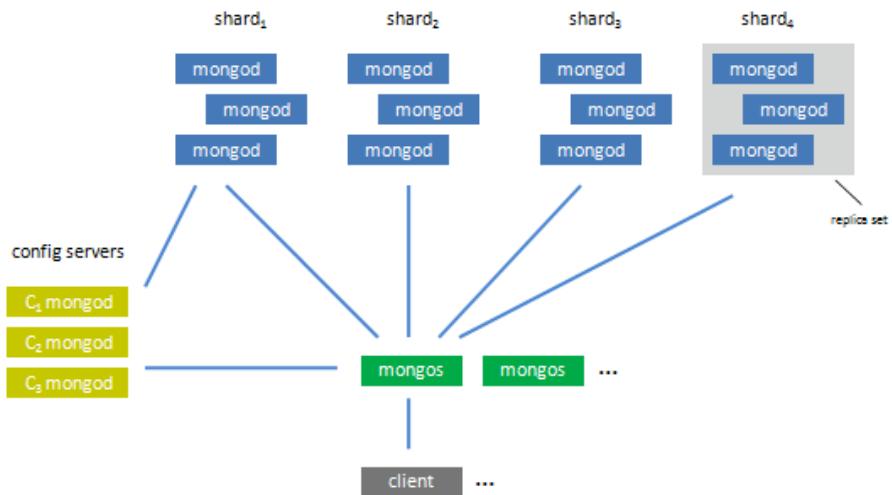
## Sharding

Since Version 1.6 MongoDB supports horizontal scaling via an automatic sharding architecture to distribute data across “thousands of nodes” with automatic balancing of load and data as well as automatic failover (cf. [MHC<sup>+</sup>10b], [MDH<sup>+</sup>10]).

Sharding is understood to be “the partitioning of data among multiple machines in an order-preserving manner” by the MongoDB documentation. The MongoDB mentions Yahoo!’s PNUTS<sup>22</sup> ([CRS<sup>+</sup>08]) and Google’s Bigtable ([CDG<sup>+</sup>06]) as important influences for the partitioning scheme implemented in MongoDB (cf. [MDH<sup>+</sup>10, MongoDB’s Auto-Sharding, Scaling Model]). Sharding in MongoDB “occurs on a per-collection basis, not on the database as a whole”. In a setup configured for sharding, MongoDB automatically detects which collections grow much faster than the average so that they become subject to sharding while the other collections may still reside on single nodes. MongoDB also detects imbalances in the load different shards have to handle and can automatically rebalance data to reduce disproportionate load distribution (cf. [MDH<sup>+</sup>10, MongoDB’s Auto-Sharding]).

Sharding in MongoDB is built on top of *replica sets* which have been discussed above. This means that for each partition of data a number of nodes forming a replica set is in charge: at any time one of these servers is primary and handles all write requests which then get propagated to the secondary servers to replicate changes and keep the set in-sync; if the primary node fails the remaining nodes elect a new primary via consensus so that the operation of the replica set is continued. This way, automated failover is provided for each shard (cf. [MDH<sup>+</sup>10, MongoDB’s Auto-Sharding, Balancing and Failover]).

**Sharding Architecture** A MongoDB shard cluster is built up by three components as depicted in figure 5.2 (cf. [MDH<sup>+</sup>10, Architectural Overview]):



**Figure 5.2.:** MongoDB – Sharding Components (taken from [MDH<sup>+</sup>10, Architectural Overview])

**Shards** consisting of servers that run `mongod` processes and store data. To ensure availability and automated failover in production systems, each shard typically consists of multiple servers comprising a replica set.

<sup>22</sup>Platform for Nimble Universal Table Storage

**Config Servers** “store the cluster’s metadata, which includes basic information on each shard server and the chunks contained therein”. Chunks are contiguous ranges of data from collections which are ordered by the sharding key and stored on the shards (sharding keys and chunks are discussed in more detail below). Each config server stores the complete chunk metadata and is able to derive on which shards a particular document resides. The data on config servers is kept consistent via a two-phase commit protocol and a special replication scheme (i.e. Master Slave Replication or Replica Sets are not used for this purpose). The metadata stored on config servers becomes read only if any of these servers is unreachable; in this state, the database is still writable, but it becomes impossible to redistribute data among the shards (cf. also [MHB<sup>+</sup>10b]).

**Routing Services** are server-side `mongos`-processes executing read and write requests on behalf of client applications. They are in charge of looking up the shards to be read from or written to via the config servers, connecting to these shards, executing the requested operation and returning the results to client applications, also merging results of the request execution on different shards. This makes a distributed MongoDB setup look like a single server towards client applications which do not have to be aware of sharding; even the programming language libraries do not have to take sharding into account. The `mongos` processes do not persist any state and do not coordinate with one another within a sharded setup. In addition, they are described to be lightweight by the MongoDB documentation, so that it is uncritical to run any number of `mongos` processes. The `mongos` processes may be executed on any type of server—shards, config servers as well as application servers.

Shards, config and routing servers can be organized in different ways on physical or virtual servers as described by the MongoDB documentation (cf. [MDH<sup>+</sup>10, Server Layout]). A minimal sharding setup requires at least two shards, one config and one routing server (cf. [MHB<sup>+</sup>10a, Introduction]).

**Sharding Scheme** As discussed above, partitioning in MongoDB occurs based on collections. For each collection, a number of fields can be configured by which the documents shall get partitioned. If MongoDB then detects imbalances in load and data size for a collection it partitions its documents by the configured keys while preserving their order. If e.g. the document field `name` is configured as a shard key for the collection `users`, the metadata about this collection held on the config servers may look like figure 5.3.

collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	shard <sub>2</sub>
users	{ name : 'Nessman' }	{ name : 'Ogden' }	shard <sub>4</sub>
...			

**Figure 5.3.:** MongoDB – Sharding Metadata Example (taken from [MDH<sup>+</sup>10, Architectural Overview, Shards, Shard Keys])

**Sharding Persistence** Documents of a collection are stored in so called *chunks*, contiguous ranges of data. They are identified by the triple (`collection`, `minkey`, `maxkey`) with `minkey` and `maxkey` as the minimum and maximum value for the document field(s) chosen as sharding keys. If chunks grow to a configured size they are split into two new chunks (cf. [MDH<sup>+</sup>10, Architectural Overview]). As of version 1.6, chunk “splits happen as a side effect of inserting (and are transparent)” towards client applications. Chunks can get migrated in the background among the shard servers to evenly distribute data as well as load; this is done by a “sub-system called Balancer, which constantly monitors shards loads and [...] moves chunks around if it finds an imbalance” (cf. [MSL10]).

The MongoDB manual points out that it is important to choose sharding keys that are “*granular* enough to ensure an even distribution of data”. This means that there should be a high number of distinct values in the document fields chosen as sharding keys, so that it is always possible to split chunks. A counterexample might be the `name` field mentioned above: if one chunk only contains documents with popular names like “Smith” it cannot be split into smaller chunks. This is due to the fact that a chunk is identified by the abovementioned triple of (`collection`, `minkey`, `maxkey`): if `minkey` and `maxkey` already have the same value and sharding shall preserve the order of documents, such a chunk cannot be split up. To avoid such issues it is also possible to specify compound sharding keys (such as `lastname`, `firstname`; cf. [MDH<sup>+</sup>10, Architectural Overview]).

**Setting up a Sharded MongoDB Cluster** is done via the following steps (cf. [MHB<sup>+</sup>10a, Configuring the Shard Cluster], [MCH<sup>+</sup>10]):

1. Start up a number shards via the `mongod` command with the `-shardsrv` flag:

```
./mongod --shardsrv <further parameters>
```

2. Start up a number of config servers via the `mongod` command with the `-configsvr` flag:

```
./mongod --configsvr <further parameters>
```

3. Start up a number of routing servers via the `mongos` command which are pointed to one config server via the `-configdb` flag:

```
./mongos --configdb <config server host>:<port> <further parameters>
```

4. Connect to a routing server, switch to its admin database and add the shards:

```
./mongo <routig server host>:<routing server port>/admin
> db.runCommand( { addshard : "<shard hostname>:<port>" } )
```

5. Enable sharding for a database and therein contained collection(s):

```
./mongo <routig server host>:<routing server port>/admin
> db.runCommand( { enablesharding : "<db>" } )
> db.runCommand( { shardcollection : "<db>.<collection>",
      key : {<sharding key(s)>} } )
```

It is also possible to add database nodes from a non-sharded environment to a sharded one without downtime. To achieve this, they can be added as shards in step 4 like any other shard server (cf. [HL10]).

**Failures** in a sharded environment have the following effects, depending on the type of failure (cf. [MHB<sup>+</sup>10b]):

- The **failure of a mongos routing process** is uncritical as there may—and in production environments: should—be more than one routing process. In case of a failure of one of these processes, it can simple be launched again or requests from client applications may be redirected to another routing process.
- The **failure of a single mondod process within a shard** do not affect the availability of this shard if it is distributed on different servers comprising a replica set. The aforementioned notes on failures of replica set nodes apply here again.
- The **failure of all mongodb processes comprising a shard** makes the shard unavailable for read and write operations. However, operations concerning other shards are not affected in this failure scenario.
- The **failure of a config server** does not affect read and write operations for shards but makes it impossible to split up and redistribute chunks.

**Implications of Sharding** are that it “must be ran in trusted security mode, without explicit security”, that shard keys cannot be altered as of version 1.6, and that write operations (update, insert, upsert) must include the shard key (cf. [MDN<sup>+10</sup>]).

### Limitations and Renounced Features Regarding Distribution

MongoDB does not provide multiversion-storage, multi-master replication setups or any support for version conflict reconciliation. The rationale for taking a different approach than many other NoSQL stores and not providing these features is explained by the MongoDB documentation as follows:

“Merging back old operations later, after another node has accepted writes, is a hard problem. One then has multi-master replication, with potential for conflicting writes. Typically that is handled in other products by manual version reconciliation code by developers. We think that is too much work : we want MongoDB usage to be less developer work, not more. Multi-master also can make atomic operation semantics problematic. It is possible (as mentioned above) to manually recover these events, via manual DBA effort, but we believe in large system with many, many nodes that such efforts become impractical.” (cf. [MD10, Rationale])

#### 5.2.8. Security and Authentication

According to its documentation, as of version 1.6 “Mongo supports only very basic security”. It is possible to create user accounts for databases and require them to authenticate via username and password. However, access control for authenticated users only distinguishes read and write access. A special user for a MongoDB database server process (i.e. a mongod process) is the `admin` user which has full read and write access to all databases provided by that process including the `admin` database containing administrative data. An `admin` user is also privileged to execute operations declared as administrative (cf. [MMC<sup>+10a</sup>, Mongo Security]).

To require authentication for a database server process, the `mongod` process has to be launched with the `-auth` parameter. Before that, an administrator has to be created in the `admin` database. A logged in administrator then can create accounts for read- and write-privileged users in the `system.users` collection by the `addUser` operation (cf. [MMC<sup>+10a</sup>, Configuring Authentication and Security]):

```

./mongo <parameters>           // log on as administrator
> use <database>
> db.addUser(<username>, <password> [, <true|false>])

```

The optional third parameter of the addUser operation tells whether the user shall only have read-access; if it is omitted, the user being created will have read and write access for the database.

Authentication and access control is not available for replicated and sharded setups. They can only be run securely in an environment that ensures that “only trusted machines can access database TCP ports” (cf. [MMC<sup>+</sup>10a, Running Without Security (Trusted Environment)]).

### 5.2.9. Special Features

#### Capped Collections

MongoDB allows to specify collections of a fixed size that are called *capped collections* and can be processed highly performant by MongoDB according to its manual. The size of such a capped collection is preallocated and kept constant by dropping documents in a FIFO<sup>23</sup>-manner according to the order of their insertion. If a selection is issued on a capped collection the documents of the result set are returned in the order of their insertion if they are not sorted explicitly, i.e. the sort operation is not applied to order the results by document fields<sup>24</sup>.

Capped collections have some restrictions regarding operations on documents: while insertions of documents are always possible, update operations only succeed if the modified document does not grow in size and the delete operation is not available for single documents but only for dropping all documents of the capped collection. The size of a capped collection is limited to 1<sup>9</sup> bytes on a 32-bit machine while there is no such limitation on 64-bit machine. A further limitation of capped collections is the fact that they cannot be subject to sharding. If no ordering is specified when selecting documents, they are returned in the order of their insertion. In addition to limiting the size of a capped collection it is possible to also specify the maximum number of objects to be stored (cf. [MDM<sup>+</sup>10]).

The MongoDB mentions the following use cases for capped collections (cf. [MDM<sup>+</sup>10, Applications]):

**Logging** is an application for capped collections as inserts of documents are processed at a speed comparable to file system writes while the space required for such a log is held constant.

**Caching** of small or limited numbers of precalculated objects in a LRU-manner can be done via capped collections conveniently.

**Auto Archiving** is understood by the MongoDB manual as a situation in which aged data needs to be dropped from the database. While for other databases and regular MongoDB collections scripts have to be written and scheduled to remove data by age capped collections already provide such behavior and no further effort is needed.

---

<sup>23</sup>First in, first out

<sup>24</sup>To be precise, it has to be stated that the sort operation may also be invoked with the \$natural parameter to enforce natural ordering. This can be leveraged to achieve e.g. a descending natural order as follows (cf. [MMD<sup>+</sup>10]):  
db.<collection>.find( ... ).sort( \$natural: -1 )

## GridFS

BSON objects that allow to store binary data in MongoDB are limited in size by 4MB. Therefore, MongoDB implements a specification named *GridFS* to store large binary data and provide operations on large data objects—such as videos or audios—like retrieving a limited number bytes of such an object. The GridFS specification is implemented by MongoDB transparent to client applications by dividing large data objects among multiple documents maintained in a collection and maintaining a collection containing metadata on these documents. Most programming language drivers also support the GridFS specification (cf. [CDM<sup>+</sup>10]).

The MongoDB manual considers GridFS also as an alternative to file system storage when large numbers of files have to be maintained, replicated and backed up as well as if files are expected to change often. In contrast, if a large number of small and static files has to be stored and processed, storage in MongoDB by GridFS is not regarded reasonable (cf. [Hor10]).

## Geospatial Indexes

To support location based queries like “find the closest n items to a specific location” MongoDB provides two-dimensional geospatial indexing (cf. [HMS<sup>+</sup>10]). The latitude and longitude values have to be saved in a document field that is either an object or an array with the first two elements representing coordinates, e.g.

```
{ loc : { 50, 30 } }           // coordinates in an array field
{ loc : { x : 50, y : 30 } }    // coordinates in an object field
```

If an object field is used, the object's fields do not have to have certain names and also do not have to appear in a certain order (though ordering has to be consistent among all documents).

An index on the geospatial coordinates is created using the special value `2d` instead of an order:

```
db.<collection>.createIndex( { <field> : "2d" } );
```

Without additional parameters for index creation, MongoDB suggests that latitude and longitude values shall be indexed and therefore aligns the index range to the interval  $] -180 \dots 180 [$ . If other values are indexed the index range should be given at index creation as follows:

```
db.<collection>.createIndex( { <field> : "2d" } , { min : <min-value>,
                                                       max : <max-value> } );
```

The range boundaries are exclusive which means that index values cannot take these values.

To query based on a geospatial index the query document can contain special criteria like `$near` and `$maxDistance`:

```
db.<collection>.find( { <field> : [<coordinate>, <coordinate>] } ); // exact match
db.<collection>.find( { <field> : { $near : [<coordinate>, <coordinate>] } } );
db.<collection>.find( { <field> : { $near : [<coordinate>, <coordinate>], $maxDistance : <value> } } );
```

In addition to limiting the distance in which matches shall be found it is also possible to define a shape (box or circle) covering a geospatial region as a selection criterion.

Besides the generic `find` operation MongoDB also provides a specialized way to query by geospatial criterions using the `geoNear` operation. The advantages of this operation are that it returns a distance value for each matching document and allows for diagnostics and troubleshooting. The `geoNear` operation has to be invoked using the `db.runCommand` syntax:

```
db.runCommand({geoNear: <collection>, near= [<coordinate>, <coordinate>], ...});
```

As shown in the syntax example, no field name has to be provided to the `geoNear` operation as it automatically determines the geospatial index of the collection to query.

When using the aforementioned `$near` criterion MongoDB does its calculations based on an idealized model of a flat earth where an arcdegree of latitude and longitude is the same distance at each location. Since MongoDB 1.7 a spherical model of the earth is provided so that selections can use correct spherical distances by applying the `$sphereNear` criterion (instead of `$near`), the `$centerSphere` criterion (to match circular shapes) or adding the option `spherical:true` to the parameters of the `geoNear` operation. When spherical distances are used, coordinates have to be given as decimal values in the order longitude, latitude using the radians measurement.

Geospatial indexing has some limitations in the MongoDB versions 1.6.5 (stable) and 1.7 (unstable as of December 2010). First, it is limited to indexing squares without wrapping at their outer boundaries. Second, only one geospatial index is allowed per collection. Third, MongoDB does not implement wrapping at the poles or at the -180° to 180° boundary. Lastly, geospatial indexes cannot be sharded.

# 6. Column-Oriented Databases

In this chapter a third class of NoSQL datastores is investigated: column-oriented databases. The approach to store and process data by column instead of row has its origin in analytics and business intelligence where column-stores operating in a shared-nothing massively parallel processing architecture can be used to build high-performance applications. Notable products in this field are Sybase IQ and Vertica (cf. [Nor09]). However, in this chapter the class of column-oriented stores is seen less puristic, also subsuming datastores that integrate column- and row-orientation. They are also described as “[sparse], distributed, persistent multidimensional sorted [maps]” (cf. e.g. [Int10]). The main inspiration for column-oriented datastores is Google’s Bigtable which will be discussed first in this chapter. After that there will be a brief overview of two datastores influenced by Bigtable: Hypertable and HBase. The chapter concludes with an investigation of Cassandra, which is inspired by Bigtable as well as Amazon’s Dynamo. As seen in section 2.3 on classifications of NoSQL datastores, Bigtable is subsumed differently by various authors, e.g. as a “wide columnar store” by Yen (cf. [Yen09]), as an “extensible record store” by Cattel (cf. [Cat10]) or as an entity-attribute-value<sup>1</sup> datastore by North (cf. [Nor09]). In this paper, Cassandra is discussed along with the column-oriented databases as most authors subsume it in this category and because one of its main inspirations, Google’s Bigtable, has to be introduced yet.

## 6.1. Google’s Bigtable

Bigtable is described as “a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers” (cf. [CDG<sup>+</sup>06, p. 1]). It is used by over sixty projects at Google as of 2006, including web indexing, Google Earth, Google Analytics, Orkut, and Google Docs (formerly named *Writely*)<sup>2</sup>. These projects have very different data size, infrastructure and latency requirements: “from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data” (cf. [CDG<sup>+</sup>06, p. 1]). According to Chang et al. experience at Google shows that “Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability” (cf. [CDG<sup>+</sup>06, p. 1]). Its users “like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time”. For Google as a company the design and implementation of Bigtable has shown to be advantageous as it has “gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable’s implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise” (cf. [CDG<sup>+</sup>06, p. 13]).

---

<sup>1</sup>The entity-attribute-value (EAV) datastores predate relational databases and do not provide the full feature set of RDBMSs (e.g. no comprehensive querying capabilities based on a declarative language like SQL) but their data model is richer than that of a simple key-/value-store. EAV based on modern technologies currently have a revival as they are offered by major cloud computing providers such as Amazon (SimpleDB), Google (App Engine datastore) and Microsoft (SQL Data Services; cf. [Nor09]).

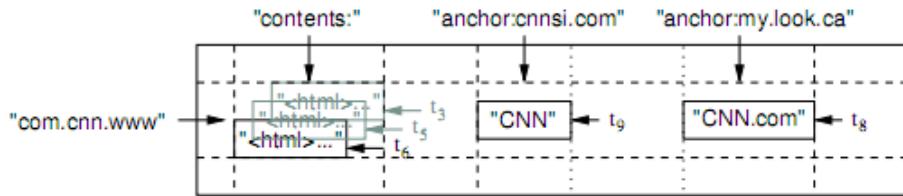
<sup>2</sup>As of 2011 a number of further notable projects such as the Google App Engine and Google fusion tables also use Bigtable.

Bigtable is described as a database by Google as “it shared many implementation strategies with databases”, e.g. parallel and main-memory databases. However, it distinguishes itself from relational databases as it “does not support a full relational data model”, but a simpler one that can be dynamically controlled by clients. Bigtable furthermore allows “clients to reason about the locality properties of the data” which are reflected “in the underlying storage” (cf. [CDG<sup>+</sup>06, p. 1]). In contrast to RDBMSs, data can be indexed by Bigtable in more than one dimension—not only row- but also column-wise. A further distinguishing proposition is that Bigtable allows data to be delivered out of memory or from disk—which can be specified via configuration.

### 6.1.1. Data Model

Chang et al. state that they “believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers.” Therefore the data model they designed for Bigtable should be “richer than simple key-value pairs, and [support] sparse semi-structured data”. On the other hand, it should remain “simple enough that it lends itself to a very efficient flat-file representation, and [...] transparent enough [...] to allow our users to tune important behaviors of the system” (cf. [CDG<sup>+</sup>06, p. 12]).

The data structure provided and processed by Google’s Bigtable is described as “a sparse, distributed, persistent multidimensional sorted map”. Values are stored as arrays of bytes which do not get interpreted by the data store. They are addressed by the triple (row-key, column-key, timestamp) (cf. [CDG<sup>+</sup>06, p. 1]).



**Figure 6.1.:** Google’s Bigtable – Example of Web Crawler Results (taken from [CDG<sup>+</sup>06, p. 2])

Figure 6.1 shows a simplified example of a Bigtable<sup>3</sup> storing information a web crawler might emit. The map contains a non-fixed number of rows representing domains read by the crawler as well as a non-fixed number of columns: the first of these columns (contents:) contains the page contents whereas the others (anchor:<domain-name>) store link texts from referring domains—each of which is represented by one dedicated column. Every value also has an associated timestamp ( $t_3, t_5, t_6$  for the page contents,  $t_9$  for the link text from *CNN Sports Illustrated*,  $t_8$  for the link text from *MY-look*). Hence, a value is addressed by the triple (domain-name, column-name, timestamp) in this example (cf. [CDG<sup>+</sup>06, p. 2]).

**Row** keys in Bigtable are strings of up to 64KB size. Rows are kept in lexicographic order and are dynamically partitioned by the datastore into so called **tablets**, “the the unit of distribution and load balancing” in Bigtable. Client applications can exploit these properties by wisely choosing row keys: as the ordering of row-keys directly influences the partitioning of rows into tablets, row ranges with a small lexicographic distance are probably split into only a few tablets, so that read operations will have only a small number of servers delivering these tablets (cf. [CDG<sup>+</sup>06, p. 2]). In the aforementioned example of figure 6.1 the domain names used as row keys are stored hierarchically descending (from a DNS point of view), so that subdomains have a smaller lexicographic distance than if the domain names were stored reversely (e.g. com.cnn.blogs, com.cnn.www in contrast to blogs.cnn.com, www.cnn.com).

<sup>3</sup>Chang et al. also refer to the datastructure itself as a *Bigtable*.

The number of **columns** per table is not limited. Columns are grouped by their key prefix into sets called *column families*. Column families are an important concept in Bigtable as they have specific properties and implications (cf. [CDG<sup>+</sup>06, p. 2]):

- They “form the basic unit of access control”, discerning privileges to list, read, modify, and add column-families.
- They are expected to store the same or a similar type of data.
- Their data gets compressed together by Bigtable.
- They have to be specified before data can be stored into a column contained in a column family.
- Their name has to be printable. In contrast, column qualifiers “may be arbitrary strings”.
- Chang et al. suggest that “that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation”.

The example of figure 6.1 shows two column families: `content` and `anchor`. The `content` column family consists of only one column whose name does not have to be qualified further. In contrast, the `anchor` column family contains two columns qualified by the domain name of the referring site.

**Timestamps**, represented as 64-bit integers, are used in Bigtable to discriminate different reversion of a cell value. The value of a timestamp is either assigned by the datastore (i.e. the actual timestamp of saving the cell value) or chosen by client applications (and required to be unique). Bigtable orders the cell values in decreasing order of their timestamp value “so that the most recent version can be read first”. In order to disburden client applications from deleting old or irrelevant revisions of cell values, an automatic garbage-collection is provided and can be parameterized per column-family by either specifying the number of revisions to keep or their maximum age (cf. [CDG<sup>+</sup>06, p. 2f]).

### 6.1.2. API

The Bigtable datastore exposes the following classes of operations to client applications (cf. [CDG<sup>+</sup>06, p. 3]):

**Read Operations** include the lookup and selection of rows by their key, the limitation of column families as well as timestamps (comparable to projections in relational databases) as well as iterators for columns.

**Write Operations for Rows** cover the creation, update and deletion of values for a columns of the particular row. Bigtable also supports “batching writes across row keys”.

**Write Operations for Tables and Column Families** include their creation and deletion.

**Administrative Operations** allow to change “cluster, table, and column family metadata, such as access control rights”.

**Server-Side Code Execution** is provided for scripts written in Google’s data processing language Sawzall (cf. [PDG<sup>+</sup>05], [Gri08]). As of 2006, such scripts are not allowed to write or modify data stored in Bigtables but “various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators”.

**MapReduce Operations** may use contents of Bigtable maps as their input source as well as output target.

Transactions are provided on a single-row basis: “Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system’s behavior in the presence of concurrent updates to the same row” (cf. [CDG<sup>+</sup>06, p.2]).

### 6.1.3. Infrastructure Dependencies

Bigtable depends on a number of technologies and services of Google’s infrastructure (cf. [CDG<sup>+</sup>06, p.3f]):

- The distributed **Google File System (GFS)** (cf. [GL03]) is used by Bigtable to persist its data and log files.
- As Bigtable typically shared machines with “wide variety of other distributed applications” it depends on a **cluster management system** “for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status”.
- Bigtable data is stored in Google’s **SSTable** file format (see section 3.3 on page 44). A SSTable is a “persistent, ordered immutable map” whose keys and values “are arbitrary byte strings”. SSTables allow applications to look up values via their keys and “iterate over all key/value pairs in a specified key range”. Internally an SSTable is represented as a sequence of blocks with a configurable, fixed size. When an SSTable is opened, a block index located at the end of the SSTable is loaded into memory. If a block from the SSTable is requested, only one disk seek is necessary to read the block after a binary search in the in-memory block index has occurred. To further enhance read performance SSTables can be loaded completely into memory.
- The “highly-available and persistent distributed lock service” **Chubby** (cf. [Bur06]) is employed by Bigtable for several tasks:
  - “[Ensure] that there is at most one active master at any time” in a Bigtable cluster
  - Store of location information for Bigtable data required to bootstrap a Bigtable instance
  - Discover tablet servers<sup>4</sup>
  - Finalize of tablet server deaths
  - Store schema information, i.e. the column-families for each table of a Bigtable instance
  - Store of access control lists

Each instance of a Chubby service consists a cluster of “five active replicas, one of which is elected to be the master and actively serve requests”. To keep the potentially fallible replicas consistent, an implementation of the Paxos consensus algorithm (cf. [Lam98]) is used<sup>5</sup>. Chubby provides a namespace for directories and (small) files and exposes a simplified file-system interfaces towards clients. Each file and directory “can act as a reader-writer lock” ([Bur06, p. 338]). Client application initiate sessions with a Chubby instance, request a lease and refresh it from time to time. If a Chubby cluster becomes unavailable, its clients—including Bigtable—instances are unable to refresh their session lease within a predefined amount of time so that their sessions expire; in case of Bigtable this results in an unavailability of Bigtable itself “after an extended period of time” (i.e. Chubby unavailability; cf. [CDG<sup>+</sup>06, p. 4]).

---

<sup>4</sup>Tablet servers will be introduced and explained in the next section.

<sup>5</sup>Details on Chubby as well difficulties met when implementing the theoretically well described Paxos consensus algorithm are described in Google’s “Paxos made live” paper (cf. [CGR07]).

### 6.1.4. Implementation

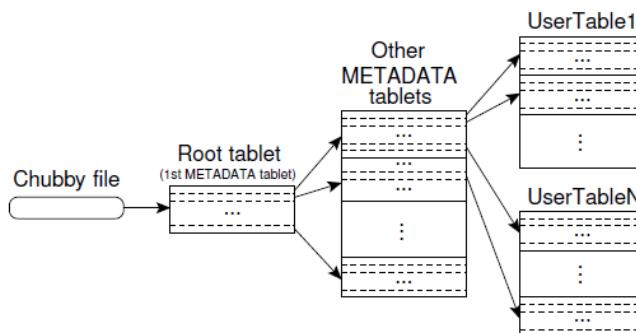
#### Components

Bigtable's implementation consists of three major components per Bigtable instance (cf. [CDG<sup>+</sup>06, p. 4]):

- Multiple **tablet servers** each of which is responsible for a number of tablets. This implies the handling of read and write requests for tablets as well as the splitting of tablets "that have grown too large"<sup>6</sup>. Tablet servers can be added and removed at runtime.
- A **client library** provided for applications to interact with Bigtable instances. The library responsible for looking up tablet servers that are in charge of data that shall be read or written, directing requests to them and providing their responses to client applications.
- One **master server** with a number of responsibilities. Firstly, it manages the tablets and tablet servers: it assigns tablets to tablet servers, detects added and removed tablet servers, and distributes workload across them. Secondly, it is responsible to process changes of a Bigtable schema, like the creation of tables and column families. Lastly, it has to garbage-collect deleted or expired files stored in GFS for the particular Bigtable instance. Despite these responsibilities the load on master servers is expected to be low as client libraries lookup tablet location information themselves and therefore "most clients never communicate with the master". As the master server is a single point of failure for a Bigtable instances it is backed up by a second machine according to Ippolito (cf. [Ipp09]).

#### Tablet Location Information

As it has been stated in the last sections, tables are dynamically split into tablets and these tablets are distributed among a multiple tablet servers which can dynamically enter and leave a Bigtable instance at runtime. Hence, Bigtable has to provide means for managing and looking up tablet locations, such that master servers can redistribute tablets and client libraries can discover the tablet servers which are in charge of certain rows of table. Figure 6.2 depicts how tablet location information is stored in Bigtable (cf. [CDG<sup>+</sup>06, p. 4f]).



**Figure 6.2.: Google's Bigtable – Tablet Location Hierarchy** (taken from [CDG<sup>+</sup>06, p. 4])

The locations of tablets are stored in a table named **METADATA** which is completely held in memory. This table is partitioned into a special first tablet (*root tablet*) and an arbitrary number of further tablets (*other METADATA tablets*). The *other METADATA tablets* contain the location information for all tablets of user tables (i.e. tables created by client applications) whereas the *root tablet* contains information about

<sup>6</sup>Typical sizes of tablets after splitting are 100-200 MB at Google according to Chang et al..

the location of the *other METADATA tablets* and is never split itself. The location information for the *root tablet* is stored in a file placed in a Chubby namespace. The location information for a tablet is stored in row that identified by the “tablet’s table identifier and its end row”<sup>7</sup>. With a size of about 1 KB per row and a tablet size of 128 MB Bigtable can address  $2^{34}$  tablets via the three-level hierarchy depicted in figure 6.2.

A client library does not read through all levels of tablet location information for each interaction with the datastore but caches tablet locations and “recursively moves up the tablet location hierarchy” if it discovers a location to be incorrect. A further optimization employed at Google is the fact that client libraries prefetch tablet locations, i.e. read tablet locations for more than one tablet whenever they read from the METADATA table.

### Master Server, Tablet Server and Tablet Lifecycles

**Tablet Lifecycle** A tablet is created, deleted and assigned to a tablet server by the master server. Each tablet of a Bigtable is assigned to at most one tablet server at a time; *at most one* is due to the fact that tablets may also be unassigned until the master server finds a tablet server that provides enough capacity to serve that tablet. Tablets may also get merged by the master server and split by a tablet server which has to notify the master server about the split<sup>8</sup>). Details about how tablets are represented at runtime and how read and write operations are applied to them is discussed in the section on tablet representation below.

**Tablet Server Lifecycle** When a tablet server starts, it creates a uniquely-named file in a predefined directory of a Chubby namespace and acquires an exclusive lock for this. The master server of a Bigtable instance constantly monitors the tablet servers by asking them whether they have still locked their file in Chubby; if a tablet server does not respond, the master server checks the Chubby directory to see whether the particular tablet server still holds its lock. If this is not the case, the master server deletes the file in Chubby and puts the tablets served by this tablet server into the set of unassigned tablets. The tablet server itself stops serving any tablets when it loses its Chubby lock. If the tablet server is still up and running but was not able to hold its lock due to e.g. network partitioning it will try to acquire that lock again if its file in Chubby has not been deleted. If this file is no longer present, the tablet server stops itself. If a tablet server is shut down in a controlled fashion by administrators, it tries to its Chubby lock, so that the master server can reassign tablets sooner (cf. [CDG<sup>+</sup>06, p. 5]).

**Master Server Lifecycle** When a master server starts up, it also places a special file into a Chubby namespace and acquires an exclusive lock for it (to prevent “concurrent master instantiations”). If the master server is not able to hold that lock so that its Chubby session expires, it takes itself down as it cannot monitor the tablet servers correctly without a reliable connection to Chubby. Hence, the availability of a Bigtable instance relies on the reliability of the connection between master server and the Chubby service used by a Bigtable instance.

In addition to registering itself via a file and its lock in Chubby, the master server processes the following steps when it starts up:

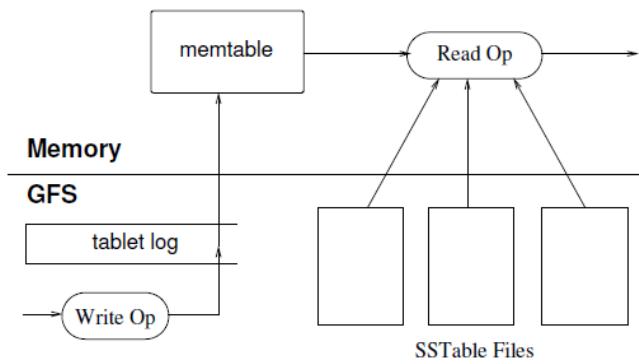
<sup>7</sup>The tuple (table, end row) is sufficient to identify a tablet due to the fact that rows are kept in ascending order by their key.

<sup>8</sup>If such a split notification gets lost (because the tablet server or the master server crashed before it has been exchanged) the master server gets aware of the split as soon as he assigns the formerly unsplitted tablet to a tablet server. The tablet server discovers the tablet split by comparing the end-row of the tablet specified by the master server compared to the end-row specified in the METADATA table: if the latter is less then the first, a split has happened of which the master server is unaware. The METADATA table contains the latest information about tablet boundaries as tablet servers have to commit tablet splittings to this table before notifying the master server.

1. Discover tablet servers that are alive via scanning the Chubby directory with tablet server files and checking their locks.
2. Connect to each alive tablet server and ask for the tablets it serves at the moment.
3. Scan the METADATA table<sup>9</sup> to construct a list of all tables and tablets. Via subtraction of the tablets already served by the running tablet servers, it derives a set of unassigned tablets.

### Tablet Representation

Figure 6.3 depicts how tablets are represented at runtime.



**Figure 6.3.: Google's Bigtable – Tablet Representation at Runtime (taken from [CDG<sup>+</sup>06, p. 6])**

All write operations on tablets are “committed to a commit log that stores redo records” and is persisted in the Google File System (GFS). The recently committed updates are put into a sorted RAM-buffer called *memtable*. When a memtable reaches a certain size, it is frozen, a new memtable is created, the frozen memtable gets transformed into the SSTable format and written to GFS; this process is called a *minor compaction*. Hence, the older updates get persisted in a sequence of SSTables on disk while the newer ones are present in memory. The information about where the SSTables comprising a tablet are located is stored in the METADATA table along with a set of pointers directing into one or more commit logs by which the memtable can be reconstructed when the tablet gets assigned to a tablet server.

Write operations are checked for well-formedness as well as authorization before they are written to the commit log and the memtable (when they are finally committed). The authorization-information is provided on column-family base and stored in a Chubby namespace.

Read operations are also checked whether they are well-formed and whether the requesting client is authorized to issue them. If a read operation is permitted, it “is executed on a merged view of the sequence of SSTables and the memtable”. The *merged view* required for read operations can be established efficiently as the SSTables and memtable are lexicographically sorted.

Besides minor compactations—the freeze, transformation and persistence of memtables as SSTables—the SSTables also get compacted from time to time. Such a *merging compaction* is executed asynchronously by a background service in a copy-on-modify fashion. The goal of merging compactations is to limit the number of SSTables which have to be considered for read operations.

A special case of merging transformations are so called *major compactations* which produce exactly one SSTable out of a number of SSTables. They are executed by Bigtable regularly to “to reclaim resources

<sup>9</sup>The scanning of the METADATA table is only possible when it is actually assigned to tablet servers. If the master server discovers that this is not the case, it puts the *root tablet* to the set of unassigned tablet, tries to assign it to a tablet server and continues its bootstrapping with scanning the METADATA table as soon as the *root tablet* is assigned.

used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data”.

During all compactations read as well as write operations can still be served. This is due to the fact that SSTables as well as a frozen memtable are immutable and only discarded if the compaction was finished successfully. In addition, a memtable for committed write operations is always present.

### 6.1.5. Refinements

In order to enhance the performance, availability and reliability of Bigtable, several refinements have been implemented at Google (cf. [CDG<sup>+</sup>06, p. 6ff]):

**Locality Groups** are groups of column-families that a client application defines and that are expected to be accessed together typically. Locality groups cause Bigtable to create a separate SSTable for each locality group within a tablet. The concept of locality groups is an approach to increase read performance by reducing the number of disk seeks as well as the amount of data to be processed for typical classes of read operations in an application. A further enhancement of read performance can be achieved by requiring a locality group to be served from memory which causes Bigtable to lazily load the associated SSTables into RAM when they are first read. Bigtable internally uses the concept of locality groups served from memory for the METADATA table.

**Compression** of SSTables can be requested by client applications for each locality group. Clients can also specify the algorithm and format to be applied for compression. The SSTable-contents are compressed block-wise by Bigtable which results in lower compression ratios compared to compressing an SSTable as a whole but “we benefit in that small portions of an SSTable can be read without decompressing the entire file”. Experiences at Google show that many applications “use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy’s scheme [...], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines.” Although optimized for speed rather than space, this two-pass scheme results in good compression ratios like 10:1 for Bigtables with similar contents than the Webtable example of figure 6.1 on page 105. This is due to the fact that similar data (e.g. documents from the same domain) are typically clustered together as row keys are lexicographically ordered and are typically chosen such that the lexicographic distance of row keys representing similar data is small.

**Caching at Tablet Servers** In order to optimize read performance, two levels of caching are implemented at the tablet servers: a cache for key-/value-pairs returned by the SSTable API (called *scan cache*) and a cache for SSTable-blocks (called *block cache*). While the scan cache optimizes read performance if the same data is read repeatedly, the block cache “is useful for applications that tend to read data that is close to the data they recently read”.

**Bloom Filters** To further enhance read performance, applications can require Bigtable to create and leverage bloom filters for a locality group. These bloom filters are used to detect whether an SSTables might contain data for certain key-/value pairs, thereby reducing the number of SSTables to be read from disk when creating the merged view of a tablet required to process read operations.

**Group Commits** “[The] throughput of lots of small mutations” to the commit log and memtable is improved by a group commit strategy (cf. [CDG<sup>+</sup>06, p. 6]).

**Reduction of Commit Logs** Each tablet server keeps only two<sup>10</sup> append-only commit log files for all tablets it serves. This is due to the fact that number of commit logs would grow very if such a file would be created and maintained on a per-tablet base, resulting in many concurrent writes to GFS, potentially large numbers of disk seeks, and less efficiency of the group commit optimization. A downside of this approach appears if tablets get reassigned (e.g. caused by tablet server crashes or tablet redistribution due to load balancing): as a tablet server typically only servers part of the tablets that were served by another server before, it has to read through the commit logs the former server produced to establish the tablet representation shown in figure 6.3 on page 110; hence, if the tablets once served by one tablet server get reassigned to n tablet servers, the commit logs have to be read n times. To tackle this issue, the commits are stored and sorted by the key-triple (table, row, log sequence number). This results in just one disk seek with a subsequent contiguous read for each tablet server that has to evaluate commit logs in order to load tablets formerly served by another tablet server. The sorting of a commit log is optimized in two ways. Firstly, a commit log is sorted lazily: when a tablet server has to serve additional tablets, it beckons the master server that it has to evaluate a commit log so that the master server can initiate the sorting of this commit log. A second optimization is how the commit logs are sorted: the master server splits them up into 64 MB chunks and initiates a parallel sort of these chunks on multiple tablet servers.

**Reducing GFS Latency Impacts** The distributed Google File Systems is not robust against latency spikes, e.g. due to server crashes or network congestion. To reduce the impact of such latencies, each tablet server uses two writer threads for commit logs, each writing to its own file. Only one of these threads is actively writing to GFS at a given time. If this active thread suffers from GFS “performance hiccups”, the commit logging is switched to the second thread. As any operation in a commit log has a unique sequence number, duplicate entries in the two commit logs can be eliminated when a tablet server loads a tablet.

**Improving Tablet Recovery** *Tablet recovery* is the process of tablet-loading done by a tablet server that a particular tablet has been assigned to. As discussed in this section and in section 6.1.4, a tablet server has to evaluate the commit logs attached that contain operations for the tablet to load. Besides the aforementioned Bloom Filter optimization, Bigtable tries to avoid that a tablet server has to read a commit log at all when recovering a tablet. This is achieved by employing two minor compactations when a tablet server stops serving a tablet. The first compaction is employed to reduce “the amount of uncompacted state in the tablet server’s commit log”. The second compaction processes the update operations that have been processed since the first compaction was started; before this second compaction is executed, the tablet server stops serving any requests. These optimizations to reduce tablet recovery time can only happen and take effect if a tablet server stops serving tablets in a controlled fashion (i.e. it has not due to a crash).

**Exploiting Immutability** Bigtable leverages in manifold ways from the fact that SSTables are immutable. Firstly, read operations to SSTables on the filesystem do not have to synchronized. Secondly, the removal of data is delegated to background processes compacting SSTables and garbage-collecting obsolete ones; hence, the removal of data can occur asynchronously and the time required for it does not have to be consumed while a request is served. Finally, when a tablet gets split the resulting child tablets inherit the SSTables from their parent tablet.

To provide efficient read and write access to the mutable memtable, a partial and temporary immutability is introduced by making memtable rows “copy-on-write and allow reads and writes to proceed in parallel”.

---

<sup>10</sup>Chang et al. first speak of only one commit log per tablet server in their section “Commit-log implementation” (cf. [CDG<sup>+</sup>06, p. 7f]). Later in this section, they introduce the optimization of two commit-log writer threads per tablet server, each writing to its own file (see below).

### 6.1.6. Lessons Learned

In the design, implementation and usage of Bigtable at Google, a lot of experience has been gained. Chang et al. especially mention the following lessons they learned:

**Failure Types in Distributed Systems** Chang et al. criticize the assumption made in many distributed protocols that large distributed systems are only vulnerable to few failures like “the standard network partitions and fail-stop failures”. In contrast, they faced a lot more issues: “memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance”. Hence, they argue that such sources of failure also have to be addressed when designing and implementing distributed systems protocols. Examples that were implemented at Google are checksumming for RPC calls as well as removing assumptions in a part of the system about the other parts of the system (e.g. that only a fixed set of errors can be returned by a service like Chubby).

**Feature Implementation** A lesson learned at Google while developing Bigtable at Google is to implement new features into such a system only if the actual usage patterns for them are known. A counterexample Chang et al. mention are general purpose distributed transactions that were planned for Bigtable but never implemented as there never was an immediate need for them. It turned out that most applications using Bigtable only needed single-row transactions. The only use case for distributed transactions that came up was the maintenance of secondary indices which can be dealt with by a “specialized mechanism [...] will be less general than distributed transactions, but will be more efficient”. Hence, general purpose implementations arising when no actual requirements and usage patterns are specified should be avoided according to Chang et al..

**System-Level Monitoring** A practical suggestion is to monitor the system as well at its clients in order to detect and analyze problems. In Bigtable e.g. the RPC being used by it produces “a detailed trace of the important actions” which helped to “detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the METADATA table when METADATA tablets are unavailable”.

**Value Simple Designs** In the eyes of Chang et al. the most important lesson to be learned from Bigtable’s development is that simplicity and clarity in design as well as code are of great value—especially for big and unexpectedly evolving systems like Bigtable. As an example they mention the tablet-server membership protocol which was designed too simple at first, refactored iteratively so that it became too complex and too much depending on seldomly used Chubby-features, and in the end was redesigned to “to a newer simpler protocol that depends solely on widely-used Chubby features” (see section 6.1.4).

## 6.2. Bigtable Derivatives

As the Bigtable code as well as the components required to operate it are not available under an open source or free software licence, open source projects have emerged that are adopting the concepts described in the Bigtable paper by Chang et al.. Notably in this field are Hypertable and HBase.

### Hypertable

Hypertable is modelled after Google’s Bigtable and inspired by “our own experience in solving large-scale data-intensive tasks” according to its developers. The project’s goal is “to set the open source standard

for highly available, petabyte scale, database systems". Hypertable is almost completely written in C++ and relies on a distributed filesystem such as Apache Hadoop's HDFS (Hadoop Distributed File System) as well as a distributed lock-manager. Regarding its data model it supports all abstractions available in Bigtable; in contrast to Hbase column-families with an arbitrary numbers of distinct columns are available in Hypertable. Tables are partitioned by ranges of row keys (like in Bigtable) and the resulting partitions get replicated between servers. The data representation and processing at runtime is also borrowed from Bigtable: "[updates] are done in memory and later flushed to disk". Hypertable has its own query language called HQL (Hypertable Query Language) and exposes a native C++ as well as a Thrift API. Originally developed by Zvents Inc., it has been open-sourced under the GPL in 2007 and is sponsored by Baidu, the leading chinese search engine since 2009 (cf. [Hyp09a], [Hyp09b], [Hyp09c], [Cat10], [Jon09], [Nor09], [Int10], [Wik11b]).

## HBase

The HBase datastore is a Bigtable-clone developed in Java as a part of Apache's MapReduce-framework Hadoop, providing a "a fault-tolerant way of storing large quantities of sparse data". Like Hypertable, HBase depends on a distributed file system (HDFS) which takes the same role as GFS in the context of Bigtable. Concepts also borrowed from Bigtable are the memory and disk usage pattern with the need for compactations of immutable or append-only files, compression of data as well as bloom filters for the reduction of disk access. HBase databases can be a source of as well as a destination for MapReduce jobs executed via Hadoop. HBase exposes a native API in Java and can also be accessed via Thrift or REST. A notable usage of HBase is the real-time messaging system of Facebook built upon HBase since 2010 (cf. [Apa11], [Nor09], [Jon09], [Int10], [Hof10a], [Wik11a]).

## 6.3. Cassandra

As a last datastore in this paper Apache Cassandra which adopts ideas and concepts of both, Amazon's Dynamo as well as Google's Bigtable (among others, cf. [LM10, p. 1f]), shall be discussed. It was originally developed by Facebook and open-sourced in 2008. Lakshman describes Cassandra as a "distributed storage system for managing structured data that is designed to scale to a very large size". It "shares many design and implementation strategies with databases" but "does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format" (cf. [Lak08], [LM10, p. 1], [Cat10]). Besides Facebook, other companies have also adopted Cassandra such as Twitter, Digg and Rackspace (cf. [Pop10a], [Eur09], [Oba09a], [Aja09]).

### 6.3.1. Origin and Major Requirements

The use-case leading to the initial design and development of Cassandra was the so entitled *Inbox Search problem* at Facebook. The worlds largest social network Facebook allows users to exchange personal messages with their contacts which appear in the inbox of a recipient. The *Inbox Search problem* can be described as finding an efficient way of storing, indexing and searching these messages.

The major requirements for the inbox search problem as well as problems of the same nature were (cf. [Lak08], [LM10, p. 1]):

- Processing of a large amount and high growth rate of data (given 100 million users as of June 2008, 250 million as of August 2009 and over 600 billion users as of January 2011; cf. [LM10, p. 1], [Car11])

- High and incremental scalability
- Cost-effectiveness
- “Reliability at massive scale” since “[outages] in the service can have significant negative impact”
- The ability to “run on top of an infrastructure of hundreds of nodes [i.e. commodity servers] (possibly spread across different datacenters)”
- A “high write throughput while not sacrificing read efficiency”
- No single point of failure
- Treatment of failures “as a norm rather than an exception”

After a year of productive usage of Cassandra at Facebook<sup>11</sup> Lakshman summed up that “Cassandra has achieved several goals – scalability, high performance, high availability and applicability” (cf. [Lak08]). In August 2009 Lakshman and Malik affirm that “Cassandra has kept up the promise so far” and state that it was also “deployed as the backend storage system for multiple services within Facebook” (cf. [LM10, p. 1]).

### 6.3.2. Data Model

An instance of Cassandra typically consists of only one table which represents a “distributed multidimensional map indexed by a key”. A table is structured by the following dimensions (cf. [LM10, p. 2], [Lak08]):

**Rows** which are identified by a string-key of arbitrary length. Operations on rows are “atomic per replica no matter how many columns are being read or written”.

**Column Families** which can occur in arbitrary number per row. As in Bigtable, column-families have to be defined in advance, i.e. before a cluster of servers comprising a Cassandra instance is launched. The number of column-families per table is not limited; however, it is expected that only a few of them are specified. A column family consists of *columns* and *supercolumns*<sup>12</sup> which can be added dynamically (i.e. at runtime) to column-families and are not restricted in number (cf. [Lak08]).

**Columns** have a name and store a number of values per row which are identified by a timestamp (like in Bigtable). Each row in a table can have a different number of columns, so a table cannot be thought of as a rectangle. Client applications may specify the ordering of columns within a column family and supercolumn which can either be by name or by timestamp.

**Supercolumns** have a name and an arbitrary number of columns associated with them. Again, the number of columns per super-column may differ per row.

Hence, values in Cassandra are addressed by the triple (*row-key*, *column-key*, *timestamp*) with column-key as *column-family:column* (for simple columns contained in the column family) or *column-family:supercolumn:column* (for columns subsumed under a supercolumn).

---

<sup>11</sup>As of 2008 it ran on more than 600 cores and stored more than 120 TB of index data for Facebook’s inbox messaging system.

<sup>12</sup>The paper by Laksman and Malik distinguishes *columns* and *supercolumns* at the level of column-families and therefore speaks of *simple column families* and *super column families*. In contrast, the earlier blog post of Lakshman distinguishes *columns* and *supercolumns*. As the latter approach describes the concept precisely and at column family level there is no further distinctive than just the type of assigned columns this paper follows Lakshman’s earlier description.

### 6.3.3. API

The API exposed to client-applications by Cassandra consists of just three operations (cf. [LM10, p. 2]):

- `get(table, key, columnName)`
- `insert(table, key, rowMutation)`
- `delete(table, key, columnName)`

The `columnName` argument of the `get` and `delete` operation identifies either a column or a supercolumn within a column family or column family as a whole .

All requests issued by client-applications get routed to an arbitrary server of a Cassandra cluster which determines the replicas serving the data for the requested key. For the write operations `insert` and `update` a quorum of replica nodes has “to acknowledge the completion of the writes”. For read operations, clients can specify which consistency guarantee they desire and based on this definition either the node closest to the client serves the request or a quorum of responses from different nodes is waited for before the request returns; hence, by defining a quorum client-applications can decide which “degree of eventual consistency” they require (cf. [LM10, p. 2]).

The API of Cassandra is exposed via Thrift. In addition, programming language libraries for Java, Ruby, Python, C# and others are available for interaction with Cassandra (cf. [Cat10], [Int10]).

### 6.3.4. System Architecture

#### Partitioning

As Cassandra is required to be incrementally scalable, machines can join and leave a cluster (or crash), so that data has to be partitioned and distributed among the nodes of a cluster in a fashion that allows repartitioning and redistribution. The data of a Cassandra table therefore gets partitioned and distributed among the nodes by a consistent hashing function that also preserves the order of row-keys. The order preservation property of the hash function is important to support range scans over the data of a table. Common problems with basic consistent hashing<sup>13</sup> are treated differently by Cassandra compared to e.g. Amazon’s Dynamo: while Dynamo hashes physical nodes to the ring multiple times (as virtual nodes<sup>14</sup>), Cassandra measures and analyzes the load information of servers and moves nodes on the consistent hash ring to get the data and processing load balanced. According to Lakshman and Malik this method has been chosen as “it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing” (cf. [LM10, p. 2], [Lak08]).

#### Replication

To achieve high scalability and durability of a Cassandra cluster, data gets replicated to a number of nodes which can be defined as a replication factor per Cassandra instance. Replication is managed by a *coordinator node* for the particular key being modified; the coordinator node for any key is the first node on the consistent hash ring that is visited when walking from the key’s position on the ring in clockwise direction (cf. [LM10, p. 3]).

---

<sup>13</sup>Non-uniform load and data distribution due to randomly distributed hash values as well as non-consideration of hardware heterogeneity (see section 3.2.1 on page 39ff)

<sup>14</sup>The number of virtual nodes per physical node depends on the hardware capabilities of the physical node.

Multiple replication strategies are provided by Cassandra:

- **Rack Unaware** is a replication strategy within a datacenter where  $N - 1^{15}$  nodes succeeding the coordinator node on the consistent hash ring are chosen to replicate data to them.
- **Rack Aware** (within a datacenter) and **Datacenter Aware** are replication strategies where by a system called ZooKeeper<sup>16</sup> a leader is elected for the cluster who is in charge of maintaining “the invariant that no node is responsible for more than  $N-1$  ranges in the ring”. The metadata about the nodes’ responsibilities for key ranges<sup>17</sup> is cached locally at each node as well as in the Zookeeper system. Nodes that have crashed and start up again therefore can determine which key-ranges they are responsible for.

The choice of replica nodes as well as the assignment of nodes and key-ranges also affects durability: to face node failures, network partitions and even entire datacenter failures, the “the preference list of a key is constructed such that the storage nodes are spread across multiple datacenters”<sup>18</sup> (cf. [LM10, p. 3]).

### Cluster Membership and Failure Detection

The membership of servers in a Cassandra cluster is managed via a Gossip-style protocol named *Scuttlebutt* (cf. [vDGT08]) which is favored because of its “very efficient CPU utilization and very efficient utilization of the gossip channel” according to Lakshman and Malik. Besides membership management, Scuttlebutt is also used “to disseminate other system related control state” in a Cassandra cluster (cf. [LM10, p. 3], [Lak08]).

Nodes within a Cassandra cluster try to locally detect whether another node is up or down to avoid connection attempts to unreachable nodes. The mechanism employed for this purpose of failure detection is based on “a modified version of the  $\Phi$  Accrual Failure Detector”. The idea behind accrual failure detectors is that no boolean value is emitted by the failure detector but a “suspicion level for [...] monitored nodes” which indicates a probability about whether they are up or down<sup>19</sup>. Lakshman and Malik state that due to their experiences “Accrual Failure Detectors are very good in both their accuracy and their speed and they also adjust well to network conditions and server load conditions” (cf. [LM10, p. 3], [Lak08]).

**Cluster Management** According to the experiences at Facebook Lakshman and Malik assume that “[a] node outage rarely signifies a permanent departure and therefore should not result in re-balancing of the partition assignment or repair of the unreachable replicas”. Hence, nodes have to be added to and removed from a cluster explicitly by an administrator (cf. [LM10, p. 3]).

**Bootstrapping Nodes** When a node is added to a cluster, it calculates a random token for its position on the hash ring. This position as well as the key-range is responsible for is stored locally at the node as well as in ZooKeeper. The node then retrieves the addresses of a few nodes of the cluster from a configuration file or by a service like ZooKeeper and announces its arrival to them which in turn spread this information to the whole cluster. By such an announcement, membership information is spread throughout the cluster so that each node can receive requests for any key and route them to the appropriate server. As the arriving node splits a range of keys another server was formerly responsible for, this part of the key-range has to be transferred from the latter to the joining node (cf. [LM10, p. 3f], [Lak08]).

<sup>15</sup>  $N$  is the replication factor specified for the Cassandra instance.

<sup>16</sup> ZooKeeper is part of Apache’s Hadoop project and provides a “centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services” (cf. [Apa10f]).

<sup>17</sup> This metadata is called *preference list* in Amazon’s Dynamo which has a similar concept.

<sup>18</sup> The term *preference list* is borrowed from Amazon’s Dynamo which has a similar concept (see section 4.1.3 on page 53ff).

<sup>19</sup> For a  $\Phi$  accrual failure detector this probability is indicated by the value of  $\Phi$ , a logarithmically scaled factor.

## Persistence

In contrast to Bigtable and its derivatives, Cassandra persists its data to local files instead of a distributed file system. However, the data representation in memory and on disk as well as the processing of read and write operations is borrowed from Bigtable (cf. [LM10, p. 4f]):

- Write operations first go to a persistent commit log and then to an in-memory data structure.
- This in-memory data structure gets persisted to disk as an immutable file if it reaches a certain threshold of size.
- All writes to disk are sequential and an index is created “for efficient lookup based on row key” (like the block-indices of SSTables used in Bigtable).
- Data files on disk get compacted from time to time by a background process.
- Read operations consider the in-memory data structure as well as the data files persisted on disk. “In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory”. It “is first consulted to check if the key being looked up does indeed exist in the given file”.

In addition, Cassandra maintains indices for column families and columns to “jump to the right chunk on disk for column retrieval” and avoid the scanning of all columns on disk. As write operations go to an append-only commit log and as data files are immutable, “[the] server instance of Cassandra is practically lockless for read/write operations”.

### 6.3.5. Implementation

#### Cassandra Nodes

A server participating in a Cassandra cluster runs modules providing the following functionality:

- Partitioning
- Cluster membership and failure detection
- Storage engine

These modules are implemented in Java and operate on an event driven software layer whose “message processing pipeline and the task pipeline are split into multiple stages along the line of the SEDA [...] architecture” (cf. [WCB01]). The cluster the membership and failure module uses nonblocking I/O for communication. “All system control messages rely on UDP based messaging while the application related messages for replication and request routing relies on TCP” (cf. [LM10, p. 4]).

Request routing is implemented via a state machine on the storage nodes which consists of the following states when a request arrives (cf. [LM10, p. 4]):

1. Identification of the nodes that are in charge of the data for the requested key
2. Route the request to the nodes identified in state 1 and wait for their responses
3. Fail the request and return to the client if the nodes contacted in state 2 do not respond within a configured amount of time
4. “[Figure] out the latest response based on timestamp”
5. “[Schedule] a repair of the data at any replica if they do not have the latest piece of data”

The storage engine module can perform writes either synchronously or asynchronously which is configurable (cf. [LM10, p. 4]).

### Commit-Logs

The commit log maintained by Cassandra node locally has to be purged from entries that have already been committed and persisted to disk in the data files. Cassandra uses a rolling log file which is limited by size; at Facebook its threshold is set to 128MB. The commit log also contains a header in which bits are set whenever the in-memory data structure gets dumped to this. If a commit log is purged as it has reached its size limit, these bits are checked to make sure that all commits are persisted via immutable data files.

The commit log can be either written to in normal mode (i.e. synchronously) or in a *fast-sync mode* in which the writes are buffered; if the latter mode is used, the in-memory data gets also buffered before writing to disk. “This implies that there is a potential of data loss on machine crash”, Laksman and Malik state.

### Data Files

Data files persisted to disk are partitioned into blocks containing data for 128 row-keys. These blocks are “demarcated by a block index” which “captures the relative offset of a key within the block and the size of its data”. To accelerate access to blocks, their indices are cached in memory. If a key needs to be read and the corresponding block index is not already in memory, the data files are read in reverse time order and their block indexes get loaded into memory.

#### 6.3.6. Lessons Learned

After three years of designing, implementing and operating Cassandra at Facebook, Lakshman and Malik mention the following lessons they have learned in this period:

**Cautious Feature Addition** Regarding the addition of new features Lakshman and Malik confirm experiences also made by Google with Bigtable, namely “not to add any new feature without understanding the effects of its usage by applications”.

**Transactions** Lakshman and Malik furthermore confirm the statement of Chang et al. that for most applications atomic operations per row are sufficient and general transactions have mainly been required for maintaining secondary indices—which can be addressed by specialized mechanisms for just this purpose.

**Failure Detection** Lakshman and Malik have tested various failure detection implementations and discovered that the time to detect a failure can increase “beyond an acceptable limit as the size of the cluster” grows; in an experiment with a cluster consisting of 100 nodes some failure detectors needed up to two minutes to discover a failed node. However, the accrual failure detector featured acceptable detection times (in the aforementioned experiment: 15 seconds).

**Monitoring** The experiences at Facebook confirm those at Google that monitoring is key to “understand the behavior of the system when subject to production workload” as well as detect errors like disks failing for non-apparent reasons. Cassandra is integrated with the distributed monitoring service Ganglia.

**Partial Centralization** Lakshman and Malik state that centralized components like ZooKeeper are useful as “having some amount of coordination is essential to making the implementation of some distributed features tractable”.

## 7. Conclusion

The aim of this paper was to give a thorough overview and introduction to the NoSQL database movement which appeared in the recent years to provide alternatives to the predominant relational database management systems. Chapter 2 discussed reasons, rationales and motives for the development and usage of nonrelational database systems. These can be summarized by the need for high scalability, the processing of large amounts of data, the ability to distribute data among many (often commodity) servers, consequently a distribution-aware design of DBMSs (instead of adding such facilities on top) as well as a smooth integration with programming languages and their data structures (instead of e.g. costly object-relational mapping). As shown in chapter 2, relational DBMSs have certain flaws and limitations regarding these requirements as they were designed in a time where hardware (especially main-memory) was expensive and full dynamic querying was expected to be the most important use case; as shown by Stonebraker et al. the situation today is very different, so a complete redesign of database management systems is suggested. Because of the limitations of relational DBMSs and today's needs, a wide range of non-relational datastores has emerged. Chapter 2 outlines several attempts to classify and characterize them.

Chapter 3 introduced concepts, techniques and patterns that are commonly used by NoSQL databases to address consistency, partitioning, storage layout, querying, and distributed data processing. Important concepts in this field—like eventual consistency and ACID vs. BASE transaction characteristics—have been discussed along with a number of notable techniques such as multi-version storage, vector clocks, state vs. operational transfer models, consistent hashing, MapReduce, and row-based vs. columnar vs. log-structured merge tree persistence.

As a first class of NoSQL databases, key-/value-stores have been examined in chapter 4. Most of these datastores heavily borrow from Amazon's Dynamo, a proprietary, fully distributed, eventual consistent key-/value-store which has been discussed in detail in this paper. The chapter also looked at popular open-source key-/value-stores like Project Voldemort, Tokyo Cabinet/Tyrant, Redis as well as MemcacheDB.

Chapter 5 has discussed document stores by observing CouchDB and MongoDB as the two major representatives of this class of NoSQL databases. These document stores provide the abstraction of documents which are flat or nested namespaces for key-/value-pairs. CouchDB is a document store written in Erlang and accessible via a RESTful HTTP-interface providing multi-version concurrency control and replication between servers. MongoDB is a datastore with additional features such as nested documents, rich dynamic querying capabilities and automatic sharding.

In chapter 6 column-stores have been discussed as a third class of NoSQL databases. Besides pure column-stores for analytics datastores integrating column- and row-orientation can be subsumed in this field. An important representative of the latter is Google's Bigtable which allows to store multidimensional maps indexed by row, column-family, column and timestamp. Via a central master server, Bigtable automatically partitions and distributes data among multiple tablet servers of a cluster. The design and implementation of the proprietary Bigtable have been adopted by open-source projects like Hypertable and HBase. The chapter concludes with an examination of Apache Cassandra which integrates the full-distribution and eventual consistency of Amazon's Dynamo with the data model of Google's Bigtable.

## A. Further Reading, Listening and Watching

In addition to the references in the bibliography of this paper, the following resources are suggested to the interested reader.

### SQL vs. NoSQL databases

- *One Size Fits All: An Idea whose Time has Come and Gone* by Michael Stonebraker and Uğur Çetintemel:  
[http://www.cs.brown.edu/~ugur/fits\\_all.pdf](http://www.cs.brown.edu/~ugur/fits_all.pdf)
- *What Should I do? – Choosing SQL, NoSQL or Both for Scalable Web Apps* by Todd Hoff:  
<http://voltdb.com/webcast-choosing-sql-nosql-or-both-scalable-web-apps>
- *SQL Databases Don't Scale* by Adam Wiggins:  
[http://adam.heroku.com/past/2009/7/6/sql\\_databases\\_dont\\_scale/](http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/)
- *6 Reasons Why Relational Database Will Be Superseded* by Robin Bloor:  
<http://www.havemacwillblog.com/2008/11/6-reasons-why-relational-database-will-be-superseded/>

### Concepts, Techniques and Patterns

#### Introduction and Overview

- *Scalability, Availability & Stability Patterns* by Jonas Bonér:  
<http://www.slideshare.net/jboner/scalability-availability-stability-patterns>
- *Architecting for the Cloud – Horizontal Scalability via Transient, Shardable, Share-Nothing Resources* by Adam Wiggins:  
<http://www.infoq.com/presentations/Horizontal-Scalability>
- *Cluster-based scalable network services* by Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer and Paul Gauthier:  
<http://www.cs.berkeley.edu/~brewer/cs262b/TACC.pdf>

#### CAP-Theorem, BASE and Eventual Consistency

- *Brewer's CAP Theorem* by Julian Browne:  
<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>
- *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services* by Seth Gilbert and Nancy Lynch:  
<http://www.cs.utsa.edu/~shxu/CS6393-Fall2007/presentation/paper-18.pdf>

- *Errors in Database Systems, Eventual Consistency, and the CAP Theorem* by Michael Stonebraker:  
<http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext>
- *BASE: An Acid Alternative* by Dan Pritchett:  
<http://queue.acm.org/detail.cfm?id=1394128>
- *Eventually consistent* by Werner Vogels:  
[http://www.allthingsdistributed.com/2007/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2007/12/eventually_consistent.html)  
[http://www.allthingsdistributed.com/2008/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)
- *I love eventual consistency but...* by James Hamilton:  
<http://perspectives.mvdirona.com/2010/02/24/ILoveEventualConsistencyBut.aspx>

## Paxos Consensus Protocol

- *Paxos made simple* by Leslie Lamport:  
<http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf>
- *Time, clocks, and the ordering of events in a distributed system* by Leslie Lamport:  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.3682&rep=rep1&type=pdf>
- *The part-time parliament* by Leslie Lamport:  
<http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf>

## Chord

- *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications* by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan:  
<http://www.sigcomm.org/sigcomm2001/p12-stoica.pdf>
- *The Chord/DHash Project*:  
<http://pdos.csail.mit.edu/chord/>

## MapReduce (Critical Remarks)

- *A Comparison of Approaches to Large-Scale Data Analysis* by Michael Stonebraker, Andrew Pavlo, Erik Paulson et al.:  
<http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf>
- *MapReduce: A major step backwards* by David DeWitt:  
<http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/>

## NoSQL

### Overview and Introduction

- Directory of NoSQL databases with basic information on the individual datastores:  
<http://nosql-database.org/>
- Software Engineering Radio Podcast, Episode 165: *NoSQL and MongoDB with Dwight Merriman* by Robert Blumen and Dwight Merriman:  
<http://www.se-radio.net/2010/07/episode-165-nosql-and-mongodb-with-dwight-merriman/>
- heise SoftwareArchitekTOUR Podcast (German), Episode 22: *NoSQL – Alternative zu relationalen Datenbanken* by Markus Völter, Stefan Tilkov and Mathias Meyer:  
<http://www.heise.de/developer/artikel/Episode-22-NoSQL-Alternative-zu-relationalen-Datenbanken-1027769.html>
- RadioTux Binärgewitter Podcast (German), Episode 1: *NoSQL* by Dirk Deimeke, Marc Seeger, Sven Pfleiderer and Ingo Ebel:  
<http://blog.radiotux.de/2011/01/09/binaergewitter-1-nosql/>
- *NoSQL: Distributed and Scalable Non-Relational Database Systems* by Jeremy Zawodny:  
<http://www.linux-mag.com/id/7579>
- freiesMagazin (German), issue 08/2010: *NoSQL – Jenseits der relationalen Datenbanken* by Jochen Schnelle:  
[http://www.freiesmagazin.de/mobil/freiesMagazin-2010-08-bilder.html#10\\_08\\_nosql](http://www.freiesmagazin.de/mobil/freiesMagazin-2010-08-bilder.html#10_08_nosql)

### Key-/Value Stores

- *Amazon Dynamo: The Next Generation Of Virtual Distributed Storage* by Alex Iskold:  
[http://www.readwriteweb.com/archives/amazon\\_dynamo.php](http://www.readwriteweb.com/archives/amazon_dynamo.php)
- Software Engineering Radio Podcast, Episode 162: *Project Voldemort with Jay Kreps* by Robert Blumen and Jay Kreps:  
<http://www.se-radio.net/2010/05/episode-162-project-voldemort-with-jay-kreps/>
- *Erlang eXchange 2008: Building a Transactional Data Store* (Scalaris) by Alexander Reinefeld:  
<http://video.google.com/videoplay?docid=6981137233069932108>
- *Why you won't be building your killer app on a distributed hash table* by Jonathan Ellis:  
<http://spyced.blogspot.com/2009/05/why-you-wont-be-building-your-killer.html>

### Document Databases

- freiesMagazin (German), issue 06/2010: *CouchDB – Datenbank mal anders* by Jochen Schnelle:  
[http://www.freiesmagazin.de/mobil/freiesMagazin-2010-06-bilder.html#10\\_06\\_couchdb](http://www.freiesmagazin.de/mobil/freiesMagazin-2010-06-bilder.html#10_06_couchdb)
- *Introducing MongoDB* by Eliot Horowitz:  
<http://www.linux-mag.com/id/7530>

## Column Stores

- *Distinguishing Two Major Types of Column-Stores* by Daniel Abadi:  
[http://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of\\_29.html](http://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of_29.html)
- *Cassandra – Structured Storage System over a P2P Network* by Avinash Lakshman, Prashant Malik and Karthik Ranganathan:  
<http://www.slideshare.net/jhammrb/data-presentations-cassandra-sigmod>
- *4 Months with Cassandra, a love story* by Cloudkick:  
[https://www.cloudkick.com/blog/2010/mar/02/4\\_months\\_with\\_cassandra/](https://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra/)
- *Saying Yes To NoSQL; Going Steady With Cassandra At Digg* by John Quinn:  
<http://about.digg.com/node/564>
- *up and running with cassandra* by Evan Weaver:  
<http://blog.evanweaver.com/2009/07/06/up-and-running-with-cassandra/>
- techZing! Podcast, Episode 8: *Dude, Where's My Database?!* by Justin Vincent, Jason Roberts and Jonathan Ellis:  
<http://techzinglive.com/page/75/techzing-8-dude-wheres-my-database>
- *Cassandra: Fact vs fiction* by Jonathan Ellis:  
<http://spyced.blogspot.com/2010/04/cassandra-fact-vs-fiction.html>
- *Why we're using HBase* by Cosmin Lehene:  
<http://hstack.org/why-were-using-hbase-part-1/>  
<http://hstack.org/why-were-using-hbase-part-2/>

## Graph Databases

- *Neo4j - A Graph Database That Kicks Butt* by Todd Hoff:  
<http://highscalability.com/blog/2009/6/13/neo4j-a-graph-database-that-kicks-butt>
- JAXenter (German): *Graphendatenbanken, NoSQL und Neo4j* by Peter Neubauer:  
<http://it-republik.de/jaxenter/artikel/Graphendatenbanken-NoSQL-und-Neo4j-2906.html>
- JAXenter (German): *Neo4j - die High-Performance-Graphendatenbank* by Peter Neubauer:  
<http://it-republik.de/jaxenter/artikel/Neo4j-%96-die-High-Performance-Graphendatenbank-2919.html>
- *Presentation: Graphs && Neo4j => teh awesome!* by Alex Popescu:  
<http://nosql.mypopescu.com/post/342947902/presentation-graphs-neo4j-teh-awesome>
- *Product: HyperGraphDB – A Graph Database* by Todd Hoff:  
<http://highscalability.com/blog/2010/1/26/product-hypergraphdb-a-graph-database.html>
- RadioTux (German): *Sendung über die GraphDB* by Alexander Oelling and Ingo Ebel:  
<http://blog.radiotux.de/2010/12/13/sendung-graphdb/>

## Conference Slides and Recordings

- NOSQL debrief San Francisco on 2009-06-11:  
<http://blog.oskarsson.nu/2009/06/nosql-debrief.html>  
<http://www.johnandcailin.com/blog/john/san-francisco-nosql-meetup>
- NoSQL Berlin on 2009-10-22:  
<http://www.nosqlberlin.de/>

## Evaluation and Comparison of NoSQL Databases

- *NoSQL Ecosystem* by Jonathan Ellis:  
<http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/>
- *NoSQL: If only it was that easy* by BJ Clark:  
<http://bjclark.me/2009/08/04/nosql-if-only-it-was-that-easy/>
- *The end of SQL and relational databases?* by David Intersimone:  
[http://blogs.computerworld.com/15510/the\\_end\\_of\\_sql\\_and\\_relational\\_databases\\_part\\_1\\_of\\_3](http://blogs.computerworld.com/15510/the_end_of_sql_and_relational_databases_part_1_of_3)  
[http://blogs.computerworld.com/15556/the\\_end\\_of\\_sql\\_and\\_relational\\_databases\\_part\\_2\\_of\\_3](http://blogs.computerworld.com/15556/the_end_of_sql_and_relational_databases_part_2_of_3)  
[http://blogs.computerworld.com/15641/the\\_end\\_of\\_sql\\_and\\_relational\\_databases\\_part\\_3\\_of\\_3](http://blogs.computerworld.com/15641/the_end_of_sql_and_relational_databases_part_3_of_3)
- *Performance comparison: key/value stores for language model counts* by Brendan O'Connor:  
<http://anyall.org/blog/2009/04/performance-comparison-keyvalue-stores-for-language-model-counts/>
- *Redis Performance on EC2 (aka weekend project coming)* by Michal Frackowiak:  
<http://michalfrackowiak.com/blog:redis-performance>
- *MySQL-Memcached or NOSQL Tokyo Tyrant* by Matt Yonkovit:  
<http://www.mysqlperformanceblog.com/2009/10/15/mysql-memcached-or-nosql-tokyo-tyrant-part-1/>  
[http://www.mysqlperformanceblog.com/2009/10/16/mysql\\_memcached\\_tyrant\\_part2/](http://www.mysqlperformanceblog.com/2009/10/16/mysql_memcached_tyrant_part2/)  
[http://www.mysqlperformanceblog.com/2009/10/19/mysql\\_memcached\\_tyrant\\_part3/](http://www.mysqlperformanceblog.com/2009/10/19/mysql_memcached_tyrant_part3/)
- *Redis vs MySQL vs Tokyo Tyrant (on EC2)* by Colin Howe:  
<http://colinhowe.wordpress.com/2009/04/27/redis-vs-mysql/>

## B. List of abbreviations

2PC	<b>T</b> wo- <b>p</b> hase <b>c</b> ommitt
ACID	<b>A</b> tomicity <b>C</b> onsistency <b>I</b> solation <b>D</b> urability
ACM	<b>A</b> sociation for <b>C</b> omputing <b>M</b> achinery
ADO	<b>A</b> ctive <b>X</b> <b>D</b> ata <b>O</b> bjects
aka	<b>a</b> lso <b>k</b> nown <b>a</b> s
API	<b>A</b> pplication <b>P</b> rogramming <b>I</b> nterface
BASE	<b>B</b> asically <b>A</b> vailable, <b>S</b> oft-State, <b>E</b> ventual <b>C</b> onsistency
BBC	<b>B</b> ritish <b>BC</b> orporation
BDB	<b>B</b> erkley <b>D</b> atabase
BLOB	<b>B</b> inary <b>L</b> arge <b>O</b> bject
CAP	<b>C</b> onsistency, <b>A</b> vailability, <b>P</b> artition <b>T</b> olerance
CEO	<b>C</b> hief <b>E</b> xecutive <b>O</b> fficer
CPU	<b>C</b> entral <b>P</b> rocessing <b>U</b> nit
CS	<b>C</b> omputer <b>S</b> cience
CTA	<b>C</b> onstrained tree <b>a</b> pplication
CTO	<b>C</b> hief <b>T</b> echnology <b>O</b> fficer
DNS	<b>D</b> omain <b>N</b> ame <b>S</b> ystem
DOI	<b>D</b> igital <b>O</b> bject <b>I</b> dentifier
DBA	<b>D</b> atabase <b>a</b> dministrator
EAV store	<b>E</b> ntity- <b>A</b> ttribute- <b>V</b> alue <b>s</b> tore
EC2	(Amazon's) <b>E</b> lastic <b>C</b> loud <b>C</b> omputing
EJB	<b>E</b> nterprise <b>J</b> ava <b>B</b> eans
Erlang OTP	<b>E</b> rlang <b>O</b> pen <b>T</b> elecommunication <b>P</b> latform
E-R Model	<b>E</b> ntity- <b>R</b> elationship <b>M</b> odel
FIFO	<b>F</b> irst <b>i</b> n, <b>f</b> irst <b>o</b> ut
GFS	<b>G</b> oogle <b>F</b> ile <b>S</b> ystem
GPL	<b>G</b> nu <b>G</b> eneral <b>P</b> ublic <b>L</b> icence
HA	<b>H</b> igh <b>A</b> vailability

---

HDFS	<b>H</b> adoop <b>D</b> istributed <b>F</b> ile <b>S</b> ystem
HQL	<b>H</b> ypertable <b>Q</b> uery <b>L</b> anguage
IBM	<b>I</b> nternational <b>Business <b>M</b>achines</b>
IEEE	<b>I</b> nstitute of <b>E</b> lectrical and <b>E</b> lectronics <b>E</b> ngineers
IETF	<b>I</b> nternet <b>E</b> ngineering <b>T</b> ask <b>F</b> orce
IO	<b>I</b> nput <b>O</b> utput
IP	<b>I</b> nternet <b>P</b> rotocol
JDBC	<b>J</b> ava <b>D</b> atabase <b>C</b> onnectivity
JPA	<b>J</b> ava <b>P</b> ersistence <b>API</b>
JSON	<b>J</b> ava <b>S</b> cript <b>O</b> bject <b>N</b> otation
JSR	<b>J</b> ava <b>S</b> pecification <b>R</b> equest
LINQ	<b>L</b> anguage <b>I</b> ntegrated <b>Q</b> uery
LRU	<b>L</b> east <b>r</b> ecently <b>u</b> sed
LSM(-Tree)	<b>L</b> og <b>S</b> tructured <b>M</b> erge ( <b>T</b> ree)
LZW	<b>L</b> empel- <b>Z</b> iv- <b>W</b> elch
MIT	<b>M</b> assachusetts <b>I</b> nstitute of <b>T</b> echnology
MVCC	<b>M</b> ulti- <b>v</b> ersion <b>c</b> oncurrency <b>c</b> ontrol
(Java) NIO	(Java) <b>N</b> ew <b>I</b> /O
ODBC	<b>O</b> pen <b>D</b> atabase <b>C</b> onnectivity
OLAP	<b>O</b> nline <b>A</b> nalysitical <b>P</b> rocessing
OLTP	<b>O</b> nline <b>T</b> ransaction <b>P</b> rocessing
OSCON	O'Reilly <b>O</b> pen <b>S</b> ource <b>C</b> onvention
OSS	<b>O</b> pen <b>S</b> ource <b>S</b> oftware
PCRE	<b>P</b> erl- <b>c</b> ompatible <b>R</b> egular <b>E</b> xpressions
PNUTS	(Yahoo!'s) <b>P</b> latform for <b>N</b> imble <b>U</b> niversal <b>T</b> able <b>S</b> torage
PODC	<b>P</b> rinciples <b>o</b> f distributed computing (ACM symposium)
RAC	(Oracle) <b>R</b> eal <b>A</b> pplication <b>C</b> luster
RAM	<b>R</b> andom <b>A</b> ccess <b>M</b> emory
RDF	<b>R</b> esource <b>D</b> escription <b>F</b> ramework
RDS	(Amazon) <b>R</b> elational <b>D</b> atabase <b>S</b> ervice
RDBMS	<b>R</b> elational <b>D</b> atabase <b>M</b> anagement <b>S</b> ystem
REST	<b>R</b> epresentational <b>S</b> tate <b>T</b> ransfer
RPC	<b>R</b> emote <b>P</b> rocedure <b>C</b> all

RoR	<b>Ruby on Rails</b>
RYOW	<b>R</b> ead <b>y</b> our <b>o</b> wn <b>w</b> rites (consistency model)
(Amazon) S3	(Amazon) <b>S</b> imple <b>S</b> torage <b>S</b> ervice
SAN	<b>S</b> torage <b>A</b> rea <b>N</b> etwork
SLA	<b>S</b> ervice <b>L</b> evel <b>A</b> greement
SMP	<b>S</b> ymmetric <b>m</b> ultiprocessing
SPOF	<b>S</b> ingle <b>p</b> oint <b>o</b> f failure
SSD	<b>S</b> olid <b>S</b> tate <b>D</b> isk
SQL	<b>S</b> tructured <b>Q</b> uery <b>L</b> anguage
TCP	<b>T</b> ransmission <b>C</b> ontrol <b>P</b>
TPC	<b>T</b> ransaction <b>P</b> erformance <b>P</b> rocessing <b>C</b> ouncil
US	<b>U</b> nited <b>S</b> tates
URI	<b>U</b> niform <b>R</b> esource <b>I</b> dentifier
URL	<b>U</b> niform <b>R</b> esource <b>L</b> ocator
XML	<b>E</b> xtensible <b>M</b> arkup <b>L</b> anguage

## C. Bibliography

- [10g10] 10GEN, INC: *mongoDB*. 2010. –  
<http://www.mongodb.org>
- [AGS08] AGUILERA, Marcos K. ; GOLAB, Wojciech ; SHAH, Mehul A.: A Practical Scalable Distributed B-Tree. In: *PVLDB '08: Proceedings of the VLDB Endowment* Vol. 1, VLDB Endowment, August 2008, p. 598–609. – Available online.  
<http://www.vldb.org/pvldb/1/1453922.pdf>
- [Aja09] AJATUS SOFTWARE: *The Future of Scalable Databases*. October 2009. – Blog post of 2009-10-08.  
<http://www.ajatus.in/2009/10/the-future-of-scalable-databases/>
- [Ake09] AKER, Brian: *Your guide to NoSQL*. November 2009. – Talk at OpenSQLCamp in November 2009.  
<http://www.youtube.com/watch?v=LhnGarRsKnA>
- [Ama10a] AMAZON.COM, INC.: *Amazon Simple Storage Service (Amazon S3)*. 2010. –  
<http://aws.amazon.com/s3/>
- [Ama10b] AMAZON.COM, INC.: *Amazon SimpleDB*. 2010. –  
<http://aws.amazon.com/simpledb/>
- [Apa09] APACHE SOFTWARE FOUNDATION: *Lucene*. 2009. –  
<http://lucene.apache.org/>
- [Apa10a] APACHE SOFTWARE FOUNDATION: *Apache CouchDB – Introduction*. 2008–2010. –  
<http://couchdb.apache.org/docs/intro.html>
- [Apa10b] APACHE SOFTWARE FOUNDATION: *Apache CouchDB – Technical Overview*. 2008–2010.  
–  
<http://couchdb.apache.org/docs/overview.html>
- [Apa10c] APACHE SOFTWARE FOUNDATION: *The CouchDB Project*. 2008–2010. –  
<http://couchdb.apache.org/>
- [Apa10d] APACHE SOFTWARE FOUNDATION: *The Apache Cassandra Project*. 2010. –  
<http://cassandra.apache.org/>
- [Apa10e] APACHE SOFTWARE FOUNDATION: *Introduction to CouchDB Views*. September 2010. –  
Wiki article, version 35 of 2010-09-08.  
[http://wiki.apache.org/couchdb/Introduction\\_to\\_CouchDB\\_views](http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views)
- [Apa10f] APACHE SOFTWARE FOUNDATION: *Welcome to Apache ZooKeeper!* 2010. –  
<http://hadoop.apache.org/zookeeper/>
- [Apa11] APACHE SOFTWARE FOUNDATION: *HBase*. 2011. –  
<http://hbase.apache.org/>

- [BCM<sup>+</sup>10] BANKER, Kyle ; CHODOROW, Kristina ; MERRIMAN, Dwight et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Tutorial*. August 2010. – Wiki article, version 22 of 2010-08-11.  
<http://www.mongodb.org/display/DOCS/Replica+Set+Tutorial>
- [Bem10] BEMMANN, Dennis: *Skalierbarkeit, Datenschutz und die Geschichte von StudiVZ*. January 2010. – Talk at Stuttgart Media University's Social Networks Day on 2010-01-22.  
<http://days.mi.hdm-stuttgart.de/soclnetsday09/skalierbarkeit-datenschutz-und-geschichte-von-studivz.wmv>
- [Bez93] BEZDEK, James C.: Fuzzy models—what are they, and why. In: *IEEE Transactions on Fuzzy Systems*, 1993, p. 1–6
- [BH05] BOX, Don ; HEJLSBERG, Anders: *The LINQ Project*. September 2005. –  
<http://msdn.microsoft.com/de-de/library/aa479865.aspx>
- [Blo70] BLOOM, Burton H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. In: *Communications of the ACM* 13 (1970), p. 422–426
- [Blo01] BLOCH, Joshua: *Effective Java – Programming Language Guide*. Amsterdam : Addison-Wesley Longman, 2001
- [BMH10] BANKER, Kyle ; MERRIMAN, Dwight ; HOROWITZ, Eliot: *mongoDB Manual – Admin Zone – Replication – Halted Replication*. August 2010. – Wiki article, version 18 of 2010-08-04.  
<http://www.mongodb.org/display/DOCS/Halted+Replication>
- [BO06] BISWAS, Rahul ; ORT, Ed: *The Java Persistence API – A Simpler Programming Model for Entity Persistence*. May 2006. –  
<http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>
- [Bre00] BREWER, Eric A.: *Towards Robust Distributed Systems*. Portland, Oregon, July 2000. – Keynote at the ACM Symposium on Principles of Distributed Computing (PODC) on 2000-07-19.  
<http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>
- [Bur06] BURROWS, Mike: The Chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2006 (OSDI '06), p. 335–350. – Also available online.  
<http://labs.google.com/papers/chubby-osdi06.pdf>
- [C<sup>+</sup>10] CZURA, Martin et al.: *CouchDB In The Wild*. 2008–2010. – Wiki article, version 97 of 2010-09-28.  
[http://wiki.apache.org/couchdb/CouchDB\\_in\\_the\\_wild](http://wiki.apache.org/couchdb/CouchDB_in_the_wild)
- [Can10a] CANONICAL LTD.: *ubuntu one*. 2008–2010. –  
<https://one.ubuntu.com/>
- [Can10b] CANONICAL LTD.: *ubuntu*. 2010. –  
<http://www.ubuntu.com/>
- [Can10c] CANONICAL LTD.: *UbuntuOne*. July 2010. – Wiki article, version 89 of 2010-07-21.  
<https://wiki.ubuntu.com/UbuntuOne>
- [Car11] CARLSON, Nicholas: Facebook Has More Than 600 Million Users, Goldman Tells Clients. In: *Business Insider* (2011)

- [Cat10] CATTELL, Rick: *High Performance Scalable Data Stores*. February 2010. – Article of 2010-02-22.  
<http://cattell.net/datastores/Datastores.pdf>
- [CB74] CHAMBERLIN, Donald D. ; BOYCE, Raymond F.: SEQUEL: A structured English query language. In: *SIGFIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. New York, NY, USA : ACM, 1974, p. 249–264
- [CB10] CHODOROW, Kristina ; BANKER, Kyle: *mongoDB manual – Databases*. Febrary 2010. – Wiki article, version 4 of 2010-02-04.  
<http://www.mongodb.org/display/DOCS/Databases>
- [CBH10a] CHODOROW, Kristina ; BANKER, Kyle ; HERNANDEZ, Scott: *mongoDB Manual – Collections*. June 2010. – Wiki article, version 5 of 2010-06-07.  
<http://www.mongodb.org/display/DOCS/Collections>
- [CBH<sup>+</sup>10b] CHODOROW, Kristina ; BANKER, Kyle ; HERNANDEZ, Scott et al.: *mongoDB Manual – Inserting*. August 2010. – Wiki article, version 12 of 2010-08-29.  
<http://www.mongodb.org/display/DOCS/Inserting>
- [CDG<sup>+</sup>06] CHANG, Fay ; DEAN, Jeffrey ; GHEMAWAT, Sanjay ; HSIEH, Wilson C. ; WALLACH, Deborah A. ; BURROWS, Mike ; CHANDRA, Tushar ; FIKES, Andrew ; GRUBER, Robert E.: *Bigtable: A Distributed Storage System for Structured Data*. November 2006. –  
<http://labs.google.com/papers/bigtable-osdi06.pdf>
- [CDM<sup>+</sup>10] CHODOROW, Kristina ; DIROLF, Mike ; MERRIMAN, Dwight et al.: *mongoDB Manual – GridFS*. April 2010. – Wiki article, version 13 of 2010-04-16.  
<http://www.mongodb.org/display/DOCS/GridFS>
- [CGR07] CHANDRA, Tushar D. ; GRIESEMER, Robert ; REDSTONE, Joshua: Paxos Made Live – An Engineering Perspective. In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing*. New York, NY, USA : ACM, 2007 (PODC '07), p. 398–407. – Also available online.  
[http://labs.google.com/papers/paxos\\_made\\_live.pdf](http://labs.google.com/papers/paxos_made_live.pdf)
- [CHD<sup>+</sup>10] CHODOROW, Kristina ; HOROWITZ, Eliot ; DIROLF, Mike et al.: *mongoDB Manual – Updating*. November 2010. – Wiki article, version 72 of 2010-11-12.  
<http://www.mongodb.org/display/DOCS/Updating>
- [CHM10a] CHODOROW, Kristina ; HOROWITZ, Eliot ; MERRIMAN, Dwight: *mongoDB Manual – Data Types and Conventions*. February 2010. – Wiki article, version 8 of 2010-02-25.  
<http://www.mongodb.org/display/DOCS/Data+Types+and+Conventions>
- [CHM<sup>+</sup>10b] CHODOROW, Kristina ; HOROWITZ, Eliot ; MERRIMAN, Dwight et al.: *mongoDB Manual – Database – Commands – List of Database Commands*. September 2010. – Wiki article, version 52 of 2010-09-01.  
<http://www.mongodb.org/display/DOCS/List+of+Database+Commands>
- [Chu09] CHU, Steve: *MemcacheDB*. January 2009. –  
<http://memcachedb.org/>
- [Cla09] CLARK, BJ: *NoSQL: If only it was that easy*. August 2009. – Blog post of 2009-08-04.  
<http://bjclark.me/2009/08/04/nosql-if-only-it-was-that-easy/>

- [CMB<sup>+</sup>10] CHODOROW, Kristina ; MERRIMAN, Dwight ; BANKER, Kyle et al.: *mongoDB Manual – Optimization*. November 2010. – Wiki article, version 17 of 2010-11-24.  
<http://www.mongodb.org/display/DOCS/Optimization>
- [CMH<sup>+</sup>10] CHODOROW, Kristina ; MERRIMAN, Dwight ; HOROWITZ, Eliot et al.: *mongoDB Manual – Querying*. August 2010. – Wiki article, version 23 of 2010-08-04.  
<http://www.mongodb.org/display/DOCS/Querying>
- [CNM<sup>+</sup>10] CHODOROW, Kristina ; NITZ, Ryan ; MERRIMAN, Dwight et al.: *mongoDB Manual – Removing*. March 2010. – Wiki article, version 10 of 2010-03-10.  
<http://www.mongodb.org/display/DOCS/Removing>
- [Cod70] CODD, Edgar F.: A Relational Model of Data for Large Shared Data Banks. In: *Communications of the ACM* 13 (1970), June, No. 6, p. 377–387
- [Com09a] COMPUTERWORLD: *No to SQL? Anti-database movement gains steam*. June 2009. –  
[http://www.computerworld.com/s/article/9135086/No\\_to\\_SQL\\_Anti\\_database\\_movement\\_gains\\_steam](http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam)
- [Com09b] COMPUTERWORLD: *Researchers: Databases still beat Google's MapReduce*. April 2009. –  
<http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9131526>
- [Cop10] COPELAND, Rick: *How Python, TurboGears, and MongoDB are Transforming SourceForge.net*. February 2010. – Presentation at PyCon in Atlanta on 2010-02-20.  
<http://us.pycon.org/2010/conference/schedule/event/110/>
- [Cro06] CROCKFORD, D. ; IETF (INTERNET ENGINEERING TASK FORCE) (Ed.): *The application/json Media Type for JavaScript Object Notation (JSON)*. July 2006. – RFC 4627 (Informational).  
<http://tools.ietf.org/html/rfc4627>
- [CRS<sup>+</sup>08] COOPER, Brian F. ; RAMAKRISHNAN, Raghu ; SRIVASTAVA, Utkarsh ; SILBERSTEIN, Adam ; BOHANNON, Philip ; JACOBSEN, Hans A. ; PUZ, Nick ; WEAVER, Daniel ; YERNENI, Ramana: PNUTS: Yahoo!'s hosted data serving platform. In: *Proc. VLDB Endow.* 1 (2008), August, No. 2, p. 1277–1288. – Also available online.  
<http://research.yahoo.com/files/pnnts.pdf>
- [DBC10] DIROLF, Mike ; BANKER, Kyle ; CHODOROW, Kristina: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Configuration – Adding a New Set Member*. November 2010. – Wiki article, version 5 of 2010-11-19.  
<http://www.mongodb.org/display/DOCS/Adding+a+New+Set+Member>
- [DG04] DEAN, Jeffrey ; GHEMAWAT, Sanjay: *MapReduce: simplified data processing on large clusters*. Berkeley, CA, USA, 2004. –  
<http://labs.google.com/papers/mapreduce-osdi04.pdf>
- [DHJ<sup>+</sup>07] DECANDIA, Giuseppe ; HASTORUN, Deniz ; JAMPANI, Madan ; KAKULAPATI, Gunavardhan ; LAKSHMAN, Avinash ; PILCHIN, Alex ; SIVASUBRAMANIAN, Swaminathan ; VOSSHALL, Peter ; VOGELS, Werner: *Dynamo: Amazon's Highly Available Key-value Store*. September 2007. –  
<http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf>
- [DKE06] DEMICHEL, Linda ; KEITH, Michael ; EJB 3.0 EXPERT GROUP: *JSR 220: Enterprise JavaBeans™, Version 3.0 – Java Persistence API*. May 2006. –  
<http://jcp.org/aboutJava/communityprocess/final/jsr220/>

- [Dzi10] DZIUBA, Ted: *I Can't Wait for NoSQL to Die*. March 2010. – Blogpost of 2010-03-04.  
<http://teddziuba.com/2010/03/i-cant-wait-for-nosql-to-die.html>
- [Eif09] EIFREM, Emil: *Neo4j – The Benefits of Graph Databases*. July 2009. – OSCON presentation.  
<http://www.slideshare.net/emileifrem/neo4j-the-benefits-of-graph-databases-oscon-2009>
- [Ell09a] ELLIS, Jonathan: *NoSQL Ecosystem*. November 2009. – Blog post of 2009-11-09.  
<http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/>
- [Ell09b] ELLIS, Jonathan: *Why you won't be building your killer app on a distributed hash table*. May 2009. – Blog post of 2009-05-27.  
<http://spyced.blogspot.com/2009/05/why-you-wont-be-building-your-killer.html>
- [Eri10] ERICSSON COMPUTER SCIENCE LABORATORY: *Erlang Programming Language, Official Website*. 2010. –  
<http://www.erlang.org/>
- [Eur09] EURE, Ian: *Looking to the future with Cassandra*. September 2009. – Blog post of 2009-09-09.  
<http://about.digg.com/blog/looking-future-cassandra>
- [Eva09a] EVANS, Eric: *NOSQL 2009*. May 2009. – Blog post of 2009-05-12.  
[http://blog.sym-link.com/2009/05/12/nosql\\_2009.html](http://blog.sym-link.com/2009/05/12/nosql_2009.html)
- [Eva09b] EVANS, Eric: *NoSQL: What's in a name?* October 2009. – Blog post of 2009-10-30.  
[http://www.deadcafe.org/2009/10/30/nosql\\_whats\\_in\\_a\\_name.html](http://www.deadcafe.org/2009/10/30/nosql_whats_in_a_name.html)
- [F+09] FITZPATRICK, Brad et al.: *memcached – Clients*. December 2009. – Wiki article of 2009-12-10.  
<http://code.google.com/p/memcached/wiki/Clients>
- [F+10a] FITZPATRICK, Brad et al.: *Memcached*. April 2010. –  
<http://memcached.org>
- [F+10b] FITZPATRICK, Brad et al.: *memcached – FAQ*. February 2010. – Wiki article of 2010-02-10.  
<http://code.google.com/p/memcached/wiki/FAQ>
- [F+10c] FITZPATRICK, Brad et al.: *Protocol*. March 2010. –  
<http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt>
- [FAL10a] FAL LABS: *Tokyo Cabinet: a modern implementation of DBM*. 2006–2010. –  
<http://1978th.net/tokyocabinet/>
- [FAL10b] FAL LABS: *Tokyo Tyrant: network interface of Tokyo Cabinet*. 2007–2010. –  
<http://fallabs.com/tokyotyrant/>
- [Far09] FARRELL, Enda: *Erlang at the BBC*. 2009. – Presentation at the Erlang Factory Conference in London.  
<http://www.erlang-factory.com/upload/presentations/147/EndaFarrell-ErlangFactory-London2009-ErlangattheBBC.pdf>
- [FGM<sup>+</sup>99] FIELDING, R. ; GETTYS, J. ; MOGUL, J. ; FRYSTYK, H. ; MASINTER, L. ; LEACH, P. ; BERNERS-LEE, T. ; IETF (INTERNET ENGINEERING TASK FORCE) (Ed.): *Hypertext Transfer Protocol – HTTP/1.1*. Juni 1999. – RFC 2616 (Draft Standard).  
<http://www.ietf.org/rfc/rfc2616.txt>
- [Fie00] FIELDING, Roy T.: *Architectural styles and the design of network-based software architectures*, University of California, Irvine, Diss., 2000

- [For10] FORBES, Dennis: *Getting Real about NoSQL and the SQL-Isn't-Scalable Lie*. March 2010. – Blog post of 2010-03-02.  
[http://www.yafla.com/dforbes/Getting\\_Real\\_about\\_NoSQL\\_and\\_the\\_SQL\\_Isnt\\_-Scalable\\_Lie/](http://www.yafla.com/dforbes/Getting_Real_about_NoSQL_and_the_SQL_Isnt_-Scalable_Lie/)
- [FRLL10] FERGUSON, Kevin ; RAGHUNATHAN, Vijay ; LEEDS, Randall ; LINDSAY, Shaun: *Lounge*. 2010. –  
<http://tilgovi.github.com/couchdb-lounge/>
- [GL03] GOBIOFF, Sanjay Ghemawat H. ; LEUNG, Shun-Tak: The Google File System. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), No. 5, p. 29–43. –  
<http://labs.google.com/papers/gfs-sosp2003.pdf>
- [Goo10a] GOOGLE INC.: *Google Code – protobuf*. 2010. –  
<http://code.google.com/p/protobuf/>
- [Goo10b] GOOGLE INC.: *Google Code – Protocol Buffers*. 2010. –  
<http://code.google.com/intl/de/apis/protocolbuffers/>
- [Gos07] GOSLING, James: *The Eight Fallacies of Distributed Computing*. 2007. –  
<http://blogs.sun.com/jag/resource/Fallacies.html>
- [Gra09] GRAY, Jonathan: *CAP Theorem*. August 2009. – Blog post of 2009-08-24.  
<http://devblog.streamy.com/2009/08/24/cap-theorem/>
- [Gri08] GRIESEMER, Robert: Parallelism by design: data analysis with sawzall. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York : ACM, 2008 (CGO '08), p. 3–3
- [Ham09] HAMILTON, David: *One size does not fit all*. November 2009. – Blog post of 2009-11-03.  
<http://perspectives.mvdirona.com/2009/11/03/OneSizeDoesNotFitAll.aspx>
- [HAMS08] HARIZOPOULOS, Stavros ; ABADI, Daniel J. ; MADDEN, Samuel ; STONEBRAKER, Michael: OLTP through the looking glass, and what we found there. In: *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2008, p. 981–992
- [Haz10] HAZEL, Philip: *PCRE - Perl-compatible regular expressions*. January 2010. – PCRE Man Pages, last updated 2010-01-03.  
<http://www.pcre.org/pcre.txt>
- [HCS<sup>+</sup>10] HOROWITZ, Eliot ; CHODOROW, Kristina ; STAPLE, Aaron et al.: *mongoDB Manual – Drivers*. November 2010. – Wiki article, version 92 of 2010-11-08.  
<http://www.mongodb.org/display/DOCS/Drivers>
- [Hey10] HEYMANN, Harry: *MongoDB at foursquare*. May 2010. – Presentation at MongoNYC in New York on 2010-05-21.  
<http://blip.tv/file/3704098>
- [HL10] HOROWITZ, Eliot ; LERNER, Alberto: *mongoDB Manual – Admin Zone – Sharding – Upgrading from a Non-Sharded System*. August 2010. – Wiki article, version 6 of 2010-08-05.  
<http://www.mongodb.org/display/DOCS/Upgrading+from+a+Non-Sharded+System>
- [HMC<sup>+</sup>10] HOROWITZ, Eliot ; MERRIMAN, Dwight ; CHODOROW, Kristina et al.: *mongoDB Manual – MapReduce*. November 2010. – Wiki article, version 80 of 2010-11-24.  
<http://www.mongodb.org/display/DOCS/MapReduce>

- [HMS<sup>+10</sup>] HOROWITZ, Eliot ; MERRIMAN, Dwight ; STEARN, Mathias et al.: *mongoDB Manual – Indexes – Geospatial Indexing*. November 2010. – Wiki article, version 49 of 2010-11-03.  
<http://www.mongodb.org/display/DOCS/Geospatial+Indexing>
- [Ho09a] Ho, Ricky: *NOSQL Patterns*. November 2009. – Blog post of 2009-11-15.  
<http://horicky.blogspot.com/2009/11/nosql-patterns.html>
- [Ho09b] Ho, Ricky: *Query processing for NOSQL DB*. November 2009. – Blog post of 2009-11-28.  
<http://horicky.blogspot.com/2009/11/query-processing-for-nosql-db.html>
- [Hof09a] HOFF, Todd: *And the winner is: MySQL or Memcached or Tokyo Tyrant?* October 2009. – Blog post of 2009-10-28.  
<http://highscalability.com/blog/2009/10/28/and-the-winner-is-mysql-or-memcached-or-tokyo-tyrant.html>
- [Hof09b] HOFF, Todd: *Damn, which database do I use now?* November 2009. – Blog post of 2009-11-04.  
<http://highscalability.com/blog/2009/11/4/damn-which-database-do-i-use-now.html>
- [Hof09c] HOFF, Todd: *A Yes for a NoSQL Taxonomy*. November 2009. – Blog post of 2009-11-05.  
<http://highscalability.com/blog/2009/11/5/a-yes-for-a-nosql-taxonomy.html>
- [Hof10a] HOFF, Todd: *Facebook’s New Real-time Messaging System: HBase to Store 135+ Billion Messages a Month*. November 2010. – Blog post of 2010-11-16.  
<http://highscalability.com/blog/2010/11/16/facebook-s-new-real-time-messaging-system-hbase-to-store-135.html>
- [Hof10b] HOFF, Todd: *High Scalability – Entries In Memcached*. 2010. –  
<http://highscalability.com/blog/category/memcached>
- [Hof10c] HOFF, Todd: *MySQL and Memcached: End of an Era?* February 2010. – Blog post of 2010-02-26.  
<http://highscalability.com/blog/2010/2/26/mysql-and-memcached-end-of-an-era.html>
- [Hor10] HOROWITZ, Eliot: *mongoDB Manual – GridFS – When to use GridFS*. September 2010. – Wiki article, version 2 of 2010-09-19.  
<http://www.mongodb.org/display/DOCS/When+to+use+GridFS>
- [HR10] HEINEMEIER HANSSON, David ; RAILS CORE TEAM: *Ruby on Rails*. 2010. –  
<http://rubyonrails.org/>
- [Hyp09a] HYPERTABLE: *Hypertable*. 2009. –  
<http://www.hypertable.org/index.html>
- [Hyp09b] HYPERTABLE: *Hypertable – About Hypertable*. 2009. –  
<http://www.hypertable.org/about.html>
- [Hyp09c] HYPERTABLE: *Hypertable – Sponsors*. 2009. –  
<http://www.hypertable.org/sponsors.html>
- [Int10] INTERSIMONE, David: *The end of SQL and relational databases?* February 2010. – Blog posts of 2010-02-02, 2010-02-10 and 2010-02-24.  
[http://blogs.computerworld.com/15510/the\\_end\\_of\\_sql\\_and\\_relational\\_databases\\_part\\_1\\_of\\_3](http://blogs.computerworld.com/15510/the_end_of_sql_and_relational_databases_part_1_of_3), [http://blogs.computerworld.com/15556/the\\_end\\_of\\_sql\\_and\\_relational\\_databases\\_part\\_2\\_of\\_3](http://blogs.computerworld.com/15556/the_end_of_sql_and_relational_databases_part_2_of_3), [http://blogs.computerworld.com/15641/the\\_end\\_of\\_sql\\_and\\_relational\\_databases\\_part\\_3\\_of\\_3](http://blogs.computerworld.com/15641/the_end_of_sql_and_relational_databases_part_3_of_3)

- [Ipp09] IPPOLITO, Bob: *Drop ACID and think about Data*. March 2009. – Talk at Pycon on 2009-03-28.  
<http://blip.tv/file/1949416/>
- [JBo10a] JBOSS COMMUNITY: *HIBERNATE – Relational Persistence for Java & .NET*. 2010. – <http://www.hibernate.org/>
- [JBo10b] JBOSS COMMUNITY TEAM: *JBoss Cache*. 2010. – <http://jboss.org/jbosscache>
- [Jon09] JONES, Richard: *Anti-RDBMS: A list of distributed key-value stores*. January 2009. – Blog post of 2009-01-19.  
<http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores/>
- [Jud09] JUDD, Doug: *Hypertable*. June 2009. – Presentation at NoSQL meet-up in San Francisco on 2009-06-11.  
[http://static.last.fm/johan/nosql-20090611/hypertable\\_nosql.pdf](http://static.last.fm/johan/nosql-20090611/hypertable_nosql.pdf)
- [K<sup>+</sup>10a] KREPS, Jay et al.: *Project Voldemort – A distributed database*. 2010. – <http://project-voldemort.com/>
- [K<sup>+</sup>10b] KREPS, Jay et al.: *Project Voldemort – Design*. 2010. – <http://project-voldemort.com/design.php>
- [Key09] KEYS, Adam: *It's not NoSQL, it's post-relational*. August 2009. – Blog post of 2009-08-31.  
<http://therealadam.com/archive/2009/08/31/its-not-nosql-its-post-relational/>
- [KLL<sup>+</sup>97] KARGER, David ; LEHMAN, Eric ; LEIGHTON, Tom ; LEVINE, Matthew ; LEWIN, Daniel ; PANIGRAHY, Rina: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1997, p. 654–663
- [Lak08] LAKSHMAN, Avinash: *Cassandra - A structured storage system on a P2P Network*. August 2008. – Blog post of 2008-08-25.  
[http://www.facebook.com/note.php?note\\_id=24413138919](http://www.facebook.com/note.php?note_id=24413138919)
- [Lam98] LAMPORT, Leslie: The part-time parliament. In: *ACM Transactions on Computer Systems* 16 (1998), No. 2, p. 133–169. – Also available online.  
<http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf>
- [Lin09] LIN, Leonard: *Some Notes on distributed key value Stores*. April 2009. – Blog post of 2009-04-20.  
<http://randomfoo.net/2009/04/20/some-notes-on-distributed-key-stores>
- [Lip09] LIPCON, Todd: *Design Patterns for Distributed Non-Relational Databases*. June 2009. – Presentation of 2009-06-11.  
<http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases>
- [LM09] LAKSHMAN, Avinash ; MALIK, Prashant: *Cassandra – Structured Storage System over a P2P Network*. June 2009. – Presentation at NoSQL meet-up in San Francisco on 2009-06-11.  
[http://static.last.fm/johan/nosql-20090611/cassandra\\_nosql.pdf](http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf)

- [LM10] LAKSHMAN, Avinash ; MALIK, Prashant: Cassandra – A Decentralized Structured Storage System. In: *SIGOPS Operating Systems Review* 44 (2010), April, p. 35–40. – Also available online.  
<http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf>
- [Mah10] MAHER, Jacqueline: *Building a Better Submission Form*. May 2010. – Blog post of 2010-05-25.  
<http://open.blogs.nytimes.com/2010/05/25/building-a-better-submission-form/>
- [Mat89] MATTERN, Friedemann: Virtual Time and Global States of Distributed Systems. In: *Parallel and Distributed Algorithms*, North-Holland, 1989, p. 215–226. –  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.7435&rep=rep1&type=pdf>
- [MC09] MERRIMAN, Dwight ; CHODOROW, Kristina: *mongoDB Manual – Data Types and Conventions – Internationalized Strings*. July 2009. – Wiki article, version 2 of 2009-07-30.  
<http://www.mongodb.org/display/DOCS/Internationalized+Strings>
- [MC10a] MERRIMAN, Dwight ; CHODOROW, Kristina: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Sets Limits*. November 2010. – Wiki article, version 7 of 2010-11-11.  
<http://www.mongodb.org/display/DOCS/Replica+Sets+Limits>
- [MC10b] MERRIMAN, Dwight ; CHODOROW, Kristina: *mongoDB Manual – Optimization – Query Optimizer*. February 2010. – Wiki article, version 7 of 2010-02-24.  
<http://www.mongodb.org/display/DOCS/Query+Optimizer>
- [MCB10a] MERRIMAN, Dwight ; CHODOROW, Kristina ; BANKER, Kyle: *mongoDB Manual – Querying – min and max Query Specifiers*. February 2010. – Wiki article, version 6 of 2010-02-24.  
<http://www.mongodb.org/display/DOCS/min+and+max+Query+Specifiers>
- [MCB<sup>+</sup>10b] MERRIMAN, Dwight ; CHODOROW, Kristina ; BANKER, Kyle et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Configuration*. November 2010. – Wiki article, version 49 of 2010-11-18.  
<http://www.mongodb.org/display/DOCS/Replica+Set+Configuration>
- [MCD<sup>+</sup>10] MERRIMAN, Dwight ; CHODOROW, Kristina ; DIROLF, Mike et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets*. November 2010. – Wiki article, version 51 of 2010-11-12.  
<http://www.mongodb.org/display/DOCS/Replica+Sets>
- [MCH<sup>+</sup>10] MERRIMAN, Dwight ; CHODOROW, Kristina ; HOROWITZ, Eliot et al.: *mongoDB Manual – Admin Zone – Sharding – Configuring Sharding – A Sample Configuration Session*. September 2010. – Wiki article, version 16 of 2010-09-01.  
<http://www.mongodb.org/display/DOCS/A+Sample+Configuration+Session>
- [MCS<sup>+</sup>10] MERRIMAN, Dwight ; CHODOROW, Kristina ; STEARN, Mathias et al.: *mongoDB Manual – Updating – Atomic Operations*. October 2010. – Wiki article, version 42 of 2010-10-25.  
<http://www.mongodb.org/display/DOCS/Atomic+Operations>
- [MD10] MERRIMAN, Dwight ; DIROLF, Mike: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Design Concepts*. October 2010. – Wiki article, version 17 of 2010-10-09.  
<http://www.mongodb.org/display/DOCS/Replica+Set+Design+Concepts>
- [MDC<sup>+</sup>10] MERRIMAN, Dwight ; DIROLF, Mike ; CHODOROW, Kristina et al.: *mongoDB Manual – Data Types and Conventions – Database References*. September 2010. – Wiki article, version 45 of 2010-09-23.  
<http://www.mongodb.org/display/DOCS/Database+References>

- [MDH<sup>+</sup>10] MERRIMAN, Dwight ; DIROLF, Mike ; HOROWITZ, Eliot et al.: *mongoDB Manual – Admin Zone – Sharding – Sharding Introduction*. November 2010. – Wiki article, version 56 of 2010-11-06.  
<http://www.mongodb.org/display/DOCS/Sharding+Introduction>
- [MDM<sup>+</sup>10] MURPHY, Rian ; DIROLF, Mike ; MAGNUSSON JR, Geir et al.: *mongoDB Manual – Collections – Capped Collections*. October 2010. – Wiki article, version 22 of 2010-10-27.  
<http://www.mongodb.org/display/DOCS/Capped+Collections>
- [MDN<sup>+</sup>10] MERRIMAN, Dwight ; DIROLF, Mike ; NEGYESI, Karoly et al.: *mongoDB Manual – Admin Zone – Sharding – Sharding Limits*. November 2010. – Wiki article, version 37 of 2010-11-16.  
<http://www.mongodb.org/display/DOCS/Sharding+Limits>
- [Mer10a] MERRIMAN, Dwight: *mongoDB Manual – Admin Zone – Replication – About the local database*. August 2010. – Wiki article, version 6 of 2010-08-26.  
<http://www.mongodb.org/display/DOCS/About+the+local+database>
- [Mer10b] MERRIMAN, Dwight: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Data Center Awareness*. August 2010. – Wiki article, version 3 of 2010-08-30.  
<http://www.mongodb.org/display/DOCS/Data+Center+Awareness>
- [Mer10c] MERRIMAN, Dwight: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Admin UI*. August 2010. – Wiki article, version 8 of 2010-08-05.  
<http://www.mongodb.org/display/DOCS/Replica+Set+Admin+UI>
- [Mer10d] MERRIMAN, Dwight: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Resyncing a Very Stale Replica Set Member*. August 2010. – Wiki article, version 9 of 2010-08-17.  
<http://www.mongodb.org/display/DOCS/Resyncing+a+Very+Stale+Replica+Set+Member>
- [Mer10e] MERRIMAN, Dwight: *mongoDB Manual – Admin Zone – Replication – Replication Oplog Length*. August 2010. – Wiki article, version 9 of 2010-08-07.  
<http://www.mongodb.org/display/DOCS/Replication+Oplog+Length>
- [Mer10f] MERRIMAN, Dwight: *mongoDB Manual – Querying – Mongo Query Language*. July 2010. – Wiki article, version 1 of 2010-07-23.  
<http://www.mongodb.org/display/DOCS/Mongo+Query+Language>
- [Mer10g] MERRIMAN, Dwight: *On Distributed Consistency — Part 1*. March 2010. –  
<http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1>
- [MH10a] MERRIMAN, Dwight ; HERNANDEZ, Scott: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set FAQ*. August 2010. – Wiki article, version 4 of 2010-08-08.  
<http://www.mongodb.org/display/DOCS/Replica+Set+FAQ>
- [MH10b] MERRIMAN, Dwight ; HERNANDEZ, Scott: *mongoDB Manual – Indexes – Indexing as a Background Operation*. November 2010. – Wiki article, version 15 of 2010-11-16.  
<http://www.mongodb.org/display/DOCS/Indexing+as+a+Background+Operation>
- [MHB<sup>+</sup>10a] MERRIMAN, Dwight ; HOROWITZ, Eliot ; BANKER, Kyle et al.: *mongoDB Manual – Admin Zone – Sharding – Configuring Sharding*. November 2010. – Wiki article, version 64 of 2010-11-20.  
<http://www.mongodb.org/display/DOCS/Configuring+Sharding>

- [MHB<sup>+</sup>10b] MERRIMAN, Dwight ; HOROWITZ, Eliot ; BANKER, Kyle et al.: *mongoDB Manual – Admin Zone – Sharding – Sharding and Failover*. September 2010. – Wiki article, version 9 of 2010-09-05.  
<http://www.mongodb.org/display/DOCS/Sharding+and+Failover>
- [MHC<sup>+</sup>10a] MERRIMAN, Dwight ; HOROWITZ, Eliot ; CHODOROW, Kristina et al.: *mongoDB Manual – Querying – Dot Notation (Reaching into Objects)*. October 2010. – Wiki article, version 23 of 2010-10-25.  
[http://www.mongodb.org/display/DOCS/Dot+Notation+\(Reaching+into+Objects\)](http://www.mongodb.org/display/DOCS/Dot+Notation+(Reaching+into+Objects))
- [MHC<sup>+</sup>10b] MERRIMAN, Dwight ; HOROWITZ, Eliot ; CHODOROW, Kristina et al.: *mongoDB Manual – Sharding*. October 2010. – Wiki article, version 37 of 2010-10-21.  
<http://www.mongodb.org/display/DOCS/Sharding>
- [MHD<sup>+</sup>10] MERRIMAN, Dwight ; HOROWITZ, Eliot ; DIROLF, Mike et al.: *mongoDB Manual – Admin Zone – Replication – Replica Sets – Replica Set Internals*. November 2010. – Wiki article, version 43 of 2010-11-10.  
<http://www.mongodb.org/display/DOCS/Replica+Set+Internals>
- [MHH10] MERRIMAN, Dwight ; HOROWITZ, Eliot ; HERNANDEZ, Scott: *mongoDB Manual – Developer FAQ – How does concurrency work*. June 2010. – Wiki article, version 17 of 2010-06-28.  
<http://www.mongodb.org/display/DOCS/How+does+concurrency+work>
- [Mic10] MICROSOFT CORPORATION: *ADO.NET Entity Framework*. 2010. –  
<http://msdn.microsoft.com/en-us/library/bb399572.aspx>
- [MJS<sup>+</sup>10] MERRIMAN, Dwight ; JR, Geir M. ; STAPLE, Aaron et al.: *mongoDB Manual – Admin Zone – Replication – Master Slave*. October 2010. – Wiki article, version 55 of 2010-10-22.  
<http://www.mongodb.org/display/DOCS/Master+Slave>
- [MKB<sup>+</sup>10] MERRIMAN, Dwight ; KREUTER, Richard ; BANKER, Kyle et al.: *mongoDB Manual – Developer FAQ – SQL to Mongo Mapping Chart*. November 2010. – Wiki article, version 47 of 2010-11-07.  
<http://www.mongodb.org/display/DOCS/SQL+to+Mongo+Mapping+Chart>
- [MMC<sup>+</sup>10a] MERRIMAN, Dwight ; MAGNUSSON JR, Geir ; CHODOROW, Kristina et al.: *mongoDB Manual – Admin Zone – Security and Authentication*. November 2010. – Wiki article, version 31 of 2010-11-20.  
<http://www.mongodb.org/display/DOCS/Security+and+Authentication>
- [MMC<sup>+</sup>10b] MURPHY, Rian ; MAGNUSSON JR, Geir ; CHODOROW, Kristina et al.: *mongoDB Manual – Querying – Server-side Code Execution*. October 2010. – Wiki article, version 35 of 2010-10-24.  
<http://www.mongodb.org/display/DOCS/Server-side+Code+Execution>
- [MMD<sup>+</sup>10] MURPHY, Rian ; MAGNUSSON JR, Geir ; DIROLF, Mike et al.: *mongoDB Manual – Querying – Sorting and Natural Order*. February 2010. – Wiki article, version 14 of 2010-02-03.  
<http://www.mongodb.org/display/DOCS/Sorting+and+Natural+Order>
- [MMG<sup>+</sup>10] MURPHY, Rian ; MERRIMAN, Dwight ; GEIR MAGNUSSON JR et al.: *mongoDB Manual – Admin Zone – Replication*. August 2010. – Wiki article, version 83 of 2010-08-05.  
<http://www.mongodb.org/display/DOCS/Replication>
- [MMH<sup>+</sup>10] MURPHY, Rian ; MAGNUSSON JR, Geir ; HOROWITZ, Eliot et al.: *mongoDB Manual – Indexes*. November 2010. – Wiki article, version 55 of 2010-11-23.  
<http://www.mongodb.org/display/DOCS/Indexes>

- [MMM<sup>+</sup>10a] MURPHY, Rian ; MAGNUSSON JR, Geir ; MERRIMAN, Dwight et al.: *mongoDB Manual – Data Types and Conventions – Object IDs*. November 2010. – Wiki article, version 43 of 2010-11-01.  
<http://www.mongodb.org/display/DOCS/Object+IDs>
- [MMM<sup>+</sup>10b] MURPHY, Rian ; MAGNUSSON JR, Geir ; MERRIMAN, Dwight et al.: *mongoDB Manual – Inserting – Schema Design*. July 2010. – Wiki article, version 22 of 2010-07-19.  
<http://www.mongodb.org/display/DOCS/Schema+Design>
- [MMM<sup>+</sup>10c] MURPHY, Rian ; MAGNUSSON JR, Geir ; MERRIMAN, Dwight et al.: *mongoDB Manual – Querying – Advanced Queries*. December 2010. – Wiki article, version 118 of 2010-12-23.  
<http://www.mongodb.org/display/DOCS/Advanced+Queries>
- [MMM<sup>+</sup>10d] MURPHY, Rian ; MAGNUSSON JR, Geir ; MERRIMAN, Dwight et al.: *mongoDB Manual – Updating – Updating Data in Mongo*. September 2010. – Wiki article, version 28 of 2010-09-24.  
<http://www.mongodb.org/display/DOCS/Updating+Data+in+Mongo>
- [MMM<sup>+</sup>10e] MURPHY, Rian ; MERRIMAN, Dwight ; MAGNUSSON JR, Geir et al.: *mongoDB Manual – Databases – Commands*. September 2010. – Wiki article, version 60 of 2010-09-01.  
<http://www.mongodb.org/display/DOCS/Commands>
- [MMM<sup>+</sup>10f] MURPHY, Rian ; MERRIMAN, Dwight ; MAGNUSSON JR, Geir et al.: *mongoDB Manual – Indexes – Multikeys*. September 2010. – Wiki article, version 17 of 2010-09-22.  
<http://www.mongodb.org/display/DOCS/Multikeys>
- [MMM<sup>+</sup>10g] MURPHY, Rian ; MERRIMAN, Dwight ; MAGNUSSON JR, Geir et al.: *mongoDB manual – Querying – Aggregation*. September 2010. – Wiki article, version 43 of 2010-09-24.  
<http://www.mongodb.org/display/DOCS/Aggregation>
- [MMM<sup>+</sup>10h] MURPHY, Rian ; MERRIMAN, Dwight ; MAGNUSSON JR, Geir et al.: *mongoDB Manual – Querying – Queries and Cursors*. March 2010. – Wiki article, version 21 of 2010-03-01.  
<http://www.mongodb.org/display/DOCS/Queries+and+Cursors>
- [Mon10] MONGODB TEAM: *MongoDB 1.6 Released*. August 2010. – Blog post of 2010-08-05.  
<http://blog.mongodb.org/post/908172564/mongodb-1-6-released>
- [MS10] MERRIMAN, Dwight ; STEARN, Mathias: *mongoDB Manual – Querying – Retrieving a Subset of Fields*. July 2010. – Wiki article, version 12 of 2010-07-24.  
<http://www.mongodb.org/display/DOCS/Retrieving+a+Subset+of+Fields>
- [MSB<sup>+</sup>10] MERRIMAN, Dwight ; STEARN, Mathias ; BANKER, Kyle et al.: *mongoDB Manual – Updating – findAndModify Command*. October 2010. – Wiki article, version 25 of 2010-10-25.  
<http://www.mongodb.org/display/DOCS/findAndModify+Command>
- [MSL10] MERRIMAN, Dwight ; STEARN, Mathias ; LERNER, Alberto: *mongoDB Manual – Admin Zone – Sharding – Sharding Internals – Splitting Chunks*. August 2010. – Wiki article, version 5 of 2010-08-16.  
<http://www.mongodb.org/display/DOCS/Splitting+Chunks>
- [N<sup>+</sup>10] NEWSON, Robert et al.: *couchdb-lucene*. 2010. – github project.  
<http://github.com/rnewson/couchdb-lucene>
- [Nor09] NORTH, Ken: *Databases in the cloud*. September 2009. – Article in Dr. Dobb's Magazine.  
<http://www.drdobbs.com/database/218900502>

- [OAE<sup>+</sup>10] OUSTERHOUT, John ; AGRAWAL, Parag ; ERICKSON, David ; KOZYRAKIS, Christos ; LEVERICH, Jacob ; MAZIÈRES, David ; MITRA, Subhasish ; NARAYANAN, Aravind ; PARULKAR, Guru ; ROSENBLUM, Mendel ; M.RUMBLE, Stephen ; STRATMANN, Eric ; STUTSMAN, Ryan: The case for RAMClouds: scalable high-performance storage entirely in DRAM. In: *SIGOPS Oper. Syst. Rev.* 43 (2010), January, p. 92–105. – Available online. <http://www.stanford.edu/~ouster/cgi-bin/papers/ramcloud.pdf>
- [Oba09a] OBASANJO, Dare: *Building scalable databases: Denormalization, the NoSQL movement and Digg*. September 2009. – Blog post of 2009-09-10. <http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabases-DenormalizationTheNoSQLMovementAndDigg.aspx>
- [Oba09b] OBASANJO, Dare: *Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes*. January 2009. – Blog post of 2009-01-16. <http://www.25hoursaday.com/weblog/2009/01/16/BuildingScalableDatabasesProsAnd-ConsOfVariousDatabaseShardingSchemes.aspx>
- [OCGO96] O’NEIL, Patrick ; CHENG, Edward ; GAWLICK, Dieter ; O’NEIL, Elizabeth: The log-structured merge-tree (LSM-tree). In: *Acta Inf.* 33 (1996), No. 4, p. 351–385
- [Ora10a] ORACLE CORPORATION: *Interface Serializable*. 1993, 2010. – API documentation of the Java<sup>TM</sup> Platform Standard Edition 6. <http://download.oracle.com/javase/6/docs/api/java/io/Serializable.html>
- [Ora10b] ORACLE CORPORATION: *Java New I/O APIs*. 2004, 2010. – <http://download.oracle.com/javase/1.5.0/docs/guide/nio/index.html>
- [Ora10c] ORACLE CORPORATION: *MySQL*. 2010. – <http://www.mysql.com/>
- [Ora10d] ORACLE CORPORATION: *Oracle Berkeley DB Products*. 2010. – <http://www.oracle.com/us/products/database/berkeley-db/index.html>
- [Par09] PARALLEL & DISTRIBUTED OPERATING SYSTEMS GROUP: *The Chord/DHash Project – Overview*. November 2009. – <http://pdos.csail.mit.edu/chord/>
- [PDG<sup>+</sup>05] PIKE, Rob ; DORWARD, Sean ; GRIESEMER, Robert ; QUINLAN, Sean ; INC, Google: Interpreting the Data: Parallel Analysis with Sawzall. In: *Scientific Programming Journal* Vol. 13. Amsterdam : IOS Press, January 2005, p. 227–298. – Also available online. <http://research.google.com/archive/sawzall-sciprog.pdf>
- [PLL09] PRITLOVE, Tim ; LEHNARDT, Jan ; LANG, Alexander: *CouchDB – Die moderne Key/Value-Datenbank lädt Entwickler zum Entspannen ein*. June 2009. – Chaosradio Express Episode 125, Podcast published on 2009-06-10. <http://chaosradio.ccc.de/cre125.html>
- [Pop10a] POPESCU, Alex: *Cassandra at Twitter: An Interview with Ryan King*. Februry 2010. – Blog post of 2010-02-23. <http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king>
- [Pop10b] POPESCU, Alex: *Presentation: NoSQL at CodeMash – An Interesting NoSQL categorization*. February 2010. – Blog post of 2010-02-18. <http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interesting-nosql>

- [RGO06] ROTEM-GAL-OZ, Arnon: *Fallacies of Distributed Computing Explained*. 2006. – <http://www.rgoarchitects.com/Files/fallacies.pdf>
- [Ric10] RICHPRI: *diasporatest.com – MongoDB*. November 2010. – Wiki article, version 2 of 2010-11-16. <http://www.diasporatest.com/index.php/MongoDB>
- [Rid10] RIDGEWAY, Jay: *bit.ly user history, auto-sharded*. May 2010. – Presentation at MongoNYC in New York on 2010-05-21. <http://blip.tv/file/3704043>
- [RRHS04] RAMABHADRAN, Sriram ; RATNASAMY, Sylvia ; HELLERSTEIN, Joseph M. ; SHENKER, Scott: Prefix Hash Tree – An Indexing Data Structure over Distributed Hash Tables / IRB. 2004. – Forschungsbericht. – Available online. <http://berkeley.intel-research.net/sylvia/pht.pdf>
- [RV03] REYNOLDS, Patrick ; VAHDAT, Amin: Efficient Peer-to-Peer Keyword Searching. In: *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. New York, NY, USA : Springer-Verlag New York, Inc., 2003. – ISBN 3-540-40317-5, p. 21–40. – Available online. <http://issg.cs.duke.edu/search/search.pdf>
- [S<sup>+</sup>10] SANFILIPPO, Salvatore et al.: *redis*. 2010. – <http://code.google.com/p/redis/>
- [SBc<sup>+</sup>07] STONEBRAKER, Michael ; BEAR, Chuck ; ÇETINTEMEL, Uğur ; CHERNIACK, Mitch ; GE, Tingjian ; HACHEM, Nabil ; HARIZOPOULOS, Stavros ; LIFTER, John ; ROGERS, Jennie ; ZDONIK, Stan: One Size Fits All? – Part 2: Benchmarking Results. In: *Proc. CIDR*, 2007, p. 173–184
- [Sc05] STONEBRAKER, Michael ; ÇETINTEMEL, Uğur: One Size Fits All: An Idea whose Time has Come and Gone. In: *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2005, p. 2–11. – Also available online. [http://www.cs.brown.edu/~ugur/fits\\_all.pdf](http://www.cs.brown.edu/~ugur/fits_all.pdf)
- [Sch09] SCHMIDT, Stephan: *The dark side of NoSQL*. September 2009. – Blog post of 2009-09-30. <http://codemonkeyism.com/dark-side-nosql/>
- [Sch10] SCHMIDT, Stephan: *Why NOSQL Will Not Die*. March 2010. – Blog post of 2010-03-29. <http://codemonkeyism.com/nosql-die/>
- [Sco09] SCOFIELD, Ben: *NoSQL Misconceptions*. October 2009. – Blog post of 2009-10-21. <http://www.viget.com/extend/nosql-misconceptions/>
- [Sco10] SCOFIELD, Ben: *NoSQL – Death to Relational Databases(?)*. January 2010. – Presentation at the CodeMash conference in Sandusky (Ohio), 2010-01-14. <http://www.slideshare.net/bscofield/nosql-codemash-2010>
- [See09] SEEGER, Marc: *Key-Value stores: a practical overview*. September 2009. – Paper of 2009-09-21. [http://blog.marc-seeger.de/assets/papers/Ultra\\_Large\\_Sites\\_SS09-Seeger\\_Key\\_Value\\_Stores.pdf](http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_Stores.pdf)

- [Sha09a] SHALOM, Nati: *The Common Principles Behind The NOSQL Alternatives*. December 2009. – Blog post of 2009-12-15.  
[http://natishalom.typepad.com/nati\\_shaloms\\_blog/2009/12/the-common-principles-behind-the-nosql-alternatives.html](http://natishalom.typepad.com/nati_shaloms_blog/2009/12/the-common-principles-behind-the-nosql-alternatives.html)
- [Sha09b] SHALOM, Nati: *No to SQL? Anti-database movement gains steam – My Take*. July 2009. – Blog post of 2009-07-04.  
[http://natishalom.typepad.com/nati\\_shaloms\\_blog/2009/07/no-to-sql-anti-database-movement-gains-steam-my-take.html](http://natishalom.typepad.com/nati_shaloms_blog/2009/07/no-to-sql-anti-database-movement-gains-steam-my-take.html)
- [Sha09c] SHALOM, Nati: *Why Existing Databases (RAC) are So Breakable!* November 2009. – Blog post of 2009-11-30.  
[http://natishalom.typepad.com/nati\\_shaloms\\_blog/2009/11/why-existing-databases-rac-are-so-breakable.html](http://natishalom.typepad.com/nati_shaloms_blog/2009/11/why-existing-databases-rac-are-so-breakable.html)
- [SM10] STEARN, Mathias ; MERRIMAN, Dwight: *mongoDB Manual – Inserting - Trees in MongoDB*. March 2010. – Wiki article, version 21 of 2010-03-10.  
<http://www.mongodb.org/display/DOCS/Trees+in+MongoDB>
- [SMA<sup>+</sup>07] STONEBRAKER, Michael ; MADDEN, Samuel ; ABADI, Daniel J. ; HARIZOPOULOS, Stavros ; HACHEM, Nabil ; HELLMAND, Pat: The end of an architectural era: (it's time for a complete rewrite). In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007, p. 1150–1160
- [SMK<sup>+</sup>01] STOICA, Ion ; MORRIS, Robert ; KARGER, David ; KAASHOEK, M. F. ; BALAKRISHNAN, Hari: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: *Proceedings of the ACM SIGCOMM '01 Conference*. San Diego, California, United States, August 2001, p. 149–160. – Also available online.  
<http://www.sigcomm.org/sigcomm2001/p12-stoica.pdf>
- [Ste09] STEPHENS, Bradford: *Social Media Kills the Database*. June 2009. – Blog post of 2009-06.  
<http://www.roadtofailure.com/2009/06/19/social-media-kills-the-rdbms/>
- [Sto09] STONEBRAKER, Michael: *The “NoSQL” Discussion has Nothing to Do With SQL*. November 2009. – Blog post of 2009-11-04.  
<http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext>
- [Str10] STROZZI, Carlo: *NoSQL – A relational database management system*. 2007–2010. – [http://www.strozzi.it/cgi-bin/CSA/tw7/l/en\\_US/nosql/Home%20Page](http://www.strozzi.it/cgi-bin/CSA/tw7/l/en_US/nosql/Home%20Page)
- [Tay09] TAYLOR, Bret: *How Friendfeed uses MySQL*. February 2009. – Blog post of 2009-02-27.  
<http://bret.appspot.com/entry/how-friendfeed-uses-mysql>
- [Tec09] TECHNOLOGY REVIEW: *Designing for the cloud*. July/August 2009. – Interview with Dwight Merriman (CEO and cofounder of 10gen).  
<http://www.technologyreview.com/video/?vid=356>
- [vDGT08] VAN RENESSE, Robbert ; DUMITRIU, Dan ; GOUGH, Valient ; THOMAS, Chris: Efficient reconciliation and flow control for anti-entropy protocols. In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. New York, NY, USA : ACM, 2008 (LADIS '08), p. 6:1–6:7. – Also available online.  
<http://www.cs.cornell.edu/projects/ladis2008/materials/rvr.pdf>
- [Vog07] VOGELS, Werner: *Eventually consistent*. December 2007. – Blog post of 2007-12-19.  
[http://www.allthingsdistributed.com/2007/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2007/12/eventually_consistent.html)

- 
- [Vog08] VOGELS, Werner: *Eventually consistent*. December 2008. – Blog post of 2008-12-23.  
[http://www.allthingsdistributed.com/2008/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)
  - [WCB01] WELSH, Matt ; CULLER, David ; BREWER, Eric: SEDA: an architecture for well-conditioned, scalable internet services. In: *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2001 (SOSP '01), p. 230–243.  
– Also available online.  
<http://www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf>
  - [Whi07] WHITE, Tom: *Consistent Hashing*. November 2007. – Blog post of 2007-11-27.  
[http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent\\_hash.html](http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html)
  - [Wig09] WIGGINS, Adam: *SQL Databases Don't Scale*. June 2009. – Blog post of 2009-07-06.  
[http://adam.heroku.com/past/2009/7/6/sql\\_databases\\_dont\\_scale/](http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/)
  - [Wik10] WIKIPEDIA: *Fallacies of Distributed Computing*. 2010. – Wiki article, version of 2010-03-04.  
[http://en.wikipedia.org/wiki/Fallacies\\_of\\_Distributed\\_Computing](http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing)
  - [Wik11a] WIKIPEDIA: *HBase*. January 2011. – Wiki article of 2011-01-03.  
<http://en.wikipedia.org/wiki/HBase>
  - [Wik11b] WIKIPEDIA: *Hypertable*. January 2011. – Wiki article of 2011-01-21.  
<http://en.wikipedia.org/wiki/Hypertable>
  - [Yen09] YEN, Stephen: *NoSQL is a horseless carriage*. November 2009. –  
<http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf>

# Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Ion Stoica<sup>†</sup>, Robert Morris<sup>‡</sup>, David Liben-Nowell<sup>‡</sup>, David R. Karger<sup>‡</sup>, M. Frans Kaashoek<sup>‡</sup>, Frank Dabek<sup>‡</sup>, Hari Balakrishnan<sup>‡</sup>

## Abstract—

A fundamental problem that confronts peer-to-peer applications is the efficient location of the node that stores a desired data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis and simulations show that Chord is scalable: communication cost and the state maintained by each node scale logarithmically with the number of Chord nodes.

## I. INTRODUCTION

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, in which each node runs software with equivalent functionality. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is efficient location of data items. The contribution of this paper is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

The *Chord protocol* supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses consistent hashing [12] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and requires relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumes that each node is aware of most of the other nodes in the system, an approach that does not scale well to large numbers of nodes. In contrast, each Chord node needs “routing” information about only a few other nodes. Because the routing table is distributed, a Chord node communicates with other nodes in order to perform a lookup. In the steady state, in an  $N$ -node system, each node maintains information about only  $O(\log N)$  other nodes, and resolves all lookups via  $O(\log N)$  messages to other nodes. Chord maintains its routing information as nodes join and leave the sys-

tem.

A Chord node requires information about  $O(\log N)$  other nodes for *efficient* routing, but performance degrades gracefully when that information is out of date. This is important in practice because nodes will join and leave arbitrarily, and consistency of even  $O(\log N)$  state may be hard to maintain. Only one piece of information per node need be correct in order for Chord to guarantee correct (though possibly slow) routing of queries; Chord has a simple algorithm for maintaining this information in a dynamic environment.

The contributions of this paper are the Chord algorithm, the proof of its correctness, and simulation results demonstrating the strength of the algorithm. We also report some initial results on how the Chord routing protocol can be extended to take into account the physical network topology. Readers interested in an application of Chord and how Chord behaves on a small Internet testbed are referred to Dabek *et al.* [9]. The results reported by Dabek *et al.* are consistent with the simulation results presented in this paper.

The rest of this paper is structured as follows. Section II compares Chord to related work. Section III presents the system model that motivates the Chord protocol. Section IV presents the Chord protocol and proves several of its properties. Section V presents simulations supporting our claims about Chord’s performance. Finally, we summarize our contributions in Section VII.

## II. RELATED WORK

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

To clarify comparisons with related work, we will assume in this section a Chord-based application that maps keys onto values. A value can be an address, a document, or an arbitrary data item. A Chord-based application would store and find each value at the node to which the value’s key maps.

DNS provides a lookup service, with host names as keys and IP addresses (and other host information) as values. Chord could provide the same service by hashing each host name to a key [7]. Chord-based DNS would require no special servers, while ordinary DNS relies on a set of special root servers. DNS requires manual management of the routing information (NS records) that allows clients to navigate the name server hierarchy; Chord automatically maintains the correctness of the analogous routing information. DNS only works well when host names are structured to reflect administrative boundaries; Chord imposes no naming structure. DNS is specialized to the task of finding named hosts or services, while Chord can also be used to find

<sup>†</sup>University of California, Berkeley, [istoica@cs.berkeley.edu](mailto:istoica@cs.berkeley.edu)

<sup>‡</sup>MIT Laboratory for Computer Science, [{rtm, dln, karger, kaashoek, fdabek, hari}@lcs.mit.edu](mailto:{rtm, dln, karger, kaashoek, fdabek, hari}@lcs.mit.edu)

Authors in reverse alphabetical order.

data objects that are not tied to particular machines.

The Freenet peer-to-peer storage system [5], [6], like Chord, is decentralized and symmetric and automatically adapts when hosts leave and join. Freenet does not assign responsibility for documents to specific servers; instead, its lookups take the form of searches for cached copies. This allows Freenet to provide a degree of anonymity, but prevents it from guaranteeing retrieval of existing documents or from providing low bounds on retrieval costs. Chord does not provide anonymity, but its lookup operation runs in predictable time and always results in success or definitive failure.

The Ohaha system uses a consistent hashing-like algorithm map documents to nodes, and Freenet-style query routing [20]. As a result, it shares some of the weaknesses of Freenet. Archival Intermemory uses an off-line computed tree to map logical addresses to machines that store the data [4].

The Globe system [2] has a wide-area location service to map object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search shortcuts [25]. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

The distributed data location protocol developed by Plaxton *et al.* [21] is perhaps the closest algorithm to the Chord protocol. The Tapestry lookup protocol [26], used in OceanStore [13], is a variant of the Plaxton algorithm. Like Chord, it guarantees that queries make no more than a logarithmic number of hops and that keys are well-balanced. The Plaxton protocol's main advantage over Chord is that it ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. Chord, on the other hand, is substantially less complicated and handles concurrent node joins and failures well. Pastry [23] is a prefix-based lookup protocol that has properties similar to Chord. Like Tapestry, Pastry takes into account network topology to reduce the routing latency. However, Pastry achieves this at the cost of a more elaborated join protocol which initializes the routing table of the new node by using the information from nodes along the path traversed by the join message.

CAN uses a  $d$ -dimensional Cartesian coordinate space (for some fixed  $d$ ) to implement a distributed hash table that maps keys onto values [22]. Each node maintains  $O(d)$  state, and the lookup cost is  $O(dN^{1/d})$ . Thus, in contrast to Chord, the state maintained by a CAN node does not depend on the network size  $N$ , but the lookup cost increases faster than  $\log N$ . If  $d = \log N$ , CAN lookup times and storage needs match Chord's. However, CAN is not designed to vary  $d$  as  $N$  (and thus  $\log N$ ) varies, so this match will only occur for the “right”  $N$  corresponding to the fixed  $d$ . CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes. Chord also has the advantage that its correctness is robust in the face of partially incorrect routing information.

Chord's routing procedure may be thought of as a one-dimensional analogue of the Grid location system (GLS) [15]. GLS relies on real-world geographic location information to route its queries; Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to Grid's.

Napster [18] and Gnutella [11] provide a lookup operation to find data in a distributed set of peers. They search based on user-supplied keywords, while Chord looks up data with unique identifiers. Use of keyword search presents difficulties in both systems. Napster uses a central index, resulting in a single point of failure. Gnutella floods each query over the whole system, so its communication and processing costs are high in large systems.

Chord has been used as a basis for a number of subsequent research projects. The *Chord File System (CFS)* stores files and meta-data in a peer-to-peer system, using Chord to locate storage blocks [9]. New analysis techniques have shown that Chord's stabilization algorithms (with minor modifications) maintain good lookup performance despite continuous failure and joining of nodes [16]. Chord has been evaluated as a tool to serve DNS [7] and to maintain a distributed public key database for secure name resolution [1].

### III. SYSTEM MODEL

Chord simplifies the design of peer-to-peer systems and applications based on it by addressing these difficult problems:

- **Load balance:** Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.
- **Decentralization:** Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.
- **Scalability:** The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.
- **Availability:** Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.
- **Flexible naming:** Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

The Chord software takes the form of a library to be linked with the applications that use it. The application interacts with Chord in two main ways. First, the Chord library provides a `lookup(key)` function that yields the IP address of the node responsible for the key. Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for. This allows the application software to, for example, move corresponding values to their new homes when a new node joins.

The application using Chord is responsible for providing any desired authentication, caching, replication, and user-friendly

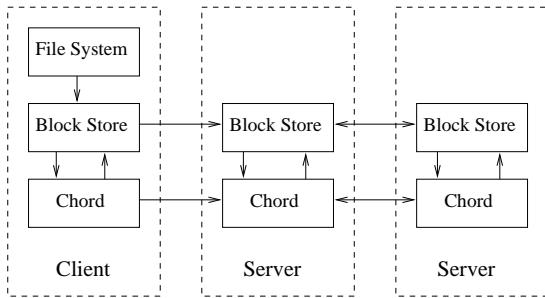


Fig. 1. Structure of an example Chord-based distributed storage system.

naming of data. Chord’s flat key-space eases the implementation of these features. For example, an application could authenticate data by storing it under a Chord key derived from a cryptographic hash of the data. Similarly, an application could replicate data by storing it under two distinct Chord keys derived from the data’s application-level identifier.

The following are examples of applications for which Chord can provide a good foundation:

**Cooperative mirroring**, in which multiple providers of content cooperate to store and serve each others’ data. The participants might, for example, be a set of software development projects, each of which makes periodic releases. Spreading the total load evenly over all participants’ hosts lowers the total cost of the system, since each participant need provide capacity only for the average load, not for that participant’s peak load. Dabek *et al.* describe a realization of this idea that uses Chord to map data blocks onto servers; the application interacts with Chord achieve load balance, data replication, and latency-based server selection [9].

**Time-shared storage** for nodes with intermittent connectivity. If someone wishes their data to be always available, but their server is only occasionally available, they can offer to store others’ data while they are connected, in return for having their data stored elsewhere when they are disconnected. The data’s name can serve as a key to identify the (live) Chord node responsible for storing the data item at any given time. Many of the same issues arise as in the cooperative mirroring application, though the focus here is on availability rather than load balance.

**Distributed indexes** to support Gnutella- or Napster-like keyword search. A key in this application could be derived from the desired keywords, while values could be lists of machines offering documents with those keywords.

**Large-scale combinatorial search**, such as code breaking. In this case keys are candidate solutions to the problem (such as cryptographic keys); Chord maps these keys to the machines responsible for testing them as solutions.

We have built several peer-to-peer applications using Chord. The structure of a typical application is shown in Figure 1. The highest layer implements application-specific functions such as file-system meta-data. The next layer implements a general-purpose distributed hash table that multiple applications use to insert and retrieve data blocks identified with unique keys. The distributed hash table takes care of storing, caching, and replication of blocks. The distributed hash table uses Chord to identify

the node responsible for storing a block, and then communicates with the block storage server on that node to read or write the block.

#### IV. THE CHORD PROTOCOL

This section describes the Chord protocol. The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes. In this paper we assume that communication in the underlying network is both symmetric (if  $A$  can route to  $B$ , then  $B$  can route to  $A$ ), and transitive (if  $A$  can route to  $B$  and  $B$  can route to  $C$ , then  $A$  can route to  $C$ ).

##### A. Overview

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. Chord assigns keys to nodes with *consistent hashing* [12], [14], which has several desirable properties. With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an  $N^{th}$  node joins (or leaves) the network, only a  $O(1/N)$  fraction of the keys are moved to a different location—this is clearly the minimum necessary to maintain a balanced load.

Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of “routing” information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with other nodes. In an  $N$ -node network, each node maintains information about only  $O(\log N)$  other nodes, and a lookup requires  $O(\log N)$  messages.

##### B. Consistent Hashing

The consistent hash function assigns each node and key an  $m$ -bit *identifier* using SHA-1 [10] as a base hash function. A node’s identifier is chosen by hashing the node’s IP address, while a key identifier is produced by hashing the key. We will use the term “key” to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term “node” will refer to both the node and its identifier under the hash function. The identifier length  $m$  must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered on an *identifier circle* modulo  $2^m$ . Key  $k$  is assigned to the first node whose identifier is equal to or follows (the identifier of)  $k$  in the identifier space. This node is called the *successor node* of key  $k$ , denoted by  $\text{successor}(k)$ . If identifiers are represented as a circle of numbers from 0 to  $2^m - 1$ , then  $\text{successor}(k)$  is the first node clockwise from  $k$ . In the remainder of this paper, we will also refer to the identifier circle as the *Chord ring*.

Figure 2 shows a Chord ring with  $m = 6$ . The Chord ring has 10 nodes and stores five keys. The successor of identifier 10 is node 14, so key 10 would be located at node 14. Similarly, keys 24 and 30 would be located at node 32, key 38 at node 38, and key 54 at node 56.

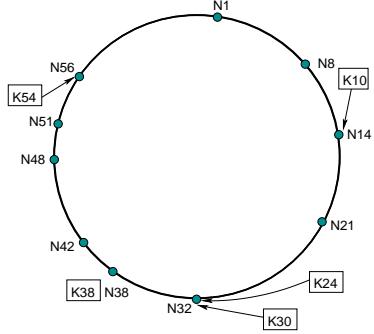


Fig. 2. An identifier circle (ring) consisting of 10 nodes storing five keys.

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node  $n$  joins the network, certain keys previously assigned to  $n$ 's successor now become assigned to  $n$ . When node  $n$  leaves the network, all of its assigned keys are reassigned to  $n$ 's successor. No other changes in assignment of keys to nodes need occur. In the example above, if a node were to join with identifier 26, it would capture the key with identifier 24 from the node with identifier 32.

The following results are proven in the papers that introduced consistent hashing [12], [14]:

*Theorem IV.1:* For any set of  $N$  nodes and  $K$  keys, with high probability:

1. Each node is responsible for at most  $(1 + \epsilon)K/N$  keys
2. When an  $(N + 1)^{st}$  node joins or leaves the network, responsibility for  $O(K/N)$  keys changes hands (and only to or from the joining or leaving node).

When consistent hashing is implemented as described above, the theorem proves a bound of  $\epsilon = O(\log N)$ . The consistent hashing paper shows that  $\epsilon$  can be reduced to an arbitrarily small constant by having each node run  $\Omega(\log N)$  virtual nodes, each with its own identifier. In the remainder of this paper, we will analyze all bounds in terms of work per virtual node. Thus, if each real node runs  $v$  virtual nodes, all bounds should be multiplied by  $v$ .

The phrase “with high probability” bears some discussion. A simple interpretation is that the nodes and keys are randomly chosen, which is plausible in a non-adversarial model of the world. The probability distribution is then over random choices of keys and nodes, and says that such a random choice is unlikely to produce an unbalanced distribution. A similar model is applied to analyze standard hashing. Standard hash functions distribute data well when the set of keys being hashed is random. When keys are not random, such a result cannot be guaranteed—indeed, for any hash function, there exists some key set that is terribly distributed by the hash function (e.g., the set of keys that all map to a single hash bucket). In practice, such potential bad sets are considered unlikely to arise. Techniques have also been developed [3] to introduce randomness in the hash function; given any set of keys, we can choose a hash function at random so that the keys are well distributed with high probability over the choice of hash function. A similar technique can be applied to consistent hashing; thus the “high probability” claim in the theorem above. Rather than select a random hash func-

tion, we make use of the SHA-1 hash which is expected to have good distributional properties.

Of course, once the random hash function has been chosen, an adversary can select a badly distributed set of keys for that hash function. In our application, an adversary can generate a large set of keys and insert into the Chord ring only those keys that map to a particular node, thus creating a badly distributed set of keys. As with standard hashing, however, we expect that a non-adversarial set of keys can be analyzed as if it were random. Using this assumption, we state many of our results below as “high probability” results.

### C. Simple Key Location

This section describes a simple but slow Chord lookup algorithm. Succeeding sections will describe how to extend the basic algorithm to increase efficiency, and how to maintain the correctness of Chord’s routing information.

Lookups could be implemented on a Chord ring with little per-node state. Each node need only know how to contact its current successor node on the identifier circle. Queries for a given identifier could be passed around the circle via these successor pointers until they encounter a pair of nodes that straddle the desired identifier; the second in the pair is the node the query maps to.

Figure 3(a) shows pseudocode that implements simple key lookup. Remote calls and variable references are preceded by the remote node identifier, while local variable references and procedure calls omit the local node. Thus  $n.\text{foo}()$  denotes a remote procedure call of procedure **foo** on node  $n$ , while  $n.\text{bar}$ , without parentheses, is an RPC to fetch a variable *bar* from node  $n$ . The notation  $(a, b]$  denotes the segment of the Chord ring obtained by moving clockwise from (but not including)  $a$  until reaching (and including)  $b$ .

Figure 3(b) shows an example in which node 8 performs a lookup for key 54. Node 8 invokes *find\_successor* for key 54 which eventually returns the successor of that key, node 56. The query visits every node on the circle between nodes 8 and 56. The result returns along the reverse of the path followed by the query.

### D. Scalable Key Location

The lookup scheme presented in the previous section uses a number of messages linear in the number of nodes. To accelerate lookups, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as each node knows its correct successor.

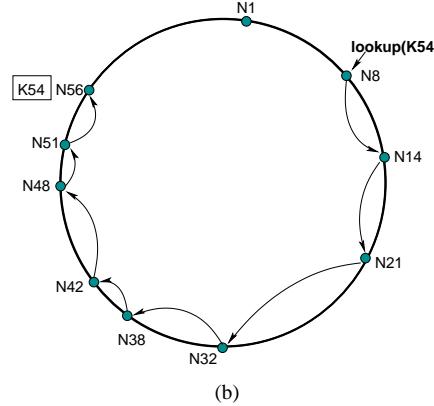
As before, let  $m$  be the number of bits in the key/node identifiers. Each node  $n$  maintains a routing table with up to  $m$  entries (we will see that in fact only  $O(\log n)$  are distinct), called the *finger table*. The  $i^{th}$  entry in the table at node  $n$  contains the identity of the *first* node  $s$  that succeeds  $n$  by at least  $2^{i-1}$  on the identifier circle, i.e.,  $s = \text{successor}(n + 2^{i-1})$ , where  $1 \leq i \leq m$  (and all arithmetic is modulo  $2^m$ ). We call node  $s$  the  $i^{th}$  finger of node  $n$ , and denote it by  $n.\text{finger}[i]$  (see Table I). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of  $n$  is the immediate successor of  $n$  on the circle; for convenience we often refer to the first finger as the *successor*.

```

// ask node n to find the successor of id
n.find_successor(id)
if (id ∈ (n, successor])
    return successor;
else
    //forward the query around the circle
    return successor.find_successor(id);

```

(a)



(b)

Fig. 3. (a) Simple (but slow) pseudocode to find the successor node of an identifier  $id$ . Remote procedure calls and variable lookups are preceded by the remote node. (b) The path taken by a query from node 8 for key 54, using the pseudocode in Figure 3(a).

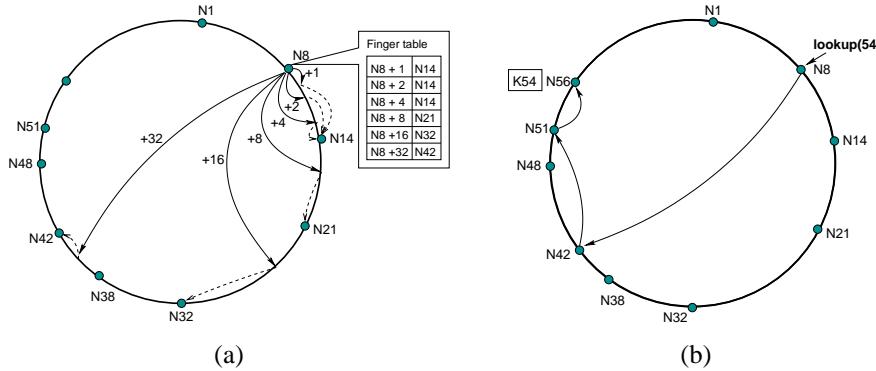


Fig. 4. (a) The finger table entries for node 8. (b) The path a query for key 54 starting at node 8, using the algorithm in Figure 5.

Notation	Definition
$\text{finger}[k]$	first node on circle that succeeds $(n + 2^{k-1}) \bmod 2^m$ , $1 \leq k \leq m$
$\text{successor}$	the next node on the identifier circle; $\text{finger}[1].\text{node}$
$\text{predecessor}$	the previous node on the identifier circle

TABLE I

Definition of variables for node  $n$ , using  $m$ -bit identifiers.

```

// ask node n to find the successor of id
n.find_successor(id)
if (id ∈ (n, successor])
    return successor;
else
    n' = closest_preceding_node(id);
    return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
for i = m downto 1
    if (finger[i] ∈ (n, id))
        return finger[i];
return n;

```

Fig. 5. Scalable key lookup using the finger table.

The example in Figure 4(a) shows the finger table of node 8. The first finger of node 8 points to node 14, as node 14 is the first node that succeeds  $(8 + 2^0) \bmod 2^6 = 9$ . Similarly, the last finger of node 8 points to node 42, as node 42 is the first node that succeeds  $(8 + 2^5) \bmod 2^6 = 40$ .

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to directly determine the successor of an arbitrary key  $k$ . For example, node 8 in Figure 4(a) cannot determine the successor of key 34 by itself, as this successor (node 38) does not appear in node 8's finger table.

Figure 5 shows the pseudocode of the  $\text{find\_successor}$  opera-

tion, extended to use finger tables. If  $id$  falls between  $n$  and its successor,  $\text{find\_successor}$  is finished and node  $n$  returns its successor. Otherwise,  $n$  searches its finger table for the node  $n'$  whose ID most immediately precedes  $id$ , and then invokes  $\text{find\_successor}$  at  $n'$ . The reason behind this choice of  $n'$  is that the closer  $n'$  is to  $id$ , the more it will know about the identifier circle in the region of  $id$ .

As an example, consider the Chord circle in Figure 4(b), and suppose node 8 wants to find the successor of key 54. Since the largest finger of node 8 that precedes 54 is node 42, node 8 will ask node 42 to resolve the query. In turn, node 42 will determine the largest finger in its finger table that precedes 54, i.e., node 51. Finally, node 51 will discover that its own successor, node

56, succeeds key 54, and thus will return node 56 to node 8.

Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier. From this intuition follows a theorem:

*Theorem IV.2:* With high probability, the number of nodes that must be contacted to find a successor in an  $N$ -node network is  $O(\log N)$ .

*Proof:* Suppose that node  $n$  wishes to resolve a query for the successor of  $k$ . Let  $p$  be the node that immediately precedes  $k$ . We analyze the number of query steps to reach  $p$ .

Recall that if  $n \neq p$ , then  $n$  forwards its query to the closest predecessor of  $k$  in its finger table. Consider the  $i$  such that node  $p$  is in the interval  $[n + 2^{i-1}, n + 2^i]$ . Since this interval is not empty (it contains  $p$ ), node  $n$  will contact its  $i^{\text{th}}$  finger, the first node  $f$  in this interval. The distance (number of identifiers) between  $n$  and  $f$  is at least  $2^{i-1}$ . But  $f$  and  $p$  are both in the interval  $[n + 2^{i-1}, n + 2^i]$ , which means the distance between them is at most  $2^{i-1}$ . This means  $f$  is closer to  $p$  than to  $n$ , or equivalently, that the distance from  $f$  to  $p$  is at most half the distance from  $n$  to  $p$ .

If the distance between the node handling the query and the predecessor  $p$  halves in each step, and is at most  $2^m$  initially, then within  $m$  steps the distance will be one, meaning we have arrived at  $p$ .

In fact, as discussed above, we assume that node and key identifiers are random. In this case, the number of forwardings necessary will be  $O(\log N)$  with high probability. After  $2 \log N$  forwardings, the distance between the current query node and the key  $k$  will be reduced to at most  $2^m/N^2$ . The probability that any other node is in this interval is at most  $1/N$ , which is negligible. Thus, the next forwarding step will find the desired node. ■

In the section reporting our experimental results (Section V), we will observe (and justify) that the average lookup time is  $\frac{1}{2} \log N$ .

Although the finger table contains room for  $m$  entries, in fact only  $O(\log N)$  fingers need be stored. As we just argued in the above proof, no node is likely to be within distance  $2^m/N^2$  of any other node. Thus, the  $i^{\text{th}}$  finger of the node, for any  $i \leq m - 2 \log N$ , will be equal to the node's immediate successor with high probability and need not be stored separately.

### E. Dynamic Operations and Failures

In practice, Chord needs to deal with nodes joining the system and with nodes that fail or leave voluntarily. This section describes how Chord handles these situations.

#### E.1 Node Joins and Stabilization

In order to ensure that lookups execute correctly as the set of participating nodes changes, Chord must ensure that each node's successor pointer is up to date. It does this using a "stabilization" protocol that each node runs periodically in the background and which updates Chord's finger tables and successor pointers.

Figure 6 shows the pseudocode for joins and stabilization. When node  $n$  first starts, it calls  $n.join(n')$ , where  $n'$  is any

```

// create a new Chord ring.
n.create()
  predecessor = nil;
  successor = n;

// join a Chord ring containing node n'.
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
    successor = x;
  successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
  next = next + 1;
  if (next > m)
    next = 1;
  finger[next] = find_successor(n + 2^{ext-1});

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
  if (predecessor has failed)
    predecessor = nil;

```

Fig. 6. Pseudocode for stabilization.

known Chord node, or  $n.create()$  to create a new Chord network. The  $join()$  function asks  $n'$  to find the immediate successor of  $n$ . By itself,  $join()$  does not make the rest of the network aware of  $n$ .

Every node runs  $stabilize()$  periodically to learn about newly joined nodes. Each time node  $n$  runs  $stabilize()$ , it asks its successor for the successor's predecessor  $p$ , and decides whether  $p$  should be  $n$ 's successor instead. This would be the case if node  $p$  recently joined the system. In addition,  $stabilize()$  notifies node  $n$ 's successor of  $n$ 's existence, giving the successor the chance to change its predecessor to  $n$ . The successor does this only if it knows of no closer predecessor than  $n$ .

Each node periodically calls  $fix\_fingers$  to make sure its finger table entries are correct; this is how new nodes initialize their finger tables, and it is how existing nodes incorporate new nodes into their finger tables. Each node also runs  $check\_predecessor$  periodically, to clear the node's predecessor pointer if  $n.predecessor$  has failed; this allows it to accept a new predecessor in  $notify$ .

As a simple example, suppose node  $n$  joins the system, and its ID lies between nodes  $n_p$  and  $n_s$ . In its call to  $join()$ ,  $n$  acquires  $n_s$  as its successor. Node  $n_s$ , when notified by  $n$ , acquires  $n$  as its predecessor. When  $n_p$  next runs  $stabilize()$ , it asks  $n_s$  for its predecessor (which is now  $n$ );  $n_p$  then acquires  $n$  as its successor. Finally,  $n_p$  notifies  $n$ , and  $n$  acquires  $n_p$  as its predecessor. At this point, all predecessor and successor pointers

are correct. At each step in the process,  $n_s$  is reachable from  $n_p$  using successor pointers; this means that lookups concurrent with the join are not disrupted. Figure 7 illustrates the join procedure, when  $n$ 's ID is 26, and the IDs of  $n_s$  and  $n_p$  are 21 and 32, respectively.

As soon as the successor pointers are correct, calls to `find_successor()` will reflect the new node. Newly-joined nodes that are not yet reflected in other nodes' finger tables may cause `find_successor()` to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor (`finger[1]`) pointers through the newly-joined nodes until the correct predecessor is reached. Eventually `fix_fingers()` will adjust finger table entries, eliminating the need for these linear scans.

The following result, proved in [24], shows that the inconsistent state caused by concurrent joins is transient.

*Theorem IV.3:* If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network.

In other words, after some time each node is able to reach any other node in the network by following successor pointers.

Our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages. This stabilization protocol by itself won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space. These pathological cases cannot be produced by any sequence of ordinary node joins. If produced, these cases can be detected and repaired by periodic sampling of the ring topology [24].

## E.2 Impact of Node Joins on Lookups

In this section, we consider the impact of node joins on lookups. We first consider correctness. If joining nodes affect some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors. The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in  $O(\log N)$  steps. The second case is where successor pointers are correct, but fingers are inaccurate. This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail. The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause. This pause can be short, since stabilization fixes successor pointers quickly.

Now let us consider performance. Once stabilization has completed, the new nodes will have no effect beyond increasing the  $N$  in the  $O(\log N)$  lookup time. If stabilization has not yet completed, existing nodes' finger table entries may not reflect the new nodes. The ability of finger entries to carry queries long distances around the identifier ring does not depend on exactly which nodes the entries point to; the distance halving argument depends only on ID-space distance. Thus the fact that finger table entries may not reflect new nodes does not significantly affect lookup speed. The main way in which newly joined nodes can influence lookup speed is if the new nodes' IDs are between

the target's predecessor and the target. In that case the lookup will have to be forwarded through the intervening nodes, one at a time. But unless a tremendous number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible. Formally, we can state the following result. We call a Chord ring *stable* if all its successor and finger pointers are correct.

*Theorem IV.4:* If we take a stable network with  $N$  nodes with correct finger pointers, and another set of up to  $N$  nodes joins the network, and all successor pointers (but perhaps not all finger pointers) are correct, then lookups will still take  $O(\log N)$  time with high probability.

*Proof:* The original set of fingers will, in  $O(\log N)$  time, bring the query to the old predecessor of the correct node. With high probability, at most  $O(\log N)$  new nodes will land between any two old nodes. So only  $O(\log N)$  new nodes will need to be traversed along successor pointers to get from the old predecessor to the new predecessor. ■

More generally, as long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups will continue to take  $O(\log N)$  hops. We can achieve such adjustment by repeatedly carrying out lookups to update our fingers. It follows that lookups perform well so long as  $\Omega(\log^2 N)$  rounds of stabilization happen between any  $N$  node joins.

## E.3 Failure and Replication

The correctness of the Chord protocol relies on the fact that each node knows its successor. However, this invariant can be compromised if nodes fail. For example, in Figure 4, if nodes 14, 21, and 32 fail simultaneously, node 8 will not know that node 38 is now its successor, since it has no finger pointing to 38. An incorrect successor will lead to incorrect lookups. Consider a query for key 30 initiated by node 8. Node 8 will return node 42, the first node it knows about from its finger table, instead of the correct successor, node 38.

To increase robustness, each Chord node maintains a *successor list* of size  $r$ , containing the node's first  $r$  successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All  $r$  successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of  $r$ . Assuming each node fails independently with probability  $p$ , the probability that all  $r$  successors fail simultaneously is only  $p^r$ . Increasing  $r$  makes the system more robust.

Handling the successor list requires minor changes in the pseudocode in Figures 5 and 6. A modified version of the *stabilize* procedure in Figure 6 maintains the successor list. Successor lists are stabilized as follows: node  $n$  reconciles its list with its successor  $s$  by copying  $s$ 's successor list, removing its last entry, and prepending  $s$  to it. If node  $n$  notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor. At that point,  $n$  can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, `fix_fingers` and `stabilize` will correct finger table entries and successor list entries pointing to the failed node.

A modified version of the *closest\_preceding\_node* procedure in Figure 5 searches not only the finger table but also the suc-

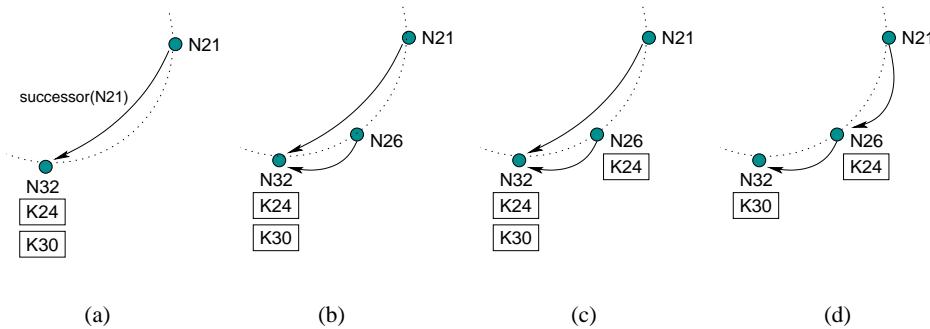


Fig. 7. Example illustrating the join operation. Node 26 joins the system between nodes 21 and 32. The arcs represent the successor relationship. (a) Initial state: node 21 points to node 32; (b) node 26 finds its successor (i.e., node 32) and points to it; (c) node 26 copies all keys less than 26 from node 32; (d) the stabilize procedure updates the successor of node 21 to node 26.

sor list for the most immediate predecessor of  $id$ . In addition, the pseudocode needs to be enhanced to handle node failures. If a node fails during the *find\_successor* procedure, the lookup proceeds, after a timeout, by trying the next best predecessor among the nodes in the finger table and the successor list.

The following results quantify the robustness of the Chord protocol, by showing that neither the success nor the performance of Chord lookups is likely to be affected even by massive simultaneous failures. Both theorems assume that the successor list has length  $r = \Omega(\log N)$ .

*Theorem IV.5:* If we use a successor list of length  $r = \Omega(\log N)$  in a network that is initially stable, and then every node fails with probability  $1/2$ , then with high probability *find\_successor* returns the closest living successor to the query key.

*Proof:* Before any nodes fail, each node was aware of its  $r$  immediate successors. The probability that all of these successors fail is  $(1/2)^r$ , so with high probability every node is aware of its immediate living successor. As was argued in the previous section, if the invariant that every node is aware of its immediate successor holds, then all queries are routed properly, since every node except the immediate predecessor of the query has at least one better node to which it will forward the query. ■

*Theorem IV.6:* In a network that is initially stable, if every node then fails with probability  $1/2$ , then the expected time to execute *find\_successor* is  $O(\log N)$ .

*Proof:* Due to space limitations we omit the proof of this result, which can be found in the technical report [24]. ■

Under some circumstances the preceding theorems may apply to malicious node failures as well as accidental failures. An adversary may be able to make some set of nodes fail, but have no control over the choice of the set. For example, the adversary may be able to affect only the nodes in a particular geographical region, or all the nodes that use a particular access link, or all the nodes that have a certain IP address prefix. As was discussed above, because Chord node IDs are generated by hashing IP addresses, the IDs of these failed nodes will be effectively random, just as in the failure case analyzed above.

The successor list mechanism also helps higher-layer software replicate data. A typical application using Chord might store replicas of the data associated with a key at the  $k$  nodes succeeding the key. The fact that a Chord node keeps track of

its  $r$  successors means that it can inform the higher layer software when successors come and go, and thus when the software should propagate data to new replicas.

#### E.4 Voluntary Node Departures

Since Chord is robust in the face of failures, a node voluntarily leaving the system could be treated as a node failure. However, two enhancements can improve Chord performance when nodes leave voluntarily. First, a node  $n$  that is about to leave may transfer its keys to its successor before it departs. Second,  $n$  may notify its predecessor  $p$  and successor  $s$  before leaving. In turn, node  $p$  will remove  $n$  from its successor list, and add the last node in  $n$ 's successor list to its own list. Similarly, node  $s$  will replace its predecessor with  $n$ 's predecessor. Here we assume that  $n$  sends its predecessor to  $s$ , and the last node in its successor list to  $p$ .

#### F. More Realistic Analysis

Our analysis above gives some insight into the behavior of the Chord system, but is inadequate in practice. The theorems proven above assume that the Chord ring starts in a stable state and then experiences joins or failures. In practice, a Chord ring will never be in a stable state; instead, joins and departures will occur continuously, interleaved with the stabilization algorithm. The ring will not have time to stabilize before new changes happen. The Chord algorithms can be analyzed in this more general setting. Other work [16] shows that if the stabilization protocol is run at a certain rate (dependent on the rate at which nodes join and fail) then the Chord ring remains continuously in an “almost stable” state in which lookups are fast and correct.

## V. SIMULATION RESULTS

In this section, we evaluate the Chord protocol by simulation. The packet-level simulator uses the lookup algorithm in Figure 5, extended with the successor lists described in Section IV-E.3, and the stabilization algorithm in Figure 6.

#### A. Protocol Simulator

The Chord protocol can be implemented in an *iterative* or *recursive* style. In the iterative style, a node resolving a lookup initiates all communication: it asks a series of nodes for information from their finger tables, each time moving closer on the

Chord ring to the desired successor. In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor. The simulator implements the Chord protocol in an iterative style.

During each stabilization step, a node updates its immediate successor and one other entry in its successor list or finger table. Thus, if a node's successor list and finger table contain a total of  $k$  unique entries, each entry is refreshed once every  $k$  stabilization rounds. Unless otherwise specified, the size of the successor list is one, that is, a node knows only its immediate successor. In addition to the optimizations described on Section IV-E.4, the simulator implements one other optimization. When the predecessor of a node  $n$  changes,  $n$  notifies its old predecessor  $p$  about the new predecessor  $p'$ . This allows  $p$  to set its successor to  $p'$  without waiting for the next stabilization round.

The delay of each packet is exponentially distributed with mean of 50 milliseconds. If a node  $n$  cannot contact another node  $n'$  within 500 milliseconds,  $n$  concludes that  $n'$  has left or failed. If  $n'$  is an entry in  $n$ 's successor list or finger table, this entry is removed. Otherwise  $n$  informs the node from which it learnt about  $n'$  that  $n'$  is gone. When a node on the path of a lookup fails, the node that initiated the lookup tries to make progress using the next closest finger preceding the target key.

A lookup is considered to have succeeded if it reaches the current successor of the desired key. This is slightly optimistic: in a real system, there might be periods of time in which the real successor of a key has not yet acquired the data associated with the key from the previous successor. However, this method allows us to focus on Chord's ability to perform lookups, rather than on the higher-layer software's ability to maintain consistency of its own data.

### B. Load Balance

We first consider the ability of consistent hashing to allocate keys to nodes evenly. In a network with  $N$  nodes and  $K$  keys we would like the distribution of keys to nodes to be tight around  $N/K$ .

We consider a network consisting of  $10^4$  nodes, and vary the total number of keys from  $10^5$  to  $10^6$  in increments of  $10^5$ . For each number of keys, we run 20 experiments with different random number generator seeds, counting the number of keys assigned to each node in each experiment. Figure 8(a) plots the mean and the 1st and 99th percentiles of the number of keys per node. The number of keys per node exhibits large variations that increase linearly with the number of keys. For example, in all cases some nodes store no keys. To clarify this, Figure 8(b) plots the probability density function (PDF) of the number of keys per node when there are  $5 \times 10^5$  keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or  $9.1 \times$  the mean value. For comparison, the 99th percentile is  $4.6 \times$  the mean value.

One reason for these variations is that node identifiers do not uniformly cover the entire identifier space. From the perspective of a single node, the amount of the ring it "owns" is determined by the distance to its immediate predecessor. The distance to each of the other  $n - 1$  nodes is uniformly distributed over the range  $[0, m]$ , and we are interested in the minimum of these dis-

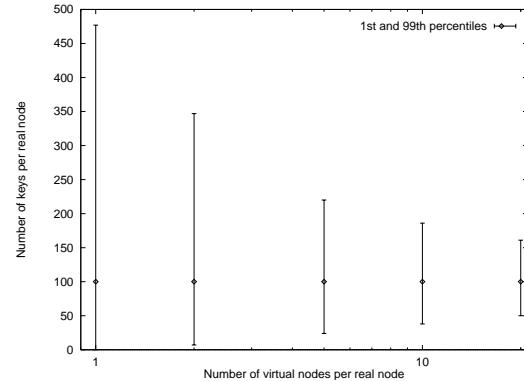


Fig. 9. The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has  $10^4$  real nodes and stores  $10^6$  keys.

tance. It is a standard fact that the distribution of this minimum is tightly approximated by an exponential distribution with mean  $2^m/N$ . Thus, for example, the owned region exceeds twice the average value (of  $2^m/N$ ) with probability  $e^{-2}$ .

Chord makes the number of keys per node more uniform by associating keys with virtual nodes, and mapping multiple virtual nodes (with unrelated identifiers) to each real node. This provides a more uniform coverage of the identifier space. For example, if we allocate  $\log N$  randomly chosen virtual nodes to each real node, with high probability each of the  $N$  bins will contain  $O(\log N)$  virtual nodes [17].

To verify this hypothesis, we perform an experiment in which we allocate  $r$  virtual nodes to each real node. In this case keys are associated with virtual nodes instead of real nodes. We consider again a network with  $10^4$  real nodes and  $10^6$  keys. Figure 9 shows the 1st and 99th percentiles for  $r = 1, 2, 5, 10$ , and  $20$ , respectively. As expected, the 99th percentile decreases, while the 1st percentile increases with the number of virtual nodes,  $r$ . In particular, the 99th percentile decreases from  $4.8 \times$  to  $1.6 \times$  the mean value, while the 1st percentile increases from  $0$  to  $0.5 \times$  the mean value. Thus, adding virtual nodes as an indirection layer can significantly improve load balance. The tradeoff is that each real node now needs  $r$  times as much space to store the finger tables for its virtual nodes.

We make several observations with respect to the complexity incurred by this scheme. First, the asymptotic value of the query path length, which now becomes  $O(\log(N \log N)) = O(\log N)$ , is not affected. Second, the total identifier space covered by the virtual nodes<sup>1</sup> mapped on the same real node is with high probability an  $O(1/N)$  fraction of the total, which is the same on average as in the absence of virtual nodes. Since the number of queries handled by a node is roughly proportional to the total identifier space covered by that node, the worst-case number of queries handled by a node does not change. Third, while the routing state maintained by a node is now  $O(\log^2 N)$ , this value is still reasonable in practice; for  $N = 10^6$ ,  $\log^2 N$  is only 400. Finally, while the number of control messages ini-

<sup>1</sup>The identifier space covered by a virtual node represents the interval between the node's identifier and the identifier of its predecessor. The identifier space covered by a real node is the sum of the identifier spaces covered by its virtual nodes.

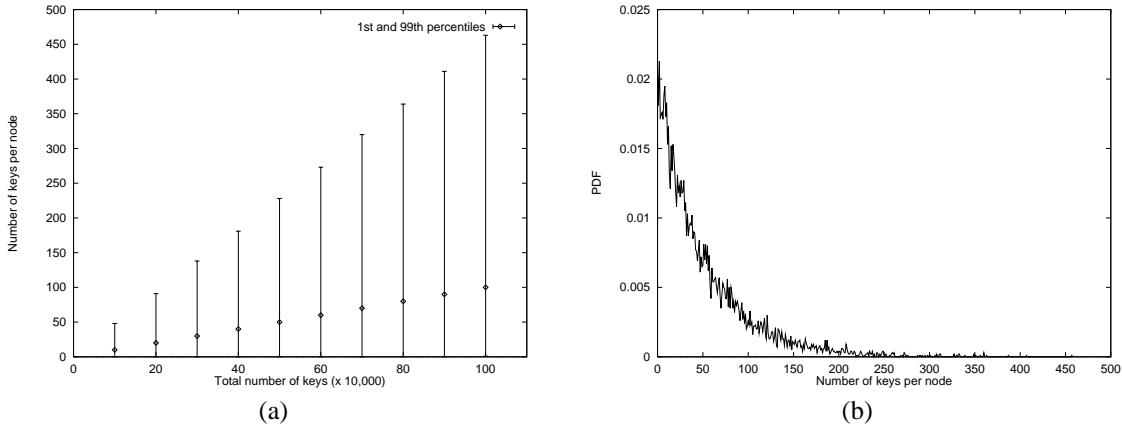


Fig. 8. (a) The mean and 1st and 99th percentiles of the number of keys stored per node in a  $10^4$  node network. (b) The probability density function (PDF) of the number of keys per node. The total number of keys is  $5 \times 10^5$ .

tiated by a node increases by a factor of  $O(\log N)$ , the asymptotic number of control messages received from other nodes is not affected. To see why is this, note that in the absence of virtual nodes, with “reasonable” probability a real node is responsible for  $O(\log N/N)$  of the identifier space. Since there are  $O(N \log N)$  fingers in the entire system, the number of fingers that point to a real node is  $O(\log^2 N)$ . In contrast, if each real node maps  $\log N$  virtual nodes, with high probability each real node is responsible for  $O(1/N)$  of the identifier space. Since there are  $O(N \log^2 N)$  fingers in the entire system, with high probability the number of fingers that point to the virtual nodes mapped on the same real node is still  $O(\log^2 N)$ .

### C. Path Length

Chord’s performance depends in part on the number of nodes that must be visited to resolve a query. From Theorem IV.2, with high probability, this number is  $O(\log N)$ , where  $N$  is the total number of nodes in the network.

To understand Chord’s routing performance in practice, we simulated a network with  $N = 2^k$  nodes, storing  $100 \times 2^k$  keys in all. We varied  $k$  from 3 to 14 and conducted a separate experiment for each value. Each node in an experiment picked a random set of keys to query from the system, and we measured each query’s path length.

Figure 10(a) plots the mean, and the 1st and 99th percentiles of path length as a function of  $k$ . As expected, the mean path length increases logarithmically with the number of nodes, as do the 1st and 99th percentiles. Figure 10(b) plots the PDF of the path length for a network with  $2^{12}$  nodes ( $k = 12$ ).

Figure 10(a) shows that the path length is about  $\frac{1}{2} \log_2 N$ . The value of the constant term ( $\frac{1}{2}$ ) can be understood as follows. Consider a node making a query for a randomly chosen key. Represent the distance in identifier space between node and key in binary. The most significant (say  $i^{th}$ ) bit of this distance can be corrected to 0 by following the node’s  $i^{th}$  finger. If the next significant bit of the distance is 1, it too needs to be corrected by following a finger, but if it is 0, then no  $i - 1^{st}$  finger is followed—instead, we move on the the  $i - 2^{nd}$  bit. In general, the number of fingers we need to follow will be the number of ones in the binary representation of the distance from node to query. Since the node identifiers are randomly distributed, we

expect half the bits to be ones. As discussed in Theorem IV.2, after the  $\log N$  most-significant bits have been fixed, in expectation there is only one node remaining between the current position and the key. Thus the average path length will be about  $\frac{1}{2} \log_2 N$ .

### D. Simultaneous Node Failures

In this experiment, we evaluate the impact of a massive failure on Chord’s performance and on its ability to perform correct lookups. We consider a network with  $N = 1,000$  nodes, where each node maintains a successor list of size  $r = 20 = 2 \log_2 N$  (see Section IV-E.3 for a discussion on the size of the successor list). Once the network becomes stable, each node is made to fail with probability  $p$ . After the failures occur, we perform 10,000 random lookups. For each lookup, we record the number of timeouts experienced by the lookup, the number of nodes contacted during the lookup (including attempts to contact failed nodes), and whether the lookup found the key’s true current successor. A timeout occurs when a node tries to contact a failed node. The number of timeouts experienced by a lookup is equal to the number of failed nodes encountered by the lookup operation. To focus the evaluation on Chord’s performance immediately after failures, before it has a chance to correct its tables, these experiments stop stabilization just before the failures occur and do not remove the fingers pointing to failed nodes from the finger tables. Thus the failed nodes are detected only when they fail to respond during the lookup protocol.

Table II shows the mean, and the 1st and the 99th percentiles of the path length for the first 10,000 lookups after the failure occurs as a function of  $p$ , the fraction of failed nodes. As expected, the path length and the number of timeouts increases as the fraction of nodes that fail increases.

To interpret these results better, we next estimate the mean path length of a lookup when each node has a successor list of size  $r$ . By an argument similar to the one used in Section V-C, a successor list of size  $r$  eliminates the last  $\frac{1}{2} \log_2 r$  hops from the lookup path on average. The mean path length of a lookup becomes then  $\frac{1}{2} \log_2 N - \frac{1}{2} \log_2 r + 1$ . The last term (1) accounts for accessing the predecessor of the queried key once this predecessor is found in the successor list of the previous node. For  $N = 1,000$  and  $r = 20$ , the mean path length is 3.82,

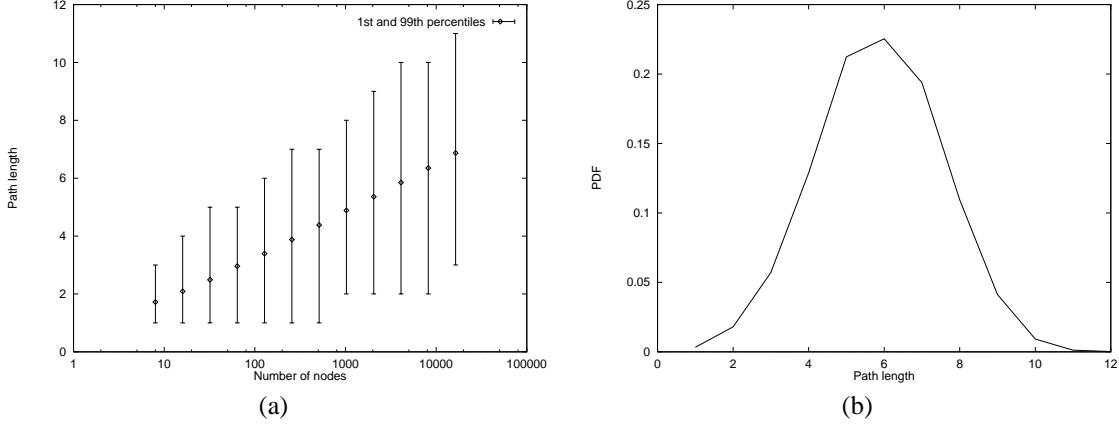


Fig. 10. (a) The path length as a function of network size. (b) The PDF of the path length in the case of a  $2^{12}$  node network.

Fraction of failed nodes	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)
0	3.84 (2, 5)	0.0 (0, 0)
0.1	4.03 (2, 6)	0.60 (0, 2)
0.2	4.22 (2, 6)	1.17 (0, 3)
0.3	4.44 (2, 6)	2.02 (0, 5)
0.4	4.69 (2, 7)	3.23 (0, 8)
0.5	5.09 (3, 8)	5.10 (0, 11)

TABLE II

The path length and the number of timeouts experienced by a lookup as function of the fraction of nodes that fail simultaneously. The 1st and the 99th percentiles are in parenthesis. Initially, the network has 1,000 nodes.

which is very close to the value of 3.84 shown in Table II for  $p = 0$ .

Let  $x$  denote the progress made in the identifier space towards a target key during a particular lookup iteration, when there are no failures in the system. Next, assume that each node fails independently with probability  $p$ . As discussed in Section IV-E.3, during each lookup iteration every node selects the largest *alive* finger (from its finger table) that precedes the target key. Thus the progress made during the same lookup iteration in the identifier space is  $x$  with probability  $(1 - p)$ , roughly  $x/2$  with probability  $p * (1 - p)$ , roughly  $x/2^2$  with probability  $p^2 * (1 - p)$ , and so on. The expected progress made towards the target key is then  $\sum_{i=0}^{\infty} \frac{x}{2^i} (1 - p)p^i = x(1 - p)/(1 - p/2)$ . As a result, the mean path length becomes approximately  $\frac{1}{2} \log_d N - \frac{1}{2} \log_d r + 1$ , where  $d = 1.7 = \log_2 \left(\frac{1-p/2}{1-p}\right)$ . As an example, the mean path length for  $p = 0.5$  is 5.76. One reason for which the predicted value is larger than the measured value in Table II is because the series used to evaluate  $d$  is finite in practice. This leads us to underestimating the value of  $d$ , which in turn leads us to overestimating the mean path length.

Now, let us turn our attention to the number of timeouts. Let  $m$  be the mean number of nodes contacted during a lookup operation. The expected number of timeouts experienced during a lookup operation is  $m * p$ , and the mean path length is  $l = m * (1 - p)$ . Given the mean path length in Table II, the expected number of timeouts is 0.45 for  $p = 0.1$ , 1.06 for  $p = 0.2$ , 1.90 for  $p = 0.3$ , 3.13 for  $p = 0.4$ , and 5.06 for  $p = 0.5$ . These

values match well the measured number of timeouts shown in Table II.

Finally, we note that in our simulations all lookups were successfully resolved, which supports the robustness claim of Theorem IV.5.

#### E. Lookups During Stabilization

In this experiment, we evaluate the performance and accuracy of Chord lookups when nodes are continuously joining and leaving. The leave procedure uses the departure optimizations outlined in Section IV-E.4. Key lookups are generated according to a Poisson process at a rate of one per second. Joins and voluntary leaves are modeled by a Poisson process with a mean arrival rate of  $R$ . Each node runs the stabilization routine at intervals that are uniformly distributed in the interval [15, 45] seconds; recall that only the successor and one finger table entry are stabilized for each call, so the expected interval between successive stabilizations of a given finger table entry is much longer than the average stabilization period of 30 seconds. The network starts with 1,000 nodes, and each node maintains again a successor list of size  $r = 20 = 2 \log_2 N$ . Note that even though there are no failures, timeouts may still occur during the lookup operation; a node that tries to contact a finger that has just left will time out.

Table III shows the means and the 1st and 90th percentiles of the path length and the number of timeouts experienced by the lookup operation as a function of the rate  $R$  at which nodes join and leave. A rate  $R = 0.05$  corresponds to one node joining and leaving every 20 seconds on average. For comparison, recall that each node invokes the stabilize protocol once every 30 seconds. Thus,  $R$  ranges from a rate of one join and leave per 1.5 stabilization periods to a rate of 12 joins and 12 leaves per one stabilization period.

As discussed in Section V-D, the mean path length in steady state is about  $\frac{1}{2} \log_2 N - \frac{1}{2} \log_2 r + 1$ . Again, since  $N = 1,000$  and  $r = 20$ , the mean path length is 3.82. As shown in Table III, the measured path length is very close to this value and does not change dramatically as  $R$  increases. This is because the number of timeouts experienced by a lookup is relatively small, and thus it has minimal effect on the path length. On the other hand, the number of timeouts increases with  $R$ . To understand this result,

Node join/leave rate (per second/per stab. period)	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)	Lookup failures (per 10,000 lookups)
0.05 / 1.5	3.90 (1, 9)	0.05 (0, 2)	0
0.10 / 3	3.83 (1, 9)	0.11 (0, 2)	0
0.15 / 4.5	3.84 (1, 9)	0.16 (0, 2)	2
0.20 / 6	3.81 (1, 9)	0.23 (0, 3)	5
0.25 / 7.5	3.83 (1, 9)	0.30 (0, 3)	6
0.30 / 9	3.91 (1, 9)	0.34 (0, 4)	8
0.35 / 10.5	3.94 (1, 10)	0.42 (0, 4)	16
0.40 / 12	4.06 (1, 10)	0.46 (0, 5)	15

TABLE III

The path length and the number of timeouts experienced by a lookup as function of node join and leave rates. The 1st and the 99th percentiles are in parentheses. The network has roughly 1,000 nodes.

consider the following informal argument.

Let us consider a particular finger pointer  $f$  from node  $n$  and evaluate the fraction of lookup traversals of *that finger* that encounter a timeout (by symmetry, this will be the same for all fingers). From the perspective of that finger, history is made up of an interleaving of three types of events: (1) stabilizations of that finger, (2) departures of the node pointed at by the finger, and (3) lookups that traverse the finger. A lookup causes a timeout if the finger points at a departed node. This occurs precisely when the event immediately preceding the lookup was a departure—if the preceding event was a stabilization, then the node currently pointed at is alive; similarly, if the previous event was a lookup, then *that* lookup timed out and caused eviction of that dead finger pointer. So we need merely determine the fraction of lookup events in the history that are immediately preceded by a departure event.

To simplify the analysis we assume that, like joins and leaves, stabilization is run according to a Poisson process. Our history is then an interleaving of three Poisson processes. The fingered node departs as a Poisson process at rate  $R' = R/N$ . Stabilization of that finger occurs (and detects such a departure) at rate  $S$ . In each stabilization round, a node stabilizes either a node in its finger table or a node in its successor list (there are  $3 \log N$  such nodes in our case). Since the stabilization operation reduces to a lookup operation (see Figure 6), each stabilization operation will use  $l$  fingers on the average, where  $l$  is the mean lookup path length.<sup>2</sup> As result, the rate at which a finger is touched by the stabilization operation is  $S = (1/30) * l/(3 \log N)$  where  $1/30$  is the average rate at which each node invokes stabilization. Finally, lookups using that finger are also a Poisson process. Recall that lookups are generated (globally) as a Poisson process with rate of one lookup per second. Each such lookup uses  $l$  fingers on average, while there are  $N \log N$  fingers in total. Thus a particular finger is used with probability  $l/(N \log N)$ , meaning that the finger gets used according to a Poisson process at rate  $L = l/(N \log N)$ .

We have three interleaved Poisson processes (the lookups, departures, and stabilizations). Such a union of Poisson processes is itself a Poisson process with rate equal to the sum of the three

<sup>2</sup>Actually, since  $2 \log N$  of the nodes belong to the successor list, the mean path length of the stabilization operation is smaller than the the mean path length of the lookup operation (assuming the requested keys are randomly distributed). This explains in part the underestimation bias in our computation.

underlying rates. Each time an “event” occurs in this union process, it is assigned to one of the three underlying processes with probability proportional to those processes rates. In other words, the history seen by a node looks like a random sequence in which each event is a departure with probability

$$\begin{aligned} p_t &= \frac{R'}{R' + S + L} = \frac{\frac{R}{N}}{\frac{R}{N} + \frac{l}{90 \log N} + \frac{l}{N \log N}} \\ &= \frac{R}{R + \frac{l \cdot N}{90 \log N} + \frac{l}{\log N}}. \end{aligned}$$

In particular, the event immediately preceding any lookup is a departure with this probability. This is the probability that the lookup encounters the timeout. Finally, the expected number of timeouts experienced by a lookup operation is  $l * p_t = R/(R/l + N/(90 \log N) + 1/\log(N))$ . As examples, the expected number of timeouts is 0.041 for  $R = 0.05$ , and 0.31 for  $R = 0.4$ . These values are reasonable close to the measured values shown in Table III.

The last column in Table III shows the number of lookup failures per 10,000 lookups. The reason for these lookup failures is state inconsistency. In particular, despite the fact that each node maintains a successor list of  $2 \log_2 N$  nodes, it is possible that for short periods of time a node may point to an incorrect successor. Suppose at time  $t$ , node  $n$  knows both its first and its second successor,  $s_1$  and  $s_2$ . Assume that just after time  $t$ , a new node  $s$  joins the network between  $s_1$  and  $s_2$ , and that  $s_1$  leaves before  $n$  had the chance to discover  $s$ . Once  $n$  learns that  $s_1$  has left,  $n$  will replace it with  $s_2$ , the closest successor  $n$  knows about. As a result, for any key  $id \in (n, s)$ ,  $n$  will return node  $s_2$  instead of  $s$ . However, the next time  $n$  invokes stabilization for  $s_2$ ,  $n$  will learn its correct successor  $s$ .

#### F. Improving Routing Latency

While Chord ensures that the average path length is only  $\frac{1}{2} \log_2 N$ , the lookup *latency* can be quite large. This is because the node identifiers are randomly distributed, and therefore nodes close in the identifier space can be far away in the underlying network. In previous work [8] we attempted to reduce lookup latency with a simple extension of the Chord protocol that exploits only the information already in a node’s finger table. The idea was to choose the next-hop finger based on both progress in identifier space and latency in the underlying

Number of fingers' successors (s)	Stretch (10th, 90th percentiles)			
	Iterative		Recursive	
	3-d space	Transit stub	3-d space	Transit stub
1	7.8 (4.4, 19.8)	7.2 (4.4, 36.0)	4.5 (2.5, 11.5)	4.1 (2.7, 24.0)
2	7.2 (3.8, 18.0)	7.1 (4.2, 33.6)	3.5 (2.0, 8.7)	3.6 (2.3, 17.0)
4	6.1 (3.1, 15.3)	6.4 (3.2, 30.6)	2.7 (1.6, 6.4)	2.8 (1.8, 12.7)
8	4.7 (2.4, 11.8)	4.9 (1.9, 19.0)	2.1 (1.4, 4.7)	2.0 (1.4, 8.9)
16	3.4 (1.9, 8.4)	2.2 (1.7, 7.4)	1.7 (1.2, 3.5)	1.5 (1.3, 4.0)

TABLE IV

The stretch of the lookup latency for a Chord system with  $2^{16}$  nodes when the lookup is performed both in the iterative and recursive style. Two network models are considered: a 3-d Euclidean space, and a transit stub network.

network, trying to maximize the former while minimizing the latter. While this protocol extension is simple to implement and does not require any additional state, its performance is difficult to analyze [8]. In this section, we present an alternate protocol extension, which provides better performance at the cost of slightly increasing the Chord state and message complexity. We emphasize that we are actively exploring techniques to minimize lookup latency, and we expect further improvements in the future.

The main idea of our scheme is to maintain a set of alternate nodes for each finger (that is, nodes with similar identifiers that are roughly equivalent for routing purposes), and then route the queries by selecting the closest node among the alternate nodes according to some network proximity metric. In particular, every node associates with each of its fingers,  $f$ , a list of  $s$  immediate successors of  $f$ . In addition, we modify the *find\_successor* function in Figure 5 accordingly: instead of simply returning the largest finger,  $f$ , that precedes the queried ID, the function returns the closest node (in terms of networking distance) among  $f$  and its  $s$  successors. For simplicity, we choose  $s = r$ , where  $r$  is the length of the successor list; one could reduce the storage requirements for the routing table by maintaining, for each finger  $f$ , only the closest node  $n$  among  $f$ 's  $s$  successors. To update  $n$ , a node can simply ask  $f$  for its successor list, and then ping each node in the list. The node can update  $n$  either periodically, or when it detects that  $n$  has failed. Observe that this heuristic can be applied *only* in the recursive (not the iterative) implementation of lookup, as the original querying node will have no distance measurements to the fingers of each node on the path.

To illustrate the efficacy of this heuristic, we consider a Chord system with  $2^{16}$  nodes and two network topologies:

- **3-d space:** The network distance is modeled as the geometric distance in a 3-dimensional space. This model is motivated by recent research [19] showing that the network latency between two nodes in the Internet can be modeled (with good accuracy) as the geometric distance in a  $d$ -dimensional Euclidean space, where  $d \geq 3$ .
- **Transit stub:** A transit-stub topology with 5,000 nodes, where link latencies are 50 milliseconds for intra-transit domain links, 20 milliseconds for transit-stub links and 1 milliseconds for intra-stub domain links. Chord nodes are randomly assigned to stub nodes. This network topology aims to reflect the hierarchical organization of today's Internet.

We use the *lookup stretch* as the main metric to evaluate our heuristic. The lookup stretch is defined as the ratio between the (1) latency of a Chord lookup from the time the lookup is initiated to the time the result is returned to the initiator, and the (2) latency of an optimal lookup using the underlying network. The latter is computed as the round-trip time between the initiator and the server responsible for the queried ID.

Table IV shows the median, the 10th and the 99th percentiles of the lookup stretch over 10,000 lookups for both the iterative and the recursive styles. The results suggest that our heuristic is quite effective. The stretch decreases significantly as  $s$  increases from one to 16.

As expected, these results also demonstrate that recursive lookups execute faster than iterative lookups. Without any latency optimization, the recursive lookup style is expected to be approximately twice as fast as the iterative style: an iterative lookup incurs a round-trip latency per hop, while a recursive lookup incurs a one-way latency.

Note that in a 3-d Euclidean space the expected distance from a node to the closest node from a set of  $s + 1$  random nodes is proportional to  $(s + 1)^{1/3}$ . Since the number of Chord hops does not change as  $s$  increases, we expect the lookup latency to be also proportional to  $(s + 1)^{1/3}$ . This observation is consistent with the results presented in Table IV. For instance, for  $s = 16$ , we have  $17^{1/3} = 2.57$ , which is close to the observed reduction of the median value of the lookup stretch from  $s = 1$  to  $s = 16$ .

## VI. FUTURE WORK

Work remains to be done in improving Chord's resilience against network partitions and adversarial nodes as well as its efficiency.

Chord can detect and heal partitions whose nodes know of each other. One way to obtain this knowledge is for every node to know of the same small set of initial nodes. Another approach might be for nodes to maintain long-term memory of a random set of nodes they have encountered in the past; if a partition forms, the random sets in one partition are likely to include nodes from the other partition.

A malicious or buggy set of Chord participants could present an incorrect view of the Chord ring. Assuming that the data Chord is being used to locate is cryptographically authenticated, this is a threat to availability of data rather than to authenticity. One way to check global consistency is for each node  $n$  to periodically ask other nodes to do a Chord lookup for  $n$ ; if the lookup does not yield node  $n$ , this could be an indication for

victims that they are not seeing a globally consistent view of the Chord ring.

Even  $\frac{1}{2} \log_2 N$  messages per lookup may be too many for some applications of Chord, especially if each message must be sent to a random Internet host. Instead of placing its fingers at distances that are all powers of 2, Chord could easily be changed to place its fingers at distances that are all integer powers of  $1 + 1/d$ . Under such a scheme, a single routing hop could decrease the distance to a query to  $1/(1+d)$  of the original distance, meaning that  $\log_{1+d} N$  hops would suffice. However, the number of fingers needed would increase to  $\log N / (\log(1 + 1/d)) \approx O(d \log N)$ .

## VII. CONCLUSION

Many distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. It offers a powerful primitive: given a key, it determines the node responsible for storing the key's value, and does so efficiently. In the steady state, in an  $N$ -node network, each node maintains routing information for only  $O(\log N)$  other nodes, and resolves all lookups via  $O(\log N)$  messages to other nodes.

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct. Our theoretical analysis and simulation results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

We believe that Chord will be a valuable component for peer-to-peer, large-scale distributed applications such as cooperative file sharing, time-shared available storage systems, distributed indices for document and service discovery, and large-scale distributed computing platforms. Our initial experience with Chord has been very promising. We have already built several peer-to-peer applications using Chord, including a cooperative file sharing application [9]. The software is available at <http://pdos.lcs.mit.edu/chord/>.

## REFERENCES

- [1] AJMANI, S., CLARKE, D., MOH, C.-H., AND RICHMAN, S. ConChord: Cooperative SDSI certifi cate storage and name resolution. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).
- [2] BAKKER, A., AMADE, E., BALLINTIJN, G., KUZ, I., VERKAIK, P., VAN DER WIJK, I., VAN STEEN, M., AND TANENBAUM, A. The Globe distribution network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)* (San Diego, CA, June 2000), pp. 141–152.
- [3] CARTER, J. L., AND WEGMAN, M. N. Universal classes of hash functions. *Journal of Computer and System Sciences* 18, 2 (1979), 143–154.
- [4] CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.
- [5] CLARKE, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.
- [6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). <http://freenet.sourceforge.net>.
- [7] COX, R., MUTHITACHAROEN, A., AND MORRIS, R. Serving DNS using Chord. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).
- [8] DABEK, F. A cooperative fi le system. Master's thesis, Massachusetts Institute of Technology, September 2001.
- [9] DABEK, F., KAASHOEK, F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01* (Banff, Canada, 2001), pp. 202–215.
- [10] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfi eld, VA, Apr. 1995.
- [11] Gnutella. <http://gnutella.wego.com/>.
- [12] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663.
- [13] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.
- [14] LEWIN, D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT, 1998. Available at the MIT Library, <http://thesis.mit.edu/>.
- [15] LI, J., JANNOTTI, J., DE COUTO, D., KARGER, D., AND MORRIS, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking* (Boston, Massachusetts, August 2000), pp. 120–130.
- [16] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. R. Observations on the dynamic evolution of peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).
- [17] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.
- [18] Napster. <http://www.napster.com/>.
- [19] NG, T. S. E., AND ZHANG, H. Towards global network positioning. In *ACM SIGCOMM Internet Measurements Workshop 2001* (San Francisco, CA, Nov. 2001).
- [20] Ohaha. Smart decentralized peer-to-peer sharing. <http://www.ohaha.com/design.html>.
- [21] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.
- [22] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001), pp. 161–172.
- [23] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001), pp. 329–350.
- [24] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. Tech. Rep. TR-819, MIT LCS, 2001. <http://www.pdos.lcs.mit.edu/chord/papers/>.
- [25] VAN STEEN, M., HAUCK, F., BALLINTIJN, G., AND TANENBAUM, A. Algorithmic design of the Globe wide-area location service. *The Computer Journal* 41, 5 (1998), 297–310.
- [26] ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.

# Parallax: Virtual Disks for Virtual Machines

Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre,  
Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield\*

{dmeyer, gitika, brendan, geoffrey, feeley, norm, andy}@cs.ubc.ca

Department of Computer Science

University of British Columbia

Vancouver, BC, Canada

## ABSTRACT

Parallax is a distributed storage system that uses virtualization to provide storage facilities specifically for virtual environments. The system employs a novel architecture in which storage features that have traditionally been implemented directly on high-end storage arrays and switches are relocated into a federation of *storage VMs*, sharing the same physical hosts as the VMs that they serve. This architecture retains the single administrative domain and OS agnosticism achieved by array- and switch-based approaches, while lowering the bar on hardware requirements and facilitating the development of new features. Parallax offers a comprehensive set of storage features including frequent, low-overhead snapshot of virtual disks, the “gold-mastering” of template images, and the ability to use local disks as a persistent cache to dampen burst demand on networked storage.

## Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management—*Storage Hierarchies*; D.4.7 [Operating Systems]: Organization and Design—*Distributed Systems*

## General Terms

Design, Experimentation, Measurement, Performance

## 1. INTRODUCTION

In current deployments of hardware virtualization, storage facilities severely limit the flexibility and freedom of virtual machines.

Perhaps the most important aspect of the resurgence of virtualization is that it allows complex modern software—the operating system and applications that run on a computer—to be completely encapsulated in a virtual machine. The encapsulation afforded by the VM abstraction is without parallel: it allows whole systems to easily be quickly provisioned, duplicated, rewound, and migrated across physical hosts without disrupting execution. The benefits of

---

\*also of XenSource, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.  
Copyright 2008 ACM 978-1-60558-013-5/08/04 ...\$5.00.

this encapsulation have been demonstrated by numerous interesting research projects that allow VMs to travel through space [24, 2, 13], time [4, 12, 32], and to be otherwise manipulated [30].

Unfortunately, while both system software and platform hardware such as CPUs and chipsets have evolved rapidly in support of virtualization, storage has not. While “storage virtualization” is widely available, the term is something of a misnomer in that it is largely used to describe the aggregation and repartitioning of disks at very coarse time scales for use by physical machines. VM deployments are limited by modern storage systems because the storage primitives available for use by VMs are not nearly as nimble as the VMs that consume them. Operations such as remapping volumes across hosts and checkpointing disks are frequently clumsy and esoteric on high-end storage systems, and are simply unavailable on lower-end commodity storage hardware.

This paper describes *Parallax*, a system that attempts to *use* virtualization in order to provide advanced storage services for virtual machines. Parallax takes advantage of the structure of a virtualized environment to move storage enhancements that are traditionally implemented on arrays or in storage switches out onto the consuming physical hosts. Each host in a Parallax-based cluster runs a *storage VM*, which is a virtual appliance [23] specifically for storage that serves virtual disks to the VMs that run alongside it. The encapsulation provided by virtualization allows these storage features to remain behind the block interface, agnostic to the OS that uses them, while moving their implementation into a context that facilitates improvement and innovation.

Parallax is effectively a cluster volume manager for virtual disks: each physical host shares access to a single, globally visible block device, which is collaboratively managed to present individual virtual disk images (VDIs) to VMs. The system has been designed with considerations specific to the emerging uses of virtual machines, resulting in some particularly unusual directions. Most notably, we desire very frequent (i.e., every 10ms) snapshots. This capability allows the fine-grained rewinding of the disk to arbitrary points in its history, which makes virtual machine snapshots much more powerful. In addition, since our goal is to present virtual disks to VMs, we intentionally do not support sharing of VDIs. This eliminates the requirement for a distributed lock manager, and dramatically simplifies our design.

In this paper, we describe the design and implementation of Parallax as a storage system for the Xen virtual machine monitor. We demonstrate that the VM-based design allows Parallax to be implemented in user-space, allowing for a very fast development cycle. We detail a number of interesting aspects of Parallax: the optimizations required to maintain high throughput over fine grained block addressing, our fast snapshot facility, and the ability to mitigate congestion of shared storage by caching to local disks.

## 1.1 Related Work

Despite the many storage-related challenges present in virtualized environments, we are aware of only two other storage systems that cater specifically to VM deployments: Ventana [20] and VMware’s VMFS [29].

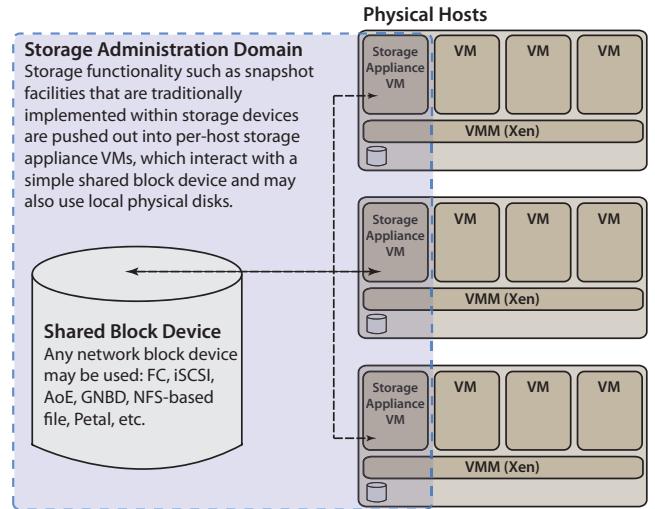
Ventana attempts to provide support for virtual machines at the file system level, effectively virtualizing the file system namespace and allowing individual VMs to share underlying file objects where possible. File system virtualization is a fundamentally different approach to the block-level virtualization provided by Parallax. Ventana provides an improved degree of “transparency” into the contents of virtual disks, but sacrifices generality in order to achieve it. Windows VMs, for instance, cannot be hosted off of the NFS interface that the Ventana server presents. Ventana’s authors do not evaluate its performance, but do mention that the system suffers as the number of branches (equivalent to snapshots in Parallax) increases.

VMFS is a commercial block-level storage virtualization system intended for use with VMware ESX. VMFS is certainly the most similar known system to Parallax; both approaches specifically address virtualized environments by providing distributed facilities to convert one large shared volume into a number of virtual disks for use by VMs. As it is proprietary software, little is known about the internals of VMFS’s design. However, it acts largely as a cluster file system, specifically tuned to host image files. Virtual disks themselves are stored within VMFS as VMDK [28] images. VMDK is a image format for virtual disks, similar to QCOW [17] and VHD [18], which provides sparseness and allows images to be “chained”. The performance of chained images decays linearly as the number of snapshots increases in addition to imposing overheads for open file handles and in-memory caches for each open image. In addition to chaining capabilities provided by VMDK, VMFS employs a redo log-based checkpoint facility that has considerable performance limitations [26]. Parallax directly manages the contents of disk images, and provides fine-grained sharing and snapshots as core aspects of its design.

Another approach that addresses issues similar to those of Parallax has been undertaken in recent work by the Emulab developers at the University of Utah [5]. In order to provide snapshots for Xen-based VMs, the researchers modified Linux LVM (Logical Volume Management) to provide a branching facility. No details are currently available on this implementation.

Beyond VM-specific approaches, many other systems provide virtual volumes in block-level storage, most notably FAB [7] and its predecessor Petal [14]. Both systems, particularly FAB, aim to provide a SAN-like feature set at a low total system cost. Both systems also support snapshots; the ability to snapshot in FAB is best manifest in Olive [10, 1].

Parallax differs from these prior block-level virtual disk systems in three ways. First, Parallax assumes the availability of a single shared block device, such as an iSCSI or FiberChannel LUN, NFS-based file, or Petal-like virtual disk, while FAB and similar systems compose a shared volume from a federation of storage devices. Whereas other systems must focus on coordination among distributed storage nodes, Parallax focuses on coordinating distributed clients sharing a network attached disk. By relying on virtualized storage in this manner, we address fundamentally different challenges. Second, because we provide the abstraction of a local disk to virtualized guest operating systems, we can make a reasonable assumption that disk images will be single-writer. This simplifies our system and enables aggressive performance optimization. Third, Parallax’s design and virtualized infrastructure enables us to rethink the traditional boundaries of a network storage system. In



**Figure 1:** Parallax is designed as a set of per-host storage appliances that share access to a common block device, and present virtual disks to client VMs.

addition, among block-level virtualization systems, only Olive [1] has a snapshot of comparable performance to ours. Olive’s snapshots have more complicated failure semantics than those of Parallax and the system imposes delays on write operations issued during a snapshot.

WAFL [9] has very similar goals to those of Parallax, and as a consequence results in a very similar approach to block address virtualization. WAFL is concerned with maintaining historical versions of the files in a network-attached storage system. It uses tree-based mapping structures to represent divergences between snapshots and to allow data to be written to arbitrary locations on the underlying disk. Parallax applies similar techniques at a finer granularity allowing snapshots of individual virtual disks, effectively the analogue of a single file in a WAFL environment. Moreover, Parallax has been designed to support arbitrary numbers of snapshots, as opposed to the hard limit of 255 snapshots available from current WAFL-based systems.

Many other systems have provided snapshots as a storage system feature, ranging from file system-level support in ZFS [22] to block-level volume management systems like LVM2 [21]. In every case these systems suffer from either a limited range of supported environments, severely limited snapshot functionality, or both. These limitations make them ill-suited for general deployment in virtualized storage infrastructures.

## 2. CLUSTERED STORAGE APPLIANCES

Figure 1 presents a high-level view of the structure of a Parallax-based cluster. Parallax provides block virtualization by interposing between individual virtual machines and the physical storage layer. The virtualized environment allows the storage virtualization service to be physically co-located with its clients. From an architectural perspective, this structure makes Parallax unique: the storage system runs in an isolated VM on each host and is administratively separate from the client VMs running alongside it; effectively, Parallax allows the storage system to be pushed out to include slices of each machine that uses it.

In this section, we describe the set of specific design considerations that have guided our implementation, and then present an overview of the system’s structure.

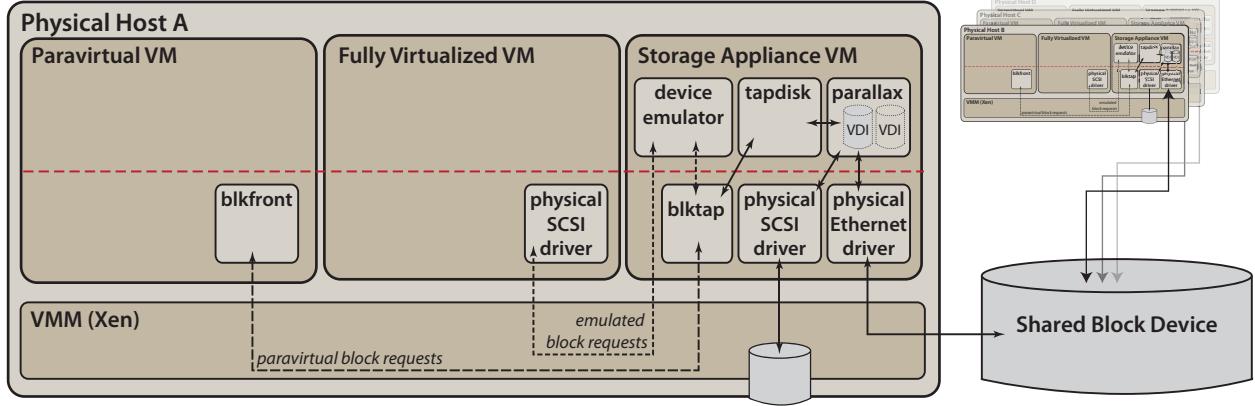


Figure 2: Overview of the Parallax system architecture.

## 2.1 Design Considerations

Parallax's design is based on four high-level themes:

**Agnosticism and isolation.** Parallax is implemented as a collaborative set of storage *appliances*; as shown in Figure 1, each physical host in a cluster contains a *storage VM* which is responsible for providing storage to other virtual machines running on that host. This VM isolates storage management and delivery to a single container that is administratively separate from the rest of the system. This design has been used previously to insulate running VMs from device driver crashes [6, 15], allowing drivers to be transparently restarted. Parallax takes this approach a step further to isolate storage virtualization in addition to driver code.

Isolating storage virtualization to individual per-host VMs results in a system that is agnostic to both the OSes that run in other VMs on the host, and the physical storage that backs VM data. A single cluster-wide administrator can manage the Parallax instances on each host, unifying the storage management role.

**Blocks not files.** In keeping with the goal of remaining agnostic to OSes running within individual VMs, Parallax operates at the block, rather than file-system, level. Block-level virtualization provides a narrow interface to storage, and allows Parallax to present simple virtual disks to individual VMs. While virtualization at the block level yields an agnostic and simple implementation, it also presents a set of challenges. The “semantic gap” introduced by virtualizing the system at a low level obscures higher-level information that could aid in identifying opportunities for sharing, and complicates request dependency analysis for the disk scheduler, as discussed in Section 5.1.

**Minimize lock management.** Distributed storage has historically implied some degree of concurrency control. Write sharing of disk data, especially at the file system level, typically involves the introduction of some form of distributed lock manager. Lock management is a very complex service to provide in a distributed setting and is notorious for difficult failure cases and recovery mechanisms. Moreover, although write conflict resolution is a well-investigated area of systems research, it is one for which no general solutions exist.

Parallax's design is premised on the idea that data sharing in a cluster environment should be provided by application-level services with clearly defined APIs, where concurrency and conflicts may be managed with application semantics in mind. Therefore, it *explicitly excludes* support for write-sharing of individual virtual disk images. The system ensures that each VDI has at most one writer, greatly reducing the need for concurrency control. Some

degree of concurrency management is still required, but only when performing administrative operations such as creating new VDIs, and in very coarse-grained allocations of writable areas on disk. Locking operations are explicitly not required as part of the normal data path or for snapshot operations.

**Snapshots as a primitive operation.** In existing storage systems, the ability to snapshot storage has typically been implemented as an afterthought, and for very limited use cases such as the support of backup services. Post-hoc implementations of snapshot facilities are typically complex, involve inefficient techniques such as redo logs [29], or impose hard limits on the maximum number of snapshots [9]. Our belief in constructing Parallax has been that the ability to take and preserve very frequent, low-overhead snapshots is an enabling storage feature for a wide variety of VM-related applications such as high-availability, debugging, and continuous data protection. As such, the system has been designed to incorporate snapshots from the ground up, representing each virtual disk as a set of radix-tree based block mappings that may be chained together as a potentially infinite series of copy-on-write (CoW) instances.

## 2.2 System structure

Figure 2 shows an overview of Parallax's architecture and allows a brief discussion of components that are presented in more detail throughout the remainder of the paper.

As discussed above, each physical host in the cluster contains a storage appliance VM that is responsible for mediating accesses to an underlying block storage device by presenting individual virtual disks to other VMs running on the host. This storage VM allows a single, cluster-wide administrative domain, allowing functionality that is currently implemented within enterprise storage hardware to be pushed out and implemented on individual hosts. The result is that advanced storage features, such as snapshot facilities, may be implemented in software and delivered above commodity network storage targets.

Parallax itself runs as a user-level daemon in the Storage Appliance VM, and uses Xen's block tap driver [31] to handle block requests. The block tap driver provides a very efficient interface for forwarding block requests from VMs to daemon processes that run in user space of the storage appliance VM. The user space portion of block tap defines an asynchronous disk interface and spawns a *tapdisk* process when a new VM disk is connected. Parallax is implemented as a tapdisk library, and acts as a single block virtualization service for all client VMs on the physical host.

Each Parallax instance shares access to a single shared block de-

vice. We place no restrictions as to what this device need be, so long as it is sharable and accessible as a block target in all storage VM instances. In practice we most often target iSCSI devices, but other device types work equally well. We have chosen that approach as it requires the lowest common denominator of shared storage, and allows Parallax to provide VM storage on the broadest possible set of targets.

Virtual machines that interact with Parallax are presented with entire virtual disks. Xen allows disks to be accessed using both emulated and paravirtualized interfaces. In the case of emulation, requests are handled by a device emulator that presents an IDE controller to the client VM. Emulated devices generally have poor performance, due to the context switching required to emulate individual accesses to device I/O memory. For performance, clients may install paravirtual device drivers, which are written specifically for Xen-based VMs and allow a fast, shared-memory transport on which batches of block requests may be efficiently forwarded. By presenting virtual disks over traditional block device interfaces as a storage primitive to VMs, Parallax supports any OS capable of running on the virtualized platform, meeting the goal of agnosticism.

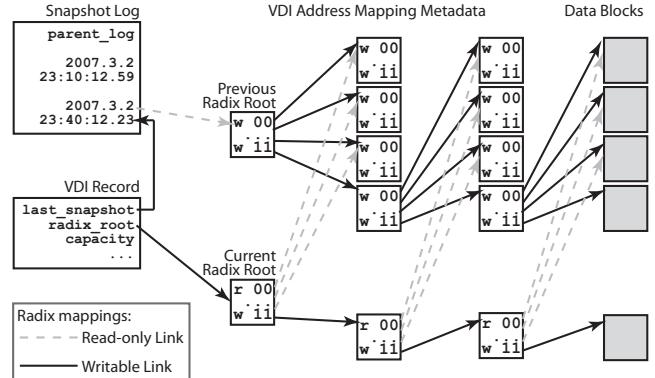
The storage VM is connected directly to physical device hardware for block and network access. Including physical block device drivers in the storage VM allows a storage administrator the ability to do live upgrades of block device drivers in an active cluster. This is an area of future exploration for us, but a very similar approach has been described previously [6].

### 3. VIRTUAL DISK IMAGES

Virtual Disk Images (VDIs) are the core abstraction provided by Parallax to virtual machines. A VDI is a single-writer virtual disk which may be accessed in a location transparent manner from any of the physical hosts in the Parallax cluster. Table 1 presents a summary of the administrative operations that may be performed on VDIs; these operations are available through the command line of the storage VM. There are three core operations, allowing VDIs to be created, deleted, and snapshotted. These are the only operations required to actively manage VDIs; once created, they may be attached to VMs as would any other block device. In addition to the three core operations, Parallax provides some convenience operations that allow an administrator to view catalogues of VDIs, snapshots associated with a particular VDI, and to “tag” particular snapshots with a human-readable alias, facilitating creation of new VDIs based on that snapshot in the future. An additional convenience function produces a simple visualization of the VDIs in the system as well as tagged snapshots.

#### 3.1 VDIs as Block Address Spaces

In order to achieve the design goals that have been outlined regarding VDI functionality, in particular the ability to take fast and frequent snapshots, Parallax borrows heavily from techniques used to manage virtual memory. A Parallax VDI is effectively a single *block* address space, represented by a radix tree that maps virtual block addresses to physical block addresses. Virtual addresses are a continuous range from zero to the size of the virtual disk, while physical addresses reflect the actual location of a block on the shared blockstore. The current Parallax implementation maps virtual addresses using 4K blocks, which are chosen to intentionally match block sizes used on x86 OS implementations. Mappings are stored in 3-level radix trees, also based on 4K blocks. Each of the radix metadata pages stores 512 64-bit global block address pointers, and the high-order bit is used to indicate that a link is read-only. This layout results in a maximum VDI size of 512GB (9 address bits per tree-level, 3 levels, and 4K data blocks yields



**Figure 3: Parallax radix tree (simplified with short addresses) and COW behaviour.**

$2^{9 \times 3} * 2^{12} = 2^{39} = 512\text{GB}$ . Adding a level to the radix tree extends this by a factor of  $2^9$  to 256TB and has a negligible effect on performance for small volumes (less than 512GB) as only one additional metadata node per active VDI need be cached. Parallax’s address spaces are sparse; zeroed addresses indicate that the range of the tree beyond the specified link is non-existent and must be allocated. In this manner, the creation of new VDIs involves the allocation of only a single, zeroed, root block. Parallax will then populate both data and metadata blocks as they are written to the disk. In addition to sparseness, references can be shared across descendant radix trees in order to implement snapshots.

#### 3.2 Snapshots

A snapshot in Parallax is a read-only image of an entire disk at a particular point in time. Like many other systems, Parallax always ensures that snapshots are *crash consistent*, which means that snapshots will capture a file system state that could have resulted from a crash [1] [14] [19] [27] [20]. While this may necessitate running an application or file system level disk check such as fsck, it is unlikely that any block-level system can offer stronger guarantees about consistency without coordination with applications and file systems.

Snapshots can be taken of a disk not currently in use, or they can be taken on a disk during its normal operation. In this latter case, the snapshot semantics are strictly *asynchronous*; snapshots are issued directly into the stream of I/O requests in a manner similar to write barriers. The snapshot is said to be “complete” when the structures associated with the snapshot are correctly placed on disk. These snapshot semantics enable Parallax to complete a snapshot without pausing or delaying the I/O requests, by allowing both pre-snapshot and post-snapshot I/O to complete on their respective views of the disk after the completion of the snapshot. Such an approach is ideal when issuing snapshots in rapid succession since the resulting snapshots have very little overhead, as we will show.

To implement snapshots, we use the high-order bit of block addresses in the radix tree to indicate that the block pointed to is read-only. All VDI mappings are traversed from a given radix root down the tree, and a read-only link indicates that the entire subtree is read-only. In taking a snapshot, Parallax simply copies the root block of the radix tree and marks all of its references as read-only. The original root need not be modified as it is only referenced by a snapshot log that is implicitly read-only. The entire process usually requires just three block-write operations, two of which can be performed concurrently.

The result of a snapshot is illustrated in Figure 3. The figure

<code>create(name, [snapshot]) → VDI_id</code>	Create a new VDI, optionally based on an existing snapshot. The provided name is for administrative convenience, whereas the returned VDI identifier is globally unique.
<code>delete(VDI_id)</code>	Mark the specified VDI as deleted. When the garbage collector is run, the VDI and all snapshots are freed.
<code>snapshot(VDI_id) → snap_id</code>	Request a snapshot of the specified VDI.
<code>list() → VDI_list</code>	Return a list of VDIs in the system.
<code>snap_list(VDI_id) → snap_list</code>	Return the log of snapshots associated with the specified VDI.
<code>snap_label(snap_id, name)</code>	Label the specified snapshot with a human-readable name.
<code>tree() → (tree view of VDIs)</code>	Produce a diagram of the current system-wide VDI tree (see Figure 4 for an example.)

Table 1: VDI Administrative Interfaces.

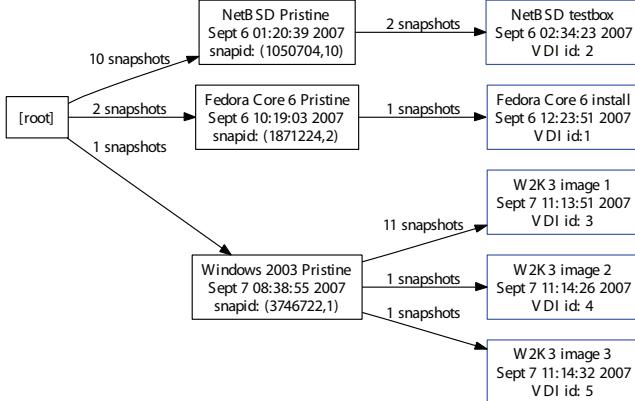


Figure 4: VDI Tree View—Visualizing the Snapshot Log.

shows a simplified radix tree mapping six-bit block addresses with two address bits per radix page. In the figure, a VDI has had a snapshot taken, and subsequently had a block of data written at virtual block address 111111 (*binary*). The snapshot operation copies the radix tree root block and redirects the VDI record to point to the new root. All of the links from the new root are made read-only, as indicated by the “*r*” flags and the dashed grey arrows in the diagram.

Copying a radix tree block always involves marking all links from that block as read-only. A snapshot is completed using one such block copy operation, following which the VM continues to run using the new radix tree root. At this point, data writes may not be applied in-place as there is no direct path of writable links from the root to any data block. The write operation shown in the figure copies every radix tree block along the path from the root to the data (two blocks in this example) and the newly-copied branch of the radix tree is linked to a freshly allocated data block. All links to newly allocated (or copied) blocks are writable links, allowing successive writes to the same or nearby data blocks to proceed with in-place modification of the radix tree. The active VDI that results is a copy-on-write version of the previous snapshot.

The address of the old radix root is appended, along with the current time-stamp, to a *snapshot log*. The snapshot log represents a history of all of a given VDI’s snapshots.

Parallax enforces the invariant that radix roots in snaplogs are immutable. However, they may be used as a reference to create a new VDI. The common approach to interacting with a snapshot is to create a writable VDI clone from it and to interact with that. A VM’s snapshot log represents a chain of dependent images from the current writable state of the VDI, back to an initial disk. When a new VDI is created from an existing snapshot, its snapshot log is made to link back to the snapshot on which it is based. Therefore,

the set of all snapshot logs in the system form a forest, linking all of the radix roots for all VDIs, which is what Parallax’s VDI tree operation generates, as shown in Figure 4. This aggregate snaplog tree is not explicitly represented, but may be composed by walking individual logs backwards from all writable VDI roots.

From a single-host perspective, the VDI and its associated radix mapping tree and snapshot logs are largely sufficient for Parallax to operate. However, these structures present several interesting challenges that are addressed in the following sections. Section 4 explains how the shared block device is managed to allow multiple per-host Parallax instances to concurrently access data without conflicts or excessive locking complexity. Parallax’s radix trees, described above, are very fine grained, and risk the introduction of a great deal of per-request latency. The system takes considerable effort, described in Section 5, to manage the request stream to eliminate these overheads.

## 4. THE SHARED BLOCKSTORE

Traditionally, distributed storage systems rely on distributed lock management to handle concurrent access to shared data structures within the cluster. In designing Parallax, we have attempted to avoid distributed locking wherever possible, with the intention that even in the face of disconnection<sup>1</sup> or failure, individual Parallax nodes should be able to continue to function for a reasonable period of time while an administrator resolves the problem. This approach has guided our management of the shared blockstore in determining how data is laid out on disk, and where locking is required.

### 4.1 Extent-based Access

The physical blockstore is divided, at start of day, into fixed-size extents. These extents are large (2GB in our current implementation) and represent a lockable single-allocator region. “Allocators” at this level are physical hosts—Parallax instances—rather than the consumers of individual VDIs. These extents are typed; with the exception of a special system extent at the start of the blockstore, extents either contain data or metadata. Data extents hold the actual data written by VMs to VDIs, while metadata extents hold radix tree blocks and snapshot logs. This division of extent content is made to clearly identify metadata, which facilitates garbage collection. In addition, it helps preserve linearity in the placement of data blocks, by preventing metadata from becoming intermingled with data. Both data and metadata extents start with an allocation bitmap that indicates which blocks are in use.

When a Parallax-based host attaches to the blockstore, it will exclusively lock a data and a metadata extent for its use. At this point, it is free to modify unallocated regions of the extent with no additional locking.<sup>2</sup> In order to survive disconnection from the

<sup>1</sup>This refers to disconnection from other hosts. A connection to the actual shared blockstore is still required to make forward progress.

<sup>2</sup>This is a white lie – there is a very coarse-grained lock on the allocation bitmaps used with the garbage collector, see Section 4.3.

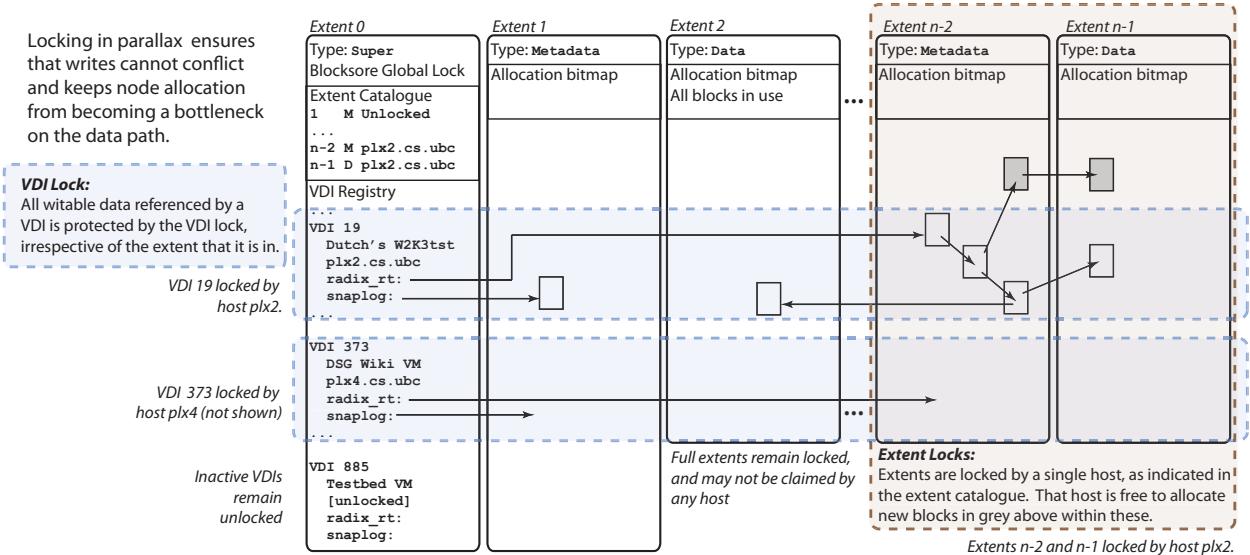


Figure 5: Blockstore Layout.

lock manager, Parallax nodes may lock additional unused extents to allow room for additional allocation beyond the capacity of active extents. We will likely optimize this further in the future by arranging for connected Parallax instances to each lock a share of the unallocated extents, further reducing the already very limited need for allocation-related locking.

The system extent at the front of the blockstore contains a small number of blockstore-wide data structures. In addition to system-wide parameters, like the size of the blockstore and the size of extents, it has a catalogue of all fixed-size extents in the system, their type (system, data, metadata, and unused), and their current lock-holder. It also contains the VDI registry, a tree of VDI structs, each stored in an individual block, describing all active VDIs in the system. VDIs also contain persistent lock fields and may be locked by individual Parallax instances. Locking a VDI struct provides two capabilities. First, the locker is free to write data within the VDI struct, as is required when taking a snapshot where the radix root address must be updated. Second, with the VDI struct locked, a Parallax instance is allowed to issue in-place writes to *any* blocks, data or metadata, referenced as writable through the VDI's radix root. The second of these properties is a consequence of the fact that a given (data or metadata) block is only ever marked writable within a *single* radix tree.

Figure 5 illustrates the structure of Parallax's blockstore, and demonstrates how extent locks allow a host to act as a single writer for new allocations within a given extent, while VDI locks allow access to allocated VDI blocks across all extents on the blockstore.

## 4.2 Lock Management

The protocols and data structures in Parallax have been carefully designed to minimize the need for coordination. Locking is required only for infrequent operations: to claim an extent from which to allocate new data blocks, to gain write access to an inactive VDI, or to create or delete VDIs. Unless an extent has exhausted its free space, no VDI read, write, or snapshot operation requires any coordination at all.

The VDI and extent locks work in tandem to ensure that the VDI owner can safely write to the VDI irrespective of its physical location in the cluster, even if the VDI owner migrates from one host

to another while running. The Parallax instance that holds the VDI lock is free to write to existing writable blocks in that VDI on *any* extent on the shared blockstore. Writes that require allocations, such as writes to read-only or sparse regions of a VDI's address space, are allocated within the extents that the Parallax instance has locked. As a VM moves across hosts in the cluster, its VDI is managed by different Parallax instances. The only effect of this movement is that new blocks will be allocated from a different extent.

The independence that this policy affords to each Parallax instance improves the scalability and reliability of the entire cluster. The scalability benefits are clear: with no lock manager acting as a bottleneck, the only limiting factor for throughput is the shared storage medium. Reliability is improved because Parallax instances can continue running in the absence of a lock manager as long as they have free space in the extents they have already claimed. Nodes that anticipate heavy block allocation can simply lock extra extents in advance.

In the case that a Parallax instance has exhausted its free space or cannot access the shared block device, the local disk cache described in Section 6.2.5 could be used for temporary storage until connectivity is restored.

Because it is unnecessary for data access, the lock manager can be very simple. In our implementation we designate a single node to be the lock manager. When the manager process instantiates, it writes its address into the special extent at the start of the blockstore, and other nodes use this address to contact the lock manager with lock requests for extents or VDIs. Failure recovery is not currently automated, but the system's tolerance for lock manager failure makes manual recovery feasible.

## 4.3 Garbage Collection

Parallax nodes are free to allocate new data to any free blocks within their locked extents. Combined with the copy-on-write nature of Parallax, this makes deletion a challenge. Our approach to reclaiming deleted data is to have users simply mark radix root nodes as deleted, and to then run a garbage collector that tracks metadata references across the entire shared blockstore and frees any unallocated blocks.

---

**Algorithm 1** The Parallax Garbage Collector

---

1. Checkpoint Block Allocation Maps (BMaps) of extents.
2. Initialize the Reachability Map (RMap) to zero.
3. For each VDI in the VDI registry:
  - If VDI is not marked as deleted:
    - Mark its radix root in the RMap.
    - For each snapshot in its snaplog
      - If snapshot is not marked as deleted:
        - Mark its radix root in the RMap.
  4. For each Metadata extent:
    - Scan its RMap. If a page is marked:
      - Mark all pages (in the RMap) that it points to.
  5. Repeat step 4 for each level in the radix tree.
  6. For each VDI in the VDI registry:
    - If VDI is marked as not deleted:
      - Mark each page of its snaplog in the RMap.
  7. For each extent:
    - Lock the BMap.
    - For each unmarked bit in the RMap:
      - If it is marked in the BMap as well as in the checkpointed copy of the BMap :
        - Unmark the BMap entry and reclaim the block.
    - Unlock the BMap.

---

Parallax’s garbage collector is described as Algorithm 1. It is similar to a mark-and-sweep collector, except that it has a fixed, static set of passes. This is possible because we know that the maximum length of any chain of references is the height of the radix trees. As a result we are able to scan the metadata blocks in (disk) order rather than follow them in the arbitrary order that they appear in the radix trees. The key data structure managed by the garbage collector is the *Reachability Map* (RMap), an in-memory bitmap with one bit per block in the blockstore; each bit indicates whether the corresponding block is reachable.

A significant goal in the design of the garbage collector is that it interfere as little as possible with the ongoing work of Parallax. While the garbage collector is running, Parallax instances are free to allocate blocks, create snapshots and VDIs, and delete snapshots and VDIs. Therefore the garbage collector works on a “checkpoint” of the state of the system at the point in time that it starts. Step 1 takes an on-disk read-only copy of all block allocation maps (BMaps) in the system. Initially, only the radix roots of VDIs and their snapshots are marked as reachable. Subsequent passes mark blocks that are reachable from these radix roots and so on. In Step 5, the entire RMap is scanned every time. This results in re-reading nodes that are high in the tree, a process that could be made more efficient at the cost of additional memory. The only blocks that the collector considers as candidates for deallocation are those that were marked as allocated in the checkpoint taken in Step 1 (see Step 7). The only time that the collector interferes with ongoing Parallax operations is when it updates the (live) allocation bitmap for an extent to indicate newly deallocated blocks. For this operation it must coordinate with the Parallax instance that owns the extent to avoid simultaneous updates, thus the BMap must be locked in Step 7. Parallax instances claim many free blocks at once when looking at the allocation bitmap (currently 10,000), so this lock suffers little contention.

We discuss the performance of our garbage collector during our system evaluation in Section 6.2.3.

## 4.4 Radix Node Cache

Parallax relies on caching of radix node blocks to mitigate the overheads associated with radix tree traversal. There are two aspects of Parallax’s design that makes this possible. First, single-writer semantics of virtual disk images remove the need for any cache coherency mechanisms. Second, the ratio of data to metadata is approximately 512:1, which makes caching a large proportion of the radix node blocks for any virtual disk feasible. With our current default cache size of just 64MB we can fully accommodate a working set of nearly 32GB of data. We expect that a production-grade Parallax system will be able to dedicate a larger portion of its RAM to caching radix nodes. To maintain good performance, our cache must be scaled linearly with the working set of data.

The cache replacement algorithm is a simple numerical hashing based on block address. Since this has the possibility of thrashing or evicting a valuable root node in favour of a low-level radix node, we have plan to implement and evaluate a more sophisticated page replacement algorithm in the future.

## 4.5 Local Disk Cache

Our local disk cache allows persistent data to be written by a Parallax host without contacting the primary shared storage. The current implementation is in a prototype phase. We envision several eventual applications for this approach. The first is to mitigate the effects of degraded network operation by temporarily using the disk as a cache. We evaluate this technique in Section 6.2.5. In the future we plan to use this mechanism to support fully disconnected operation of a physical host.

The local disk cache is designed as a log-based ring of write requests that would have otherwise been sent to the primary storage system. The write records are stored in a file or raw partition on the local disk. In addition to its normal processing, Parallax consumes write records from the front of the log and sends them to the primary storage system. By maintaining the same write ordering we ensure that the consistency of the remote storage system is maintained. When the log is full, records must be flushed to primary storage before request processing can continue. In the event of a physical host crash, all virtual disks (which remain locked) must be quiesced before the virtual disk can be remounted.

A drawback to this approach is that it incorporates the physical host’s local disk into the failure model of the storage system. Users must be willing to accept the minimum of the reliability of the local disk and that of the storage system. For many users, this will mean that a single disk is unacceptable as a persistent cache, and that the cache must be stored redundantly to multiple local disks.

## 5. THE BLOCK REQUEST STREAM

While Parallax’s fine-grained address mapping trees provide efficient snapshots and sharing of block data, they risk imposing a high performance cost on block requests. At worst, accessing a block on disk can incur three dependent metadata reads that precede the actual data access. Given the high cost of access to block devices, it is critical to reduce this overhead. However, Parallax is presenting virtual block devices to the VMs that use it; it must be careful to provide the semantics that OSes expect from their disks. This section discusses how Parallax aggressively optimizes the block request stream while ensuring the correct handling of block data.

### 5.1 Consistency and Durability

Parallax is designed to allow guest operating systems to issue and receive I/O requests with the same semantics that they would to a local disk. VMs see a virtual SCSI-like block device; our current implementation allows a guest to have up to 64 requests in-flight,

and in-flight requests may complete in any order. Parallax does not currently support any form of tag or barrier operation, although this is an area of interest for future work; at the moment guest OSes must allow the request queue to drain in order to ensure that all issued writes have hit the disk. We expect that the addition of barriers will further improve our performance by better saturating the request pipeline.

While in-flight requests may complete out of order, Parallax must manage considerable internal ordering complexity. Consider that each *logical* block request, issued by a guest, will result in a number of *component* block requests to read, and potentially update metadata and finally data on disk. Parallax must ensure that these component requests are carefully ordered to provide both the consistency and durability expected by the VM. These expectations may be satisfied through the following two invariants:

1. Durability is the guest expectation that acknowledged write requests indicate that data has been written to disk.<sup>3</sup> To provide durability, Parallax cannot notify the guest operating system that a logical I/O request has completed until all component I/O requests have committed to physical storage.
2. Consistency is the guest expectation that its individual block requests are atomic—that while system crashes may lose in-flight logical requests, Parallax will not leave its own metadata in an invalid state.

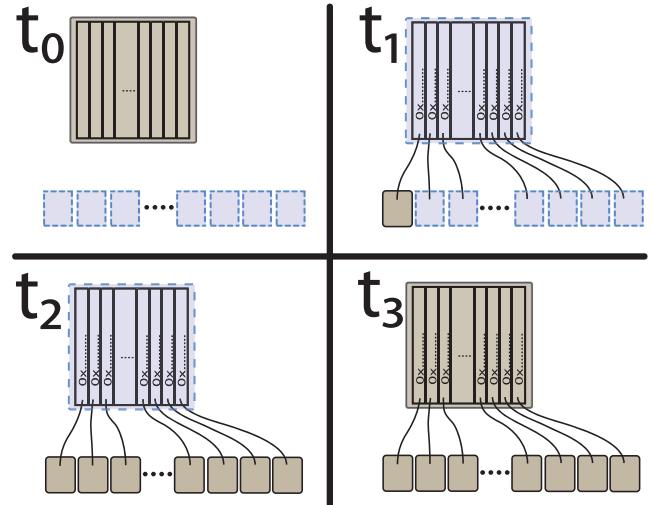
In satisfying both of these properties, Parallax uses what are effectively soft updates [16]. All dependent data and metadata are written to disk before updates are made that reference this data from the radix tree, described in the previous section. For any VDI, all address lookups must start at the radix root. When a write is being made, either all references from the top of the tree down to the data block being written are writable, in which case the write may be made in-place, or there is an intermediate reference that is read-only or sparse. In cases where such a reference exists, Parallax is careful to write all tree data below that reference to disk *before* updating the reference on disk. Thus, to satisfy consistency for each logical request, Parallax must not modify nodes in the on-disk tree until all component requests affecting lower levels of the tree have been committed to disk.

We refer to the block that contains this sparse or read-only reference as a *commit node*, as updates to it will atomically add all of the new blocks written below it to the lookup tree. In the case of a crash, some nodes may have been written to disk without their commit nodes. This is acceptable, because without being linked into a tree, they will never be accessed, and the corresponding write will have failed. The orphaned nodes can be returned to the blockstore through garbage collection.

## 5.2 Intra-request Dependencies

Logical requests that are otherwise independent can share commit nodes in the tree. During writes, this can lead to nodes upon which multiple logical requests are dependent. In the case of a shared commit node, we must respect the second invariant for both nodes independently. In practice this is a very common occurrence.

This presents a problem in scheduling the write of the shared commit node. In Figure 6, we provide an example of this behaviour. The illustration shows a commit node and its associated data at four monotonically increasing times. At each time, nodes and data



**Figure 6: Example of a shared write dependency.**

blocks that are flushed to disk and synchronized in memory appear darker in color, and are bordered with solid lines. Those blocks that appear lighter and are bordered with dashed lines have been modified in memory but those modifications have not yet reached disk.

The illustration depicts the progress of  $n$  logical write requests,  $a_0$  through  $a_n$ , all of which are sequential and share a commit node. For simplicity, this example will consider what is effectively a radix tree with a single radix node; the Parallax pipeline behaves analogously when a full tree is present. At time  $t_0$ , assume for the purpose of illustration that we have a node, in memory and synchronized to disk, that contains no references to data blocks. At this time we receive the  $n$  requests in a single batch, we begin processing the requests issuing the data blocks to the disk, and updating the root structure in memory. At time  $t_1$  we have made all updates to the root block in memory, and a write of one of the data blocks has been acknowledged by the storage system. We would like to complete the logical request  $a_0$  as quickly as possible but we cannot flush the commit node in its given form, because it still contains references to data blocks that have not been committed to disk. In this example, we wait. At time  $t_2$ , all data blocks have successfully been committed to disk; this is the soonest time that we can finally proceed to flush the commit node. Once that request completes at time  $t_3$ , we can notify the guest operating system that the associated I/O operations have completed successfully.

The latency for completing request  $a_0$  is thus the sum of the time required to write the data for the subsequent  $n - 1$  requests, plus the time required to flush the commit node. The performance impact can be further compounded by the dependency requirements imposed by a guest file system. These dependencies are only visible to Parallax in that the guest file system may stop issuing requests to Parallax due to the increased latency on some previously issued operation.

For this reason, commit nodes are the fundamental “dial” for trading off batching versus latency in the request pipeline. In the case of sequential writes, where all outstanding writes (of which there are a finite number) share a common commit node, it is possible in our current implementation that all in-flight requests must complete before any notifications may be passed back to the guest, resulting in bubbles while we wait for the guest to refill the request

<sup>3</sup>Or has at least been acknowledged as being written by the physical block device.

pipeline in response to completion notifications. We intend to address this by limiting the number of outstanding logical requests that are dependent on a given commit node, and forcing the node to be written once this number exceeds a threshold, likely half of the maximum in-flight requests. Issuing intermediate versions of the commit node will trade off a small number of additional writes for better interleaving of notifications to the guest. This technique was employed in [8]. As a point of comparison, we have disabled the dependency tracking between nodes, allowing them to be flushed immediately. Such an approach yields a 5% increase in sequential write performance, thought it is obviously unsafe for normal operation. With correct flushing of intermediate results we may be able to close this performance gap.

### 5.3 Snapshots in the Pipeline

Our snapshot semantics enable Parallax to complete a snapshot without pausing or delaying I/O requests, by allowing both pre-snapshot and post-snapshot operations to complete on their respective views of the disk after the completion of the snapshot. This capability is facilitated by both our single-writer assumptions and our client-oriented design. In systems where distributed writes to shared data must be managed, a linearizability of I/O requests around snapshots must be established, otherwise there can be no consensus about the correct state of a snapshot. In other systems, this requires pausing the I/O stream to some degree. A simple approach is to drain the I/O queue entirely [14], while a more complicated approach is to optimistically assume success and retry I/O that conflicts with the snapshot [1]. Linearization in Parallax comes naturally because each VDI is being written to by at most one physical host.

## 6. EVALUATION

We now consider Parallax’s performance. As discussed in previous sections, the design of our system includes a number of factors that we expect to impose considerable overheads on performance. Block address virtualization is provided by the Parallax daemon, which runs in user space in an isolated VM and therefore incurs context-switching on every batch of block requests. Additionally, our address mapping metadata involves 3-level radix trees, which risks a dramatic increase in the latency of disk accesses due to seeks on uncached metadata blocks.

There are two questions that this performance analysis attempts to answer. First, what are the overheads that Parallax imposes on the processing of I/O requests? Second, what are the performance implications of the virtual machine specific features that Parallax provides? We address these questions in turn, using sequential read and write [3] (in Section 6.1.1) and PostMark [11] (in Section 6.1.2) to answer the first and using a combination of micro and macro-benchmarks to address the second.

In all tests, we use IBM eServer x306 machines, each node with a 3.2 GHz Pentium-4 processor, 1 GByte of RAM, and an Intel e1000 GbE network interface. Storage is provided by a NetApp FAS3070<sup>4</sup> exporting an iSCSI LUN over gigabit links. We access the filer in all cases using the Linux open-iSCSI software initiator (v2.0.730, and kernel module v1.1-646) running in domain 0. We have been developing against Xen 3.1.0 as a base. One notable modification that we have made to Xen has been to double

<sup>4</sup>We chose to benchmark against the FAS 3070 because it is simply the fastest iSCSI target available to us. This is the UBC CS department filer, and so has required very late-night benchmarking efforts. The FAS provides a considerable amount of NVRAM on the write path, which explains the asymmetric performance between read and write in many of our benchmark results.

the maximum number of block requests, from 32 to 64, that a guest may issue at any given time, by allocating an additional shared ring page in the split block (blkback) driver. The standard 32-slot rings were shown to be a bottleneck when connecting to iSCSI over a high capacity network.

### 6.1 Overall performance

It is worth providing a small amount of additional detail on each of the test configurations that we compare. Our analysis compares access to the block device from Xen’s domain 0 (dom0 in the graphs), to the block device directly connected to a guest VM using the block back driver (blkback), and to Parallax. Parallax virtualizes block access through blktap [31], which facilitates the development of user-mode storage drivers.

Accessing block devices from dom0 has the least overhead, in that there is no extra processing required on block requests and dom0 has direct access to the network interface. This configuration is effectively the same as unvirtualized Linux with respect to block performance. In addition, in dom0 tests, the full system RAM and both hyperthreads are available to dom0. In the following cases, the memory and hyperthreads are equally divided between dom0 (which acts as the Storage VM<sup>5</sup>) and a guest VM.

In the “Direct” case, we access the block device from a guest VM over Xen’s blkback driver. In this case, the guest runs a block driver that forwards requests over a shared memory ring to a driver (blkback) in dom0, where they are issued to the iSCSI stack. Dom0 receives direct access to the relevant guest pages, so there is no copy overhead, but this case does incur a world switch between the client VM and dom0 for each batch of requests.

Finally, in the case of Parallax, the configuration is similar to the direct case, but when requests arrive at the dom0 kernel module (blktap instead of blkback), they are passed on to the Parallax daemon running in user space. Parallax issues reads and writes to the Linux kernel using Linux’s asynchronous I/O interface (libaio), which are then issued to the iSCSI stack.

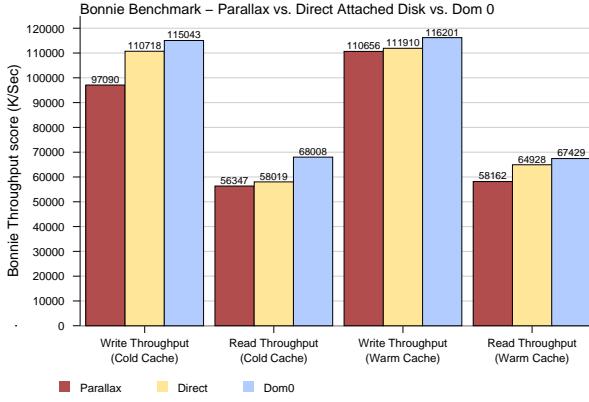
Reported performance measures a best of 3 runs for each category. The alternate convention of averaging several runs results in slightly lower performance for dom0 and direct configurations relative to Parallax. Memory and CPU overheads were shown to be too small to warrant their inclusion here.

#### 6.1.1 Sequential I/O

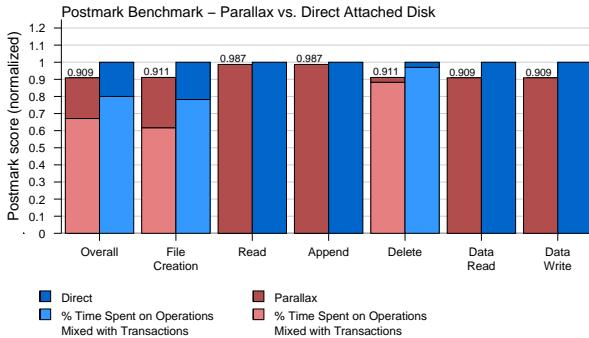
For each of the three possible configurations, we ran Bonnie++ twice in succession. The first run provided cold-cache data points, while the second allows Parallax to populate its radix node cache<sup>6</sup>. The strong write performance in the warm cache case demonstrates that Parallax is able to maintain write performance near the effective line speed of a 1Gbps connection. Our system performance is within 5% of dom0. At the same time, the 12% performance degradation in the cold cache case underscores the importance of caching in Parallax, as doing so limits the overheads involved in radix tree traversal. As we have focused our efforts to date on tuning the write path, we have not yet sought aggressive optimizations for read operations. This is apparent in the Bonnie++ test, as we can see read performance slipping to more than 14% lower than that of our non-virtualized dom0 configuration.

<sup>5</sup>We intend to explore a completely isolated Storage VM configuration as part of future work on live storage system upgrades.

<sup>6</sup>In the read path, this may also have some effect on our filer’s caching; however, considering the small increase in read throughput and the fact that a sequential read is easily predictable, we conclude that these effects are minimal.



**Figure 7:** System throughput as reported by Bonnie++ during a first (cold) and second (warm) run.



**Figure 8:** PostMark results running against network available filer (normalized).

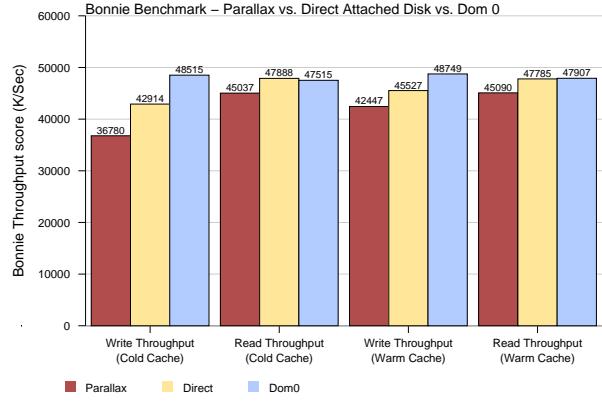
### 6.1.2 PostMark

Figure 8 shows the results of running PostMark on the Parallax and directly attached configurations. PostMark is designed to model a heavy load placed on many small files [11]. The performance of Parallax is comparable to and slightly lower than that of the directly connected configuration. In all cases we fall within 10% of a directly attached block device. File creation and deletion are performed during and after the transaction phase of the PostMark test, respectively. We have merged both phases, and illustrated the relative time spent in each.

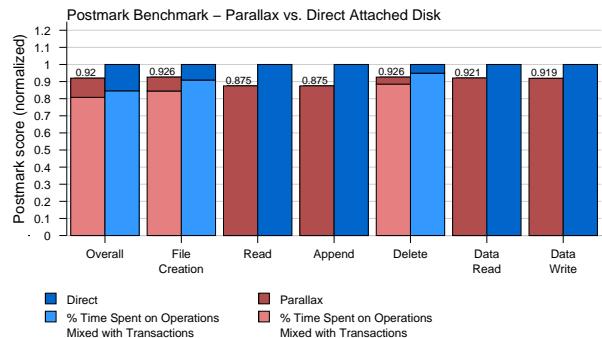
### 6.1.3 Local Disk Performance

To demonstrate that a high-end storage array with NVRAM is not required to maintain Parallax’s general performance profile, we ran the same tests using a commodity local disk as a target. Our disk was a Hitachi Deskstar 7K80, which is an 80GB, 7,200 RPM SATA drive with an 8MB cache. The results of Bonnie++ are shown in Figure 9. Again, the importance of maintaining a cache of intermediate radix nodes is clear. Once the system has been in use for a short time, the write overheads drop to 13%, while read overheads are shown to be less than 6%. In this case, Parallax’s somewhat higher I/O requirements increase the degree to which the local disk acts as a bottleneck. The lack of tuning of read operations is not apparent at this lower throughput.

In Figure 10 we show the results of running the PostMark test



**Figure 9:** System throughput against a local disk as reported by Bonnie++ during a first (cold) and second (warm) run.



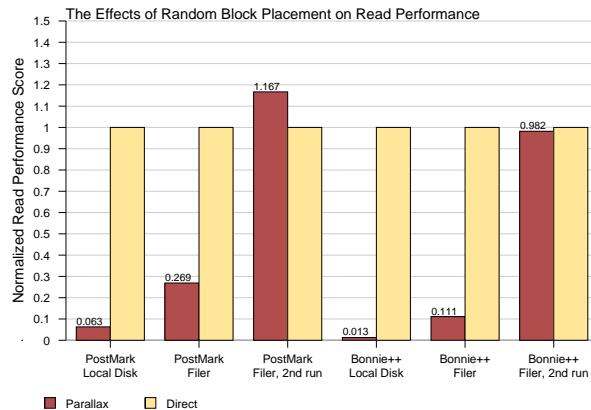
**Figure 10:** PostMark results running against a local disk (normalized).

with a local disk, as above. Similarly, the results show a only small performance penalty when Parallax is used without the advantages of striping disks or a large write cache.

## 6.2 Measuring Parallax’s Features

### 6.2.1 Disk Fragmentation

While our approach to storage provides many beneficial properties, it raises concerns over how performance will evolve as a block-store ages. The natural argument against any copy-on-write based system is that the resulting fragmentation of blocks will prove detrimental to performance. In Parallax, fragmentation occurs when the block addresses visible to the guest VM are sequentially placed, but the corresponding physical addresses are not. This can come as a result of several usage scenarios. First, when a snapshot is deleted, it can fragment the allocation bitmaps forcing future sequential writes to be placed non-linearly. Second, if a virtual disk is sparse, future writes may be placed far from other blocks that are adjacent in the block address space. Similarly, when snapshots are used, the CoW behaviour can force written blocks to diverging locations on the physical medium. Third, the interleaving of writes to multiple VDIs will result in data for each virtual disk being placed together on the physical medium. Finally, VM migration will cause the associated Parallax virtual disks to be moved to new physical hosts, which will in turn allocate from different extents. Thus data



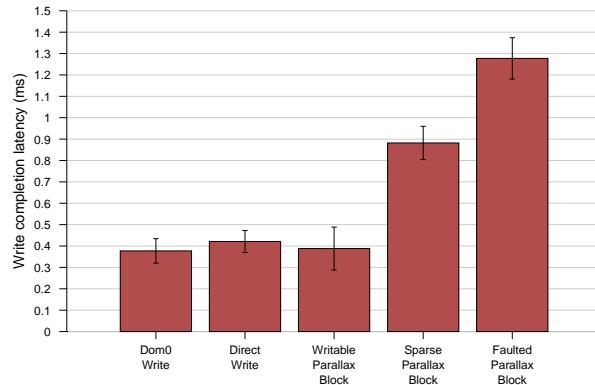
**Figure 11:** The effects of a worst case block allocation scheme on Parallax performance.

allocations after migration will not be located near those that occurred before migration. Note however that fragmentation will not result from writing data to blocks that are not marked read-only, as this operation will be done in place. In addition, sequential writes that target a read-only or sparse region of a virtual disk will remain sequential when they are written to newly allocated regions. This is true even if the original write-protected blocks were not linear on disk, due to fragmentation.

Thus, as VDIs are created, deleted, and snapshotted, we intuitively expect that some fragmentation of the physical media will occur, potentially incurring seeks even when performing sequential accesses to the virtual disk. To explore this possibility further, we modified our allocator to place new blocks randomly in the extent, simulating a worst-case allocation of data. We then benchmarked local disk and filer read performance against the resulting VDI, as shown in Figure 11.

Even though this test is contrived to place extreme stress on disk performance, the figure presents three interesting results. First, although it would be difficult to generate such a degenerate disk in the normal use of Parallax, in this worst case scenario, random block placement does incur a considerable performance penalty, especially on a commodity disk. In addition, the test confirms that the overheads for Bonnie++, which emphasizes sequential disk access, are higher than those for PostMark, which emphasizes smaller reads from a wider range of the disk. Interestingly, the third result is that when the workload is repeated, the filer is capable of regaining most of the lost performance, and even outperforms PostMark with sequential allocation. Although a conclusive analysis is complicated by the encapsulated nature of the filer, this result demonstrates that the increased reliance on disk striping, virtualized block addressing, and intelligent caching makes the fragmentation problem both difficult to characterize and compelling. It punctuates the observation made by Stein et al [25], that storage stacks have become incredibly complex and that naive block placement does not necessarily translate to worse case performance - indeed it can prove beneficial.

As a block management system, Parallax is well positioned to tackle the fragmentation problem directly. We are currently enhancing the garbage collector to allow arbitrary block remapping. This facility will be used to defragment VDIs and data extents, and to allow the remapping of performance-sensitive regions of disk into large contiguous regions that may be directly referenced at



**Figure 12:** Single write request latency for dom0, direct attached disks, and three potential Parallax states. A 95% confidence interval is shown.

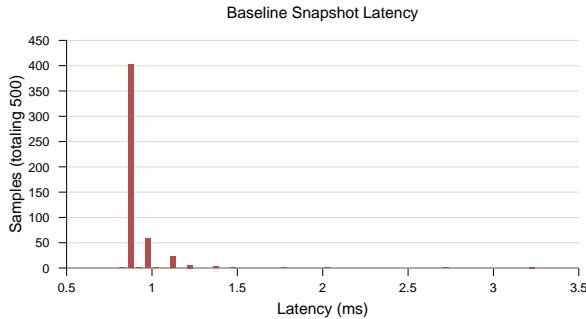
higher levels in the metadata tree, much like the concept of superpages in virtual memory. These remapping operations are independent of the data path, just like the rest of the garbage collector. Ultimately, detailed analysis of these features, combined with a better characterization of realistic workloads, will be necessary to evaluate this aspect of Parallax’s performance.

### 6.2.2 Radix tree overheads

In order to provide insight into the servicing of individual block requests, we use a simple microbenchmark to measure the various overheads. There are three distinct kinds of nodes in a radix tree. A node may be writable, which allows in-place modification. It may be sparse, in that it is marked as non-existent by its parent. Finally, it may be read-only, requiring that the contents be copied to a newly block in order to process write requests. We instrumented Parallax to generate each of these types of nodes at the top level of the tree, to highlight their differences. When non-writable nodes are reached at lower levels in the tree, the performance impact will be less notable. Figure 12 shows the results. Unsurprisingly, when a single block is written, Parallax performs very similarly to the other configurations, because writing is done in place. When a sparse node is reached at the top of the radix tree, Parallax must perform writes on intermediate radix nodes, the radix root, and the actual data. Of these writes, the radix root can only complete after all other requests have finished, as was discussed in Section 5. The faulted case is similar in that it too requires a serialized write, but it also carries additional overheads in reading and copying intermediate tree nodes.

### 6.2.3 Garbage collection

As described in Section 4.3, the Parallax garbage collector works via sequential scans of all metadata extents. As a result, the performance of the garbage collector is determined by the speed of reading metadata and the amount of metadata, and is independent of both the complexity of the forest of VDIs and their snapshots and the number of deleted VDIs. We’ve run the garbage collector on full blockstores ranging in size from 10GB to 50GB, and we characterize its performance by the amount of data it can process (measured as the size of the blockstore) per unit time. Its performance is linear at a rate of 0.96GB/sec. This exceeds the line speed of the storage array, because leaf nodes do not need to be read to determine if they can be collected.



**Figure 13: Snapshot latency of running VM during constant checkpointing.**

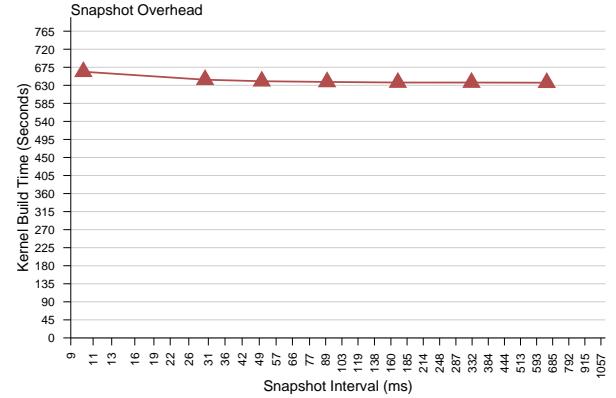
The key to the good performance of the garbage collector is that the Reachability Map is stored in memory. In contrast to the Block Allocation Maps of each extent which are always scanned sequentially, the RMap is accessed in random order. This puts a constraint on the algorithm’s scalability. Since the RMap contains one bit per blockstore block, each 1GB of memory in the garbage collector allows it to manage 32TB of storage. To move beyond those constraints, RMap pages can be flushed to disk. We look forward to having to address this challenge in the future, should we be confronted with a sufficiently large Parallax installation.

#### 6.2.4 Snapshots

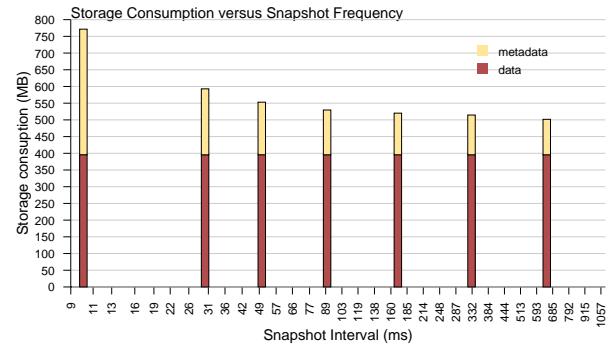
To establish baseline performance, we first measured the general performance of checkpointing the storage of a running but idle VM. We completed 500 checkpoints in a tight loop with no delay. A histogram of the time required by each checkpoint is given in Figure 13. The maximum observed snapshot latency in this test was 3.25ms. This is because the 3 writes required for most snapshots can be issued with a high degree of concurrency and are often serviced by the physical disk’s write cache. In this test, more than 90% of snapshots completed within a single millisecond; however, it is difficult to establish a strong bound on snapshot latency. The rate at which snapshots may be taken depends on the performance of the underlying storage and the load on Parallax’s I/O request pipeline. If the I/O pipeline is full, the snapshot request may be delayed as Parallax services other requests. Average snapshot latency is generally under 10ms, but under very heavy load we have observed average snapshot latency to be as high as 30ms.

Next we measured the effects of varying snapshot rates during the decompression and build of a Linux 2.6 kernel. In Figure 14 we provide results for various sub-second snapshot intervals. While this frequency may seem extreme, it explores a reasonable space for applications that require near continuous state capture. Larger snapshot intervals were tested as well, but had little effect on performance. The snapshot interval is measured as the average time between successive snapshots and includes the actual time required to complete the snapshot. By increasing the snapshot rate from 1 per second to 100 per second we incur only a 4% performance overhead. Furthermore, the majority of this increase occurs as we move from a 20ms to 10ms interval.

Figure 15 depicts the results of the same test in terms of data and metadata creation. The data consumption is largely fixed over all tests because kernel compilation does not involve overwriting previously written data, thus the snapshots have little effect on the number of data blocks created. In the extreme, taking snapshots



**Figure 14: Measuring performance effects of various snapshot intervals on a Linux Kernel decompression and compilation.**



**Figure 15: Measuring data consumption at various snapshot intervals on a Linux Kernel decompression and compilation.**

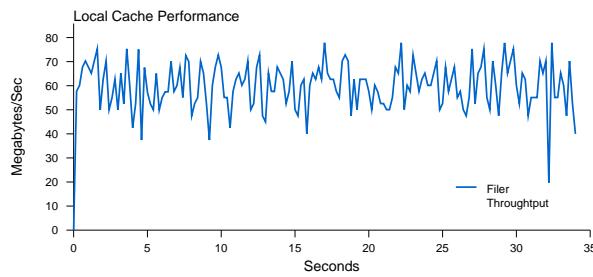
every 10ms, 65,852 snapshots were created, each consuming just 5.84KB of storage on average. This accounted for 375 MB of metadata, roughly equal in size to the 396 MB of data that was written.

Snapshot per Write	877.921 seconds	1188.59 MB
Snapshot per Batch	764.117 seconds	790.46 MB

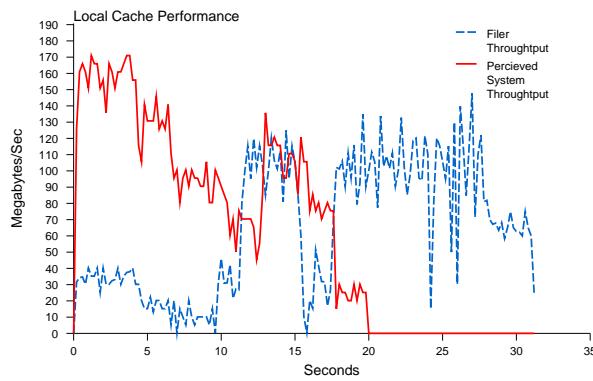
**Table 2: Alternate snapshot configurations.**

To further explore the potential of snapshots, we created two alternate modes to investigate even more fine-grained state capture in Parallax. In the first case we snapshot after each batch of requests; this enables data retention without capturing the unchanging disk states between writes. In our second snapshot mode, we perform a snapshot after every write request. Owing to the experimental nature of this code, our implementation is unoptimized. Even though the results are good, we expect there is significant room for improvement<sup>7</sup>. The impact on the performance of the kernel compile is shown in Table 2. When taking a snapshot after every data write, for every data block we consume 3 metadata blocks for the radix tree nodes and a few bytes for the entry in the snapshot log.

<sup>7</sup>Our current implementation does not support concurrent snapshots; we will remove this restriction in the future.



**Figure 16: Performance of bursted write traffic.**



**Figure 17: Performance of bursted write traffic with local disk caching.**

### 6.2.5 Local Disk Cache

We evaluated our local disk cache to illustrate the advantage of shaping the traffic of storage clients accessing a centralized network storage device. We have not yet fully explored the performance of caching to local disk in all scenarios, as its implementation is still in an early phase. The following experiment is not meant to exhaustively explore the implications of this technique, merely to illustrate its use and current implementation. In addition, the local disk cache demonstrates the ease with which new features may be added to Parallax, owing to its clean isolation from both the physical storage system and the guest operating system. The local disk cache is currently implemented in less than 500 lines of code.

In Figure 16, we show the time required to process 500MB of write traffic by 4 clients simultaneously. This temporary saturation of the shared storage resource may come as a result of an unusual and temporary increase in load, such as occurs when a system is initially brought online. This scenario results in a degradation of per-client performance, even as the overall throughput is high.

In Figure 17 we performed the same test with the help of our local disk cache. The Storage VMs each quickly recognized increased latency in their I/O requests to the filer and enabled their local caches. As a result, clients perceived an aggregate increase in throughput, because each local disk can be accessed without interference from competing clients. In the background, writes that had been made to the local cache were flushed to network storage without putting too much strain on the shared resource. Clients processed the workload in significantly less time (18-20 seconds). A short time after the job completed, the cache was fully drained, though this background process was transparent to users.

### 6.2.6 Metadata consumption

While there are some large metadata overheads, particularly in the initial extent, we expect that metadata consumption in Parallax will be dominated by the storage of radix nodes. Measuring this consumption is difficult, because it is parameterized by not only the image size, but also the sparseness of the images, the system-wide frequency and quality of snapshots, and the degree of sharing involved. To simplify this problem, we consider only the rate of radix nodes per data block on an idealized system.

In a full tree of height three with no sparseness we must create a radix node for every 512 blocks of data, an additional node for every 262,144 blocks of data, and finally a root block for the whole disk. With a standard 4KB blocks size, for 512GB of data, we must store just over 1GB of data in the form of radix nodes. Naturally for a non-full radix tree, this ratio could be larger. However, we believe that in a large system, the predominant concern is the waste created by duplication of highly redundant system images — a problem we explicitly address.

## 7. CONCLUSIONS AND FUTURE WORK

Parallax is a system that attempts to provide storage virtualization specifically for virtual machines. The system moves functionality, such as volume snapshots, that is commonly implemented on expensive storage hardware out into a software implementation running within a VM on the physical host that consumes the storage. This approach is a novel organization for a storage system, and allows a storage administrator access to a cluster-wide administration domain for storage. Despite its use of several potentially high-overhead techniques, such as a user-level implementation and fine-grained block mappings through 3-level radix trees, Parallax achieves good performance against both a very fast shared storage target and a commodity local disk.

We are actively exploring a number of improvements to the system including the establishing of a dedicated storage VM, the use of block remapping to recreate the sharing of common data as VDIs diverge, the creation of superpage-style mappings to avoid the overhead of tree traversals for large contiguous extents, and exposing Parallax's snapshot and dependency tracking features as primitives to the guest file system. As an alternative to using a single network available disk, we are designing a mode of operation in which Parallax itself will manage multiple physical volumes. This may prove a lower cost alternative to large sophisticated arrays.

We are continually making performance improvements to Parallax. As part of these efforts we are also testing Parallax on a wider array of hardware. We plan to deploy Parallax as part of an experimental VM-based hosting environment later this year. This will enable us to refine our designs and collect more realistic data on Parallax's performance. An open-source release of Parallax, with current performance data, is available at:  
<http://dsg.cs.ubc.ca/parallax/>.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough and encouraging feedback. They would also like to thank Michael Sanderson and the UBC CS technical staff for their enthusiastic support, which was frequently beyond the call of duty. This work is supported by generous grants from Intel Research and the National Science and Engineering Research Council of Canada.

## 8. REFERENCES

- [1] M. K. Aguilera, S. Spence, and A. Veitch. Olive: distributed point-in-time branching storage for real systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI 2006)*, pages 367–380, Berkeley, CA, USA, May 2006.
- [2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.
- [3] R. Coker. Bonnie++. <http://www.coker.com.au/bonnie++>.
- [4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)*, December 2002.
- [5] E. Eide, L. Stoller, and J. Lepreau. An experimentation workbench for replayable networking research. In *Proceedings of the Fourth USENIX Symposium on Networked Systems Design & Implementation*, April 2007.
- [6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS-1)*, October 2004.
- [7] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. C. Veitch. Fab: Enterprise storage systems on a shoestring. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, Lihue (Kauai), Hawaii, USA*, pages 169–174, May 2003.
- [8] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 307–320, October 2007.
- [9] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Francisco, CA, USA, January 1994.
- [10] M. Ji. Instant snapshots in a federated array of bricks., January 2005.
- [11] J. Katcher. Postmark: a new file system benchmark, 1997.
- [12] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005*, pages 1–15, Berkeley, CA, April 2005.
- [13] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY*, pages 40–46, June 2002.
- [14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.
- [15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI 2004)*, pages 17–30, December 2004.
- [16] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *FREENIX Track: 1999 USENIX Annual TC*, pages 1–18, Monterey, CA, June 1999.
- [17] M. McLoughlin. The QCOW image format. <http://www.gnome.org/~markmc/qcow-image-format.html>.
- [18] Microsoft TechNet. Virtual hard disk image format specification. <http://microsoft.com/technet/virtualserver/downloads/vhdspec.mspx>.
- [19] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.
- [20] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI 2006)*, pages 353–366, Berkeley, CA, USA, May 2006.
- [21] Red Hat, Inc. LVM architectural overview. [http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Cluster\\_Logical\\_Volume\\_Manager/LVM\\_definition.html](http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Cluster_Logical_Volume_Manager/LVM_definition.html).
- [22] O. Rodeh and A. Teperman. zFS – A scalable distributed file system using object disks. In *MSS '03: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, Washington, DC, USA, April 2003.
- [23] C. Sapuntzakis and M. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems*, pages 55–60, May 2003.
- [24] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)*, December 2002.
- [25] L. Stein. Stupid file systems are better. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 5–5, Berkeley, CA, USA, 2005.
- [26] VMWare, Inc. Performance Tuning Best Practices for ESX Server 3. [http://www.vmware.com/pdf/vi\\_performance\\_tuning.pdf](http://www.vmware.com/pdf/vi_performance_tuning.pdf).
- [27] VMWare, Inc. Using vmware esx server system and vmware virtual infrastructure for backup, restoration, and disaster recovery. [www.vmware.com/pdf/esx\\_backup\\_wp.pdf](http://www.vmware.com/pdf/esx_backup_wp.pdf).
- [28] VMWare, Inc. Virtual machine disk format. <http://www.vmware.com/interfaces/vmdk.html>.
- [29] VMWare, Inc. VMware VMFS product datasheet. [http://www.vmware.com/pdf/vmfs\\_datasheet.pdf](http://www.vmware.com/pdf/vmfs_datasheet.pdf).
- [30] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 148–162, Brighton, UK, October 2005.
- [31] A. Warfield. *Virtual Devices for Virtual Machines*. PhD thesis, University of Cambridge, 2006.
- [32] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI 2004)*, pages 77–90, December 2004.

# Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems\*

Antony Rowstron<sup>1</sup> and Peter Druschel<sup>2\*\*</sup>

<sup>1</sup> Microsoft Research Ltd, St. George House,  
1 Guildhall Street, Cambridge, CB2 3NH, UK.  
[antr@microsoft.com](mailto:antr@microsoft.com)

<sup>2</sup> Rice University MS-132, 6100 Main Street,  
Houston, TX 77005-1892, USA.  
[druschel@cs.rice.edu](mailto:druschel@cs.rice.edu)

**Abstract.** This paper presents the design and evaluation of Pastry, a scalable, distributed object location and routing substrate for wide-area peer-to-peer applications. Pastry performs application-level routing and object location in a potentially very large overlay network of nodes connected via the Internet. It can be used to support a variety of peer-to-peer applications, including global data storage, data sharing, group communication and naming.

Each node in the Pastry network has a unique identifier (nodeId). When presented with a message and a key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops.

Pastry is completely decentralized, scalable, and self-organizing; it automatically adapts to the arrival, departure and failure of nodes. Experimental results obtained with a prototype implementation on an emulated network of up to 100,000 nodes confirm Pastry's scalability and efficiency, its ability to self-organize and adapt to node failures, and its good network locality properties.

## 1 Introduction

Peer-to-peer Internet applications have recently been popularized through file sharing applications like Napster, Gnutella and FreeNet [1, 2, 8]. While much of the attention has been focused on the copyright issues raised by these particular applications, peer-to-peer systems have many interesting technical aspects like decentralized control, self-organization, adaptation and scalability. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric.

---

\* Appears in Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). Heidelberg, Germany, November 2001.

\*\* Work done in part while visiting Microsoft Research, Cambridge, UK.

There are currently many projects aimed at constructing peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [1, 2, 5, 8, 10, 15]. One of the key problems in large-scale peer-to-peer applications is to provide efficient algorithms for object location and routing within the network. This paper presents Pastry, a generic peer-to-peer object location and routing scheme, based on a self-organizing overlay network of nodes connected to the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliable. Moreover, Pastry has good route locality properties.

Pastry is intended as general substrate for the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems. Several application have been built on top of Pastry to date, including a global, persistent storage utility called PAST [11, 21] and a scalable publish/subscribe system called SCRIBE [22]. Other applications are under development.

Pastry provides the following capability. Each node in the Pastry network has a unique numeric identifier (nodeId). When presented with a message and a numeric key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. The expected number of routing steps is  $O(\log N)$ , where  $N$  is the number of Pastry nodes in the network. At each Pastry node along the route that a message takes, the application is notified and may perform application-specific computations related to the message.

Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Because nodeIds are randomly assigned, with high probability, the set of nodes with adjacent nodeId is diverse in geography, ownership, jurisdiction, etc. Applications can leverage this, as Pastry can route to one of  $k$  nodes that are numerically closest to the key. A heuristic ensures that among a set of nodes with the  $k$  closest nodeIds to the key, the message is likely to first reach a node “near” the node from which the message originates, in terms of the proximity metric.

Applications use these capabilities in different ways. PAST, for instance, uses a fileId, computed as the hash of the file’s name and owner, as a Pastry key for a file. Replicas of the file are stored on the  $k$  Pastry nodes with nodeIds numerically closest to the fileId. A file can be looked up by sending a message via Pastry, using the fileId as the key. By definition, the lookup is guaranteed to reach a node that stores the file as long as one of the  $k$  nodes is live. Moreover, it follows that the message is likely to first reach a node near the client, among the  $k$  nodes; that node delivers the file and consumes the message. Pastry’s notification mechanisms allow PAST to maintain replicas of a file on the  $k$  nodes closest to the key, despite node failure and node arrivals, and using only local coordination among nodes with adjacent nodeIds. Details on PAST’s use of Pastry can be found in [11, 21].

As another sample application, in the SCRIBE publish/subscribe System, a list of subscribers is stored on the node with nodeId numerically closest to the topicId of a topic, where the topicId is a hash of the topic name. That node forms a rendez-vous point for publishers and subscribers. Subscribers send a message via Pastry using the

topicId as the key; the registration is recorded at each node along the path. A publisher sends data to the rendez-vous point via Pastry, again using the topicId as the key. The rendez-vous point forwards the data along the multicast tree formed by the reverse paths from the rendez-vous point to all subscribers. Full details of Scribe's use of Pastry can be found in [22].

These and other applications currently under development were all built with little effort on top of the basic capability provided by Pastry. The rest of this paper is organized as follows. Section 2 presents the design of Pastry, including a description of the API. Experimental results with a prototype implementation of Pastry are presented in Section 3. Related work is discussed in Section 4 and Section 5 concludes.

## 2 Design of Pastry

A Pastry system is a self-organizing overlay network of nodes, where each node routes client requests and interacts with local instances of one or more applications. Any computer that is connected to the Internet and runs the Pastry node software can act as a Pastry node, subject only to application-specific security policies.

Each node in the Pastry peer-to-peer overlay network is assigned a 128-bit node identifier (nodeId). The nodeId is used to indicate a node's position in a circular nodeId space, which ranges from 0 to  $2^{128} - 1$ . The nodeId is assigned randomly when a node joins the system. It is assumed that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space. For instance, nodeIds could be generated by computing a cryptographic hash of the node's public key or its IP address. As a result of this random assignment of nodeIds, with high probability, nodes with adjacent nodeIds are diverse in geography, ownership, jurisdiction, network attachment, etc.

Assuming a network consisting of  $N$  nodes, Pastry can route to the numerically closest node to a given key in less than  $\lceil \log_{2^b} N \rceil$  steps under normal operation ( $b$  is a configuration parameter with typical value 4). Despite concurrent node failures, eventual delivery is guaranteed unless  $\lfloor |L|/2 \rfloor$  nodes with *adjacent* nodeIds fail simultaneously ( $|L|$  is a configuration parameter with a typical value of 16 or 32). In the following, we present the Pastry scheme.

For the purpose of routing, nodeIds and keys are thought of as a sequence of digits with base  $2^b$ . Pastry routes messages to the node whose nodeId is numerically closest to the given key. This is accomplished as follows. In each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or  $b$  bits) longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. To support this routing procedure, each node maintains some routing state, which we describe next.

### 2.1 Pastry node state

Each Pastry node maintains a *routing table*, a *neighborhood set* and a *leaf set*. We begin with a description of the routing table. A node's routing table,  $R$ , is organized into

Nodeld 10233102			
Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232
Routing table			
-0-2212102	<b>1</b>	-2-2301203	-3-1203203
<b>0</b>	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	<b>2</b>	10-3-23302
102-0-0230	102-1-1302	102-2-2302	<b>3</b>
1023-0-322	1023-1-000	1023-2-121	<b>3</b>
10233-0-01	<b>1</b>	10233-2-32	
<b>0</b>		102331-2-0	
		<b>2</b>	
Neighborhood set			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

**Fig. 1.** State of a hypothetical Pastry node with nodeId 10233102,  $b = 2$ , and  $l = 8$ . All numbers are in base 4. The top row of the routing table is row zero. The shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeId. The nodeIds in each entry have been split to show the *common prefix with 10233102 - next digit - rest of nodeId*. The associated IP addresses are not shown.

$\lceil \log_2 N \rceil$  rows with  $2^b - 1$  entries each. The  $2^b - 1$  entries at row  $n$  of the routing table each refer to a node whose nodeId shares the present node's nodeId in the first  $n$  digits, but whose  $n + 1$ th digit has one of the  $2^b - 1$  possible values other than the  $n + 1$ th digit in the present node's id.

Each entry in the routing table contains the IP address of one of potentially many nodes whose nodeId have the appropriate prefix; in practice, a node is chosen that is close to the present node, according to the proximity metric. We will show in Section 2.5 that this choice provides good locality properties. If no node is known with a suitable nodeId, then the routing table entry is left empty. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, on average, only  $\lceil \log_2 N \rceil$  rows are populated in the routing table.

The choice of  $b$  involves a trade-off between the size of the populated portion of the routing table (approximately  $\lceil \log_2 N \rceil \times (2^b - 1)$  entries) and the maximum number of hops required to route between any pair of nodes ( $\lceil \log_2 N \rceil$ ). With a value of  $b = 4$  and  $10^6$  nodes, a routing table contains on average 75 entries and the expected number of routing hops is 5, whilst with  $10^9$  nodes, the routing table contains on average 105 entries, and the expected number of routing hops is 7.

The neighborhood set  $M$  contains the nodeIds and IP addresses of the  $|M|$  nodes that are closest (according the proximity metric) to the local node. The neighborhood set is not normally used in routing messages; it is useful in maintaining locality properties, as discussed in Section 2.5. The leaf set  $L$  is the set of nodes with the  $|L|/2$  numerically closest larger nodeIds, and the  $|L|/2$  nodes with numerically closest smaller nodeIds, relative to the present node's nodeId. The leaf set is used during the message routing, as described below. Typical values for  $|L|$  and  $|M|$  are  $2^b$  or  $2 \times 2^b$ .

How the various tables of a Pastry node are initialized and maintained is the subject of Section 2.4. Figure 1 depicts the state of a hypothetical Pastry node with the nodeId 10233102 (base 4), in a system that uses 16 bit nodeIds and a value of  $b = 2$ .

## 2.2 Routing

The Pastry routing procedure is shown in pseudo code form in Table 1. The procedure is executed whenever a message with key  $D$  arrives at a node with nodeId  $A$ . We begin by defining some notation.

$R_l^i$ : the entry in the routing table  $R$  at column  $i$ ,  $0 \leq i < 2^b$  and row  $l$ ,  $0 \leq l < \lfloor 128/b \rfloor$ .  
 $L_i$ : the  $i$ -th closest nodeId in the leaf set  $L$ ,  $-\lfloor |L|/2 \rfloor \leq i \leq \lfloor |L|/2 \rfloor$ , where negative/positive indices indicate nodeIds smaller/larger than the present nodeId, respectively.

$D_l$ : the value of the  $l$ 's digit in the key  $D$ .

$shl(A, B)$ : the length of the prefix shared among  $A$  and  $B$ , in digits.

```

(1) if ( $L_{-\lfloor |L|/2} \leq D \leq L_{\lfloor |L|/2}$ ) {
(2)   //  $D$  is within range of our leaf set
(3)   forward to  $L_i$ , s.th.  $|D - L_i|$  is minimal;
(4) } else {
(5)   // use the routing table
(6)   Let  $l = shl(D, A)$ ;
(7)   if ( $R_l^{D_l} \neq null$ ) {
(8)     forward to  $R_l^{D_l}$ ;
(9)   }
(10)  else {
(11)    // rare case
(12)    forward to  $T \in L \cup R \cup M$ , s.th.
(13)       $shl(T, D) \geq l$ ,
(14)       $|T - D| < |A - D|$ 
(15)  }
(16) }
```

**Table 1.** Pseudo code for Pastry core routing algorithm.

Given a message, the node first checks to see if the key falls within the range of nodeIds covered by its leaf set (line 1). If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose nodeId is closest to the key (possibly the present node) (line 3).

If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit (lines 6–8). In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable (line 11–14), in which case the message is forwarded to a node that shares a prefix with the key at least as long as the local node, and is numerically closer to the key than the present node's id.

Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. And, unless  $\lfloor |L|/2 \rfloor$  adjacent nodes in the leaf set have failed simultaneously, at least one of those nodes must be live.

This simple routing procedure always converges, because each step takes the message to a node that either (1) shares a longer prefix with the key than the local node, or (2) shares as long a prefix with, but is numerically closer to the key than the local node.

*Routing performance* It can be shown that the expected number of routing steps is  $\lceil \log_2 N \rceil$  steps, assuming accurate routing tables and no recent node failures. Briefly, consider the three cases in the routing procedure. If a message is forwarded using the routing table (lines 6–8), then the set of nodes whose ids have a longer prefix match with the key is reduced by a factor of  $2^b$  in each step, which means the destination is reached in  $\lceil \log_2 N \rceil$  steps. If the key is within range of the leaf set (lines 2–3), then the destination node is at most one hop away.

The third case arises when the key is not covered by the leaf set (i.e., it is still more than one hop away from the destination), but there is no routing table entry. Assuming accurate routing tables and no recent node failures, this means that a node with the appropriate prefix does not exist (lines 11–14). The likelihood of this case, given the uniform distribution of nodeIds, depends on  $|L|$ . Analysis shows that with  $|L| = 2^b$  and  $|L| = 2 \times 2^b$ , the probability that this case arises during a given message transmission is less than .02 and 0.006, respectively. When it happens, no more than one additional routing step results with high probability.

In the event of many simultaneous node failures, the number of routing steps required may be at worst linear in  $N$ , while the nodes are updating their state. This is a loose upper bound; in practice, routing performance degrades gradually with the number of recent node failures, as we will show experimentally in Section 3.1. Eventual message delivery is guaranteed unless  $\lfloor |L|/2 \rfloor$  nodes with consecutive nodeIds fail simultaneously. Due to the expected diversity of nodes with adjacent nodeIds, and with a reasonable choice for  $|L|$  (e.g.  $2^b$ ), the probability of such a failure can be made very low.

### 2.3 Pastry API

Next, we briefly outline Pastry’s application programming interface (API). The presented API is slightly simplified for clarity. Pastry exports the following operations:

**nodeId = pastryInit(Credentials, Application)** causes the local node to join an existing Pastry network (or start a new one), initialize all relevant state, and return the local node’s nodeId. The application-specific credentials contain information needed to authenticate the local node. The application argument is a handle to the application object that provides the Pastry node with the procedures to invoke when certain events happen, e.g., a message arrival.

**route(msg,key)** causes Pastry to route the given message to the node with nodeId numerically closest to the key, among all live Pastry nodes.

Applications layered on top of Pastry must export the following operations:

**deliver(msg,key)** called by Pastry when a message is received and the local node's nodeId is numerically closest to key, among all live nodes.

**forward(msg,key,nextId)** called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to NULL terminates the message at the local node.

**newLeafs(leafSet)** called by Pastry whenever there is a change in the local node's leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set.

Several applications have been built on top of Pastry using this simple API, including PAST [11, 21] and SCRIBE [22], and several applications are under development.

## 2.4 Self-organization and adaptation

In this section, we describe Pastry's protocols for handling the arrival and departure of nodes in the Pastry network. We begin with the arrival of a new node that joins the system. Aspects of this process pertaining to the locality properties of the routing tables are discussed in Section 2.5.

*Node arrival* When a new node arrives, it needs to initialize its state tables, and then inform other nodes of its presence. We assume the new node knows initially about a nearby Pastry node  $A$ , according to the proximity metric, that is already part of the system. Such a node can be located automatically, for instance, using “expanding ring” IP multicast, or be obtained by the system administrator through outside channels.

Let us assume the new node's nodeId is  $X$ . (The assignment of nodeIds is application-specific; typically it is computed as the SHA-1 hash of its IP address or its public key). Node  $X$  then asks  $A$  to route a special “join” message with the key equal to  $X$ . Like any message, Pastry routes the join message to the existing node  $Z$  whose id is numerically closest to  $X$ .

In response to receiving the “join” request, nodes  $A$ ,  $Z$ , and all nodes encountered on the path from  $A$  to  $Z$  send their state tables to  $X$ . The new node  $X$  inspects this information, may request state from additional nodes, and then initializes its own state tables, using a procedure described below. Finally,  $X$  informs any nodes that need to be aware of its arrival. This procedure ensures that  $X$  initializes its state with appropriate values, and that the state in all other affected nodes is updated.

Since node  $A$  is assumed to be in proximity to the new node  $X$ ,  $A$ 's neighborhood set to initialize  $X$ 's neighborhood set. Moreover,  $Z$  has the closest existing nodeId to  $X$ , thus its leaf set is the basis for  $X$ 's leaf set. Next, we consider the routing table, starting at row zero. We consider the most general case, where the nodeIds of  $A$  and  $X$  share no common prefix. Let  $A_i$  denote node  $A$ 's row of the routing table at level  $i$ . Note that the entries in row zero of the routing table are independent of a node's nodeId. Thus,  $A_0$  contains appropriate values for  $X_0$ . Other levels of  $A$ 's routing table are of no use to  $X$ , since  $A$ 's and  $X$ 's ids share no common prefix.

However, appropriate values for  $X_1$  can be taken from  $B_1$ , where  $B$  is the first node encountered along the route from  $A$  to  $Z$ . To see this, observe that entries in  $B_1$  and

$X_1$  share the same prefix, because  $X$  and  $B$  have the same first digit in their nodeId. Similarly,  $X$  obtains appropriate entries for  $X_2$  from node  $C$ , the next node encountered along the route from  $A$  to  $Z$ , and so on.

Finally,  $X$  transmits a copy of its resulting state to each of the nodes found in its neighborhood set, leaf set, and routing table. Those nodes in turn update their own state based on the information received. One can show that at this stage, the new node  $X$  is able to route and receive messages, and participate in the Pastry network. The total cost for a node join, in terms of the number of messages exchanged, is  $O(\log_{2^b} N)$ . The constant is about  $3 \times 2^b$ .

Pastry uses an optimistic approach to controlling concurrent node arrivals and departures. Since the arrival/departure of a node affects only a small number of existing nodes in the system, contention is rare and an optimistic approach is appropriate. Briefly, whenever a node  $A$  provides state information to a node  $B$ , it attaches a timestamp to the message.  $B$  adjusts its own state based on this information and eventually sends an update message to  $A$  (e.g., notifying  $A$  of its arrival).  $B$  attaches the original timestamp, which allows  $A$  to check if its state has since changed. In the event that its state has changed, it responds with its updated state and  $B$  restarts its operation.

*Node departure* Nodes in the Pastry network may fail or depart without warning. In this section, we discuss how the Pastry network handles such node departures. A Pastry node is considered failed when its immediate neighbors in the nodeId space can no longer communicate with the node.

To replace a failed node in the leaf set, its neighbor in the nodeId space contacts the live node with the largest index on the side of the failed node, and asks that node for its leaf table. For instance, if  $L_i$  failed for  $\lfloor |L|/2 \rfloor < i < 0$ , it requests the leaf set from  $L_{-\lfloor |L|/2 \rfloor}$ . Let the received leaf set be  $L'$ . This set partly overlaps the present node's leaf set  $L$ , and it contains nodes with nearby ids not presently in  $L$ . Among these new nodes, the appropriate one is then chosen to insert into  $L$ , verifying that the node is actually alive by contacting it. This procedure guarantees that each node lazily repairs its leaf set unless  $\lfloor |L|/2 \rfloor$  nodes with adjacent nodeIds have failed simultaneously. Due to the diversity of nodes with adjacent nodeIds, such a failure is very unlikely even for modest values of  $|L|$ .

The failure of a node that appears in the routing table of another node is detected when that node attempts to contact the failed node and there is no response. As explained in Section 2.2, this event does not normally delay the routing of a message, since the message can be forwarded to another node. However, a replacement entry must be found to preserve the integrity of the routing table.

To repair a failed routing table entry  $R_l^d$ , a node contacts first the node referred to by another entry  $R_l^i$ ,  $i \neq d$  of the same row, and asks for that node's entry for  $R_l^d$ . In the event that none of the entries in row  $l$  have a pointer to a live node with the appropriate prefix, the node next contacts an entry  $R_{l+1}^i$ ,  $i \neq d$ , thereby casting a wider net. This procedure is highly likely to eventually find an appropriate node if one exists.

The neighborhood set is not normally used in the routing of messages, yet it is important to keep it current, because the set plays an important role in exchanging information about nearby nodes. For this purpose, a node attempts to contact each member of the neighborhood set periodically to see if it is still alive. If a member is not respond-

ing, the node asks other members for their neighborhood tables, checks the distance of each of the newly discovered nodes, and updates its own neighborhood set accordingly.

Experimental results in Section 3.2 demonstrate Pastry’s effectiveness in repairing the node state in the presence of node failures, and quantify the cost of this repair in terms of the number of messages exchanged.

## 2.5 Locality

In the previous sections, we discussed Pastry’s basic routing properties and discussed its performance in terms of the expected number of routing hops and the number of messages exchanged as part of a node join operation. This section focuses on another aspect of Pastry’s routing performance, namely its properties with respect to locality. We will show that the route chosen for a message is likely to be “good” with respect to the proximity metric.

Pastry’s notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops or geographic distance. It is assumed that the application provides a function that allows each Pastry node to determine the “distance” of a node with a given IP address to itself. A node with a lower distance value is assumed to be more desirable. An application is expected to implement this function depending on its choice of a proximity metric, using network services like traceroute or Internet subnet maps, and appropriate caching and approximation techniques to minimize overhead.

Throughout this discussion, we assume that the proximity space defined by the chosen proximity metric is Euclidean; that is, the triangulation inequality holds for distances among Pastry nodes. This assumption does not hold in practice for some proximity metrics, such as the number of IP routing hops in the Internet. If the triangulation inequality does not hold, Pastry’s basic routing is not affected; however, the locality properties of Pastry routes may suffer. Quantifying the impact of such deviations is the subject of ongoing work.

We begin by describing how the previously described procedure for node arrival is augmented with a heuristic that ensures that routing table entries are chosen to provide good locality properties.

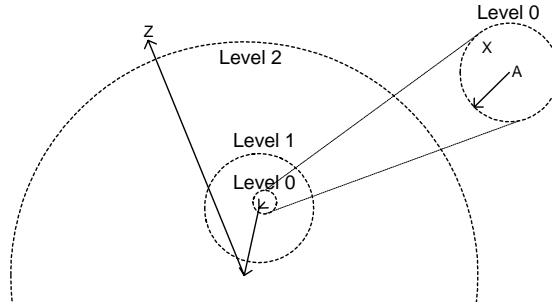
*Locality in the routing table* In Section 2.4, we described how a newly joining node initializes its routing table. Recall that a newly joining node  $X$  asks an existing node  $A$  to route a join message using  $X$  as the key. The message follows a path through nodes  $A, B, \dots$ , and eventually reaches node  $Z$ , which is the live node with the numerically closest nodeId to  $X$ . Node  $X$  initialized its routing table by obtaining the  $i$ -th row of its routing table from the  $i$ -th node encountered along the route from  $A$  to  $Z$ .

The property we wish to maintain is that all routing table entries refer to a node that is near the present node, according to the proximity metric, among all live nodes with a prefix appropriate for the entry. Let us assume that this property holds prior to node  $X$ ’s joining the system, and show how we can maintain the property as node  $X$  joins.

First, we require that node  $A$  is near  $X$ , according to the proximity metric. Since the entries in row zero of  $A$ ’s routing table are close to  $A$ ,  $A$  is close to  $X$ , and we assume that the triangulation inequality holds in the proximity space, it follows that the entries

are relatively near  $A$ . Therefore, the desired property is preserved. Likewise, obtaining  $X$ 's neighborhood set from  $A$  is appropriate.

Let us next consider row one of  $X$ 's routing table, which is obtained from node  $B$ . The entries in this row are near  $B$ , however, it is not clear how close  $B$  is to  $X$ . Intuitively, it would appear that for  $X$  to take row one of its routing table from node  $B$  does not preserve the desired property, since the entries are close to  $B$ , but not necessarily to  $X$ . In reality, the entries tend to be reasonably close to  $X$ . Recall that the entries in each successive row are chosen from an exponentially decreasing set size. Therefore, the expected distance from  $B$  to its row one entries ( $B_1$ ) is much larger than the expected distance traveled from node  $A$  to  $B$ . As a result,  $B_1$  is a reasonable choice for  $X_1$ . This same argument applies for each successive level and routing step, as depicted in Figure 2.



**Fig. 2.** Routing step distance versus distance of the representatives at each level (based on experimental data). The circles around the  $n$ -th node along the route from  $A$  to  $Z$  indicate the average distance of the node's representatives at level  $n$ . Note that  $X$  lies within each circle.

After  $X$  has initialized its state in this fashion, its routing table and neighborhood set approximate the desired locality property. However, the quality of this approximation must be improved to avoid cascading errors that could eventually lead to poor route locality. For this purpose, there is a second stage in which  $X$  requests the state from each of the nodes in its routing table and neighborhood set. It then compares the distance of corresponding entries found in those nodes' routing tables and neighborhood sets, respectively, and updates its own state with any closer nodes it finds. The neighborhood set contributes valuable information in this process, because it maintains and propagates information about nearby nodes regardless of their nodeId prefix.

Intuitively, a look at Figure 2 illuminates why incorporating the state of nodes mentioned in the routing and neighborhood tables from stage one provides good representatives for  $X$ . The circles show the average distance of the entry from each node along the route, corresponding to the rows in the routing table. Observe that  $X$  lies within each circle, albeit off-center. In the second stage,  $X$  obtains the state from the entries discovered in stage one, which are located at an average distance equal to the perimeter of each respective circle. These states must include entries that are appropriate for  $X$ , but were not discovered by  $X$  in stage one, due to its off-center location.

Experimental results in Section 3.2 show that this procedure maintains the locality property in the routing table and neighborhood sets with high fidelity. Next, we discuss how the locality in Pastry’s routing tables affects Pastry routes.

*Route locality* The entries in the routing table of each Pastry node are chosen to be close to the present node, according to the proximity metric, among all nodes with the desired nodeId prefix. As a result, in each routing step, a message is forwarded to a relatively close node with a nodeId that shares a longer common prefix or is numerically closer to the key than the local node. That is, each step moves the message closer to the destination in the nodeId space, while traveling the least possible distance in the proximity space.

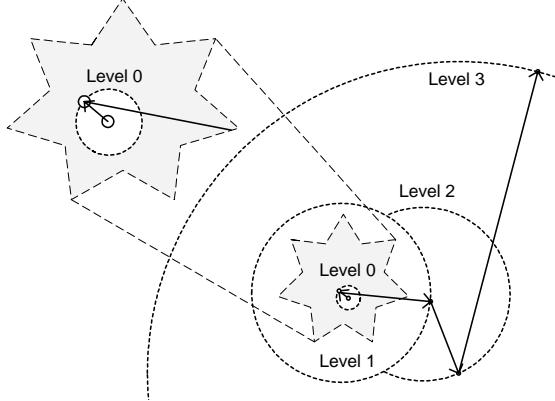
Since only local information is used, Pastry minimizes the distance of the next routing step with no sense of global direction. This procedure clearly does not guarantee that the shortest path from source to destination is chosen; however, it does give rise to relatively good routes. Two facts are relevant to this statement. First, given a message was routed from node  $A$  to node  $B$  at distance  $d$  from  $A$ , the message cannot subsequently be routed to a node with a distance of less than  $d$  from  $A$ . This follows directly from the routing procedure, assuming accurate routing tables.

Second, the expected distance traveled by a messages during each successive routing step is exponentially increasing. To see this, observe that an entry in the routing table in row  $l$  is chosen from a set of nodes of size  $N/2^l$ . That is, the entries in successive rows are chosen from an exponentially decreasing number of nodes. Given the random and uniform distribution of nodeIds in the network, this means that the expected distance of the closest entry in each successive row is exponentially increasing.

Jointly, these two facts imply that although it cannot be guaranteed that the distance of a message from its source increases monotonically at each step, a message tends to make larger and larger strides with no possibility of returning to a node within  $d_i$  of any node  $i$  encountered on the route, where  $d_i$  is the distance of the routing step taken away from node  $i$ . Therefore, the message has nowhere to go but towards its destination. Figure 3 illustrates this effect.

*Locating the nearest among  $k$  nodes* Some peer-to-peer application we have built using Pastry replicate information on the  $k$  Pastry nodes with the numerically closest nodeIds to a key in the Pastry nodeId space. PAST, for instance, replicates files in this way to ensure high availability despite node failures. Pastry naturally routes a message with the given key to the live node with the numerically closest nodeId, thus ensuring that the message reaches one of the  $k$  nodes as long as at least one of them is live.

Moreover, Pastry’s locality properties make it likely that, along the route from a client to the numerically closest node, the message first reaches a node near the client, in terms of the proximity metric, among the  $k$  numerically closest nodes. This is useful in applications such as PAST, because retrieving a file from a nearby node minimizes client latency and network load. Moreover, observe that due to the random assignment of nodeIds, nodes with adjacent nodeIds are likely to be widely dispersed in the network. Thus, it is important to direct a lookup query towards a node that is located relatively near the client.



**Fig. 3.** Sample trajectory of a typical message in the Pastry network, based on experimental data. The message cannot re-enter the circles representing the distance of each of its routing steps away from intermediate nodes. Although the message may partly “turn back” during its initial steps, the exponentially increasing distances traveled in each step cause it to move toward its destination quickly.

Recall that Pastry routes messages towards the node with the nodeId closest to the key, while attempting to travel the smallest possible distance in each step. Therefore, among the  $k$  numerically closest nodes to a key, a message tends to first reach a node near the client. Of course, this process only approximates routing to the nearest node. Firstly, as discussed above, Pastry makes only local routing decisions, minimizing the distance traveled on the next step with no sense of global direction. Secondly, since Pastry routes primarily based on nodeId prefixes, it may miss nearby nodes with a different prefix than the key. In the worst case,  $k/2 - 1$  of the replicas are stored on nodes whose nodeIds differ from the key in their domain at level zero. As a result, Pastry will first route towards the nearest among the  $k/2 + 1$  remaining nodes.

Pastry uses a heuristic to overcome the prefix mismatch issue described above. The heuristic is based on estimating the density of nodeIds in the nodeId space using local information. Based on this estimation, the heuristic detects when a message approaches the set of  $k$  numerically closest nodes, and then switches to numerically nearest address based routing to locate the nearest replica. Results presented in Section 3.3 show that Pastry is able to locate the nearest node in over 75%, and one of the two nearest nodes in over 91% of all queries.

## 2.6 Arbitrary node failures and network partitions

Throughout this paper, it is assumed that Pastry nodes fail silently. Here, we briefly discuss how a Pastry network could deal with arbitrary nodes failures, where a failed node continues to be responsive, but behaves incorrectly or even maliciously. The Pastry routing scheme as described so far is deterministic. Thus, it is vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. Repeated queries could thus fail each time, since they normally take the same route.

In applications where arbitrary node failures must be tolerated, the routing can be randomized. Recall that in order to avoid routing loops, a message must always be forwarded to a node that shares a longer prefix with the destination, or shares the same prefix length as the current node but is numerically closer in the nodeId space than the current node. However, the choice among multiple nodes that satisfy this criterion can be made randomly. In practice, the probability distribution should be biased towards the best choice to ensure low average route delay. In the event of a malicious or failed node along the path, the query may have to be repeated several times by the client, until a route is chosen that avoids the bad node. Furthermore, the protocols for node join and node failure can be extended to tolerate misbehaving nodes. The details are beyond the scope of this paper.

Another challenge are IP routing anomalies in the Internet that cause IP hosts to be unreachable from certain IP hosts but not others. The Pastry routing is tolerant of such anomalies; Pastry nodes are considered live and remain reachable in the overlay network as long as they are able to communicate with their immediate neighbors in the nodeId space. However, Pastry's self-organization protocol may cause the creation of multiple, isolated Pastry overlay networks during periods of IP routing failures. Because Pastry relies almost exclusively on information exchange within the overlay network to self-organize, such isolated overlays may persist after full IP connectivity resumes.

One solution to this problem involves the use of IP multicast. Pastry nodes can periodically perform an expanding ring multicast search for other Pastry nodes in their vicinity. If isolated Pastry overlays exist, they will be discovered eventually, and reintegrated. To minimize the cost, this procedure can be performed randomly and infrequently by Pastry nodes, only within a limited range of IP routing hops from the node, and only if no search was performed by another nearby Pastry node recently. As an added benefit, the results of this search can also be used to improve the quality of the routing tables.

### 3 Experimental results

In this section, we present experimental results obtained with a prototype implementation of Pastry. The Pastry node software was implemented in Java. To be able to perform experiments with large networks of Pastry nodes, we also implemented a network emulation environment, permitting experiments with up to 100,000 Pastry nodes.

All experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 6GBytes of main memory, running True64 UNIX, version 4.0F. The Pastry node software was implemented in Java and executed using Compaq's Java 2 SDK, version 1.2.2-6 and the Compaq FastVM, version 1.2.2-4.

In all experiments reported in this paper, the Pastry nodes were configured to run in a single Java VM. This is largely transparent to the Pastry implementation—the Java runtime system automatically reduces communication among the Pastry nodes to local object invocations.

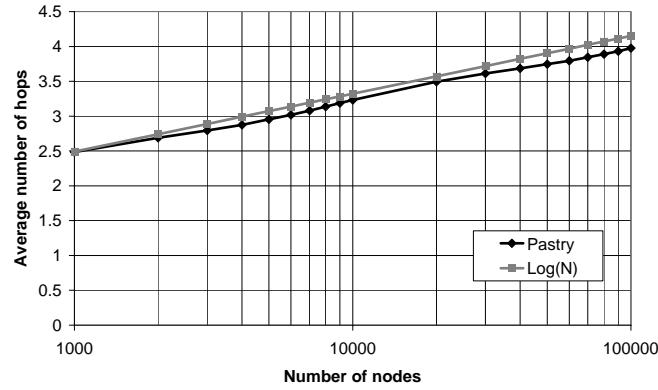
The emulated network environment maintains distance information between the Pastry nodes. Each Pastry node is assigned a location in a plane; coordinates in the plane are randomly assigned in the range [0, 1000]. Nodes in the Internet are not uni-

formly distributed in a Euclidean space; instead, there is a strong clustering of nodes and the triangulation inequality doesn't always hold. We are currently performing emulations based on a more realistic network topology model taken from [26]. Early results indicate that overall, Pastry's locality related routing properties are not significantly affected by this change.

A number of Pastry properties are evaluated experimentally. The first set of results demonstrates the basic performance of Pastry routing. The routing tables created within the Pastry nodes are evaluated in Section 3.2. In Section 3.3 we evaluate Pastry's ability to route to the nearest among the  $k$  numerically closest nodes to a key. Finally, in 3.4 the properties of Pastry under node failures are considered.

### 3.1 Routing performance

The first experiment shows the number of routing hops as a function of the size of the Pastry network. We vary the number of Pastry nodes from 1,000 to 100,000 in a network where  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ . In each of 200,000 trials, two Pastry nodes are selected at random and a message is routed between the pair using Pastry.

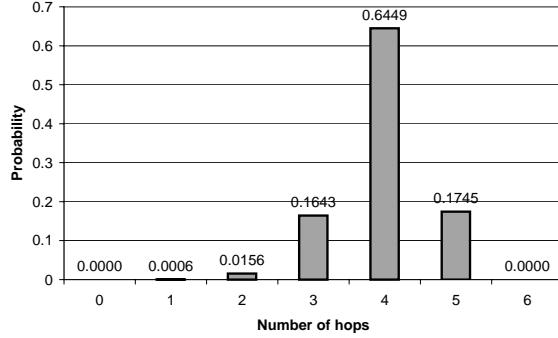


**Fig. 4.** Average number of routing hops versus number of Pastry nodes,  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$  and 200,000 lookups.

Figure 4 show the average number of routing hops taken, as a function of the network size. “Log N” shows the value  $\log_2 N$  and is included for comparison. ( $\lceil \log_2 N \rceil$  is the expected maximum number of hops required to route in a network containing  $N$  nodes). The results show that the number of route hops scale with the size of the network as predicted.

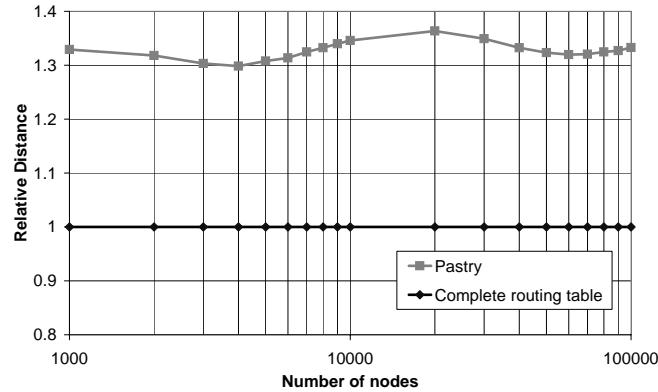
Figure 5 shows the distribution of the number of routing hops taken, for a network size of 100,000 nodes, in the same experiment. The results show that the maximum route length is ( $\lceil \log_2 N \rceil$ ) ( $\lceil \log_2 100,000 \rceil = 5$ ), as expected.

The second experiment evaluated the locality properties of Pastry routes. It compares the relative distance a message travels using Pastry, according to the proximity



**Fig. 5.** Probability versus number of routing hops,  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ ,  $N = 100,000$  and 200,000 lookups.

metric, with that of a fictitious routing scheme that maintains complete routing tables. The distance traveled is the sum of the distances between consecutive nodes encountered along the route in the emulated network. For the fictitious routing scheme, the distance traveled is simply the distance between the source and the destination node. The results are normalized to the distance traveled in the fictitious routing scheme. The goal of this experiment is to quantify the cost, in terms of distance traveled in the proximity space, of maintaining only small routing tables in Pastry.



**Fig. 6.** Route distance versus number of Pastry nodes,  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ , and 200,000 lookups.

The number of nodes varies between 1,000 and 100,000,  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ . 200,000 pairs of Pastry nodes are selected and a message is routed between each pair. Figure 6 shows the results for Pastry and the fictitious scheme (labeled “Complete routing tables”). The results show that the Pastry routes are only approximately 30% to 40% longer. Considering that the routing tables in Pastry contain only approximately

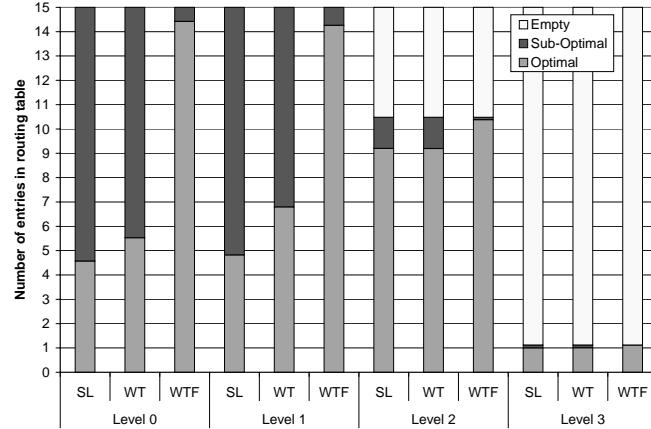
$\lceil \log_2 N \rceil \times (2^b - 1)$  entries, this result is quite good. For 100,000 nodes the Pastry routing tables contain approximately 75 entries, compared to 99,999 in the case of complete routing tables.

We also determined the routing throughput, in messages per second, of a Pastry node. Our unoptimized Java implementation handled over 3,000 messages per second. This indicates that the routing procedure is very lightweight.

### 3.2 Maintaining the network

Figure 7 shows the quality of the routing tables with respect to the locality property, and how the extent of information exchange during a node join operation affects the quality of the resulting routing tables vis-à-vis locality. In this experiment, 5,000 nodes join the Pastry network one by one. After all nodes joined, the routing tables were examined. The parameters are  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ .

Three options were used to gather information when a node joins. “SL” is a hypothetical method where the joining node considers only the appropriate row from each node along the route from itself to the node with the closest existing nodeId (see Section 2.4). With “WT”, the joining node fetches the entire state of each node along the path, but does not fetch state from the resulting entries. This is equivalent to omitting the second stage. “WTF” is the actual method used in Pastry, where state is fetched from each node that appears in the tables after the first stage.



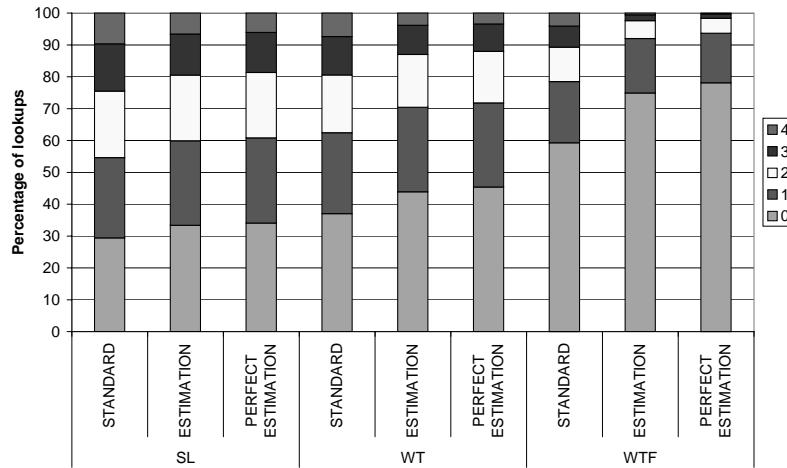
**Fig. 7.** Quality of routing tables (locality),  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$  and 5,000 nodes.

The results are shown in Figure 7. For levels 0 to 3, we show the quality of the routing table entries with each method. With 5,000 nodes and  $b = 4$ , levels 2 and 3 are not fully populated, which explains the missing entries shown. “Optimal” means that the best (i.e., closest according to the proximity metric) node appeared in a routing table entry, “sub-optimal” means that an entry was not the closest or was missing.

The results show that Pastry’s method of node integration (“WTF”) is highly effective in initializing the routing tables with good locality. On average, less than 1 entry per level of the routing table is not the best choice. Moreover, the comparison with “SL” and “WT” shows that less information exchange during the node join operation comes at a dramatic cost in routing table quality with respect to locality.

### 3.3 Replica routing

The next experiment examines Pastry’s ability to route to one of the  $k$  closest nodes to a key, where  $k = 5$ . In particular, the experiment explores Pastry’s ability to locate one of the  $k$  nodes near the client. In a Pastry network of 10,000 nodes with  $b = 3$  and  $|L| = 8$ , 100,000 times a Pastry node and a key are chosen randomly, and a message is routed using Pastry from the node using the key. The first of the  $k$  numerically closest nodes to the key that is reached along the route is recorded.



**Fig. 8.** Number of nodes closer to the client than the node discovered. ( $b = 3$ ,  $|L| = 8$ ,  $|M| = 16$ , 10,000 nodes and 100,000 message routes).

Figure 8 shows the percentage of lookups that reached the closest node, according to the proximity metric (0 closer nodes), the second closest node (1 closer node), and so forth. Results are shown for the three different protocols for initializing a new node’s state, with (“Estimation”) and without (“Standard”) the heuristic mentioned in Section 2.5, and for an idealized, optimal version of the heuristic (“Perfect estimation”). Recall that the heuristic estimates the nodeId space coverage of other nodes’ leaf sets, using an estimate based on its own leaf sets coverage. The “Perfect estimation” ensures that this estimate of a node’s leaf set coverage is correct for every node.

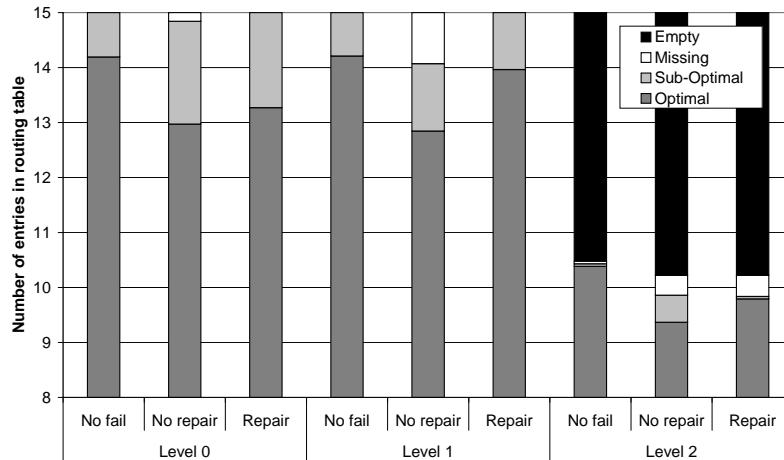
Without the heuristic and the standard node joining protocol (WTF), Pastry is able to locate the closest node 68% of the time, and one of the top two nodes 87% of the time. With the heuristic routing option, this figures increase to 76% and 92%, respectively.

The lesser routing table quality resulting from the “SL” and “WT” methods for node joining have a strong negative effect on Pastry’s ability to locate nearby nodes, as one would expect. Also, the heuristic approach is only approximately 2% worse than the best possible results using perfect estimation.

The results show that Pastry is effective in locating a node near the client in the vast majority of cases, and that the use of the heuristic is effective.

### 3.4 Node failures

The next experiment explores Pastry’s behavior in the presence of node failures. A 5,000 node Pastry network is used with  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ ,  $k = 5$ . Then, 10% (500) randomly selected nodes fail silently. After these failures, a key is chosen at random, and two Pastry nodes are randomly selected. A message is routed from these two nodes to the key, and this is repeated 100,000 times (200,000 lookups total). Initially, the node state repair facilities in Pastry were disabled, which allows us to measure the full impact of the failures on Pastry’s routing performance. Next, the node state repair facilities were enabled, and another 200,000 lookups were performed from the same Pastry nodes to the same key.

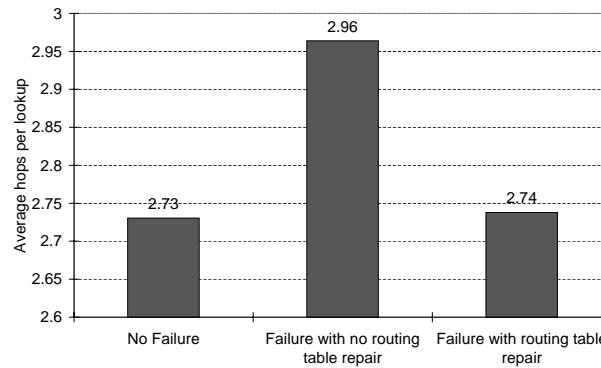


**Fig. 9.** Quality of routing tables before and after 500 node failures,  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$  and 5,000 starting nodes.

Figure 9 shows the average routing table quality across all nodes for levels 0–2, as measured before the failures, after the failures, and after the repair. Note that in this figure, missing entries are shown separately from sub-optimal entries. Also, recall that Pastry lazily repairs routing tables entries when they are being used. As a result, routing table entries that were not used during the 200,000 lookups are not discovered and therefore not repaired. To isolate the effectiveness of Pastry’s repair procedure, we excluded table entries that were never used.

The results show that Pastry recovers all missing table entries, and that the quality of the entries with respect to locality (fraction of optimal entries) approaches that before the failures. At row zero, the average number of best entries after the repair is approximately one below that prior to the failure. However, although this can't be seen in the figure, our results show that the actual distance between the suboptimal and the optimal entries is very small. This is intuitive, since the average distance of row zero entries is very small. Note that the increase in empty entries at levels 1 and 2 after the failures is due to the reduction in the total number of Pastry nodes, which increases the sparseness of the tables at the upper rows. Thus, this increase does not constitute a reduction in the quality of the tables.

Figure 10 shows the impact of failures and repairs on the route quality. The left bar shows the average number of hops before the failures; the middle bar shows the average number of hops after the failures, and before the tables were repaired. Finally, the right bar shows the average number of hops after the repair. The data shows that without repairs, the stale routing table state causes a significant deterioration of route quality. After the repair, however, the average number of hops is only slightly higher than before the failures.



**Fig. 10.** Number of routing hops versus node failures,  $b = 4$ ,  $|L| = 16$ ,  $|M| = 32$ , 200,000 lookups and 5,000 nodes with 500 failing.

We also measured the average cost, in messages, for repairing the tables after node failure. In our experiments, a total of 57 remote procedure calls were needed on average per failed node to repair all relevant table entries.

## 4 Related Work

There are currently several peer-to-peer systems in use, and many more are under development. Among the most prominent are file sharing facilities, such as Gnutella [2] and Freenet [8]. The Napster [1] music exchange service provided much of the original motivation for peer-to-peer systems, but it is not a pure peer-to-peer system because its database is centralized. All three systems are primarily intended for the large-scale

sharing of data files; reliable content location is not guaranteed or necessary in this environment. In Gnutella, the use of a broadcast based protocol limits the system's scalability and incurs a high bandwidth requirement. Both Gnutella and Freenet are not guaranteed to find an existing object.

Pastry, along with Tapestry [27], Chord [24] and CAN [19], represent a second generation of peer-to-peer routing and location schemes that were inspired by the pioneering work of systems like FreeNet and Gnutella. Unlike that earlier work, they guarantee a definite answer to a query in a bounded number of network hops, while retaining the scalability of FreeNet and the self-organizing properties of both FreeNet and Gnutella.

Pastry and Tapestry bear some similarity to the work by Plaxton et al [18] and to routing in the landmark hierarchy [25]. The approach of routing based on address prefixes, which can be viewed as a generalization of hypercube routing, is common to all these schemes. However, neither Plaxton nor the landmark approach are fully self-organizing. Pastry and Tapestry differ in their approach to achieving network locality and to supporting replication, and Pastry appears to be less complex.

The Chord protocol is closely related to both Pastry and Tapestry, but instead of routing towards nodes that share successively longer address prefixes with the destination, Chord forwards messages based on numerical difference with the destination address. Unlike Pastry and Tapestry, Chord makes no explicit effort to achieve good network locality. CAN routes messages in a  $d$ -dimensional space, where each node maintains a routing table with  $O(d)$  entries and any node can be reached in  $O(dN^{1/d})$  routing hops. Unlike Pastry, the routing table does not grow with the network size, but the number of routing hops grows faster than  $\log N$ .

Existing applications built on top of Pastry include PAST [11, 21] and SCRIBE [22]. Other peer-to-peer applications that were built on top of a generic routing and location substrate like Pastry are OceanStore [15] (Tapestry) and CFS [9] (Chord). FarSite [5] uses a conventional distributed directory service, but could potentially be built on top of a system like Pastry. Pastry can be seen as an overlay network that provides a self-organizing routing and location service. Another example of an overlay network is the Overcast system [12], which provides reliable single-source multicast streams.

There has been considerable work on routing in general, on hypercube and mesh routing in parallel computers, and more recently on routing in ad hoc networks, for example GRID [17]. In Pastry, we assume an existing infrastructure at the network layer, and the emphasis is on self-organization and the integration of content location and routing. In the interest of scalability, Pastry nodes only use local information, while traditional routing algorithms (like link-state and distance vector methods) globally propagate information about routes to each destination. This global information exchange limits the scalability of these routing algorithms, necessitating a hierarchical routing architecture like the one used in the Internet.

Several prior works consider issues in replicating Web content in the Internet, and selecting the nearest replica relative to a client HTTP query [4, 13, 14]. Pastry provides a more general infrastructure aimed at a variety of peer-to-peer applications. Another related area is that of naming services, which are largely orthogonal to Pastry's content location and routing. Lampson's Global Naming System (GNS) [16] is an example of a scalable naming system that relies on a hierarchy of name servers that directly

corresponds to the structure of the name space. Cheriton and Mann [7] describe another scalable naming service.

Finally, attribute based and intentional naming systems [6, 3], as well as directory services [20, 23] resolve a set of attributes that describe the properties of an object to the address of an object instance that satisfies the given properties. Thus, these systems support far more powerful queries than Pastry. However, this power comes at the expense of scalability, performance and administrative overhead. Such systems could be potentially built upon Pastry.

## 5 Conclusion

This paper presents and evaluates Pastry, a generic peer-to-peer content location and routing system based on a self-organizing overlay network of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably routes a message to the live node with a nodeId numerically closest to a key. Pastry can be used as a building block in the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems.

Pastry routes to any node in the overlay network in  $O(\log N)$  steps in the absence of recent node failures, and it maintains routing tables with only  $O(\log N)$  entries. Moreover, Pastry takes into account locality when routing messages. Results with as many as 100,000 nodes in an emulated network confirm that Pastry is efficient and scales well, that it is self-organizing and can gracefully adapt to node failures, and that it has good locality properties.

## Acknowledgments

We thank Miguel Castro and the anonymous reviewers for their useful comments and feedback. Peter Druschel thanks Microsoft Research, Cambridge, UK, and the Massachusetts Institute of Technology for their support during his visits in Fall 2000 and Spring 2001, respectively, and Compaq for donating equipment used in this work.

## References

1. Napster. <http://www.napster.com/>.
2. The Gnutella protocol specification, 2000. <http://dss.clip2.com/GnutellaProtocol04.pdf>.
3. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. SOSP'99*, Kiawah Island, SC, Dec. 1999.
4. Y. Amir, A. Peterson, and D. Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proc. 12th Symposium on Distributed Computing*, Andros, Greece, Sept. 1998.
5. W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS'2000*, Santa Clara, CA, 2000.
6. M. Bowman, L. L. Peterson, and A. Yeatts. Univers: An attribute-based name server. *Software — Practice and Experience*, 20(4):403–424, Apr. 1990.

7. D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Trans. Comput. Syst.*, 7(2):147–183, May 1989.
8. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.
9. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP’01*, Banff, Canada, Oct. 2001.
10. R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
11. P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, Schloss Elmau, Germany, May 2001.
12. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O’Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. OSDI 2000*, San Diego, CA, 2000.
13. J. Kangasharju, J. W. Roberts, and K. W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proc. 4th Web Caching Workshop*, San Diego, CA, Mar. 1999.
14. J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proc. IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000.
15. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS’2000*, Cambridge, MA, November 2000.
16. B. Lampson. Designing a global name service. In *Proc. Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, Minaki, Canada, Aug. 1986.
17. J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographical ad hoc routing. In *Proc. of ACM MOBICOM 2000*, Boston, MA, 2000.
18. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.
19. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM’01*, San Diego, CA, Aug. 2001.
20. J. Reynolds. RFC 1309: Technical overview of directory services using the X.500 protocol, Mar. 1992.
21. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP’01*, Banff, Canada, Oct. 2001.
22. A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. Submitted for publication. June 2001. <http://www.research.microsoft.com/antr/SCRIBE/>.
23. M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proc. 3rd International World Wide Web Conference*, Darmstadt, Germany, 1995.
24. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM’01*, San Diego, CA, Aug. 2001.
25. P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *SIGCOMM’88*, Stanford, CA, 1988.
26. E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM96*, San Francisco, CA, 1996.
27. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.

# Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek

dpeng@google.com, fdabek@google.com

Google, Inc.

## Abstract

Updating an index of the web as documents are crawled requires continuously transforming a large repository of existing documents as new documents arrive. This task is one example of a class of data processing tasks that transform a large repository of data via small, independent mutations. These tasks lie in a gap between the capabilities of existing infrastructure. Databases do not meet the storage or throughput requirements of these tasks: Google’s indexing system stores tens of petabytes of data and processes billions of updates per day on thousands of machines. MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.

We have built Percolator, a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, we process the same number of documents per day, while reducing the average age of documents in Google search results by 50%.

## 1 Introduction

Consider the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. For example, if the same content is crawled under multiple URLs, only the URL with the highest PageRank [28] appears in the index. Each link is also inverted so that the anchor text from each outgoing link is attached to the page the link points to. Link inversion must work across duplicates: links to a duplicate of a page should be forwarded to the highest PageRank duplicate if necessary.

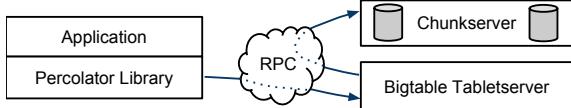
This is a bulk-processing task that can be expressed as a series of MapReduce [13] operations: one for clustering duplicates, one for link inversion, etc. It’s easy to maintain invariants since MapReduce limits the parallelism of the computation; all documents finish one processing step before starting the next. For example, when the indexing system is writing inverted links to the current highest-PageRank URL, we need not worry about its PageRank concurrently changing; a previous MapReduce step has already determined its PageRank.

Now, consider how to update that index after recrawling some small portion of the web. It’s not sufficient to run the MapReduces over just the new pages since, for example, there are links between the new pages and the rest of the web. The MapReduces must be run again over the entire repository, that is, over both the new pages and the old pages. Given enough computing resources, MapReduce’s scalability makes this approach feasible, and, in fact, Google’s web search index was produced in this way prior to the work described here. However, reprocessing the entire web discards the work done in earlier runs and makes latency proportional to the size of the repository, rather than the size of an update.

The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants. However, existing DBMSs can’t handle the sheer volume of data: Google’s indexing system stores tens of petabytes across thousands of machines [30]. Distributed storage systems like Bigtable [9] can scale to the size of our repository but don’t provide tools to help programmers maintain data invariants in the face of concurrent updates.

An ideal data processing system for the task of maintaining the web search index would be optimized for *incremental processing*; that is, it would allow us to maintain a very large repository of documents and update it efficiently as each new document was crawled. Given that the system will be processing many small updates concurrently, an ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

The remainder of this paper describes a particular incremental processing system: Percolator. Percolator provides the user with random access to a multi-PB repository. Random access allows us to process documents in-



**Figure 1:** Percolator and its dependencies

dividually, avoiding the global scans of the repository that MapReduce requires. To achieve high throughput, many threads on many machines need to transform the repository concurrently, so Percolator provides ACID-compliant transactions to make it easier for programmers to reason about the state of the repository; we currently implement snapshot isolation semantics [5].

In addition to reasoning about concurrency, programmers of an incremental system need to keep track of the state of the incremental computation. To assist them in this task, Percolator provides observers: pieces of code that are invoked by the system whenever a user-specified column changes. Percolator applications are structured as a series of observers; each observer completes a task and creates more work for “downstream” observers by writing to the table. An external process triggers the first observer in the chain by writing initial data into the table.

Percolator was built specifically for incremental processing and is not intended to supplant existing solutions for most data processing tasks. Computations where the result can’t be broken down into small updates (sorting a file, for example) are better handled by MapReduce. Also, the computation should have strong consistency requirements; otherwise, Bigtable is sufficient. Finally, the computation should be very large in some dimension (total data size, CPU required for transformation, etc.); smaller computations not suited to MapReduce or Bigtable can be handled by traditional DBMSs.

Within Google, the primary application of Percolator is preparing web pages for inclusion in the live web search index. By converting the indexing system to an incremental system, we are able to process individual documents as they are crawled. This reduced the average document processing latency by a factor of 100, and the average age of a document appearing in a search result dropped by nearly 50 percent (the age of a search result includes delays other than indexing such as the time between a document being changed and being crawled). The system has also been used to render pages into images; Percolator tracks the relationship between web pages and the resources they depend on, so pages can be reprocessed when any depended-upon resources change.

## 2 Design

Percolator provides two main abstractions for performing incremental processing at large scale: ACID transactions over a random-access repository and ob-

servers, a way to organize an incremental computation.

A Percolator system consists of three binaries that run on every machine in the cluster: a Percolator worker, a Bigtable [9] tablet server, and a GFS [20] chunkserver. All observers are linked into the Percolator worker, which scans the Bigtable for changed columns (“notifications”) and invokes the corresponding observers as a function call in the worker process. The observers perform transactions by sending read/write RPCs to Bigtable tablet servers, which in turn send read/write RPCs to GFS chunkservers. The system also depends on two small services: the timestamp oracle and the lightweight lock service. The timestamp oracle provides strictly increasing timestamps: a property required for correct operation of the snapshot isolation protocol. Workers use the lightweight lock service to make the search for dirty notifications more efficient.

From the programmer’s perspective, a Percolator repository consists of a small number of tables. Each table is a collection of “cells” indexed by row and column. Each cell contains a value: an uninterpreted array of bytes. (Internally, to support snapshot isolation, we represent each cell as a series of values indexed by timestamp.)

The design of Percolator was influenced by the requirement to run at massive scales and the lack of a requirement for extremely low latency. Relaxed latency requirements let us take, for example, a lazy approach to cleaning up locks left behind by transactions running on failed machines. This lazy, simple-to-implement approach potentially delays transaction commit by tens of seconds. This delay would not be acceptable in a DBMS running OLTP tasks, but it is tolerable in an incremental processing system building an index of the web. Percolator has no central location for transaction management; in particular, it lacks a global deadlock detector. This increases the latency of conflicting transactions but allows the system to scale to thousands of machines.

### 2.1 Bigtable overview

Percolator is built on top of the Bigtable distributed storage system. Bigtable presents a multi-dimensional sorted map to users: keys are (row, column, timestamp) tuples. Bigtable provides lookup and update operations on each row, and Bigtable row transactions enable atomic read-modify-write operations on individual rows. Bigtable handles petabytes of data and runs reliably on large numbers of (unreliable) machines.

A running Bigtable consists of a collection of tablet servers, each of which is responsible for serving several tablets (contiguous regions of the key space). A master coordinates the operation of tablet servers by, for example, directing them to load or unload tablets. A tablet is stored as a collection of read-only files in the Google

SSTable format. SSTables are stored in GFS; Bigtable relies on GFS to preserve data in the event of disk loss. Bigtable allows users to control the performance characteristics of the table by grouping a set of columns into a locality group. The columns in each locality group are stored in their own set of SSTables, which makes scanning them less expensive since the data in other columns need not be scanned.

The decision to build on Bigtable defined the overall shape of Percolator. Percolator maintains the gist of Bigtable’s interface: data is organized into Bigtable rows and columns, with Percolator metadata stored alongside in special columns (see Figure 5). Percolator’s API closely resembles Bigtable’s API: the Percolator library largely consists of Bigtable operations wrapped in Percolator-specific computation. The challenge, then, in implementing Percolator is providing the features that Bigtable does not: multirow transactions and the observer framework.

## 2.2 Transactions

Percolator provides cross-row, cross-table transactions with ACID snapshot-isolation semantics. Percolator users write their transaction code in an imperative language (currently C++) and mix calls to the Percolator API with their code. Figure 2 shows a simplified version of clustering documents by a hash of their contents. In this example, if Commit() returns false, the transaction has conflicted (in this case, because two URLs with the same content hash were processed simultaneously) and should be retried after a backoff. Calls to Get() and Commit() are blocking; parallelism is achieved by running many transactions simultaneously in a thread pool.

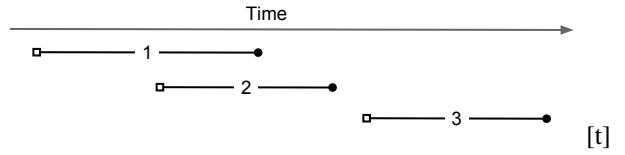
While it is possible to incrementally process data without the benefit of strong transactions, transactions make it more tractable for the user to reason about the state of the system and to avoid the introduction of errors into a long-lived repository. For example, in a transactional web-indexing system the programmer can make assumptions like: the hash of the contents of a document is always consistent with the table that indexes duplicates. Without transactions, an ill-timed crash could result in a permanent error: an entry in the document table that corresponds to no URL in the duplicates table. Transactions also make it easy to build index tables that are always up to date and consistent. Note that both of these examples require transactions that span rows, rather than the single-row transactions that Bigtable already provides.

Percolator stores multiple versions of each data item using Bigtable’s timestamp dimension. Multiple versions are required to provide snapshot isolation [5], which presents each transaction with the appearance of reading from a stable snapshot at some timestamp. Writes appear in a different, later, timestamp. Snapshot isolation pro-

```
bool UpdateDocument(Document doc) {
    Transaction t(&cluster);
    t.Set(doc.url(), "contents", "document", doc.contents());
    int hash = Hash(doc.contents());

    // dups table maps hash → canonical URL
    string canonical;
    if (!t.Get(hash, "canonical-url", "dups", &canonical)) {
        // No canonical yet; write myself in
        t.Set(hash, "canonical-url", "dups", doc.url());
    } // else this document already exists, ignore new copy
    return t.Commit();
}
```

**Figure 2:** Example usage of the Percolator API to perform basic checksum clustering and eliminate documents with the same content.



**Figure 3:** Transactions under snapshot isolation perform reads at a start timestamp (represented here by an open square) and writes at a commit timestamp (closed circle). In this example, transaction 2 would not see writes from transaction 1 since transaction 2’s start timestamp is before transaction 1’s commit timestamp. Transaction 3, however, will see writes from both 1 and 2. Transaction 1 and 2 are running concurrently: if they both write the same cell, at least one will abort.

tects against write-write conflicts: if transactions A and B, running concurrently, write to the same cell, at most one will commit. Snapshot isolation does not provide serializability; in particular, transactions running under snapshot isolation are subject to write skew [5]. The main advantage of snapshot isolation over a serializable protocol is more efficient reads. Because any timestamp represents a consistent snapshot, reading a cell requires only performing a Bigtable lookup at the given timestamp; acquiring locks is not necessary. Figure 3 illustrates the relationship between transactions under snapshot isolation.

Because it is built as a client library accessing Bigtable, rather than controlling access to storage itself, Percolator faces a different set of challenges implementing distributed transactions than traditional PDBMSs. Other parallel databases integrate locking into the system component that manages access to the disk: since each node already mediates access to data on the disk it can grant locks on requests and deny accesses that violate locking requirements.

By contrast, any node in Percolator can (and does) issue requests to directly modify state in Bigtable: there is no convenient place to intercept traffic and assign locks. As a result, Percolator must explicitly maintain locks. Locks must persist in the face of machine failure; if a lock could disappear between the two phases of com-

key	bal:data	bal:lock	bal:write
Bob	6: 5: \$10	6: 5:	6: data @ 5 5:
Joe	6: 5: \$2	6: 5:	6: data @ 5 5:

1. Initial state: Joe's account contains \$2 dollars, Bob's \$10.

	7:\$3	7: <b>I am primary</b>	7: 6: data @ 5
Bob	6: 5: \$10	6: 5:	5:
Joe	6: 5: \$2	6: 5:	6: data @ 5 5:

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

	7: \$3 6: 5: \$10	7: I am primary 6: 5:	7: 6: data @ 5 5:
Bob	7: <b>\$9</b> 6: 5: \$2	7: <b>primary @ Bob.bal</b> 6: 5:	7: 6: data @ 5 5:
Joe			

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

	8: 7: \$3 6: 5: \$10	8: 7: 6: 5:	8: <b>data @ 7</b> 7: 6: data @ 5 5:
Bob	7: \$9 6: 5: \$2	7: primary @ Bob.bal 6: 5:	7: 6: data @ 5 5:
Joe			

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value \$3.

	8: 7: \$3 6: 5: \$10	8: 7: 6: 5:	8: data @ 7 7: 6: data @ 5 5:
Bob	8: 7: \$9 6: 5: \$2	8: 7: 6: 5:	8: <b>data @ 7</b> 7: 6: data @ 5 5:
Joe			

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

**Figure 4:** This figure shows the Bigtable writes performed by a Percolator transaction that mutates two rows. The transaction transfers 7 dollars from Bob to Joe. Each Percolator column is stored as 3 Bigtable columns: data, write metadata, and lock metadata. Bigtable's timestamp dimension is shown within each cell; 12: "data" indicates that "data" has been written at Bigtable timestamp 12. Newly written data is shown in boldface.

Column	Use
<b>c:lock</b>	An uncommitted transaction is writing this cell; contains the location of primary lock
<b>c:write</b>	Committed data present; stores the Bigtable timestamp of the data
<b>c:data</b>	Stores the data itself
<b>c:notify</b>	Hint: observers may need to run
<b>c:ack_O</b>	Observer "O" has run ; stores start timestamp of successful last run

**Figure 5:** The columns in the Bigtable representation of a Percolator column named "c."

mit, the system could mistakenly commit two transactions that should have conflicted. The lock service must provide high throughput; thousands of machines will be requesting locks simultaneously. The lock service should also be low-latency; each Get() operation requires reading locks in addition to data, and we prefer to minimize this latency. Given these requirements, the lock server will need to be replicated (to survive failure), distributed and balanced (to handle load), and write to a persistent data store. Bigtable itself satisfies all of our requirements, and so Percolator stores its locks in special in-memory columns in the same Bigtable that stores data and reads or modifies the locks in a Bigtable row transaction when accessing data in that row.

We'll now consider the transaction protocol in more detail. Figure 6 shows the pseudocode for Percolator transactions, and Figure 4 shows the layout of Percolator data and metadata during the execution of a transaction. These various metadata columns used by the system are described in Figure 5. The transaction's constructor asks the timestamp oracle for a start timestamp (line 6), which determines the consistent snapshot seen by Get(). Calls to Set() are buffered (line 7) until commit time. The basic approach for committing buffered writes is two-phase commit, which is coordinated by the client. Transactions on different machines interact through row transactions on Bigtable tablet servers.

In the first phase of commit ("prewrite"), we try to lock all the cells being written. (To handle client failure, we designate one lock arbitrarily as the primary; we'll discuss this mechanism below.) The transaction reads metadata to check for conflicts in each cell being written. There are two kinds of conflicting metadata: if the transaction sees another write record after its start timestamp, it aborts (line 32); this is the write-write conflict that snapshot isolation guards against. If the transaction sees another lock at any timestamp, it also aborts (line 34). It's possible that the other transaction is just being slow to release its lock after having already committed below our start timestamp, but we consider this unlikely, so we abort. If there is no conflict, we write the lock and

```

1 class Transaction {
2     struct Write { Row row; Column col; string value; };
3     vector<Write> writes_;
4     int start_ts_;
5
6     Transaction() : start_ts_(oracle.GetTimestamp()) {}
7     void Set(Write w) { writes_.push_back(w); }
8     bool Get(Row row, Column c, string* value) {
9         while (true) {
10             bigtable::Txn T = bigtable::StartRowTransaction(row);
11             // Check for locks that signal concurrent writes.
12             if (T.Read(row, c+"lock", [0, start_ts_])) {
13                 // There is a pending lock; try to clean it and wait
14                 BackoffAndMaybeCleanupLock(row, c);
15                 continue;
16             }
17
18             // Find the latest write below our start_timestamp.
19             latest_write = T.Read(row, c+"write", [0, start_ts_]);
20             if (!latest_write.found()) return false; // no data
21             int data_ts = latest_write.start_timestamp();
22             *value = T.Read(row, c+"data", [data_ts, data_ts]);
23             return true;
24         }
25     }
26     // Prewrite tries to lock cell w, returning false in case of conflict.
27     bool Prewrite(Write w, Write primary) {
28         Column c = w.col;
29         bigtable::Txn T = bigtable::StartRowTransaction(w.row);
30
31         // Abort on writes after our start timestamp ...
32         if (T.Read(w.row, c+"write", [start_ts_, infinity])) return false;
33         // ... or locks at any timestamp.
34         if (T.Read(w.row, c+"lock", [0, infinity])) return false;
35
36         T.Write(w.row, c+"data", start_ts_, w.value);
37         T.Write(w.row, c+"lock", start_ts_,
38             {primary.row, primary.col}); // The primary's location.
39         return T.Commit();
40     }
41     bool Commit() {
42         Write primary = writes_[0];
43         vector<Write> secondaries(writes_.begin()+1, writes_.end());
44         if (!Prewrite(primary, primary)) return false;
45         for (Write w : secondaries)
46             if (!Prewrite(w, primary)) return false;
47
48         int commit_ts = oracle_.GetTimestamp();
49
50         // Commit primary first.
51         Write p = primary;
52         bigtable::Txn T = bigtable::StartRowTransaction(p.row);
53         if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
54             return false; // aborted while working
55         T.Write(p.row, p.col+"write", commit_ts,
56             start_ts_); // Pointer to data written at start_ts_
57         T.Erase(p.row, p.col+"lock", commit_ts);
58         if (!T.Commit()) return false; // commit point
59
60         // Second phase: write out write records for secondary cells.
61         for (Write w : secondaries) {
62             bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
63             bigtable::Erase(w.row, w.col+"lock", commit_ts);
64         }
65         return true;
66     }
67 } // class Transaction

```

**Figure 6:** Pseudocode for Percolator transaction protocol.

the data to each cell at the start timestamp (lines 36-38).

If no cells conflict, the transaction may commit and proceeds to the second phase. At the beginning of the second phase, the client obtains the commit timestamp from the timestamp oracle (line 48). Then, at each cell (starting with the primary), the client releases its lock and make its write visible to readers by replacing the lock with a write record. The write record indicates to readers that committed data exists in this cell; it contains a pointer to the start timestamp where readers can find the actual data. Once the primary's write is visible (line 58), the transaction must commit since it has made a write visible to readers.

A Get() operation first checks for a lock in the timestamp range [0, start\_timestamp], which is the range of timestamps visible in the transaction's snapshot (line 12). If a lock is present, another transaction is concurrently writing this cell, so the reading transaction must wait until the lock is released. If no conflicting lock is found, Get() reads the latest write record in that timestamp range (line 19) and returns the data item corresponding to that write record (line 22).

Transaction processing is complicated by the possibility of client failure (tablet server failure does not affect the system since Bigtable guarantees that written locks persist across tablet server failures). If a client fails while a transaction is being committed, locks will be left behind. Percolator must clean up those locks or they will cause future transactions to hang indefinitely. Percolator takes a lazy approach to cleanup: when a transaction A encounters a conflicting lock left behind by transaction B, A may determine that B has failed and erase its locks.

It is very difficult for A to be perfectly confident in its judgment that B is failed; as a result we must avoid a race between A cleaning up B's transaction and a not-actually-failed B committing the same transaction. Percolator handles this by designating one cell in every transaction as a synchronizing point for any commit or cleanup operations. This cell's lock is called the primary lock. Both A and B agree on which lock is primary (the location of the primary is written into the locks at all other cells). Performing either a cleanup or commit operation requires modifying the primary lock; since this modification is performed under a Bigtable row transaction, only one of the cleanup or commit operations will succeed. Specifically: before B commits, it must check that it still holds the primary lock and replace it with a write record. Before A erases B's lock, A must check the primary to ensure that B has not committed; if the primary lock is still present, then it can safely erase the lock.

When a client crashes during the second phase of commit, a transaction will be past the commit point (it has written at least one write record) but will still

have locks outstanding. We must perform roll-forward on these transactions. A transaction that encounters a lock can distinguish between the two cases by inspecting the primary lock: if the primary lock has been replaced by a write record, the transaction which wrote the lock must have committed and the lock must be rolled forward, otherwise it should be rolled back (since we always commit the primary first, we can be sure that it is safe to roll back if the primary is not committed). To roll forward, the transaction performing the cleanup replaces the stranded lock with a write record as the original transaction would have done.

Since cleanup is synchronized on the primary lock, it is safe to clean up locks held by live clients; however, this incurs a performance penalty since rollback forces the transaction to abort. So, a transaction will not clean up a lock unless it suspects that a lock belongs to a dead or stuck worker. Percolator uses simple mechanisms to determine the liveness of another transaction. Running workers write a token into the Chubby lockservice [8] to indicate they belong to the system; other workers can use the existence of this token as a sign that the worker is alive (the token is automatically deleted when the process exits). To handle a worker that is live, but not working, we additionally write the wall time into the lock; a lock that contains a too-old wall time will be cleaned up even if the worker’s liveness token is valid. To handle long-running commit operations, workers periodically update this wall time while committing.

### 2.3 Timestamps

The timestamp oracle is a server that hands out timestamps in strictly increasing order. Since every transaction requires contacting the timestamp oracle twice, this service must scale well. The oracle periodically allocates a range of timestamps by writing the highest allocated timestamp to stable storage; given an allocated range of timestamps, the oracle can satisfy future requests strictly from memory. If the oracle restarts, the timestamps will jump forward to the maximum allocated timestamp (but will never go backwards). To save RPC overhead (at the cost of increasing transaction latency) each Percolator worker batches timestamp requests across transactions by maintaining only one pending RPC to the oracle. As the oracle becomes more loaded, the batching naturally increases to compensate. Batching increases the scalability of the oracle but does not affect the timestamp guarantees. Our oracle serves around 2 million timestamps per second from a single machine.

The transaction protocol uses strictly increasing timestamps to guarantee that `Get()` returns all committed writes before the transaction’s start timestamp. To see how it provides this guarantee, consider a transaction R reading at timestamp  $T_R$  and a transaction W that com-

mitted at timestamp  $T_W < T_R$ ; we will show that R sees W’s writes. Since  $T_W < T_R$ , we know that the timestamp oracle gave out  $T_W$  before or in the same batch as  $T_R$ ; hence, W requested  $T_W$  before R received  $T_R$ . We know that R can’t do reads before receiving its start timestamp  $T_R$  and that W wrote locks before requesting its commit timestamp  $T_W$ . Therefore, the above property guarantees that W must have at least written all its locks before R did any reads; R’s `Get()` will see either the fully-committed write record or the lock, in which case W will block until the lock is released. Either way, W’s write is visible to R’s `Get()`.

### 2.4 Notifications

Transactions let the user mutate the table while maintaining invariants, but users also need a way to trigger and run the transactions. In Percolator, the user writes code (“observers”) to be triggered by changes to the table, and we link all the observers into a binary running alongside every tablet server in the system. Each observer registers a function and a set of columns with Percolator, and Percolator invokes the function after data is written to one of those columns in any row.

Percolator applications are structured as a series of observers; each observer completes a task and creates more work for “downstream” observers by writing to the table. In our indexing system, a MapReduce loads crawled documents into Percolator by running loader transactions, which trigger the document processor transaction to index the document (parse, extract links, etc.). The document processor transaction triggers further transactions like clustering. The clustering transaction, in turn, triggers transactions to export changed document clusters to the serving system.

Notifications are similar to database triggers or events in active databases [29], but unlike database triggers, they cannot be used to maintain database invariants. In particular, the triggered observer runs in a separate transaction from the triggering write, so the triggering write and the triggered observer’s writes are not atomic. Notifications are intended to help structure an incremental computation rather than to help maintain data integrity.

This difference in semantics and intent makes observer behavior much easier to understand than the complex semantics of overlapping triggers. Percolator applications consist of very few observers — the Google indexing system has roughly 10 observers. Each observer is explicitly constructed in the `main()` of the worker binary, so it is clear what observers are active. It is possible for several observers to observe the same column, but we avoid this feature so it is clear what observer will run when a particular column is written. Users do need to be wary about infinite cycles of notifications, but Percolator does nothing to prevent this; the user typically constructs

a series of observers to avoid infinite cycles.

We do provide one guarantee: at most one observer’s transaction will commit for each change of an observed column. The converse is not true, however: multiple writes to an observed column may cause the corresponding observer to be invoked only once. We call this feature message collapsing, since it helps avoid computation by amortizing the cost of responding to many notifications. For example, it is sufficient for <http://google.com> to be reprocessed periodically rather than every time we discover a new link pointing to it.

To provide these semantics for notifications, each observed column has an accompanying “acknowledgment” column for each observer, containing the latest start timestamp at which the observer ran. When the observed column is written, Percolator starts a transaction to process the notification. The transaction reads the observed column and its corresponding acknowledgment column. If the observed column was written after its last acknowledgment, then we run the observer and set the acknowledgment column to our start timestamp. Otherwise, the observer has already been run, so we do not run it again. Note that if Percolator accidentally starts two transactions concurrently for a particular notification, they will both see the dirty notification and run the observer, but one will abort because they will conflict on the acknowledgment column. We promise that at most one observer will *commit* for each notification.

To implement notifications, Percolator needs to efficiently find dirty cells with observers that need to be run. This search is complicated by the fact that notifications are rare: our table has trillions of cells, but, if the system is keeping up with applied load, there will only be millions of notifications. Additionally, observer code is run on a large number of client processes distributed across a collection of machines, meaning that this search for dirty cells must be distributed.

To identify dirty cells, Percolator maintains a special “notify” Bigtable column, containing an entry for each dirty cell. When a transaction writes an observed cell, it also sets the corresponding notify cell. The workers perform a distributed scan over the notify column to find dirty cells. After the observer is triggered and the transaction commits, we remove the notify cell. Since the notify column is just a Bigtable column, not a Percolator column, it has no transactional properties and serves only as a hint to the scanner to check the acknowledgment column to determine if the observer should be run.

To make this scan efficient, Percolator stores the notify column in a separate Bigtable locality group so that scanning over the column requires reading only the millions of dirty cells rather than the trillions of total data cells. Each Percolator worker dedicates several threads to the scan. For each thread, the worker chooses a portion of the

table to scan by first picking a random Bigtable tablet, then picking a random key in the tablet, and finally scanning the table from that position. Since each worker is scanning a random region of the table, we worry about two workers running observers on the same row concurrently. While this behavior will not cause correctness problems due to the transactional nature of notifications, it is inefficient. To avoid this, each worker acquires a lock from a lightweight lock service before scanning the row. This lock server need not persist state since it is advisory and thus is very scalable.

The random-scanning approach requires one additional tweak: when it was first deployed we noticed that scanning threads would tend to clump together in a few regions of the table, effectively reducing the parallelism of the scan. This phenomenon is commonly seen in public transportation systems where it is known as “platooning” or “bus clumping” and occurs when a bus is slowed down (perhaps by traffic or slow loading). Since the number of passengers at each stop grows with time, loading delays become even worse, further slowing the bus. Simultaneously, any bus behind the slow bus speeds up as it needs to load fewer passengers at each stop. The result is a clump of buses arriving simultaneously at a stop [19]. Our scanning threads behaved analogously: a thread that was running observers slowed down while threads “behind” it quickly skipped past the now-clean rows to clump with the lead thread and failed to pass the lead thread because the clump of threads overloaded tablet servers. To solve this problem, we modified our system in a way that public transportation systems cannot: when a scanning thread discovers that it is scanning the same row as another thread, it chooses a new random location in the table to scan. To further the transportation analogy, the buses (scanner threads) in our city avoid clumping by teleporting themselves to a random stop (location in the table) if they get too close to the bus in front of them.

Finally, experience with notifications led us to introduce a lighter-weight but semantically weaker notification mechanism. We found that when many duplicates of the same page were processed concurrently, each transaction would conflict trying to trigger reprocessing of the same duplicate cluster. This led us to devise a way to notify a cell without the possibility of transactional conflict. We implement this weak notification by writing *only* to the Bigtable “notify” column. To preserve the transactional semantics of the rest of Percolator, we restrict these weak notifications to a special type of column that cannot be written, only notified. The weaker semantics also mean that multiple observers may run and commit as a result of a single weak notification (though the system tries to minimize this occurrence). This has become an important feature for managing conflicts; if an observer

frequently conflicts on a hotspot, it often helps to break it into two observers connected by a non-transactional notification on the hotspot.

## 2.5 Discussion

One of the inefficiencies of Percolator relative to a MapReduce-based system is the number of RPCs sent per work-unit. While MapReduce does a single large read to GFS and obtains all of the data for 10s or 100s of web pages, Percolator performs around 50 individual Bigtable operations to process a single document.

One source of additional RPCs occurs during commit. When writing a lock, we must do a read-modify-write operation requiring two Bigtable RPCs: one to read for conflicting locks or writes and another to write the new lock. To reduce this overhead, we modified the Bigtable API by adding conditional mutations which implements the read-modify-write step in a single RPC. Many conditional mutations destined for the same tablet server can also be batched together into a single RPC to further reduce the total number of RPCs we send. We create batches by delaying lock operations for several seconds to collect them into batches. Because locks are acquired in parallel, this adds only a few seconds to the latency of each transaction; we compensate for the additional latency with greater parallelism. Batching also increases the time window in which conflicts may occur, but in our low-contention environment this has not proved to be a problem.

We also perform the same batching when reading from the table: every read operation is delayed to give it a chance to form a batch with other reads to the same tablet server. This delays each read, potentially greatly increasing transaction latency. A final optimization mitigates this effect, however: prefetching. Prefetching takes advantage of the fact that reading two or more values in the same row is essentially the same cost as reading one value. In either case, Bigtable must read the entire SSTable block from the file system and decompress it. Percolator attempts to predict, each time a column is read, what other columns in a row will be read later in the transaction. This prediction is made based on past behavior. Prefetching, combined with a cache of items that have already been read, reduces the number of Bigtable reads the system would otherwise do by a factor of 10.

Early in the implementation of Percolator, we decided to make all API calls blocking and rely on running thousands of threads per machine to provide enough parallelism to maintain good CPU utilization. We chose this thread-per-request model mainly to make application code easier to write, compared to the event-driven model. Forcing users to bundle up their state each of the (many) times they fetched a data item from the table would have made application development much more difficult. Our

experience with thread-per-request was, on the whole, positive: application code is simple, we achieve good utilization on many-core machines, and crash debugging is simplified by meaningful and complete stack traces. We encountered fewer race conditions in application code than we feared. The biggest drawbacks of the approach were scalability issues in the Linux kernel and Google infrastructure related to high thread counts. Our in-house kernel development team was able to deploy fixes to address the kernel issues.

## 3 Evaluation

Percolator lies somewhere in the performance space between MapReduce and DBMSs. For example, because Percolator is a distributed system, it uses far more resources to process a fixed amount of data than a traditional DBMS would; this is the cost of its scalability. Compared to MapReduce, Percolator can process data with far lower latency, but again, at the cost of additional resources required to support random lookups. These are engineering tradeoffs which are difficult to quantify: how much of an efficiency loss is too much to pay for the ability to add capacity endlessly simply by purchasing more machines? Or: how does one trade off the reduction in development time provided by a layered system against the corresponding decrease in efficiency?

In this section we attempt to answer some of these questions by first comparing Percolator to batch processing systems via our experiences with converting a MapReduce-based indexing pipeline to use Percolator. We'll also evaluate Percolator with microbenchmarks and a synthetic workload based on the well-known TPC-E benchmark [1]; this test will give us a chance to evaluate the scalability and efficiency of Percolator relative to Bigtable and DBMSs.

All of the experiments in this section are run on a subset of the servers in a Google data center. The servers run the Linux operating system on x86 processors; each machine is connected to several commodity SATA drives.

### 3.1 Converting from MapReduce

We built Percolator to create Google's large "base" index, a task previously performed by MapReduce. In our previous system, each day we crawled several billion documents and fed them along with a repository of existing documents through a series of 100 MapReduces. The result was an index which answered user queries. Though not all 100 MapReduces were on the critical path for every document, the organization of the system as a series of MapReduces meant that each document spent 2-3 days being indexed before it could be returned as a search result.

The Percolator-based indexing system (known as Caffeine [25]), crawls the same number of documents,

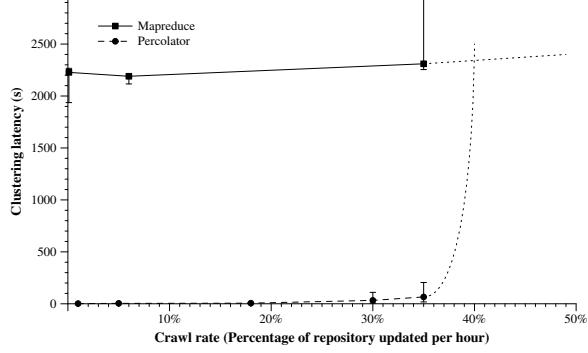
but we feed each document through Percolator as it is crawled. The immediate advantage, and main design goal, of Caffeine is a reduction in latency: the median document moves through Caffeine over 100x faster than the previous system. This latency improvement grows as the system becomes more complex: adding a new clustering phase to the Percolator-based system requires an extra lookup for each document rather than an extra scan over the repository. Additional clustering phases can also be implemented in the same transaction rather than in another MapReduce; this simplification is one reason the number of observers in Caffeine (10) is far smaller than the number of MapReduces in the previous system (100). This organization also allows for the possibility of performing additional processing on only a subset of the repository without rescanning the entire repository.

Adding additional clustering phases isn't free in an incremental system: more resources are required to make sure the system keeps up with the input, but this is still an improvement over batch processing systems where no amount of resources can overcome delays introduced by stragglers in an additional pass over the repository. Caffeine is essentially immune to stragglers that were a serious problem in our batch-based indexing system because the bulk of the processing does not get held up by a few very slow operations. The radically-lower latency of the new system also enables us to remove the rigid distinctions between large, slow-to-update indexes and smaller, more rapidly updated indexes. Because Percolator frees us from needing to process the repository each time we index documents, we can also make it larger: Caffeine's document collection is currently 3x larger than the previous system's and is limited only by available disk space.

Compared to the system it replaced, Caffeine uses roughly twice as many resources to process the same crawl rate. However, Caffeine makes good use of the extra resources. If we were to run the old indexing system with twice as many resources, we could either increase the index size or reduce latency by at most a factor of two (but not do both). On the other hand, if Caffeine were run with half the resources, it would not be able to process as many documents per day as the old system (but the documents it did produce would have much lower latency).

The new system is also easier to operate. Caffeine has far fewer moving parts: we run tablet servers, Percolator workers, and chunkservers. In the old system, each of a hundred different MapReduces needed to be individually configured and could independently fail. Also, the "peaky" nature of the MapReduce workload made it hard to fully utilize the resources of a datacenter compared to Percolator's much smoother resource usage.

The simplicity of writing straight-line code and the ability to do random lookups into the repository makes developing new features for Percolator easy. Under



**Figure 7:** Median document clustering delay for Percolator (dashed line) and MapReduce (solid line). For MapReduce, all documents finish processing at the same time and error bars represent the min, median, and max of three runs of the clustering MapReduce. For Percolator, we are able to measure the delay of individual documents, so the error bars represent the 5th- and 95th-percentile delay on a per-document level.

MapReduce, random lookups are awkward and costly. On the other hand, Caffeine developers need to reason about concurrency where it did not exist in the MapReduce paradigm. Transactions help deal with this concurrency, but can't fully eliminate the added complexity.

To quantify the benefits of moving from MapReduce to Percolator, we created a synthetic benchmark that clusters newly crawled documents against a billion-document repository to remove duplicates in much the same way Google's indexing pipeline operates. Documents are clustered by three clustering keys. In a real system, the clustering keys would be properties of the document like redirect target or content hash, but in this experiment we selected them uniformly at random from a collection of 750M possible keys. The average cluster in our synthetic repository contains 3.3 documents, and 93% of the documents are in a non-singleton cluster. This distribution of keys exercises the clustering logic, but does not expose it to the few extremely large clusters we have seen in practice. These clusters only affect the latency tail and not the results we present here. In the Percolator clustering implementation, each crawled document is immediately written to the repository to be clustered by an observer. The observer maintains an index table for each clustering key and compares the document against each index to determine if it is a duplicate (an elaboration of Figure 2). MapReduce implements clustering of continually arriving documents by repeatedly running a sequence of three clustering MapReduces (one for each clustering key). The sequence of three MapReduces processes the entire repository and any crawled documents that accumulated while the previous three were running.

This experiment simulates clustering documents crawled at a uniform rate. Whether MapReduce or Percolator performs better under this metric is a function of the how frequently documents are crawled (the crawl rate)

and the repository size. We explore this space by fixing the size of the repository and varying the rate at which new documents arrive, expressed as a percentage of the repository crawled per hour. In a practical system, a very small percentage of the repository would be crawled per hour: there are over 1 trillion web pages on the web (and ideally in an indexing system’s repository), far too many to crawl a reasonable fraction of in a single day. When the new input is a small fraction of the repository (low crawl rate), we expect Percolator to outperform MapReduce since MapReduce must map over the (large) repository to cluster the (small) batch of new documents while Percolator does work proportional only to the small batch of newly arrived documents (a lookup in up to three index tables per document). At very large crawl rates where the number of newly crawled documents approaches the size of the repository, MapReduce will perform better than Percolator. This cross-over occurs because streaming data from disk is much cheaper, per byte, than performing random lookups. At the cross-over the total cost of the lookups required to cluster the new documents under Percolator equals the cost to stream the documents and the repository through MapReduce. At crawl rates higher than that, one is better off using MapReduce.

We ran this benchmark on 240 machines and measured the median delay between when a document is crawled and when it is clustered. Figure 7 plots the median latency of document processing for both implementations as a function of crawl rate. When the crawl rate is low, Percolator clusters documents faster than MapReduce as expected; this scenario is illustrated by the leftmost pair of points which correspond to crawling 1 percent of documents per hour. MapReduce requires approximately 20 minutes to cluster the documents because it takes 20 minutes just to process the repository through the three MapReduces (the effect of the few newly crawled documents on the runtime is negligible). This results in an average delay between crawling a document and clustering of around 30 minutes: a random document waits 10 minutes after being crawled for the previous sequence of MapReduces to finish and then spends 20 minutes being processed by the three MapReduces. Percolator, on the other hand, finds a newly loaded document and processes it in two seconds on average, or about 1000x faster than MapReduce. The two seconds includes the time to find the dirty notification and run the transaction that performs the clustering. Note that this 1000x latency improvement could be made arbitrarily large by increasing the size of the repository.

As the crawl rate increases, MapReduce’s processing time grows correspondingly. Ideally, it would be proportional to the combined size of the repository and the input which grows with the crawl rate. In practice, the running time of a small MapReduce like this is limited by strag-

	Bigtable	Percolator	Relative
Read/s	15513	14590	0.94
Write/s	31003	7232	0.23

**Figure 8:** The overhead of Percolator operations relative to Bigtable. Write overhead is due to additional operations Percolator needs to check for conflicts.

glers, so the growth in processing time (and thus clustering latency) is only weakly correlated to crawl rate at low crawl rates. The 6 percent crawl rate, for example, only adds 150GB to a 1TB data set; the extra time to process 150GB is in the noise. The latency of Percolator is relatively unchanged as the crawl rate grows until it suddenly increases to effectively infinity at a crawl rate of 40% per hour. At this point, Percolator saturates the resources of the test cluster, is no longer able to keep up with the crawl rate, and begins building an unbounded queue of unprocessed documents. The dotted asymptote at 40% is an extrapolation of Percolator’s performance beyond this breaking point. MapReduce is subject to the same effect: eventually crawled documents accumulate faster than MapReduce is able to cluster them, and the batch size will grow without bound in subsequent runs. In this particular configuration, however, MapReduce can sustain crawl rates in excess of 100% (the dotted line, again, extrapolates performance).

These results show that Percolator can process documents at orders of magnitude better latency than MapReduce in the regime where we expect real systems to operate (single-digit crawl rates).

### 3.2 Microbenchmarks

In this section, we determine the cost of the transactional semantics provided by Percolator. In these experiments, we compare Percolator to a “raw” Bigtable. We are only interested in the relative performance of Bigtable and Percolator since any improvement in Bigtable performance will translate directly into an improvement in Percolator performance. Figure 8 shows the performance of Percolator and raw Bigtable running against a single tablet server. All data was in the tablet server’s cache during the experiments and Percolator’s batching optimizations were disabled.

As expected, Percolator introduces overhead relative to Bigtable. We first measure the number of random writes that the two systems can perform. In the case of Percolator, we execute transactions that write a single cell and then commit; this represents the worst case for Percolator overhead. When doing a write, Percolator incurs roughly a factor of four overhead on this benchmark. This is the result of the extra operations Percolator requires for commit beyond the single write that Bigtable issues: a read to check for locks, a write to add the lock, and a second write to remove the lock record. The read, in particular, is more expensive than a write and accounts

for most of the overhead. In this test, the limiting factor was the performance of the tablet server, so the additional overhead of fetching timestamps is not measured. We also tested random reads: Percolator performs a single Bigtable operation per read, but that read operation is somewhat more complex than the raw Bigtable operation (the Percolator read looks at metadata columns in addition to data columns).

### 3.3 Synthetic Workload

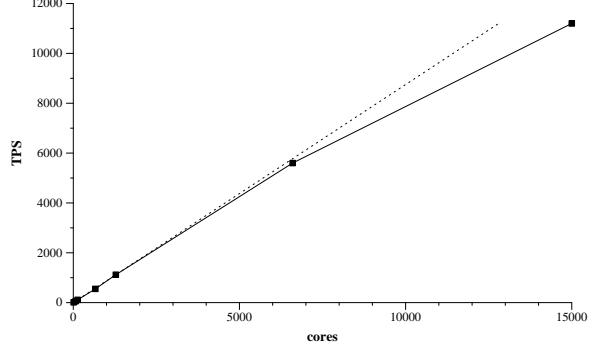
To evaluate Percolator on a more realistic workload, we implemented a synthetic benchmark based on TPC-E [1]. This isn't the ideal benchmark for Percolator since TPC-E is designed for OLTP systems, and a number of Percolator's tradeoffs impact desirable properties of OLTP systems (the latency of conflicting transactions, for example). TPC-E is a widely recognized and understood benchmark, however, and it allows us to understand the cost of our system against more traditional databases.

TPC-E simulates a brokerage firm with customers who perform trades, market search, and account inquiries. The brokerage submits trade orders to a market exchange, which executes the trade and updates broker and customer state. The benchmark measures the number of trades executed. On average, each customer performs a trade once every 500 seconds, so the benchmark scales by adding customers and associated data.

TPC-E traditionally has three components – a customer emulator, a market emulator, and a DBMS running stored SQL procedures. Since Percolator is a client library running against Bigtable, our implementation is a combined customer/market emulator that calls into the Percolator library to perform operations against Bigtable. Percolator provides a low-level Get/Set/iterator API rather than a high-level SQL interface, so we created indexes and did all the ‘query planning’ by hand.

Since Percolator is an incremental processing system rather than an OLTP system, we don't attempt to meet the TPC-E latency targets. Our average transaction latency is 2 to 5 seconds, but outliers can take several minutes. Outliers are caused by, for example, exponential backoff on conflicts and Bigtable tablet unavailability. Finally, we made a small modification to the TPC-E transactions. In TPC-E, each trade result increases the broker's commission and increments his trade count. Each broker services a hundred customers, so the average broker must be updated once every 5 seconds, which causes repeated write conflicts in Percolator. In Percolator, we would implement this feature by writing the increment to a side table and periodically aggregating each broker's increments; for the benchmark, we choose to simply omit this write.

Figure 9 shows how the resource usage of Percolator scales as demand increases. We will measure resource

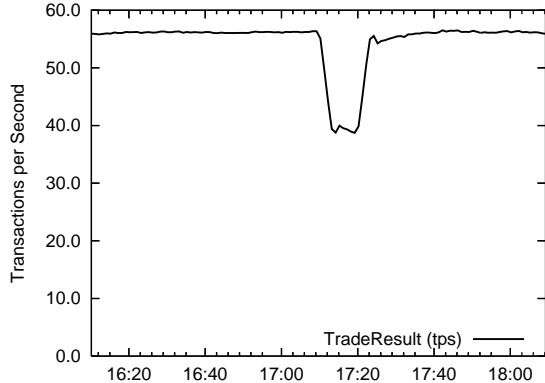


**Figure 9:** Transaction rate on a TPC-E-like benchmark as a function of cores used. The dotted line shows linear scaling.

usage in CPU cores since that is the limiting resource in our experimental environment. We were able to procure a small number of machines for testing, but our test Bigtable cell shares the disk resources of a much larger production cluster. As a result, disk bandwidth is not a factor in the system's performance. In this experiment, we configured the benchmark with increasing numbers of customers and measured both the achieved performance and the number of cores used by all parts of the system including cores used for background maintenance such as Bigtable compactions. The relationship between performance and resource usage is essentially linear across several orders of magnitude, from 11 cores to 15,000 cores.

This experiment also provides an opportunity to measure the overheads in Percolator relative to a DBMS. The fastest commercial TPC-E system today performs 3,183 tpsE using a single large shared-memory machine with 64 Intel Nehalem cores with 2 hyperthreads per core [33]. Our synthetic benchmark based on TPC-E performs 11,200 tps using 15,000 cores. This comparison is very rough: the Nehalem cores in the comparison machine are significantly faster than the cores in our test cell (small-scale testing on Nehalem processors shows that they are 20-30% faster per-thread compared to the cores in the test cluster). However, we estimate that Percolator uses roughly 30 times more CPU per transaction than the benchmark system. On a cost-per-transaction basis, the gap is likely much less than 30 since our test cluster uses cheaper, commodity hardware compared to the enterprise-class hardware in the reference machine.

The conventional wisdom on implementing databases is to “get close to the iron” and use hardware as directly as possible since even operating system structures like disk caches and schedulers make it hard to implement an efficient database [32]. In Percolator we not only interposed an operating system between our database and the hardware, but also several layers of software and network links. The conventional wisdom is correct: this arrangement has a cost. There are substantial overheads in



**Figure 10:** Recovery of tps after 33% tablet server mortality

preparing requests to go on the wire, sending them, and processing them on a remote machine. To illustrate these overheads in Percolator, consider the act of mutating the database. In a DBMS, this incurs a function call to store the data in memory and a system call to force the log to hardware controlled RAID array. In Percolator, a client performing a transaction commit sends multiple RPCs to Bigtable, which commits the mutation by logging it to 3 chunkservers, which make system calls to actually flush the data to disk. Later, that same data will be compacted into minor and major sstables, each of which will be again replicated to multiple chunkservers.

The CPU inflation factor is the cost of our layering. In exchange, we get scalability (our fastest result, though not directly comparable to TPC-E, is more than 3x the current official record [33]), and we inherit the useful features of the systems we build upon, like resilience to failures. To demonstrate the latter, we ran the benchmark with 15 tablet servers and allowed the performance to stabilize. Figure 10 shows the performance of the system over time. The dip in performance at 17:09 corresponds to a failure event: we killed a third of the tablet servers. Performance drops immediately after the failure event but recovers as the tablets are reloaded by other tablet servers. We allowed the killed tablet servers to restart so performance eventually returns to the original level.

#### 4 Related Work

Batch processing systems like MapReduce [13, 22, 24] are well suited for efficiently transforming or analyzing an entire corpus: these systems can simultaneously use a large number of machines to process huge amounts of data quickly. Despite this scalability, re-running a MapReduce pipeline on each small batch of updates results in unacceptable latency and wasted work. Overlapping or pipelining the adjacent stages can reduce latency [10], but straggler shards still set the minimum time to complete the pipeline. Percolator avoids the expense of repeated scans by, essentially, creating indexes

on the keys used to cluster documents; one of criticisms leveled by Stonebraker and DeWitt in their initial critique of MapReduce [16] was that MapReduce did not support such indexes.

Several proposed modifications to MapReduce [18, 26, 35] reduce the cost of processing changes to a repository by allowing workers to randomly read a base repository while mapping over only newly arrived work. To implement clustering in these systems, we would likely maintain a repository per clustering phase. Avoiding the need to re-map the entire repository would allow us to make batches smaller, reducing latency. DryadInc [31] attacks the same problem by reusing identical portions of the computation from previous runs and allowing the user to specify a merge function that combines new input with previous iterations’ outputs. These systems represent a middle-ground between mapping over the entire repository using MapReduce and processing a single document at a time with Percolator.

Databases satisfy many of the requirements of an incremental system: a RDBMS can make many independent and concurrent changes to a large corpus and provides a flexible language for expressing computation (SQL). In fact, Percolator presents the user with a database-like interface: it supports transactions, iterators, and secondary indexes. While Percolator provides distributed transactions, it is by no means a full-fledged DBMS: it lacks a query language, for example, as well as full relational operations such as join. Percolator is also designed to operate at much larger scales than existing parallel databases and to deal better with failed machines. Unlike Percolator, database systems tend to emphasize latency over throughput since a human is often waiting for the results of a database query.

The organization of data in Percolator mirrors that of shared-nothing parallel databases [7, 15, 4]. Data is distributed across a number of commodity machines in shared-nothing fashion: the machines communicate only via explicit RPCs; no shared memory or shared disks are used. Data stored by Percolator is partitioned by Bigtable into tablets of contiguous rows which are distributed among machines; this mirrors the declustering performed by parallel databases.

The transaction management of Percolator builds on a long line of work on distributed transactions for database systems. Percolator implements snapshot isolation [5] by extending multi-version timestamp ordering [6] across a distributed system using two-phase commit.

An analogy can be drawn between the role of observers in Percolator to incrementally move the system towards a “clean” state and the incremental maintenance of materialized views in traditional databases (see Gupta and Mumick [21] for a survey of the field). In practice, while some indexing tasks like clustering documents by

contents could be expressed in a form appropriate for incremental view maintenance it would likely be hard to express the transformation of a raw document into an indexed document in such a form.

The utility of parallel databases and, by extension, a system like Percolator, has been questioned several times [17] over their history. Hardware trends have, in the past, worked against parallel databases. CPUs have become so much faster than disks that a few CPUs in a shared-memory machine can drive enough disk heads to service required loads without the complexity of distributed transactions: the top TPC-E benchmark results today are achieved on large shared-memory machines connected to a SAN. This trend is beginning to reverse itself, however, as the enormous datasets like those Percolator is intended to process become far too large for a single shared-memory machine to handle. These datasets require a distributed solution that can scale to 1000s of machines, while existing parallel databases can utilize only 100s of machines [30]. Percolator provides a system that is scalable enough for Internet-sized datasets by sacrificing some (but not all) of the flexibility and low-latency of parallel databases.

Distributed storage systems like Bigtable have the scalability and fault-tolerance properties of MapReduce but provide a more natural abstraction for storing a repository. Using a distributed storage system allows for low-latency updates since the system can change state by mutating the repository rather than rewriting it. However, Percolator is a data transformation system, not only a data storage system: it provides a way to structure computation to transform that data. In contrast, systems like Dynamo [14], Bigtable, and PNUTS [11] provide highly available data storage without the attendant mechanisms of transformation. These systems can also be grouped with the NoSQL databases (MongoDB [27], to name one of many): both offer higher performance and scale better than traditional databases, but provide weaker semantics.

Percolator extends Bigtable with multi-row, distributed transactions, and it provides the observer interface to allow applications to be structured around notifications of changed data. We considered building the new indexing system directly on Bigtable, but the complexity of reasoning about concurrent state modification without the aid of strong consistency was daunting. Percolator does not inherit all of Bigtable's features: it has limited support for replication of tables across data centers, for example. Since Bigtable's cross data center replication strategy is consistent only on a per-tablet basis, replication is likely to break invariants between writes in a distributed transaction. Unlike Dynamo and PNUTS which serve responses to users, Percolator is willing to accept the lower availability of a single data center in return for stricter consistency.

Several research systems have, like Percolator, extended distributed storage systems to include strong consistency. Sinfonia [3] provides a transactional interface to a distributed repository. Earlier published versions of Sinfonia [2] also offered a notification mechanism similar to the Percolator's observer model. Sinfonia and Percolator differ in their intended use: Sinfonia is designed to build distributed infrastructure while Percolator is intended to be used directly by applications (this probably explains why Sinfonia's authors dropped its notification mechanism). Additionally, Sinfonia's mini-transactions have limited semantics compared to the transactions provided by RDBMSs or Percolator: the user must specify a list of items to compare, read, and write prior to issuing the transaction. The mini-transactions are sufficient to create a wide variety of infrastructure but could be limiting for application builders.

CloudTPS [34], like Percolator, builds an ACID-compliant datastore on top of a distributed storage system (HBase [23] or Bigtable). Percolator and CloudTPS systems differ in design, however: the transaction management layer of CloudTPS is handled by an intermediate layer of servers called local transaction managers that cache mutations before they are persisted to the underlying distributed storage system. By contrast, Percolator uses clients, directly communicating with Bigtable, to coordinate transaction management. The focus of the systems is also different: CloudTPS is intended to be a backend for a website and, as such, has a stronger focus on latency and partition tolerance than Percolator.

ElasTraS [12], a transactional data store, is architecturally similar to Percolator; the Owning Transaction Managers in ElasTraS are essentially tablet servers. Unlike Percolator, ElasTraS offers limited transactional semantics (Sinfonia-like mini-transactions) when dynamically partitioning the dataset and has no support for structuring computation.

## 5 Conclusion and Future Work

We have built and deployed Percolator and it has been used to produce Google's websearch index since April, 2010. The system achieved the goals we set for reducing the latency of indexing a single document with an acceptable increase in resource usage compared to the previous indexing system.

The TPC-E results suggest a promising direction for future investigation. We chose an architecture that scales linearly over many orders of magnitude on commodity machines, but we've seen that this costs a significant 30-fold overhead compared to traditional database architectures. We are very interested in exploring this tradeoff and characterizing the nature of this overhead: how much is fundamental to distributed storage systems, and how much can be optimized away?

## Acknowledgments

Percolator could not have been built without the assistance of many individuals and teams. We are especially grateful to the members of the indexing team, our primary users, and the developers of the many pieces of infrastructure who never failed to improve their services to meet our increasingly large demands.

## References

- [1] TPC benchmark E standard specification version 1.9.0. Tech. rep., Transaction Processing Performance Council, September 2009.
- [2] AGUILERA, M. K., KARAMANOLIS, C., MERCHANT, A., SHAH, M., AND VEITCH, A. Building distributed applications using Sinfonia. Tech. rep., Hewlett-Packard Labs, 2006.
- [3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07* (2007), ACM, pp. 159–174.
- [4] BARU, C., FECTEAU, G., GOYAL, A., HSIAO, H.-I., JHINGRAN, A., PADMANABHAN, S., WILSON, W., AND I HSIAO, A. G. H. DB2 parallel edition, 1995.
- [5] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O’NEIL, E., AND O’NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD* (New York, NY, USA, 1995), ACM, pp. 1–10.
- [6] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Computer Surveys 13*, 2 (1981), 185–221.
- [7] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 4–24.
- [8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *7th OSDI* (Nov. 2006).
- [9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIRES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th OSDI* (Nov. 2006), pp. 205–218.
- [10] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTEN, J. M. MapReduce online. In *7th NSDI* (2010).
- [11] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!’s hosted data serving platform. In *Proceedings of VLDB* (2008).
- [12] DAS, S., AGRAWAL, D., AND ABBADI, A. E. ElasTraS: An elastic transactional data store in the cloud. In *USENIX HotCloud* (June 2009).
- [13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *6th OSDI* (Dec. 2004), pp. 137–150.
- [14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon’s highly available key-value store. In *SOSP ’07* (2007), pp. 205–220.
- [15] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), 44–62.
- [16] DEWITT, D., AND STONEBRAKER, M. MapReduce: A major step backwards. <http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/>.
- [17] DEWITT, D. J., AND GRAY, J. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec.* 19, 4 (1990), 104–112.
- [18] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. Twister: A runtime for iterative MapReduce. In *The First International Workshop on MapReduce and its Applications* (2010).
- [19] GERSHENSON, C., AND PINEDA, L. A. Why does public transport not arrive on time? The pervasiveness of equal headway instability. *PLoS ONE* 4, 10 (10 2009).
- [20] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. vol. 37, pp. 29–43.
- [21] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques, and applications, 1995.
- [22] Hadoop. <http://hadoop.apache.org/>.
- [23] HBase. <http://hbase.apache.org/>.
- [24] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys ’07* (New York, NY, USA, 2007), ACM, pp. 59–72.
- [25] IYER, S., AND CUTTS, M. Help test some next-generation infrastructure. <http://googlewebmastercentral.blogspot.com/2009/08/help-test-some-next-generation.html>, August 2009.
- [26] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *SoCC ’10: Proceedings of the 1st ACM symposium on cloud computing* (2010), pp. 51–62.
- [27] MongoDB. <http://mongodb.org/>.
- [28] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.
- [29] PATON, N. W., AND DÍAZ, O. Active database systems. *ACM Computing Surveys 31*, 1 (1999), 63–103.
- [30] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD ’09* (June 2009), ACM.
- [31] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *USENIX workshop on Hot Topics in Cloud Computing* (2009).
- [32] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM* 24, 7 (1981), 412–418.
- [33] NEC Express5800/A1080a-E TPC-E results. [http://www.tpc.org/tfce/results/tfce\\_result\\_detail.asp?id=110033001](http://www.tpc.org/tfce/results/tfce_result_detail.asp?id=110033001), Mar. 2010.
- [34] WEI, Z., PIERRE, G., AND CHI, C.-H. CloudTPS: Scalable transactions for Web applications in the cloud. Tech. Rep. IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010. <http://www.globule.org/publi/CSTWAC ircs53.html>.
- [35] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *2nd USENIX workshop on Hot Topics in Cloud Computing* (2010).

# Interpreting the Data: Parallel Analysis with Sawzall

Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan  
Google, Inc.

## Abstract

Very large data sets often have a flat but regular structure and span multiple disks and machines. Examples include telephone call records, network logs, and web document repositories. These large data sets are not amenable to study using traditional database techniques, if only because they can be too large to fit in a single relational database. On the other hand, many of the analyses done on them can be expressed using simple, easily distributed computations: filtering, aggregation, extraction of statistics, and so on.

We present a system for automating such analyses. A filtering phase, in which a query is expressed using a new procedural programming language, emits data to an aggregation phase. Both phases are distributed over hundreds or even thousands of computers. The results are then collated and saved to a file. The design—including the separation into two phases, the form of the programming language, and the properties of the aggregators—exploits the parallelism inherent in having data and computation distributed across many machines.

## 1 Introduction

Many data sets are too large, too dynamic, or just too unwieldy to be housed productively in a relational database. One common scenario is a set of many plain files—sometimes amounting to petabytes of data—distributed across many disks on many computers (Figure 1). The files in turn comprise many records, organized along some obvious axes such as time or geography. Examples might include a web page repository used to construct the index for an internet search engine, the system health records from thousands of on-line server machines, telephone call records or other business transaction logs, network packet traces, web server query logs, or even higher-level data such as satellite imagery.

Quite often the analyses applied to these data sets can be expressed simply, using operations much less sophisticated than a general SQL query. For instance, we might wish to count records that satisfy a certain property, or extract them, or search for anomalous records, or construct frequency histograms of the values of certain fields in the records. In other cases the analyses might be more

intricate but still be expressible as a series of simpler steps, operations that can be mapped easily onto a set of records in files.

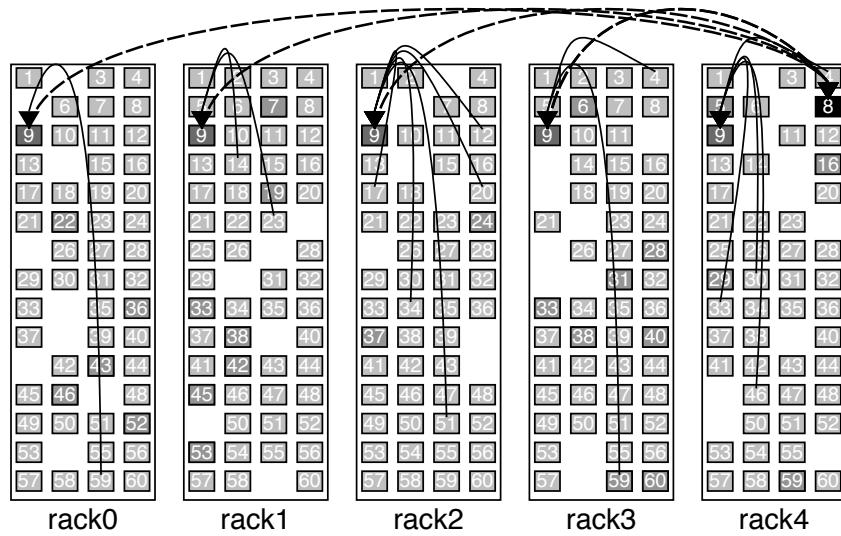


Figure 1: Five racks of 50-55 working computers each, with four disks per machine. Such a configuration might have a hundred terabytes of data to be processed, distributed across some or all of the machines. Tremendous parallelism can be achieved by running a filtering phase independently on all 250+ machines and aggregating their emitted results over a network between them (the arcs). Solid arcs represent data flowing from the analysis machines to the aggregators; dashed arcs represent the aggregated data being merged, first into one file per aggregation machine and then to a single final, collated output file.

Not all problems are like this, of course. Some benefit from the tools of traditional programming—arithmetic, patterns, storage of intermediate values, functions, and so on—provided by procedural languages such as Python [1] or Awk [12]. If the data is unstructured or textual, or if the query requires extensive calculation to arrive at the relevant data for each record, or if many distinct but related calculations must be applied during a single scan of the data set, a procedural language is often the right tool. Awk was specifically designed for making data mining easy. Although it is used for many other things today, it was inspired by a tool written by Marc Rochkind that executed procedural code when a regular expression matched a record in telephone system log data [13]. Even today, there are large Awk programs in use for mining telephone logs. Languages such as C

or C++, while capable of handling such tasks, are more awkward to use and require more effort on the part of the programmer. Still, Awk and Python are not panaceas; for instance, they have no inherent facilities for processing data on multiple machines.

Since the data records we wish to process do live on many machines, it would be fruitful to exploit the combined computing power to perform these analyses. In particular, if the individual steps can be expressed as query operations that can be evaluated one record at a time, we can distribute the calculation across all the machines and achieve very high throughput. The results of these operations will then require an aggregation phase. For example, if we are counting records, we need to gather the counts from the individual machines before we can report the total count.

We therefore break our calculations into two phases. The first phase evaluates the analysis on each record individually, while the second phase aggregates the results (Figure 2). The system described in this paper goes even further, however. The analysis in the first phase is expressed in a new procedural programming language that executes one record at a time, in isolation, to calculate query results for each record. The second phase is restricted to a set of predefined aggregators that process the intermediate results generated by the first phase. By restricting the calculations to this model, we can achieve very high throughput. Although not all calculations fit this model well, the ability to harness a thousand or more machines with a few lines of code provides some compensation.

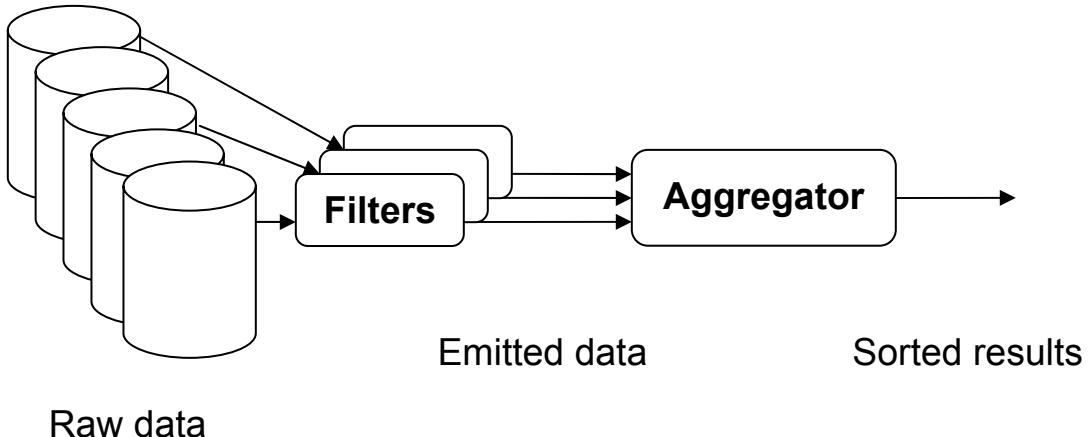


Figure 2: The overall flow of filtering, aggregating, and collating. Each stage typically involves less data than the previous.

Of course, there are still many subproblems that remain to be solved. The calculation must be divided into pieces and distributed across the machines holding the data, keeping the computation as near the data as possible to avoid network bottlenecks. And when there are many machines there is a high probability of some of them failing during the analysis, so the system must be

fault tolerant. These are difficult and interesting problems, but they should be handled without the involvement of the user of the system. Google has several pieces of infrastructure, including GFS [9] and MapReduce [8], that cope with fault tolerance and reliability and provide a powerful framework upon which to implement a large, parallel system for distributed analysis. We therefore hide the details from the user and concentrate on the job at hand: expressing the analysis cleanly and executing it quickly.

## 2 Overview

A typical job will process anywhere from a gigabyte to many terabytes of data on hundreds or even thousands of machines in parallel, some executing the query while others aggregate the results. An analysis may consume months of CPU time, but with a thousand machines that will only take a few hours of real time.

Our system's design is influenced by two observations.

First, if the querying operations are commutative across records, the order in which the records are processed is unimportant. We can therefore work through the input in arbitrary order.

Second, if the aggregation operations are commutative, the order in which the intermediate values are processed is unimportant. Moreover, if they are also associative, the intermediate values can be grouped arbitrarily or even aggregated in stages. As an example, counting involves addition, which is guaranteed to be the same independent of the order in which the values are added and independent of any intermediate subtotals that are formed to coalesce intermediate values.

The constraints of commutativity and associativity are not too restrictive; they permit a wide range of valuable analyses, including: counting, filtering, sampling, generating histograms, finding the most frequent items, and many more.

The set of aggregations is limited but the query phase can involve more general computations, which we express in a new interpreted, procedural programming language called Sawzall.<sup>1</sup> (An interpreted language is fast enough: most of the programs are small and on large data sets the calculation tends to be I/O bound, as is discussed in the section on performance.)

An analysis operates as follows. First the input is divided into pieces to be processed separately, perhaps individual files or groups of records, which are located on multiple storage nodes.

Next, a Sawzall interpreter is instantiated for each piece of data. This will involve many machines, perhaps the same machines that store the data or perhaps a different, nearby set. There will often be more data pieces than computers, in which case scheduling and load balancing software will dispatch the pieces to machines as they complete the processing of prior pieces.

---

<sup>1</sup>Not related to the portable reciprocating power saw trademark of the Milwaukee Electric Tool Corporation.

The Sawzall program operates on each input record individually. The output of the program is, for each record, zero or more intermediate values—integers, strings, key-value pairs, tuples, etc.—to be combined with values from other records.

These intermediate values are sent to further computation nodes running the aggregators, which collate and reduce the intermediate values to create the final results. In a typical run, the majority of machines will run Sawzall and a smaller fraction will run the aggregators, reflecting not only computational overhead but also network load balancing issues; at each stage, the amount of data flowing is less than at the stage before (see Figure 2).

Once all the processing is complete, the results will be spread across multiple files, one per aggregation machine. A final step collects, collates, and formats the final results, to be viewed or stored in a single file.

### 3 A Simple Example

These ideas may become clearer in the context of a simple example. Let's say our input is a set of files comprising records that each contain one floating-point number. This complete Sawzall program will read the input and produce three results: the number of records, the sum of the values, and the sum of the squares of the values.

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

The first three lines declare the aggregators `count`, `total`, and `sum_of_squares`. The keyword `table` introduces an aggregator type; aggregators are called tables in Sawzall even though they may be singletons. These particular tables are `sum` tables; they add up the values emitted to them, `ints` or `floats` as appropriate.

For each input record, Sawzall initializes the pre-defined variable `input` to the uninterpreted byte string of the input record. Therefore, the line

```
x: float = input;
```

converts the input record from its external representation into a native floating-point number, which is stored in a local variable `x`. Finally, the three `emit` statements send intermediate values to the aggregators.

When run, the program is instantiated once for each input record. Local variables declared by the program are created anew for each instance, while tables declared in the program are shared by all instances. Emitted values are summed up by the global tables. When all records have been processed, the values in the tables are saved to one or more files.

The next few sections outline some of the Google infrastructure this system is built upon: protocol buffers, the Google File System, the Workqueue, and MapReduce. Subsequent sections describe the language and other novel components of the system in more detail.

## 4 Protocol Buffers

Although originally conceived for defining the messages communicated between servers, Google's *protocol buffers* are also used to describe the format of permanent records stored on disk. They serve a purpose similar to XML, but have a much denser, binary representation, which is in turn often wrapped by an additional compression layer for even greater density.

Protocol buffers are described by a data description language (DDL) that defines the content of the messages. A *protocol compiler* takes this language and generates executable code to manipulate the protocol buffers. A flag to the protocol compiler specifies the output language: C++, Java, Python, and so on. For each protocol buffer type in the DDL file, the output of the protocol compiler includes the native data structure definition, code to access the fields, marshaling and unmarshaling code to translate between the native representation and the external binary format, and debugging and pretty-printing interfaces. The generated code is compiled and linked with the application to provide efficient, clean access to the records. There is also a tool suite for examining and debugging stored protocol buffers.

The DDL forms a clear, compact, extensible notation describing the layout of the binary records and naming the fields. Protocol buffer types are roughly analogous to C structs: they comprise named, typed fields. However, the DDL includes two additional properties for each field: a distinguishing integral tag, used to identify the field in the binary representation, and an indication of whether the field is required or optional. These additional properties allow for backward-compatible extension of a protocol buffer, by marking all new fields as optional and assigning them an unused tag.

For example, the following describes a protocol buffer with two required fields. Field *x* has tag 1, and field *y* has tag 2.

```
parsed message Point {  
    required int32 x = 1;  
    required int32 y = 2;  
};
```

To extend this two-dimensional point, one can add a new, optional field with a new tag. All existing Points stored on disk remain readable; they are compatible with the new definition since the new field is optional.

```
parsed message Point {
    required int32 x = 1;
    required int32 y = 2;
    optional string label = 3;
};
```

Most of the data sets our system operates on are stored as records in protocol buffer format. The protocol compiler was extended by adding support for Sawzall to enable convenient, efficient I/O of protocol buffers in the new language.

## 5 Google File System (GFS)

The data sets are often stored in GFS, the Google File System [9]. GFS provides a reliable distributed storage system that can grow to petabyte scale by keeping data in 64-megabyte “chunks” stored on disks spread across thousands of machines. Each chunk is replicated, usually 3 times, on different machines so GFS can recover seamlessly from disk or machine failure.

GFS runs as an application-level file system with a traditional hierarchical naming scheme. The data sets themselves have regular structure, represented by a large set of individual GFS files each around a gigabyte in size. For example, a document repository (the result of a web crawl) holding a few billion HTML pages might be stored as several thousand files each storing a million or so documents of a few kilobytes each, compressed.

## 6 Workqueue and MapReduce

The business of scheduling a job to run on a cluster of machines is handled by software called (somewhat misleadingly) the Workqueue. In effect, the Workqueue creates a large-scale time sharing system out of an array of computers and their disks. It schedules jobs, allocates resources, reports status, and collects the results.

The Workqueue is similar to several other systems such as Condor [16]. We often overlay a Workqueue cluster and a GFS cluster on the same set of machines. Since GFS is a storage system, its CPUs are often lightly loaded, and the free computing cycles can be used to run Workqueue jobs.

MapReduce [8] is a software library for applications that run on the Workqueue. It performs three primary services. First, it provides an execution model for programs that operate on many data items in parallel. Second, it isolates the application from the details of running a distributed program, including issues such as data distribution, scheduling, and fault tolerance. Finally, when possible it schedules the computations so each unit runs on the machine or rack that holds its GFS data, reducing the load on the network.

As the name implies, the execution model consists of two phases: a first phase that *maps* an execution across all the items in the data set; and a second phase that *reduces* the results of the first phase to arrive at a final answer. For example, a sort program using MapReduce would map a standard sort algorithm upon each of the files in the data set, then reduce the individual results by running a merge sort to produce the final output. On a thousand-machine cluster, a MapReduce implementation can sort a terabyte of data at an aggregate rate of a gigabyte per second [9].

Our data processing system is built on top of MapReduce. The Sawzall interpreter runs in the map phase. It is instantiated in parallel on many machines, with each instantiation processing one file or perhaps GFS chunk. The Sawzall program executes once for each record of the data set. The output of this map phase is a set of data items to be accumulated in the aggregators. The aggregators run in the reduce phase to condense the results to the final output.

The following sections describe these pieces in more detail.

## 7 Sawzall Language Overview

The query language, Sawzall, operates at about the level of a type-safe scripting language. For problems that can be solved in Sawzall, the resulting code is much simpler and shorter—by a factor of ten or more—than the corresponding C++ code in MapReduce.

The syntax of statements and expressions is borrowed largely from C; `for` loops, `while` loops, `if` statements and so on take their familiar form. Declarations borrow from the Pascal tradition:

```
i: int;      # a simple integer declaration
i: int = 0;  # a declaration with an initial value
```

The basic types include integer (`int`), a 64-bit signed value; floating point (`float`), a double-precision IEEE value; and special integer-like types called `time` and `fingerprint`, also 64 bits long. The `time` type has microsecond resolution and the libraries include convenient functions for decomposing and manipulating these values. The `fingerprint` type represents an internally-computed hash of another value, which makes it easy to construct data structures such as aggregators indexed by the fingerprint of a datum. As another example, one might use the fingerprint of a URL or of its HTML contents to do efficient comparison or fair selection among a set of documents.

There are also two primitive array-like types: `bytes`, similar to a C array of `unsigned char`; and `string`, which is defined to hold characters from the Unicode character set. There is no “character” type; the elements of byte arrays and strings are `int` values, although an `int` is capable of holding a much larger value than will fit in a byte or string element.

Compound types include arrays, maps (an overloaded term in this paper), and tuples. Arrays are indexed by integer values, while maps are like associative arrays or Python dictionaries and may be indexed by any type, with the indices unordered and the storage for the elements created on demand. Finally, tuples represent arbitrary groupings of data, like a C struct or Pascal record. A `typedef`-like mechanism allows any type to be given a shorthand name.

Conversion operators translate values from one type to another, and encapsulate a wide range of conversions. For instance, to extract the floating point number represented by a string, one simply converts the value:

```
f: float;  
s: string = "1.234";  
f = float(s);
```

Some conversions take parameters; for example

```
string(1234, 16)
```

generates the hexadecimal representation of the integer, while

```
string(utf8_bytes, "UTF-8")
```

will interpret the byte array in UTF-8 representation and return a string of the resulting Unicode character values.

For convenience, and to avoid the stutter endemic in some languages’ declaration syntaxes, in initializations the appropriate conversion operation (with default parameter values) is supplied implicitly by the compiler. Thus

```
b: bytes = "Hello, world!\n";
```

is equivalent to the more verbose

```
b: bytes = bytes("Hello, world!\n", "UTF-8");
```

Values of any type can be converted into strings to aid debugging.

One of the most important conversion operations is associated with protocol buffers. A compile-time directive in Sawzall, `proto`, somewhat analogous to C’s `#include` directive, imports the DDL for a protocol buffer from a file and defines the Sawzall tuple type that describes the layout. Given that tuple description, one can convert the input protocol buffer to a Sawzall value.

For each input record, the special variable `input` is initialized by the interpreter to the uninterpreted byte array holding the data, typically a protocol buffer. In effect, the execution of the Sawzall program for each record begins with the implicit statement:

```
input: bytes = next_record_from_input();
```

Therefore, if the file `some_record.proto` includes the definition of a protocol buffer of type `Record`, the following code will parse each input record and store it in the variable `r`:

```
proto "some_record.proto" # define 'Record'  
r: Record = input; # convert input to Record
```

The language has a number of other traditional features such as functions and a wide selection of intrinsic library calls. Among the intrinsics are functions that connect to existing code for internationalization, document parsing, and so on.

## 7.1 Input and Aggregation

Although at the statement level Sawzall is a fairly ordinary language, it has two very unusual features, both in some sense outside the language itself:

1. A Sawzall program defines the operations to be performed on a single record of the data. There is nothing in the language to enable examining multiple input records simultaneously, or even to have the contents of one input record influence the processing of another.
2. The only output primitive in the language is the `emit` statement, which sends data to an external aggregator that gathers the results from each record and correlates and processes the result.

Thus the usual behavior of a Sawzall program is to take the `input` variable, parse it into a data structure by a conversion operation, examine the data, and emit some values. We saw this pattern in the simple example of Section 3.

Here is a more representative Sawzall program. Given a set of logs of the submissions to our source code management system, this program will show how the rate of submission varies through the week, at one-minute resolution:

```
proto "p4stat.proto"  
submitsthroughweek: table sum[minute: int] of count: int;  
  
log: P4ChangelistStats = input;  
t: time = log.time; # microseconds  
  
minute: int = minuteof(t)+60*(hourof(t)+24*(dayofweek(t)-1));
```

```
emit submitsthroughweek[minute] <- 1;
```

The program begins by importing the protocol buffer description from the file `p4stat.proto`. In particular, it declares the type `P4ChangelistStats`. (The programmer must know the type that will arise from the `proto` directive, but this is an accepted property of the protocol buffer DDL.)

The declaration of `submitsthroughweek` appears next. It defines a table of sum values, indexed by the integer `minute`. Note that the index value in the declaration of the table is given an optional name (`minute`). This name serves no semantic purpose, but makes the declaration easier to understand and provides a label for the field in the aggregated output.

The declaration of `log` converts the input byte array (using code generated by the `proto` directive) into the Sawzall type `P4ChangelistStats`, a tuple, and stores the result in the variable `log`. Then we pull out the time value and, for brevity, store it in the variable `t`.

The next declaration has a more complicated initialization expression that uses some built-in functions to extract the cardinal number of the minute of the week from the time value.

Finally, the `emit` statement counts the submission represented in this record by adding its contribution to the particular minute of the week in which it appeared.

To summarize, this program, for each record, extracts the time stamp, bins the time to a minute within the week, and adds one to the count for that minute. Then, implicitly, it starts again on the next record.

When we run this program over all the submission logs—which span many months—and plot the result, we see an aggregated view of how activity tends to vary through the week, minute by minute. The output looks like this:

```
submitsthroughweek[0] = 27
submitsthroughweek[1] = 31
submitsthroughweek[2] = 52
submitsthroughweek[3] = 41
...
submitsthroughweek[10079] = 34
```

When plotted, the graph looks like Figure 3.

The salient point is not the data itself, of course, but the simplicity of the program that extracts it.

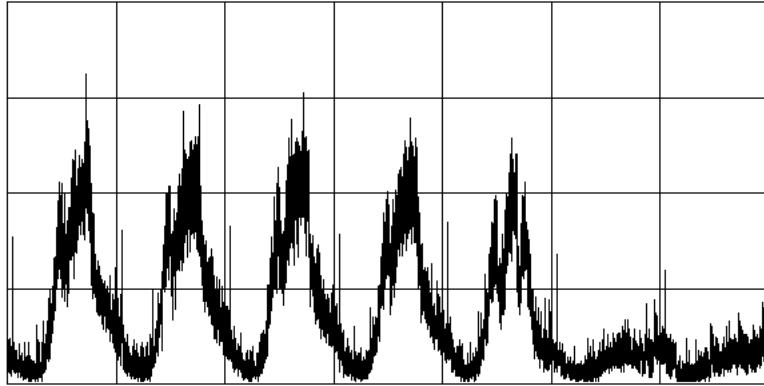


Figure 3: Frequency of submits to the source code repository through the week. The graph starts at midnight Monday morning.

## 7.2 More About Aggregators

Aggregation is done outside the language for a couple of reasons. A more traditional language would use the language itself to correlate results, but some of the aggregation algorithms are sophisticated and best implemented in a native language and packaged in some form. More important, drawing an explicit line between filtering and aggregation enables a high degree of parallelism, even though it hides the parallelism from the language itself. Nowhere in Sawzall is the multiplicity of records apparent, yet a typical Sawzall job operates on billions of records, often on hundreds or thousands of machines simultaneously.

Focusing attention on the aggregators encourages the creation of unusual statistical collectors. There are a number of aggregators available; here is an incomplete list, with examples:

- Collection:

```
c: table collection of string;
```

A simple list of all the emitted values, including all duplicates, in arbitrary order.

- Sample:

```
s: table sample(100) of string;
```

Like collection, but chooses an unbiased sample of the emitted values. The size of the desired sample is provided as a parameter.

- Sum:

```
s: table sum of { count: int, revenue: float };
```

The summation of all the emitted values. The values must be arithmetic or composed of

arithmetic values, as is the tuple in the example. For compound values, the components are summed elementwise. In the example shown, if `count` were always 1 when emitted, the average revenue could be computed after the run by dividing `revenue` by `count`.

- Maximum:

```
m: table maximum(10) of string weight length: int;
```

The highest-weighted values. The values are tagged with the weight, and the value with the highest weight is chosen. The parameter (10 in the example) specifies the number of values to keep. The weight, introduced by the obvious keyword, has its type (here `int`) provided in the declaration and its value in the `emit` statement. For the example given,

```
emit m <- s weight len(s);
```

will report the ten longest strings in the data.

- Quantile:

```
q: table quantile(101) of response_in_ms: int;
```

Use the set of emitted values to construct a cumulative probability distribution represented by the quantile values for each increment of probability. (The algorithm is a distributed variant of that of Greenwald and Khanna [10].) The example could be used to see how system response varies. With parameter 101, it computes percentiles; table indices in that case range from 0 to 100, so the element at index 50 records the median time, while the element at index 99 is the time value for which 99% of the response times were equal or lower.

- Top:

```
t: table top(10) of language: string;
```

Estimate which values are the most popular. (By contrast, the maximum table finds the items with the highest weight, not the highest frequency of occurrence.)

For our example,

```
emit t <- language_of_document(input);
```

will estimate the ten most frequently occurring languages found in a document repository.

For large data sets, it can be prohibitively expensive to find the precise ordering of frequency of occurrence, but there are efficient estimators. The top table uses a distributed variant of the algorithm by Charikar, Chen, and Farach-Colton [5]. The algorithm is approximate: with high probability it returns approximately the correct top elements. Its commutativity and associativity are also approximate: changing the order that the input is processed can change the final results. To compensate, in addition to computing the counts of the elements, we

compute the estimated error of those counts. If the error is small compared to the counts, the quality of the results is high, while if the error is relatively large, chances are the results are bad. The algorithm used in the top tables works well for skewed distributions but produces unreliable results when all the values occur with similar frequency. For our analyses this is rarely an issue.

- Unique:

```
u: table unique(10000) of string;
```

The unique table is unusual. It reports the estimated size of the population of unique items emitted to it. A sum table could be used to count the total number of elements, but a unique table will ignore duplicates; in effect it computes the size of the set of input values. The unique table is also unusual in that its output is always a count, regardless of the type of the values emitted to it. The parameter specifies the size of the internal table used to do the estimation; a value of 10000 generates a final value within  $\pm 2\%$  of the correct answer with 95% probability. (For set size  $N$ , the standard deviation is about  $N \times param^{-1/2}$ .)

### 7.3 Implementing an Aggregator

Occasionally, ideas arise for new aggregators to be supported by Sawzall. Adding new aggregators is fairly easy, although not trivial. Their implementations are connected to the Sawzall runtime and interact with system internals, managing low-level data formats and distributed computations. Moreover, most aggregators handle many different Sawzall data types, further complicating their implementations. There is a support in the Sawzall runtime to ease these tasks, but we have not lavished these libraries with the same care given the Sawzall language.

The Sawzall runtime itself manages the plumbing of emitting values, passing data between machines, and so on, while a set of five functions specific to each aggregator class handles type checking, allocation of aggregation state, merging emitted values, packaging the data into an intermediate form suitable for further merging, and producing the final output data. To add a new aggregator, the programmer implements this five-function interface and links it with the Sawzall runtime. For simple aggregators, such as `sum`, the implementation is straightforward. For more sophisticated aggregators, such as `quantile` and `top`, care must be taken to choose an algorithm that is commutative, associative and reasonably efficient for distributed processing. Our smallest aggregators are implemented in about 200 lines of C++, while the largest require about 1000.

Some aggregators can process data as part of the mapping phase to reduce the network bandwidth to the aggregators. For instance, a `sum` table can add the individual elements locally, emitting only an occasional subtotal value to the remote aggregator. In MapReduce terminology, this is the *combining* phase of the MapReduce run, a sort of middle optimization step between map and reduce.

It would be possible to create a new aggregator-specification language, or perhaps extend Sawzall to handle the job. However, we have found that on the rare occasion that a new aggregator is needed, it's been easy to implement. Moreover, an aggregator's performance is crucial for the performance of the system, and since it needs to interact with the Mapreduce framework, it is best expressed in C++, the language of the Sawzall runtime and Mapreduce.

## 7.4 Indexed Aggregators

An aggregator can be indexed, which in effect creates a distinct individual aggregator for each unique value of the index. The index can be of any Sawzall type and can be compounded to construct multidimensional aggregators.

For example, if we are examining web server logs, the table

```
table top(1000) [country: string][hour: int] of request: string;
```

could be used to find the 1000 most popular request strings for each country, for each hour.

The Sawzall runtime automatically creates individual aggregators as new index values arise, similarly to maps adding entries as new key values occur. The aggregation phase will collate values according to index and generate the appropriate aggregated values for each distinct index value.

As part of the collation, the values are sorted in index order, to facilitate merging values from different machines. When the job completes, the values are therefore arranged in index order, which means the output from the aggregators is in index order.

The indices themselves form a useful source of information. To return to the web server example above, after the run the set of values recorded in the `country` index form a record of the set of countries from which requests were received. In effect, the properties of index generation turn the indices into a way to recover sets of values. The indices resulting from

```
t1: table sum[country: string] of int
```

would be equivalent to the values collected by

```
t2: table collection of country: string
```

with duplicates removed. Its output would be of the form

```
t1["china"] = 123456  
t1["japan"] = 142367  
...
```

which the user would need to post-process with a separate tool to extract the set of countries.

## 8 System Model

Now that the basic features of the language have been presented, we can give an overview of the high-level system model by which data analyses are conducted.

The system operates in a batch execution style: the user submits a job, which runs on a fixed set of files, and collects the output at the end of the run. The input format and location (typically a set of files in GFS) and the output destination are specified outside the language, as arguments to the command that submits the job to the system.

The command is called `saw` and its basic parameters define the name of the Sawzall program to be run, a pattern that matches the names of the files that hold the data records to be processed, and a set of files to receive the output. A typical job might be instantiated like this:

```
saw --program code.szl \
    --workqueue testing \
    --input_files /gfs/cluster1/2005-02-0[1-7]/submits.* \
    --destination /gfs/cluster2/$USER/output@100
```

The `program` flag names the file containing the Sawzall source of the program to be run, while the `workqueue` flag names the Workqueue cluster on which to run it. The `input_files` argument accepts standard Unix shell file-name-matching metacharacters to identify the files to be processed. The `destination` argument names the output files to be generated, using the notation that an at sign (@) introduces the number of files across which to distribute the results.

After the run, the user collects the data using a tool that accepts a `source` argument with the same notation as the `destination` argument to `saw`:

```
dump --source /gfs/cluster2/$USER/output@100 --format csv
```

This program merges the output data distributed across the named files and prints the final results in the specified format.

When a `saw` job request is received by the system, a Sawzall processor is invoked to verify that the program is syntactically valid. If it is, the source code is sent to the Workqueue machines for execution. The number of such machines is determined by looking at the size of the input and the number of input files, with a default upper limit proportional to the size of the Workqueue. A flag to `saw` can override this behavior and set the number explicitly.

These machines each compile the Sawzall program and run it on their portion of the input data, one record at a time. The emitted values are passed to the Sawzall runtime system, which collects them and performs an initial stage of aggregation locally, to reduce the amount of intermediate data sent to the aggregation machines. The runtime sends the data when the memory used for intermediate values reaches a threshold, or when input processing is complete.

The aggregation machines, whose number is determined by the count in the destination flag to `saw`, collect the intermediate outputs from the execution machines and merge them to each produce a sorted subset of the output. To ensure the merging process sees all the necessary data, the intermediate outputs are assigned to aggregation machines deterministically according to the particular table and any index values it may have. Each aggregation machine's output is written to a file in GFS; the output of the entire job is a set of files, one per aggregation machine. Aggregation is therefore a parallel operation that can take full advantage of the cluster's resources.

In general, the generation of the output is fairly evenly spread across the aggregation machines. Each element of a table has relatively modest size and the elements are spread across the different outputs. Collection tables are an exception; such tables can be of arbitrary size. To avoid overloading a single aggregation machine, the values inside a collection table are spread across the aggregation machines in a round-robin fashion. No aggregation machine sees all the intermediate values, but collection tables by definition do not combine duplicate results so this is not a problem.

Values stored by the aggregation machines are not in final form, but instead in an intermediate form carrying information to facilitate merging it with other values. In fact, this same intermediate form is used to pass the intermediate values from the execution machines to the aggregation machines. By leaving the values in an intermediate form, they can be merged with the values from another job. Large Sawzall jobs can be broken into pieces to be run separately, with final results constructed by merging the elements in a final phase. (This is one advantage of this system over plain MapReduce; even MapReduce can have problems with jobs that run for days or weeks of real time.)

This multi-stage merge—at execution machines, aggregations machines, and perhaps across jobs—is the reason the aggregators must be associative to work well. They must also be commutative, since the order in which the input records are processed is undefined, as is the order of merging of the intermediate values.

Typically, the data resulting from an analysis is much smaller than the input, but there are important examples where this is not the case. For instance, a program could use an indexed collection table to organize the input along some relevant axis, in which case the output would be as large as the input. Such transformations are handled efficiently even for large data sets. The input and the output are evenly partitioned across the execution and aggregation engines respectively. Each aggregation engine sorts its portion of the output data using a disk-based sort. The resulting data can then be combined efficiently to produce the final output using a merge sort.

## 8.1 Implementation

The Sawzall language is implemented as a conventional compiler, written in C++, whose target language is an interpreted instruction set, or byte-code. The compiler and the byte-code interpreter are part of the same binary, so the user presents source code to Sawzall and the system runs it

directly. (This is a common but not universal design; Java for instance splits its compiler and interpreter into separate programs.) Actually, the Sawzall language system is structured as a library with an external interface that accepts source code and compiles and runs it, along with bindings to connect to externally-provided aggregators.

Those aggregators are implemented by `saw`, which is a program that links to the Sawzall compiler and interpreter library and is in turn implemented above MapReduce, running Sawzall in the map phase and the aggregators in the reduce phase. MapReduce manages the distribution of execution and aggregation machines, locates the computation near the GFS machines holding the data to minimize network traffic and thereby improve throughput, and handles machine failure and other faults. The system is therefore a “MapReduction” [8], but an unusual one. It actually incorporates two instances of MapReduce.

The first, quick job runs in parallel on the Workqueue to evaluate the size of the input and set up the main job. (There may be many thousands of input files and it’s worth checking their size in parallel.) The second job then runs Sawzall. `Saw` thus acts in part as a convenient wrapper for MapReduce, setting up the job automatically based on the input rather than the usual method involving a number of user-specified flags. More important is the convenience of Sawzall and its aggregators. Leaving aside the comfort afforded by its simple programming model, Sawzall is implemented within a single binary that runs whatever code is presented to it; by contrast, other MapReductions must be written in C++ and compiled separately. Moreover, the various aggregators in Sawzall provide functionality that significantly extends the power of MapReduce. It would be possible to create a MapReduce wrapper library that contained the implementation of `top` tables and others, but even then they would not be nearly as simple or convenient to use as they are from Sawzall. Moreover, the ideas for some of those aggregators grew out of the system model provided by Sawzall. Although restrictive, the focus required by the Sawzall execution model can lead to unusual and creative programming ideas.

## 8.2 Chaining

One common use of the output of a Sawzall job is to inject the resulting data into a traditional relational database for subsequent analysis. This is usually done by a separate custom program, perhaps written in Python, that transforms the data into the SQL code to create the table. We might one day provide a more direct way to achieve this injection.

Sometimes the result of a Sawzall job is provided as input to another, by a process called *chaining*. A simple example is to calculate the exact “top 10” list for some input. The Sawzall `top` table is efficient but approximate. If the exact results are important, they can be derived by a two-step process. The first step creates an indexed `sum` table to count the frequency of input values; the second uses a `maximum` table to select the most popular. The steps are separate Sawzall jobs and the first step must run to completion before the second step can commence. Although a little

clumsy, chaining is an effective way to extend the data analysis tasks that can be expressed in Sawzall.

## 9 More Examples

Here is another complete example that illustrates how Sawzall is used in practice. It processes a web document repository to answer the question: for each web domain, which page has the highest PageRank [14]? Roughly speaking, which is the most linked-to page?

```
proto "document.proto"

max_pagerank_url:
    table maximum(1) [domain: string] of url: string
        weight pagerank: int;

doc: Document = input;

emit max_pagerank_url[domain(doc.url)] <- doc.url
    weight doc.pagerank;
```

The protocol buffer format is defined in the file "document.proto". The table is called max\_pagerank\_url and will record the highest-weighted value emitted for each index. The index is the domain, the value is the URL, and the weight is the document's Page Rank. The program parses the input record and then does a relatively sophisticated emit statement. It calls the library function domain(doc.url) to extract the domain of the URL to use as the index, emits the URL itself as the value, and uses the Page Rank for the document as the weight.

When this program is run over the repository, it shows the expected result that for most sites, the most-linked page is `www.site.com`—but there are surprises. The Acrobat download site is the top page for `adobe.com`, while those who link to `banknotes.com` go right to the image gallery and `bangkok-th.com` pulls right to the `Night_Life` page.

Because Sawzall makes it easy to express calculations like this, the program is nice and short. Even using MapReduce, the equivalent straight C++ program is about a hundred lines long.

Here is an example with a multiply-indexed aggregator. We wish to look at a set of search query logs and construct a map showing how the queries are distributed around the globe.

```
proto "querylog.proto"

queries_per_degree: table sum[lat: int][lon: int] of int;
```

```

log_record: QueryLogProto = input;

loc: Location = locationinfo(log_record.ip);
emit queries_per_degree[int(loc.lat)][int(loc.lon)] <- 1;

```

The program is straightforward. We import the DDL for the query logs, declare a table indexed by integer latitude and longitude and extract the query from the log. Then we use a built-in function that looks up the incoming IP address in a database to recover the location of the requesting machine (probably its ISP), and then count 1 for the appropriate latitude/longitude bucket. The expression `int(loc.lat)` converts `loc.lat`, a float, to an integer, thereby truncating it to the degree and making it suitable as an index value. For a higher-resolution map, a more sophisticated calculation would be required.

The output of this program is an array of values suitable for creating a map, as in Figure 4.



Figure 4: Query distribution.

## 10 Execution Model

At the statement level, Sawzall is an ordinary-looking language, but from a higher perspective it has several unusual properties, all serving the goal of enabling parallelism.

The most important, of course, is that it runs on one record at a time. This means it has no memory

of other records it has processed (except through values emitted to the aggregators, outside the language). It is routine for a Sawzall job to be executing on a thousand machines simultaneously, yet the system requires no explicit communication between those machines. The only communication is from the Sawzall executions to the downstream aggregators.

To reinforce this point, consider the problem of counting input records. As we saw before, this program,

```
count: table sum of int;  
emit count <- 1;
```

will achieve the job. By contrast, consider this erroneous program, which superficially looks like it should work:

```
count: int = 0;  
count++;
```

This program fails to count the records because, *for each record*, it sets `count` to zero, increments it, and throws the result away. Running on many machines in parallel, it will throw away all the information with great efficiency.

The Sawzall program is reset to the initial state at the beginning of processing for each record. Correspondingly, after processing a record and emitting all relevant data, any resources consumed during execution—variables, temporaries, etc.—can be discarded. Sawzall therefore uses an arena allocator [11], resetting the arena to the initial state after each record is processed.

Sometimes it is useful to do non-trivial initialization before beginning execution. For instance, one might want to create a large array or map to be queried when analyzing each record. To avoid doing such initialization for every record, Sawzall has a declaration keyword `static` that asserts that the variable is to be initialized once (per runtime instance) and considered part of the initial runtime state for processing each record. Here is a trivial example:

```
static CJK: map[string] of string = {  
    "zh": "Chinese",  
    "ja": "Japanese",  
    "ko": "Korean",  
};
```

The `CJK` variable will be created during initialization and its value stored permanently as part of the initial state for processing each record.

The language has no reference types; it has pure value semantics. Even arrays and maps behave as values. (The implementation uses copy-on-write using reference counting to make this efficient in most cases.) Although in general this can lead to some awkwardness—to modify an array in a function, the function must return the array—for the typical Sawzall program the issue doesn't arise. In return, there is the possibility to parallelize the processing within a given record without

the need for fine-grained synchronization or the fear of aliasing, an opportunity the implementation does not yet exploit.

## 11 Domain-specific Properties of the Language

Sawzall has a number of properties selected specifically for the problem domain in which it operates. Some have already been discussed; this section examines a few others.

First, atypical of most such “little languages” [2], Sawzall is statically typed. The main reason is dependability. Sawzall programs can consume hours, even months, of CPU time in a single run, and a late-arising dynamic type error can be expensive. There are other, less obvious reasons too. It helps the implementation of the aggregators to have their type fixed when they are created. A similar argument applies to the parsing of the input protocol buffers; it helps to know the expected input type precisely. Also, it is likely that overall performance of the interpreter is improved by avoiding dynamic type-checking at run time, although we have not attempted to verify this claim. Finally, compile-time type checking and the absence of automatic conversions require the programmer to be explicit about type conversions. The only exception is variable initialization, but in that case the type is still explicit and the program remains type-safe.

Static typing guarantees that the types of variables are always known, while permitting convenience at initialization time. It is helpful that initializations like

```
t : time = "Apr 1 12:00:00 PST 2005";
```

are easy to understand, yet also type-safe.

The set of basic types and their properties are also somewhat domain-dependent. The presence of time as a basic type is motivated by the handling of time stamps from log records; it is luxurious to have a language that handles Daylight Savings Time correctly. More important (but less unusual these days), the language defines a Unicode representation for strings, yet handles a variety of external character encodings.

Two novel features of the language grew out of the demands of analyzing large data sets: the handling of undefined values, and logical quantifiers. The next two sections describe these in detail.

### 11.1 Undefined values

Sawzall does not provide any form of exception processing. Instead, it has the notion of *undefined values*, a representation of erroneous or indeterminate results including division by zero, conversion errors, I/O problems, and so on. If the program attempts to read an undefined value outside of an initialization, it will crash and provide a report of the failure.

A predicate, `def()`, can be used to test if a value is defined; it returns `true` for a defined value and `false` for an undefined value. The idiom for its use is the following:

```
v: Value = maybe_undefined();
if (def(v)) {
    calculation_using(v);
}
```

Here's an example that must handle undefined values. Let's extend the query-mapping program by adding a time axis to the data. The original program used the function `locationinfo()` to discover the location of an IP address using an external database. That program was unsafe because it would fail if the IP address could not be found in the database. In that case, `locationinfo()` would return an undefined value, but we can protect against that using a `def()` predicate.

Here is the expanded, robust, program:

```
proto "querylog.proto"

static RESOLUTION: int = 5;  # minutes; must be divisor of 60

log_record: QueryLogProto = input;

queries_per_degree: table sum[t: time][lat: int][lon: int] of int;

loc: Location = locationinfo(log_record.ip);
if (def(loc)) {
    t: time = log_record.time_usecs;
    m: int = minuteof(t);  # within the hour
    m = m - m % RESOLUTION;
    t = truncuhour(t) + time(m * int(MINUTE));
    emit queries_per_degree[t][int(loc.lat)][int(loc.lon)] <- 1;
}
```

(Notice that we just disregard the record if it has no known location; this seems a fine way to handle it.) The calculation in the `if` statement uses some built-in functions (and the built-in constant `MINUTE`) to truncate the microsecond-resolution time stamp of the record to the boundary of a 5-minute time bucket.

Given query log records spanning an extended period, this program will extract data suitable for constructing a movie showing how activity changes over time.<sup>2</sup>

A careful programmer will use `def()` to guard against the usual errors that can arise, but sometimes the error syndrome can be so bizarre that a programmer would never think of it. If a program is processing a terabyte of data, there may be some records that contain surprises; often the quality of the data set may be outside the control of the person doing the analysis, or contain occasional

---

<sup>2</sup>The movie can be seen at <http://labs.google.com/papers/sawzall.html>.

unusual values beyond the scope of the current analysis. Particularly for the sort of processing for which Sawzall is used, it will often be perfectly safe just to disregard such erroneous values.

Sawzall therefore provides a mode, set by a run-time flag, that changes the default behavior of undefined values. Normally, using an undefined value (other than in an initialization or `def()` test) will terminate the program with an error report. When the run-time flag is set, however, Sawzall simply elides all statements that depend on the undefined value. To be precise, if a statement's computation references the undefined value in any way, the statement is skipped and execution is resumed after that statement. Simple statements are just dropped. An `if` statement with an undefined condition will be dropped completely. If a loop's condition becomes undefined while the loop is active, the loop is terminated but the results of previous iterations are unaffected.

For the corrupted record, it's as though the elements of the calculation that depended on the bad value were temporarily removed from the program. Any time such an elision occurs, the run-time will record the circumstances in a special pre-defined `collection` table that serves as a log of these errors. When the run completes the user may check whether the error rate was low enough to accept the results that remain.

One way to use the flag is to debug with it off—otherwise bugs will vanish—but when it comes time to run in production on a large data set, turn the flag on so that the program won't be broken by crazy data.

This is an unusual way to treat errors, but it is very convenient in practice. The idea is related to some independent work by Rinard *et al.* [15] in which the `gcc` C compiler was modified to generate code that protected against certain errors. For example, if a program indexes off the end of an array, the generated code will make up values and the program can continue obliviously. This peculiar behavior has the empirical property of making programs like web servers much more robust against failure, even in the face of malicious attacks. The handling of undefined values in Sawzall adds a similar level of robustness.

## 11.2 Quantifiers

Although Sawzall operates on individual records, those records sometimes contain arrays or other structures that must be analyzed as a unit to discover some property. Is there an array element with this value? Do all the values satisfy some condition? To make these ideas easy to express, Sawzall provides *logical quantifiers*, a special notation analogous to the “for each”, “for any”, and “for all” quantifiers of mathematics.

Within a special construct, called a `when` statement, one defines a quantifier, a variable, and a boolean condition using the variable. If the condition is satisfied, the associated statement is executed.

Quantifier variables are declared like regular variables, but the base type (usually `int`) is prefixed by a keyword specifying the form of quantifier. For example, given an array `a`, the statement

```
when (i: some int; B(a[i]))
    F(i);
```

executes `F(i)` if and only if, for *some* value of `i`, the boolean condition `B(a[i])` is true. When `F(i)` is invoked, `i` will be bound to the value that satisfies the condition.

There are three quantifier types: `some`, which executes the statement if the condition is true for any value (if more than one satisfies, an arbitrary choice is made); `each`, which executes the statement for all the values that satisfy the condition; and `all`, which executes the statement if the condition is true for all valid values of the quantifier (and does not bind the variable in the statement).

Sawzall analyzes the conditional expression to discover how to limit the range of evaluation of a `when` statement. It is a compile-time error if the condition does not provide enough information to construct a reasonable evaluator for the quantifier. In the example above, the quantifier variable `i` is used as an index in an array expression `a[i]`, which is sufficient to define the range. For more complex conditions, the system uses the `def()` operator internally to explore the bounds safely if necessary. (Indexing beyond the bounds of an array results in an undefined value.) Consider for example scanning a sparse multidimensional array `a[i][j]`. In such an expression, the two quantifier variables `i` and `j` must pairwise define a valid entry and the implementation can use `def` to explore the matrix safely.

When statements may contain multiple quantifier variables, which in general can introduce ambiguities of logic programming [6]. Sawzall defines that if multiple variables exist, they will be bound and evaluated in the order declared. This ordering, combined with the restricted set of quantifiers, is sufficient to eliminate the ambiguities.

Here are a couple of examples. The first tests whether two arrays share a common element:

```
when(i, j: some int; a1[i] == a2[j]) {
    ...
}
```

The second expands on this. Using array slicing, indicated by `:` notation in array indices, it tests whether two arrays share a common subarray of three or more elements:

```
when(i0, i1, j0, j1: some int; a[i0:i1] == b[j0:j1] &&
    i1 >= i0+3) {
    ...
}
```

It is convenient not to write all the code to handle the edge conditions in such a test. Even if the arrays are shorter than three elements, this statement will function correctly; the evaluation of `when` statements guarantees safe execution.

In principle, the evaluation of a `when` statement is parallelizable, but we have not explored this option yet.

## 12 Performance

Although Sawzall is interpreted, that is rarely the limiting factor in its performance. Most Sawzall jobs do very little processing per record and are therefore I/O bound; for most of the rest, the CPU spends the majority of its time in various run-time operations such as parsing protocol buffers.

Nevertheless, to compare the single-CPU speed of the Sawzall interpreter with that of other interpreted languages, we wrote a couple of microbenchmarks. The first computes pixel values for displaying the Mandelbrot set, to measure basic arithmetic and loop performance. The second measures function invocation using a recursive function to calculate the first 35 Fibonacci numbers. We ran the tests on a 2.8GHz x86 desktop machine. The results, shown in Table 1, demonstrate Sawzall is significantly faster than Python, Ruby, or Perl, at least for these microbenchmarks. (We are somewhat surprised by this; the other languages are older and we would expect them to have finely tuned implementations.) On the other hand, for these microbenchmarks Sawzall is about 1.6 times slower than interpreted Java, 21 times slower than compiled Java, and 51 times slower than compiled C++.<sup>3</sup>

	Sawzall	Python	Ruby	Perl
Mandelbrot run time	12.09s	45.42s	73.59s	38.68s
factor	1.00	3.75	6.09	3.20
Fibonacci run time	11.73s	38.12s	47.18s	75.73s
factor	1.00	3.24	4.02	6.46

Table 1: Microbenchmarks. The first is a Mandelbrot set calculation:  $500 \times 500$  pixel image with 500 iterations per pixel maximum. The second uses a recursive function to calculate the first 35 Fibonacci numbers.

The key performance metric for the system is not the single-machine speed but how the speed scales as we add machines when processing a large data set. We took a 450GB sample of compressed query log data and ran a Sawzall program over it to count the occurrences of certain words. The core of the program looked schematically like this:

```
result: table sum[key: string] [month: int] [day: int] of int;
static keywords: array of string =
{ "hitchhiker", "benedict", "vytorin",
  "itanium", "aardvark" };
```

---

<sup>3</sup>The Java implementation was Sun's client Java™ 1.3.1\_02 with HotSpot™; C++ was gcc 3.2.2

```

querywords: array of string = words_from_query();
month: int = month_of_query();
day: int = day_of_query();

when (i: each int; j: some int; querywords[i] == keywords[j])
    emit result[keywords[j]][month][day] <- 1;

```

We ran the test on sets of machines varying from 50 2.4GHz Xeon computers to 600. The timing numbers are graphed in Figure 5. At 600 machines, the aggregate throughput was 1.06GB/s of compressed data or about 3.2GB/s of raw input. If scaling were perfect, performance would be proportional to the number of machines, that is, adding one machine would contribute one machine's worth of throughput. In our test, the effect is to contribute 0.98 machines.

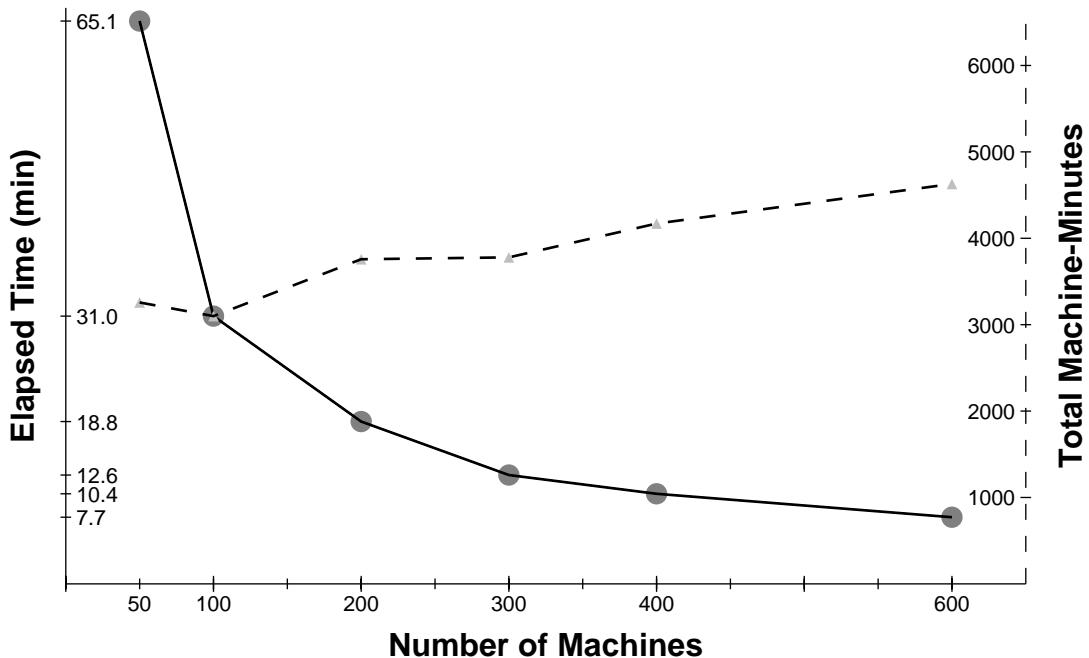


Figure 5: Performance scales well as we add machines. The solid line is elapsed time; the dashed line is the product of machines and elapsed time. Over the range of 50 to 600 machines, the machine-minutes product degrades only 30%.

## 13 Why a new language?

Why put a language above MapReduce? MapReduce is very effective; what's missing? And why a *new* language? Why not just attach an existing language such as Python to MapReduce?

The usual reasons for creating a special-purpose language apply here. Notation customized to a particular problem domain can make programs clearer, more compact, and more expressive. Capturing the aggregators in the language (and its environment) means that the programmer never has to provide one, unlike when using MapReduce. It also has led to some elegant programs as well as a comfortable way to think about data processing problems in large distributed data sets. Also, support for protocol buffers and other domain-specific types simplifies programming at a lower level. Overall, Sawzall programs tend to be around 10 to 20 times shorter than the equivalent MapReduce programs in C++ and significantly easier to write.

Other advantages of a custom language include the ability to add domain-specific features, custom debugging and profiling interfaces, and so on.

The original motivation, however, was completely different: parallelism. Separating out the aggregators and providing no other inter-record analysis maximizes the opportunity to distribute processing across records. It also provides a model for distributed processing, which in turn encourages users to think about the problem in a different light. In an existing language such as Awk [12] or Python [1], users would be tempted to write the aggregators in that language, which would be difficult to parallelize. Even if one provided a clean interface and library for aggregators in these languages, seasoned users would want to roll their own sometimes, which could introduce dramatic performance problems.

The model that Sawzall provides has proven valuable. Although some problems, such as database joins, are poor fits to the model, most of the processing we do on large data sets fits well and the benefit in notation, convenience, and expressiveness has made Sawzall a popular language at Google.

One unexpected benefit of the system arose from the constraints the programming model places on the user. Since the data flow from the input records to the Sawzall program is so well structured, it was easy to adapt it to provide fine-grained access control to individual fields within records. The system can automatically and securely wrap the user's program with a layer, itself implemented in Sawzall, that elides any sensitive fields. For instance, production engineers can be granted access to performance and monitoring information without exposing any traffic data. The details are outside the scope of this paper.

## 14 Utility

Although it has been deployed for only about 18 months, Sawzall has become one of the most widely used programming languages at Google. There are over two thousand Sawzall source files registered in our source code control system and a growing collection of libraries to aid the processing of various special-purpose data sets. Most Sawzall programs are of modest size, but the largest are several thousand lines long and generate dozens of multi-dimensional tables in a single run.

One measure of Sawzall’s utility is how much data processing it does. We monitored its use during the month of March 2005. During that time, on one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of  $3.2 \times 10^{15}$  bytes of data (2.8PB) and wrote  $9.9 \times 10^{12}$  bytes (9.3TB) (demonstrating that the term “data reduction” has some resonance). The average job therefore processed about 100GB. The jobs collectively consumed almost exactly one machine-century.

## 15 Related Work

Traditional data processing is done by storing the information in a relational database and processing it with SQL queries. Our system has many differences. First, the data sets are usually too large to fit in a relational database; files are processed *in situ* rather than being imported into a database server. Also, there are no pre-computed tables or indices; instead the purpose of the system is to construct *ad hoc* tables and indices appropriate to the computation. Although the aggregators are fixed, the presence of a procedural language to process the data enables complex, sophisticated analyses.

Sawzall is very different from SQL, combining a fairly traditional procedural language with an interface to efficient aggregators that process the results of the analysis applied to each record. SQL is excellent at database join operations, while Sawzall is not. On the other hand, Sawzall can be run on a thousand or more machines in parallel to process enormous data sets.

Brook [3] is another language for data processing, specifically graphical image processing. Although different in application area, like Sawzall it enables parallelism through a one-element-at-a-time computation model and associative reduction kernels.

A different approach to the processing of large data stores is to analyze them with a data stream model. Such systems process the data as it flows in, and their operators are dependent on the order of the input records. For example, Aurora [4] is a stream processing system that supports a (potentially large) set of standing queries on streams of data. Analogous to Sawzall’s predefinition

of its aggregators, Aurora provides a small, fixed set of operators, although two of them are escapes to user-defined functions. These operators can be composed to create more interesting queries. Unlike Sawzall, some Aurora operators work on a contiguous sequence, or window, of input values. Aurora only keeps a limited amount of data on hand, and is not designed for querying large archival stores. There is a facility to add new queries to the system, but they only operate on the recent past. Aurora’s efficiency comes from a carefully designed run-time system and a query optimizer, rather than Sawzall’s brute force parallel style.

Another stream processing system, Hancock [7], goes further and provides extensive support for storing per-query intermediate state. This is quite a contrast to Sawzall, which deliberately reverts to its initialization state after each input record. Like Aurora, Hancock concentrates on efficient operation of a single thread instead of massive parallelism.

## 16 Future Work

The throughput of the system is very high when used as intended, with thousands of machines in parallel. Because the language is modest in scope, the programs written in it tend to be small and are usually I/O bound. Therefore, although it is an interpreted language, the implementation is fast enough most of the time. Still, some of the larger or more complex analyses would be helped by more aggressive compilation, perhaps to native machine code. The compiler would run once per machine and then apply the accelerated binary to each input record. Exploratory work on such a compiler is now underway.

Occasionally a program needs to query external databases when processing a record. While support is already provided for a common set of small databases, such as the IP location information, our system could profit from an interface to query an external database. Since Sawzall processes each record independently, the system could suspend processing of one record when the program makes such a query, and continue when the query is completed. There are obvious opportunities for parallelism in this design.

Our analyses sometimes require multiple passes over the data, either multiple Sawzall passes or Sawzall passes followed by processing by another system, such as a traditional database or a program written in a general-purpose programming language. Since the language does not directly support such “chaining”, these multi-pass programs can be awkward to express in Sawzall. Language extensions are being developed that will make it easy to express the chaining of multiple passes, as are aggregator extensions allowing output to be delivered directly to external systems.

Some analyses join data from multiple input sources, often after a Sawzall preprocessing step or two. Joining is supported, but in general requires extra chaining steps. More direct support of join operations would simplify these programs.

A more radical system model would eliminate the batch-processing mode entirely. It would be convenient for tasks such as performance monitoring to have the input be fed continuously to the Sawzall program, with the aggregators keeping up with the data flow. The aggregators would be maintained in some on-line server that could be queried at any time to discover the current value of any table or table entry. This model is similar to the streaming database work [4][7] and was in fact the original thinking behind the system. However, before much implementation had been completed, the MapReduce library was created by Dean and Ghemawat and it proved so effective that the continuous system was never built. We hope to return to it one day.

## 17 Conclusions

When the problem is large enough, a new approach may be necessary. To make effective use of large computing clusters in the analysis of large data sets, it is helpful to restrict the programming model to guarantee high parallelism. The trick is to do so without unduly limiting the expressiveness of the model.

Our approach includes a new programming language called Sawzall. The language helps capture the programming model by forcing the programmer to think one record at a time, while providing an expressive interface to a novel set of aggregators that capture many common data processing and data reduction problems. In return for learning a new language, the user is rewarded by the ability to write short, clear programs that are guaranteed to work well on thousands of machines in parallel. Ironically—but vitally—the user needs to know nothing about parallel programming; the language and the underlying system take care of all the details.

It may seem paradoxical to use an interpreted language in a high-throughput environment, but we have found that the CPU time is rarely the limiting factor; the expressibility of the language means that most programs are small and spend most of their time in I/O and native run-time code. Moreover, the flexibility of an interpreted implementation has been helpful, both in ease of experimentation at the linguistic level and in allowing us to explore ways to distribute the calculation across many machines.

Perhaps the ultimate test of our system is scalability. We find linear growth in performance as we add machines, with a slope near unity. Big data sets need lots of machines; it's gratifying that lots of machines can translate into big throughput.

## 18 Acknowledgements

Geeta Chaudhry wrote the first significant Sawzall programs and gave invaluable feedback. Amit Patel, Paul Haahr and Greg Rae were enthusiastic and helpful early users. Paul Haahr created the

Page Rank example. Dick Sites and Renée French helped with the figures. We received helpful comments about the paper from Dan Bentley, Dave Hanson, Patrick Jordan, John Lamping, Dick Sites, Tom Szymanski, Deborah A. Wallach, and Lawrence You.

## References

- [1] David M. Beazley, Python Essential Reference, New Riders, Indianapolis, 2000.
- [2] Jon Bentley, Programming Pearls, CACM August 1986 v 29 n 8 pp. 711-721.
- [3] Ian Buck et al., Brook for GPUs: Stream Computing on Graphics Hardware, Proc. SIGGRAPH, Los Angeles, 2004.
- [4] Don Carney et al., Monitoring Streams – A New Class of Data Management Applications, Brown Computer Science Technical Report TR-CS-02-04. At [http://www.cs.brown.edu/research/aurora/aurora\\_tr.pdf](http://www.cs.brown.edu/research/aurora/aurora_tr.pdf).
- [5] M. Charikar, K. Chen, and M. Farach-Colton, Finding frequent items in data streams, Proc 29th Intl. Colloq. on Automata, Languages and Programming, 2002.
- [6] W. F. Clocksin and C. S. Mellish, Programming in Prolog, Springer, 1994.
- [7] Cortes et al., Hancock: A Language for Extracting Signatures from Data Streams, Proc. Sixth International Conference on Knowledge Discovery and Data Mining, Boston, 2000, pp. 9-17.
- [8] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proc 6th Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pages 137-149.
- [9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, Proc. 19th Symposium on Operating System Principles, Lake George, New York, 2003, pp. 29-43.
- [10] M. Greenwald and S. Khanna, Space-efficient online computation of quantile summaries, Proc. SIGMOD, Santa Barbara, CA, May 2001, pp. 58-66.
- [11] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes. Software–Practice and Experience, 20(1):512, January 1990.
- [12] Brian Kernighan, Peter Weinberger, and Alfred Aho, The AWK Programming Language, Addison-Wesley, Massachusetts, 1988.
- [13] Brian Kernighan, personal communication.
- [14] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, The pagerank citation algorithm: bringing order to the web, Proc. of the Seventh conference on the World Wide Web, Brisbane, Australia, April 1998.

- [15] Martin Rinard et al., Enhancing Server Reliability Through Failure-Oblivious Computing, Proc. Sixth Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pp. 303-316.
- [16] Douglas Thain, Todd Tannenbaum, and Miron Livny, Distributed computing in practice: The Condor experience, Concurrency and Computation: Practice and Experience, 2004.

# Spanner: Google’s Globally-Distributed Database

*James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman,  
Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh,  
Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura,  
David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak,  
Christopher Taylor, Ruth Wang, Dale Woodford*

*Google, Inc.*

## Abstract

Spanner is Google’s scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

## 1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google’s advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower la-

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner’s main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

provides externally consistent [16] reads and writes, and globally-consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent MapReduce executions [12], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

These features are enabled by the fact that Spanner assigns globally-meaningful commit timestamps to transactions, even though transactions may be distributed. The timestamps reflect serialization order. In addition, the serialization order satisfies external consistency (or equivalently, linearizability [20]): if a transaction  $T_1$  commits before another transaction  $T_2$  starts, then  $T_1$ 's commit timestamp is smaller than  $T_2$ 's. Spanner is the first system to provide such guarantees at global scale.

The key enabler of these properties is a new TrueTime API and its implementation. The API directly exposes clock uncertainty, and the guarantees on Spanner's timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty. Google's cluster-management software provides an implementation of the TrueTime API. This implementation keeps uncertainty small (generally less than 10ms) by using multiple modern clock references (GPS and atomic clocks).

Section 2 describes the structure of Spanner's implementation, its feature set, and the engineering decisions that went into their design. Section 3 describes our new TrueTime API and sketches its implementation. Section 4 describes how Spanner uses TrueTime to implement externally-consistent distributed transactions, lock-free read-only transactions, and atomic schema updates. Section 5 provides some benchmarks on Spanner's performance and TrueTime behavior, and discusses the experiences of F1. Sections 6, 7, and 8 describe related and future work, and summarize our conclusions.

## 2 Implementation

This section describes the structure of and rationale underlying Spanner's implementation. It then describes the *directory* abstraction, which is used to manage replication and locality, and is the unit of data movement. Finally, it describes our data model, why Spanner looks like a relational database instead of a key-value store, and how applications can control data locality.

A Spanner deployment is called a *universe*. Given that Spanner manages data globally, there will be only a handful of running universes. We currently run a test/playground universe, a development/production universe, and a production-only universe.

Spanner is organized as a set of *zones*, where each zone is the rough analog of a deployment of Bigtable

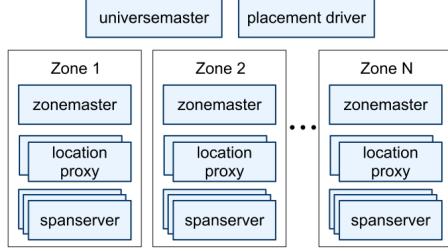


Figure 1: Spanner server organization.

servers [9]. Zones are the unit of administrative deployment. The set of zones is also the set of locations across which data can be replicated. Zones can be added to or removed from a running system as new datacenters are brought into service and old ones are turned off, respectively. Zones are also the unit of physical isolation: there may be one or more zones in a datacenter, for example, if different applications' data must be partitioned across different sets of servers in the same datacenter.

Figure 1 illustrates the servers in a Spanner universe. A zone has one *zonemaster* and between one hundred and several thousand *spanservers*. The former assigns data to spanservers; the latter serve data to clients. The per-zone *location proxies* are used by clients to locate the spanservers assigned to serve their data. The *universe master* and the *placement driver* are currently singlettons. The universe master is primarily a console that displays status information about all the zones for interactive debugging. The placement driver handles automated movement of data across zones on the timescale of minutes. The placement driver periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load. For space reasons, we will only describe the spanserver in any detail.

### 2.1 Spanserver Software Stack

This section focuses on the spanserver implementation to illustrate how replication and distributed transactions have been layered onto our Bigtable-based implementation. The software stack is shown in Figure 2. At the bottom, each spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*. A tablet is similar to Bigtable's tablet abstraction, in that it implements a bag of the following mappings:

$$(\text{key:string}, \text{timestamp:int64}) \rightarrow \text{string}$$

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multi-version database than a key-value store. A

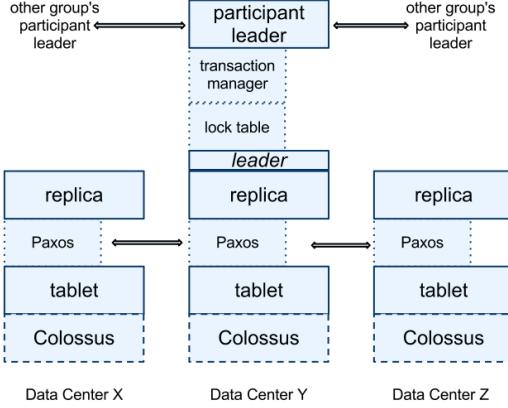


Figure 2: Spanserver software stack.

tablet's state is stored in set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System [15]).

To support replication, each spanserver implements a single Paxos state machine on top of each tablet. (An early Spanner incarnation supported multiple Paxos state machines per tablet, which allowed for more flexible replication configurations. The complexity of that design led us to abandon it.) Each state machine stores its metadata and log in its corresponding tablet. Our Paxos implementation supports long-lived leaders with time-based leader leases, whose length defaults to 10 seconds. The current Spanner implementation logs every Paxos write twice: once in the tablet's log, and once in the Paxos log. This choice was made out of expediency, and we are likely to remedy this eventually. Our implementation of Paxos is pipelined, so as to improve Spanner's throughput in the presence of WAN latencies; but writes are applied by Paxos in order (a fact on which we will depend in Section 4).

The Paxos state machines are used to implement a consistently replicated bag of mappings. The key-value mapping state of each replica is stored in its corresponding tablet. Writes must initiate the Paxos protocol at the leader; reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date. The set of replicas is collectively a *Paxos group*.

At every replica that is a leader, each spanserver implements a *lock table* to implement concurrency control. The lock table contains the state for two-phase locking: it maps ranges of keys to lock states. (Note that having a long-lived Paxos leader is critical to efficiently managing the lock table.) In both Bigtable and Spanner, we designed for long-lived transactions (for example, for report generation, which might take on the order of minutes), which perform poorly under optimistic concurrency control in the presence of conflicts. Operations

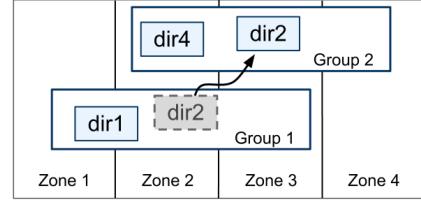


Figure 3: Directories are the unit of data movement between Paxos groups.

that require synchronization, such as transactional reads, acquire locks in the lock table; other operations bypass the lock table.

At every replica that is a leader, each spanserver also implements a *transaction manager* to support distributed transactions. The transaction manager is used to implement a *participant leader*; the other replicas in the group will be referred to as *participant slaves*. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the *coordinator leader*, and the slaves of that group as *coordinator slaves*. The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).

## 2.2 Directories and Placement

On top of the bag of key-value mappings, the Spanner implementation supports a bucketing abstraction called a *directory*, which is a set of contiguous keys that share a common prefix. (The choice of the term *directory* is a historical accident; a better term might be *bucket*.) We will explain the source of that prefix in Section 2.3. Supporting directories allows applications to control the locality of their data by choosing keys carefully.

A directory is the unit of data placement. All data in a directory has the same replication configuration. When data is moved between Paxos groups, it is moved directory by directory, as shown in Figure 3. Spanner might move a directory to shed load from a Paxos group; to put directories that are frequently accessed together into the same group; or to move a directory into a group that is closer to its accessors. Directories can be moved while client operations are ongoing. One could expect that a 50MB directory can be moved in a few seconds.

The fact that a Paxos group may contain multiple directories implies that a Spanner tablet is different from

a Bigtable tablet: the former is not necessarily a single lexicographically contiguous partition of the row space. Instead, a Spanner tablet is a container that may encapsulate multiple partitions of the row space. We made this decision so that it would be possible to colocate multiple directories that are frequently accessed together.

*Movedir* is the background task used to move directories between Paxos groups [14]. Movedir is also used to add or remove replicas to Paxos groups [25], because Spanner does not yet support in-Paxos configuration changes. Movedir is not implemented as a single transaction, so as to avoid blocking ongoing reads and writes on a bulky data move. Instead, movedir registers the fact that it is starting to move data and moves the data in the background. When it has moved all but a nominal amount of the data, it uses a transaction to atomically move that nominal amount and update the metadata for the two Paxos groups.

A directory is also the smallest unit whose geographic-replication properties (or *placement*, for short) can be specified by an application. The design of our placement-specification language separates responsibilities for managing replication configurations. Administrators control two dimensions: the number and types of replicas, and the geographic placement of those replicas. They create a menu of named options in these two dimensions (e.g., *North America, replicated 5 ways with 1 witness*). An application controls how data is replicated, by tagging each database and/or individual directories with a combination of those options. For example, an application might store each end-user’s data in its own directory, which would enable user *A*’s data to have three replicas in Europe, and user *B*’s data to have five replicas in North America.

For expository clarity we have over-simplified. In fact, Spanner will shard a directory into multiple *fragments* if it grows too large. Fragments may be served from different Paxos groups (and therefore different servers). Movedir actually moves fragments, and not whole directories, between groups.

## 2.3 Data Model

Spanner exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general-purpose transactions. The move towards supporting these features was driven by many factors. The need to support schematized semi-relational tables and synchronous replication is supported by the popularity of Megastore [5]. At least 300 applications within Google use Megastore (despite its relatively low performance) because its data model is simpler to man-

age than Bigtable’s, and because of its support for synchronous replication across datacenters. (Bigtable only supports eventually-consistent replication across datacenters.) Examples of well-known Google applications that use Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine. The need to support a SQL-like query language in Spanner was also clear, given the popularity of Dremel [28] as an interactive data-analysis tool. Finally, the lack of cross-row transactions in Bigtable led to frequent complaints; Percolator [32] was in part built to address this failing. Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings [9, 10, 19]. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.

The application data model is layered on top of the directory-bucketed key-value mappings supported by the implementation. An application creates one or more *databases* in a universe. Each database can contain an unlimited number of schematized *tables*. Tables look like relational-database tables, with rows, columns, and versioned values. We will not go into detail about the query language for Spanner. It looks like SQL with some extensions to support protocol-buffer-valued fields.

Spanner’s data model is not purely relational, in that rows must have names. More precisely, every table is required to have an ordered set of one or more primary-key columns. This requirement is where Spanner still looks like a key-value store: the primary keys form the name for a row, and each table defines a mapping from the primary-key columns to the non-primary-key columns. A row has existence only if some value (even if it is NULL) is defined for the row’s keys. Imposing this structure is useful because it lets applications control data locality through their choices of keys.

Figure 4 contains an example Spanner schema for storing photo metadata on a per-user, per-album basis. The schema language is similar to Megastore’s, with the additional requirement that every Spanner database must be partitioned by clients into one or more hierarchies of tables. Client applications declare the hierarchies in database schemas via the `INTERLEAVE IN` declarations. The table at the top of a hierarchy is a *directory table*. Each row in a directory table with key *K*, together with all of the rows in descendant tables that start with *K* in lexicographic order, forms a directory. `ON DELETE CASCADE` says that deleting a row in the directory table deletes any associated child rows. The figure also illustrates the interleaved layout for the example database: for

```

CREATE TABLE Users {
    uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
    uid INT64 NOT NULL, aid INT64 NOT NULL,
    name STRING
} PRIMARY KEY (uid, aid),
INTERLEAVE IN PARENT Users ON DELETE CASCADE;

.....


|             |
|-------------|
| Users(1)    |
| Albums(1,1) |
| Albums(1,2) |
| Users(2)    |
| Albums(2,1) |
| Albums(2,2) |
| Albums(2,3) |

 .....
..... Directory 3665
..... Directory 453
.....
```

Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

example, `Albums(2, 1)` represents the row from the `Albums` table for `user_id` 2, `album_id` 1. This interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables, which is necessary for good performance in a sharded, distributed database. Without it, Spanner would not know the most important locality relationships.

### 3 TrueTime

Method	Returns
<code>TT.now()</code>	<code>TTinterval: [earliest, latest]</code>
<code>TT.after(t)</code>	true if $t$ has definitely passed
<code>TT.before(t)</code>	true if $t$ has definitely not arrived

Table 1: TrueTime API. The argument  $t$  is of type `TTstamp`.

This section describes the TrueTime API and sketches its implementation. We leave most of the details for another paper: our goal is to demonstrate the power of having such an API. Table 1 lists the methods of the API. TrueTime explicitly represents time as a `TTinterval`, which is an interval with bounded time uncertainty (unlike standard time interfaces that give clients no notion of uncertainty). The endpoints of a `TTinterval` are of type `TTstamp`. The `TT.now()` method returns a `TTinterval` that is guaranteed to contain the absolute time during which `TT.now()` was invoked. The time epoch is analogous to UNIX time with leap-second smearing. Define the instantaneous error bound as  $\epsilon$ , which is half of the interval's width, and the average error bound as  $\bar{\epsilon}$ . The `TT.after()` and `TT.before()` methods are convenience wrappers around `TT.now()`.

Denote the absolute time of an event  $e$  by the function  $t_{abs}(e)$ . In more formal terms, TrueTime guarantees that for an invocation  $tt = TT.now()$ ,  $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$ , where  $e_{now}$  is the invocation event.

The underlying time references used by TrueTime are GPS and atomic clocks. TrueTime uses two forms of time reference because they have different failure modes. GPS reference-source vulnerabilities include antenna and receiver failures, local radio interference, correlated failures (e.g., design faults such as incorrect leap-second handling and spoofing), and GPS system outages. Atomic clocks can fail in ways uncorrelated to GPS and each other, and over long periods of time can drift significantly due to frequency error.

TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine. The majority of masters have GPS receivers with dedicated antennas; these masters are separated physically to reduce the effects of antenna failures, radio interference, and spoofing. The remaining masters (which we refer to as *Armageddon masters*) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master. All masters' time references are regularly compared against each other. Each master also cross-checks the rate at which its reference advances time against its own local clock, and evicts itself if there is substantial divergence. Between synchronizations, Armageddon masters advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift. GPS masters advertise uncertainty that is typically close to zero.

Every daemon polls a variety of masters [29] to reduce vulnerability to errors from any one master. Some are GPS masters chosen from nearby datacenters; the rest are GPS masters from farther datacenters, as well as some Armageddon masters. Daemons apply a variant of Marzullo's algorithm [27] to detect and reject liars, and synchronize the local machine clocks to the non-liars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted.

Between synchronizations, a daemon advertises a slowly increasing time uncertainty.  $\epsilon$  is derived from conservatively applied worst-case local clock drift.  $\epsilon$  also depends on time-master uncertainty and communication delay to the time masters. In our production environment,  $\epsilon$  is typically a sawtooth function of time, varying from about 1 to 7 ms over each poll interval.  $\bar{\epsilon}$  is therefore 4 ms most of the time. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at 200 microseconds/second, which together account

Operation	Timestamp Discussion	Concurrency Control	Replica Required
Read-Write Transaction	§ 4.1.2	pessimistic	leader
Read-Only Transaction	§ 4.1.4	lock-free	leader for timestamp; any for read, subject to § 4.1.3
Snapshot Read, client-provided timestamp	—	lock-free	any, subject to § 4.1.3
Snapshot Read, client-provided bound	§ 4.1.3	lock-free	any, subject to § 4.1.3

Table 2: Types of reads and writes in Spanner, and how they compare.

for the sawtooth bounds from 0 to 6 ms. The remaining 1 ms comes from the communication delay to the time masters. Excursions from this sawtooth are possible in the presence of failures. For example, occasional time-master unavailability can cause datacenter-wide increases in  $\epsilon$ . Similarly, overloaded machines and network links can result in occasional localized  $\epsilon$  spikes.

## 4 Concurrency Control

This section describes how TrueTime is used to guarantee the correctness properties around concurrency control, and how those properties are used to implement features such as externally consistent transactions, lock-free read-only transactions, and non-blocking reads in the past. These features enable, for example, the guarantee that a whole-database audit read at a timestamp  $t$  will see exactly the effects of every transaction that has committed as of  $t$ .

Going forward, it will be important to distinguish writes as seen by Paxos (which we will refer to as *Paxos writes* unless the context is clear) from Spanner client writes. For example, two-phase commit generates a Paxos write for the prepare phase that has no corresponding Spanner client write.

### 4.1 Timestamp Management

Table 2 lists the types of operations that Spanner supports. The Spanner implementation supports *read-write transactions*, *read-only transactions* (predeclared snapshot-isolation transactions), and *snapshot reads*. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as read-only transactions. Both are internally retried (clients need not write their own retry loops).

A read-only transaction is a kind of transaction that has the performance benefits of snapshot isolation [6]. A read-only transaction must be predeclared as not having any writes; it is not simply a read-write transaction without any writes. Reads in a read-only transaction execute at a system-chosen timestamp without locking, so that incoming writes are not blocked. The execution of

the reads in a read-only transaction can proceed on any replica that is sufficiently up-to-date (Section 4.1.3).

A snapshot read is a read in the past that executes without locking. A client can either specify a timestamp for a snapshot read, or provide an upper bound on the desired timestamp's staleness and let Spanner choose a timestamp. In either case, the execution of a snapshot read proceeds at any replica that is sufficiently up-to-date.

For both read-only transactions and snapshot reads, commit is inevitable once a timestamp has been chosen, unless the data at that timestamp has been garbage-collected. As a result, clients can avoid buffering results inside a retry loop. When a server fails, clients can internally continue the query on a different server by repeating the timestamp and the current read position.

#### 4.1.1 Paxos Leader Leases

Spanner's Paxos implementation uses timed leases to make leadership long-lived (10 seconds by default). A potential leader sends requests for timed *lease votes*; upon receiving a quorum of lease votes the leader knows it has a lease. A replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration. Define a leader's *lease interval* as starting when it discovers it has a quorum of lease votes, and as ending when it no longer has a quorum of lease votes (because some have expired). Spanner depends on the following disjointness invariant: for each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's. Appendix A describes how this invariant is enforced.

The Spanner implementation permits a Paxos leader to abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner constrains when abdication is permissible. Define  $s_{max}$  to be the maximum timestamp used by a leader. Subsequent sections will describe when  $s_{max}$  is advanced. Before abdicating, a leader must wait until  $TT.after(s_{max})$  is true.

#### 4.1.2 Assigning Timestamps to RW Transactions

Transactional reads and writes use two-phase locking. As a result, they can be assigned timestamps at any time

when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit.

Spanner depends on the following monotonicity invariant: within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders. A single leader replica can trivially assign timestamps in monotonically increasing order. This invariant is enforced across leaders by making use of the disjointness invariant: a leader must only assign timestamps within the interval of its leader lease. Note that whenever a timestamp  $s$  is assigned,  $s_{max}$  is advanced to  $s$  to preserve disjointness.

Spanner also enforces the following external-consistency invariant: if the start of a transaction  $T_2$  occurs after the commit of a transaction  $T_1$ , then the commit timestamp of  $T_2$  must be greater than the commit timestamp of  $T_1$ . Define the start and commit events for a transaction  $T_i$  by  $e_i^{start}$  and  $e_i^{commit}$ ; and the commit timestamp of a transaction  $T_i$  by  $s_i$ . The invariant becomes  $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$ . The protocol for executing transactions and assigning timestamps obeys two rules, which together guarantee this invariant, as shown below. Define the arrival event of the commit request at the coordinator leader for a write  $T_i$  to be  $e_i^{server}$ .

**Start** The coordinator leader for a write  $T_i$  assigns a commit timestamp  $s_i$  no less than the value of  $TT.now().latest$ , computed after  $e_i^{server}$ . Note that the participant leaders do not matter here; Section 4.2.1 describes how they are involved in the implementation of the next rule.

**Commit Wait** The coordinator leader ensures that clients cannot see any data committed by  $T_i$  until  $TT.after(s_i)$  is true. Commit wait ensures that  $s_i$  is less than the absolute commit time of  $T_i$ , or  $s_i < t_{abs}(e_i^{commit})$ . The implementation of commit wait is described in Section 4.2.1. Proof:

$$\begin{array}{ll} s_1 < t_{abs}(e_1^{commit}) & \text{(commit wait)} \\ t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) & \text{(assumption)} \\ t_{abs}(e_2^{start}) \leq t_{abs}(e_2^{server}) & \text{(causality)} \\ t_{abs}(e_2^{server}) \leq s_2 & \text{(start)} \\ s_1 < s_2 & \text{(transitivity)} \end{array}$$

### 4.1.3 Serving Reads at a Timestamp

The monotonicity invariant described in Section 4.1.2 allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read. Every replica tracks a value called *safe time*  $t_{safe}$  which is the

maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp  $t$  if  $t \leq t_{safe}$ .

Define  $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$ , where each Paxos state machine has a safe time  $t_{safe}^{Paxos}$  and each transaction manager has a safe time  $t_{safe}^{TM}$ .  $t_{safe}^{Paxos}$  is simpler: it is the timestamp of the highest-applied Paxos write. Because timestamps increase monotonically and writes are applied in order, writes will no longer occur at or below  $t_{safe}^{Paxos}$  with respect to Paxos.

$t_{safe}^{TM}$  is  $\infty$  at a replica if there are zero prepared (but not committed) transactions—that is, transactions in between the two phases of two-phase commit. (For a participant slave,  $t_{safe}^{TM}$  actually refers to the replica's leader's transaction manager, whose state the slave can infer through metadata passed on Paxos writes.) If there are any such transactions, then the state affected by those transactions is indeterminate: a participant replica does not know yet whether such transactions will commit. As we discuss in Section 4.2.1, the commit protocol ensures that every participant knows a lower bound on a prepared transaction's timestamp. Every participant leader (for a group  $g$ ) for a transaction  $T_i$  assigns a prepare timestamp  $s_{i,g}^{prepare}$  to its prepare record. The coordinator leader ensures that the transaction's commit timestamp  $s_i \geq s_{i,g}^{prepare}$  over all participant groups  $g$ . Therefore, for every replica in a group  $g$ , over all transactions  $T_i$  prepared at  $g$ ,  $t_{safe}^{TM} = \min_i(s_{i,g}^{prepare}) - 1$  over all transactions prepared at  $g$ .

### 4.1.4 Assigning Timestamps to RO Transactions

A read-only transaction executes in two phases: assign a timestamp  $s_{read}$  [8], and then execute the transaction's reads as snapshot reads at  $s_{read}$ . The snapshot reads can execute at any replicas that are sufficiently up-to-date.

The simple assignment of  $s_{read} = TT.now().latest$ , at any time after a transaction starts, preserves external consistency by an argument analogous to that presented for writes in Section 4.1.2. However, such a timestamp may require the execution of the data reads at  $s_{read}$  to block if  $t_{safe}$  has not advanced sufficiently. (In addition, note that choosing a value of  $s_{read}$  may also advance  $s_{max}$  to preserve disjointness.) To reduce the chances of blocking, Spanner should assign the oldest timestamp that preserves external consistency. Section 4.2.2 explains how such a timestamp can be chosen.

## 4.2 Details

This section explains some of the practical details of read-write transactions and read-only transactions elided earlier, as well as the implementation of a special transaction type used to implement atomic schema changes.

It then describes some refinements of the basic schemes as described.

### 4.2.1 Read-Write Transactions

Like Bigtable, writes that occur in a transaction are buffered at the client until commit. As a result, reads in a transaction do not see the effects of the transaction's writes. This design works well in Spanner because a read returns the timestamps of any data read, and uncommitted writes have not yet been assigned timestamps.

Reads within read-write transactions use wound-wait [33] to avoid deadlocks. The client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data. While a client transaction remains open, it sends keepalive messages to prevent participant leaders from timing out its transaction. When a client has completed all reads and buffered all writes, it begins two-phase commit. The client chooses a coordinator group and sends a commit message to each participant's leader with the identity of the coordinator and any buffered writes. Having the client drive two-phase commit avoids sending data twice across wide-area links.

A non-coordinator-participant leader first acquires write locks. It then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions (to preserve monotonicity), and logs a prepare record through Paxos. Each participant then notifies the coordinator of its prepare timestamp.

The coordinator leader also first acquires write locks, but skips the prepare phase. It chooses a timestamp for the entire transaction after hearing from all other participant leaders. The commit timestamp  $s$  must be greater or equal to all prepare timestamps (to satisfy the constraints discussed in Section 4.1.3), greater than  $TT.now().latest$  at the time the coordinator received its commit message, and greater than any timestamps the leader has assigned to previous transactions (again, to preserve monotonicity). The coordinator leader then logs a commit record through Paxos (or an abort if it timed out while waiting on the other participants).

Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until  $TT.after(s)$ , so as to obey the commit-wait rule described in Section 4.1.2. Because the coordinator leader chose  $s$  based on  $TT.now().latest$ , and now waits until that timestamp is guaranteed to be in the past, the expected wait is at least  $2 * \bar{\epsilon}$ . This wait is typically overlapped with Paxos communication. After commit wait, the coordinator sends the commit timestamp to the client and all other participant leaders. Each participant leader logs the transaction's outcome through Paxos. All participants apply at the same timestamp and then release locks.

### 4.2.2 Read-Only Transactions

Assigning a timestamp requires a negotiation phase between all of the Paxos groups that are involved in the reads. As a result, Spanner requires a *scope* expression for every read-only transaction, which is an expression that summarizes the keys that will be read by the entire transaction. Spanner automatically infers the scope for standalone queries.

If the scope's values are served by a single Paxos group, then the client issues the read-only transaction to that group's leader. (The current Spanner implementation only chooses a timestamp for a read-only transaction at a Paxos leader.) That leader assigns  $s_{read}$  and executes the read. For a single-site read, Spanner generally does better than  $TT.now().latest$ . Define  $LastTS()$  to be the timestamp of the last committed write at a Paxos group. If there are no prepared transactions, the assignment  $s_{read} = LastTS()$  trivially satisfies external consistency: the transaction will see the result of the last write, and therefore be ordered after it.

If the scope's values are served by multiple Paxos groups, there are several options. The most complicated option is to do a round of communication with all of the groups's leaders to negotiate  $s_{read}$  based on  $LastTS()$ . Spanner currently implements a simpler choice. The client avoids a negotiation round, and just has its reads execute at  $s_{read} = TT.now().latest$  (which may wait for safe time to advance). All reads in the transaction can be sent to replicas that are sufficiently up-to-date.

### 4.2.3 Schema-Change Transactions

TrueTime enables Spanner to support atomic schema changes. It would be infeasible to use a standard transaction, because the number of participants (the number of groups in a database) could be in the millions. Bigtable supports atomic schema changes in one datacenter, but its schema changes block all operations.

A Spanner schema-change transaction is a generally non-blocking variant of a standard transaction. First, it is explicitly assigned a timestamp in the future, which is registered in the prepare phase. As a result, schema changes across thousands of servers can complete with minimal disruption to other concurrent activity. Second, reads and writes, which implicitly depend on the schema, synchronize with any registered schema-change timestamp at time  $t$ : they may proceed if their timestamps precede  $t$ , but they must block behind the schema-change transaction if their timestamps are after  $t$ . Without TrueTime, defining the schema change to happen at  $t$  would be meaningless.

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

#### 4.2.4 Refinements

$t_{safe}^{TM}$  as defined above has a weakness, in that a single prepared transaction prevents  $t_{safe}$  from advancing. As a result, no reads can occur at later timestamps, even if the reads do not conflict with the transaction. Such false conflicts can be removed by augmenting  $t_{safe}^{TM}$  with a fine-grained mapping from key ranges to prepared-transaction timestamps. This information can be stored in the lock table, which already maps key ranges to lock metadata. When a read arrives, it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts.

$LastTS()$  as defined above has a similar weakness: if a transaction has just committed, a non-conflicting read-only transaction must still be assigned  $s_{read}$  so as to follow that transaction. As a result, the execution of the read could be delayed. This weakness can be remedied similarly by augmenting  $LastTS()$  with a fine-grained mapping from key ranges to commit timestamps in the lock table. (We have not yet implemented this optimization.) When a read-only transaction arrives, its timestamp can be assigned by taking the maximum value of  $LastTS()$  for the key ranges with which the transaction conflicts, unless there is a conflicting prepared transaction (which can be determined from fine-grained safe time).

$t_{safe}^{Paxos}$  as defined above has a weakness in that it cannot advance in the absence of Paxos writes. That is, a snapshot read at  $t$  cannot execute at Paxos groups whose last write happened before  $t$ . Spanner addresses this problem by taking advantage of the disjointness of leader-lease intervals. Each Paxos leader advances  $t_{safe}^{Paxos}$  by keeping a threshold above which future writes' timestamps will occur: it maintains a mapping  $MinNextTS(n)$  from Paxos sequence number  $n$  to the minimum timestamp that may be assigned to Paxos sequence number  $n + 1$ . A replica can advance  $t_{safe}^{Paxos}$  to  $MinNextTS(n) - 1$  when it has applied through  $n$ .

A single leader can enforce its  $MinNextTS()$  promises easily. Because the timestamps promised by  $MinNextTS()$  lie within a leader's lease, the disjointness invariant enforces  $MinNextTS()$  promises across leaders. If a leader wishes to advance  $MinNextTS()$  beyond the end of its leader lease, it must first extend its

lease. Note that  $s_{max}$  is always advanced to the highest value in  $MinNextTS()$  to preserve disjointness.

A leader by default advances  $MinNextTS()$  values every 8 seconds. Thus, in the absence of prepared transactions, healthy slaves in an idle Paxos group can serve reads at timestamps greater than 8 seconds old in the worst case. A leader may also advance  $MinNextTS()$  values on demand from slaves.

## 5 Evaluation

We first measure Spanner's performance with respect to replication, transactions, and availability. We then provide some data on TrueTime behavior, and a case study of our first client, F1.

### 5.1 Microbenchmarks

Table 3 presents some microbenchmarks for Spanner. These measurements were taken on timeshared machines: each spanserver ran on scheduling units of 4GB RAM and 4 cores (AMD Barcelona 2200MHz). Clients were run on separate machines. Each zone contained one spanserver. Clients and zones were placed in a set of datacenters with network distance of less than 1ms. (Such a layout should be commonplace: most applications do not need to distribute all of their data worldwide.) The test database was created with 50 Paxos groups with 2500 directories. Operations were standalone reads and writes of 4KB. All reads were served out of memory after a compaction, so that we are only measuring the overhead of Spanner's call stack. In addition, one unmeasured round of reads was done first to warm any location caches.

For the latency experiments, clients issued sufficiently few operations so as to avoid queuing at the servers. From the 1-replica experiments, commit wait is about 5ms, and Paxos latency is about 9ms. As the number of replicas increases, the latency stays roughly constant with less standard deviation because Paxos executes in parallel at a group's replicas. As the number of replicas increases, the latency to achieve a quorum becomes less sensitive to slowness at one slave replica.

For the throughput experiments, clients issued sufficiently many operations so as to saturate the servers'

participants	latency (ms)	
	mean	99th percentile
1	$17.0 \pm 1.4$	$75.0 \pm 34.9$
2	$24.5 \pm 2.5$	$87.6 \pm 35.9$
5	$31.5 \pm 6.2$	$104.5 \pm 52.2$
10	$30.0 \pm 3.7$	$95.6 \pm 25.4$
25	$35.5 \pm 5.6$	$100.4 \pm 42.7$
50	$42.7 \pm 4.1$	$93.7 \pm 22.9$
100	$71.4 \pm 7.6$	$131.2 \pm 17.6$
200	$150.5 \pm 11.0$	$320.3 \pm 35.1$

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

CPUs. Snapshot reads can execute at any up-to-date replicas, so their throughput increases almost linearly with the number of replicas. Single-read read-only transactions only execute at leaders because timestamp assignment must happen at leaders. Read-only-transaction throughput increases with the number of replicas because the number of effective spanservers increases: in the experimental setup, the number of spanservers equaled the number of replicas, and leaders were randomly distributed among the zones. Write throughput benefits from the same experimental artifact (which explains the increase in throughput from 3 to 5 replicas), but that benefit is outweighed by the linear increase in the amount of work performed per write, as the number of replicas increases.

Table 4 demonstrates that two-phase commit can scale to a reasonable number of participants: it summarizes a set of experiments run across 3 zones, each with 25 spanservers. Scaling up to 50 participants is reasonable in both mean and 99th-percentile, and latencies start to rise noticeably at 100 participants.

## 5.2 Availability

Figure 5 illustrates the availability benefits of running Spanner in multiple datacenters. It shows the results of three experiments on throughput in the presence of datacenter failure, all of which are overlaid onto the same time scale. The test universe consisted of 5 zones  $Z_i$ , each of which had 25 spanservers. The test database was sharded into 1250 Paxos groups, and 100 test clients constantly issued non-snapshot reads at an aggregate rate of 50K reads/second. All of the leaders were explicitly placed in  $Z_1$ . Five seconds into each test, all of the servers in one zone were killed: *non-leader* kills  $Z_2$ ; *leader-hard* kills  $Z_1$ ; *leader-soft* kills  $Z_1$ , but it gives notifications to all of the servers that they should handoff leadership first.

Killing  $Z_2$  has no effect on read throughput. Killing  $Z_1$  while giving the leaders time to handoff leadership to

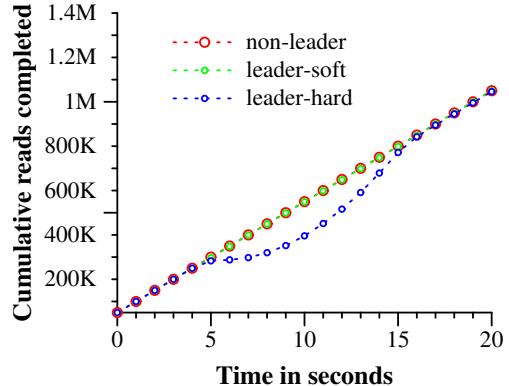


Figure 5: Effect of killing servers on throughput.

a different zone has a minor effect: the throughput drop is not visible in the graph, but is around 3-4%. On the other hand, killing  $Z_1$  with no warning has a severe effect: the rate of completion drops almost to 0. As leaders get re-elected, though, the throughput of the system rises to approximately 100K reads/second because of two artifacts of our experiment: there is extra capacity in the system, and operations are queued while the leader is unavailable. As a result, the throughput of the system rises before leveling off again at its steady-state rate.

We can also see the effect of the fact that Paxos leader leases are set to 10 seconds. When we kill the zone, the leader-lease expiration times for the groups should be evenly distributed over the next 10 seconds. Soon after each lease from a dead leader expires, a new leader is elected. Approximately 10 seconds after the kill time, all of the groups have leaders and throughput has recovered. Shorter lease times would reduce the effect of server deaths on availability, but would require greater amounts of lease-renewal network traffic. We are in the process of designing and implementing a mechanism that will cause slaves to release Paxos leader leases upon leader failure.

## 5.3 TrueTime

Two questions must be answered with respect to TrueTime: is  $\epsilon$  truly a bound on clock uncertainty, and how bad does  $\epsilon$  get? For the former, the most serious problem would be if a local clock's drift were greater than 200us/sec: that would break assumptions made by TrueTime. Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems. As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends.

Figure 6 presents TrueTime data taken at several thousand spanserver machines across datacenters up to 2200

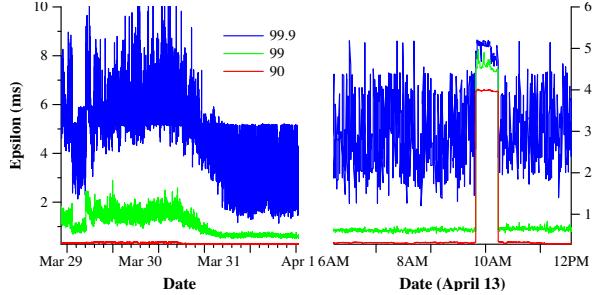


Figure 6: Distribution of TrueTime  $\epsilon$  values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

km apart. It plots the 90th, 99th, and 99.9th percentiles of  $\epsilon$ , sampled at timeslave daemons immediately after polling the time masters. This sampling elides the sawtooth in  $\epsilon$  due to local-clock uncertainty, and therefore measures time-master uncertainty (which is generally 0) plus communication delay to the time masters.

The data shows that these two factors in determining the base value of  $\epsilon$  are generally not a problem. However, there can be significant tail-latency issues that cause higher values of  $\epsilon$ . The reduction in tail latencies beginning on March 30 were due to networking improvements that reduced transient network-link congestion. The increase in  $\epsilon$  on April 13, approximately one hour in duration, resulted from the shutdown of 2 time masters at a datacenter for routine maintenance. We continue to investigate and remove causes of TrueTime spikes.

## 5.4 F1

Spanner started being experimentally evaluated under production workloads in early 2011, as part of a rewrite of Google’s advertising backend called F1 [35]. This backend was originally based on a MySQL database that was manually sharded many ways. The uncompressed dataset is tens of terabytes, which is small compared to many NoSQL instances, but was large enough to cause difficulties with sharded MySQL. The MySQL sharding scheme assigned each customer and all related data to a fixed shard. This layout enabled the use of indexes and complex query processing on a per-customer basis, but required some knowledge of the sharding in application business logic. Resharding this revenue-critical database as it grew in the number of customers and their data was extremely costly. The last resharding took over two years of intense effort, and involved coordination and testing across dozens of teams to minimize risk. This operation was too complex to do regularly: as a result, the team had to limit growth on the MySQL database by storing some

# fragments	# directories
1	>100M
2–4	341
5–9	5336
10–14	232
15–99	34
100–500	7

Table 5: Distribution of directory-fragment counts in F1.

data in external Bigtables, which compromised transactional behavior and the ability to query across all data.

The F1 team chose to use Spanner for several reasons. First, Spanner removes the need to manually reshards. Second, Spanner provides synchronous replication and automatic failover. With MySQL master-slave replication, failover was difficult, and risked data loss and downtime. Third, F1 requires strong transactional semantics, which made using other NoSQL systems impractical. Application semantics requires transactions across arbitrary data, and consistent reads. The F1 team also needed secondary indexes on their data (since Spanner does not yet provide automatic support for secondary indexes), and was able to implement their own consistent global indexes using Spanner transactions.

All application writes are now by default sent through F1 to Spanner, instead of the MySQL-based application stack. F1 has 2 replicas on the west coast of the US, and 3 on the east coast. This choice of replica sites was made to cope with outages due to potential major natural disasters, and also the choice of their frontend sites. Anecdotally, Spanner’s automatic failover has been nearly invisible to them. Although there have been unplanned cluster failures in the last few months, the most that the F1 team has had to do is update their database’s schema to tell Spanner where to preferentially place Paxos leaders, so as to keep them close to where their frontends moved.

Spanner’s timestamp semantics made it efficient for F1 to maintain in-memory data structures computed from the database state. F1 maintains a logical history log of all changes, which is written into Spanner itself as part of every transaction. F1 takes full snapshots of data at a timestamp to initialize its data structures, and then reads incremental changes to update them.

Table 5 illustrates the distribution of the number of fragments per directory in F1. Each directory typically corresponds to a customer in the application stack above F1. The vast majority of directories (and therefore customers) consist of only 1 fragment, which means that reads and writes to those customers’ data are guaranteed to occur on only a single server. The directories with more than 100 fragments are all tables that contain F1 secondary indexes: writes to more than a few fragments

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

of such tables are extremely uncommon. The F1 team has only seen such behavior when they do untuned bulk data loads as transactions.

Table 6 presents Spanner operation latencies as measured from F1 servers. Replicas in the east-coast data centers are given higher priority in choosing Paxos leaders. The data in the table is measured from F1 servers in those data centers. The large standard deviation in write latencies is caused by a pretty fat tail due to lock conflicts. The even larger standard deviation in read latencies is partially due to the fact that Paxos leaders are spread across two data centers, only one of which has machines with SSDs. In addition, the measurement includes every read in the system from two datacenters: the mean and standard deviation of the bytes read were roughly 1.6KB and 119KB, respectively.

## 6 Related Work

Consistent replication across datacenters as a storage service has been provided by Megastore [5] and DynamoDB [3]. DynamoDB presents a key-value interface, and only replicates within a region. Spanner follows Megastore in providing a semi-relational data model, and even a similar schema language. Megastore does not achieve high performance. It is layered on top of Bigtable, which imposes high communication costs. It also does not support long-lived leaders: multiple replicas may initiate writes. All writes from different replicas necessarily conflict in the Paxos protocol, even if they do not logically conflict: throughput collapses on a Paxos group at several writes per second. Spanner provides higher performance, general-purpose transactions, and external consistency.

Pavlo et al. [31] have compared the performance of databases and MapReduce [12]. They point to several other efforts that have been made to explore database functionality layered on distributed key-value stores [1, 4, 7, 41] as evidence that the two worlds are converging. We agree with the conclusion, but demonstrate that integrating multiple layers has its advantages: integrating concurrency control with replication reduces the cost of commit wait in Spanner, for example.

The notion of layering transactions on top of a replicated store dates at least as far back as Gifford’s dissertation [16]. Scatter [17] is a recent DHT-based key-value store that layers transactions on top of consistent replication. Spanner focuses on providing a higher-level interface than Scatter does. Gray and Lamport [18] describe a non-blocking commit protocol based on Paxos. Their protocol incurs more messaging costs than two-phase commit, which would aggravate the cost of commit over widely distributed groups. Walter [36] provides a variant of snapshot isolation that works within, but not across datacenters. In contrast, our read-only transactions provide a more natural semantics, because we support external consistency over all operations.

There has been a spate of recent work on reducing or eliminating locking overheads. Calvin [40] eliminates concurrency control: it pre-assigns timestamps and then executes the transactions in timestamp order. H-Store [39] and Granola [11] each supported their own classification of transaction types, some of which could avoid locking. None of these systems provides external consistency. Spanner addresses the contention issue by providing support for snapshot isolation.

VoltDB [42] is a sharded in-memory database that supports master-slave replication over the wide area for disaster recovery, but not more general replication configurations. It is an example of what has been called NewSQL, which is a marketplace push to support scalable SQL [38]. A number of commercial databases implement reads in the past, such as MarkLogic [26] and Oracle’s Total Recall [30]. Lomet and Li [24] describe an implementation strategy for such a temporal database.

Farsite derived bounds on clock uncertainty (much looser than TrueTime’s) relative to a trusted clock reference [13]: server leases in Farsite were maintained in the same way that Spanner maintains Paxos leases. Loosely synchronized clocks have been used for concurrency-control purposes in prior work [2, 23]. We have shown that TrueTime lets one reason about global time across sets of Paxos state machines.

## 7 Future Work

We have spent most of the last year working with the F1 team to transition Google’s advertising backend from MySQL to Spanner. We are actively improving its monitoring and support tools, as well as tuning its performance. In addition, we have been working on improving the functionality and performance of our backup/restore system. We are currently implementing the Spanner schema language, automatic maintenance of secondary indices, and automatic load-based resharding. Longer term, there are a couple of features that we plan to in-

vestigate. Optimistically doing reads in parallel may be a valuable strategy to pursue, but initial experiments have indicated that the right implementation is non-trivial. In addition, we plan to eventually support direct changes of Paxos configurations [22, 34].

Given that we expect many applications to replicate their data across datacenters that are relatively close to each other, TrueTime  $\epsilon$  may noticeably affect performance. We see no insurmountable obstacle to reducing  $\epsilon$  below 1ms. Time-master-query intervals can be reduced, and better clock crystals are relatively cheap. Time-master query latency could be reduced with improved networking technology, or possibly even avoided through alternate time-distribution technology.

Finally, there are obvious areas for improvement. Although Spanner is scalable in the number of nodes, the node-local data structures have relatively poor performance on complex SQL queries, because they were designed for simple key-value accesses. Algorithms and data structures from DB literature could improve single-node performance a great deal. Second, moving data automatically between datacenters in response to changes in client load has long been a goal of ours, but to make that goal effective, we would also need the ability to move client-application processes between datacenters in an automated, coordinated fashion. Moving processes raises the even more difficult problem of managing resource acquisition and allocation between datacenters.

## 8 Conclusions

To summarize, Spanner combines and extends on ideas from two research communities: from the database community, a familiar, easy-to-use, semi-relational interface, transactions, and an SQL-based query language; from the systems community, scalability, automatic sharding, fault tolerance, consistent replication, external consistency, and wide-area distribution. Since Spanner’s inception, we have taken more than 5 years to iterate to the current design and implementation. Part of this long iteration phase was due to a slow realization that Spanner should do more than tackle the problem of a globally-replicated namespace, and should also focus on database features that Bigtable was missing.

One aspect of our design stands out: the linchpin of Spanner’s feature set is TrueTime. We have shown that reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics. In addition, as the underlying system enforces tighter bounds on clock uncertainty, the overhead of the stronger semantics decreases. As a community, we should no longer depend on loosely synchronized clocks and weak time APIs in designing distributed algorithms.

## Acknowledgements

Many people have helped to improve this paper: our shepherd Jon Howell, who went above and beyond his responsibilities; the anonymous referees; and many Googlers: Atul Adya, Fay Chang, Frank Dabek, Sean Dorward, Bob Gruber, David Held, Nick Kline, Alex Thomson, and Joel Wein. Our management has been very supportive of both our work and of publishing this paper: Aristotle Balogh, Bill Coughran, Urs Hölzle, Doron Meyer, Cos Nicolaou, Kathy Polizzi, Sridhar Ramaswany, and Shivakumar Venkataraman.

We have built upon the work of the Bigtable and Megastore teams. The F1 team, and Jeff Shute in particular, worked closely with us in developing our data model and helped immensely in tracking down performance and correctness bugs. The Platforms team, and Luiz Barroso and Bob Felderman in particular, helped to make TrueTime happen. Finally, a lot of Googlers used to be on our team: Ken Ashcraft, Paul Czysz, Krzysztof Ostrowski, Amir Voskoboinik, Matthew Weaver, Theo Vassilakis, and Eric Veach; or have joined our team recently: Nathan Bales, Adam Beberg, Vadim Borisov, Ken Chen, Brian Cooper, Cian Cullinan, Robert-Jan Huijsman, Milind Joshi, Andrey Khorlin, Dawid Kuroczko, Laramie Leavitt, Eric Li, Mike Mammarella, Sunil Mushran, Simon Nielsen, Ovidiu Platon, Ananth Shrinivas, Vadim Sudorov, and Marcel van der Holst.

## References

- [1] Azza Abouzeid et al. “HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads”. *Proc. of VLDB*. 2009, pp. 922–933.
- [2] A. Adya et al. “Efficient optimistic concurrency control using loosely synchronized clocks”. *Proc. of SIGMOD*. 1995, pp. 23–34.
- [3] Amazon. *Amazon DynamoDB*. 2012.
- [4] Michael Armbrust et al. “PIQL: Success-Tolerant Query Processing in the Cloud”. *Proc. of VLDB*. 2011, pp. 181–192.
- [5] Jason Baker et al. “Megastore: Providing Scalable, Highly Available Storage for Interactive Services”. *Proc. of CIDR*. 2011, pp. 223–234.
- [6] Hal Berenson et al. “A critique of ANSI SQL isolation levels”. *Proc. of SIGMOD*. 1995, pp. 1–10.
- [7] Matthias Brantner et al. “Building a database on S3”. *Proc. of SIGMOD*. 2008, pp. 251–264.
- [8] A. Chan and R. Gray. “Implementing Distributed Read-Only Transactions”. *IEEE TOSE SE-11.2* (Feb. 1985), pp. 205–212.
- [9] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. *ACM TOCS* 26.2 (June 2008), 4:1–4:26.
- [10] Brian F. Cooper et al. “PNUTS: Yahoo!’s hosted data serving platform”. *Proc. of VLDB*. 2008, pp. 1277–1288.
- [11] James Cowling and Barbara Liskov. “Granola: Low-Overhead Distributed Transaction Coordination”. *Proc. of USENIX ATC*. 2012, pp. 223–236.

- [12] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: a flexible data processing tool”. *CACM* 53.1 (Jan. 2010), pp. 72–77.
- [13] John Douceur and Jon Howell. *Scalable Byzantine-Fault-Quantifying Clock Synchronization*. Tech. rep. MSR-TR-2003-67. MS Research, 2003.
- [14] John R. Douceur and Jon Howell. “Distributed directory service in the Farsite file system”. *Proc. of OSDI*. 2006, pp. 321–334.
- [15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google file system”. *Proc. of SOSP*. Dec. 2003, pp. 29–43.
- [16] David K. Gifford. *Information Storage in a Decentralized Computer System*. Tech. rep. CSL-81-8. PhD dissertation. Xerox PARC, July 1982.
- [17] Lisa Glendenning et al. “Scalable consistency in Scatter”. *Proc. of SOSP*. 2011.
- [18] Jim Gray and Leslie Lamport. “Consensus on transaction commit”. *ACM TODS* 31.1 (Mar. 2006), pp. 133–160.
- [19] Pat Helland. “Life beyond Distributed Transactions: an Apostate’s Opinion”. *Proc. of CIDR*. 2007, pp. 132–141.
- [20] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: a correctness condition for concurrent objects”. *ACM TOPLAS* 12.3 (July 1990), pp. 463–492.
- [21] Leslie Lamport. “The part-time parliament”. *ACM TOCS* 16.2 (May 1998), pp. 133–169.
- [22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. “Reconfiguring a state machine”. *SIGACT News* 41.1 (Mar. 2010), pp. 63–73.
- [23] Barbara Liskov. “Practical uses of synchronized clocks in distributed systems”. *Distrib. Comput.* 6.4 (July 1993), pp. 211–219.
- [24] David B. Lomet and Feifei Li. “Improving Transaction-Time DBMS Performance and Functionality”. *Proc. of ICDE* (2009), pp. 581–591.
- [25] Jacob R. Lorch et al. “The SMART way to migrate replicated stateful services”. *Proc. of EuroSys*. 2006, pp. 103–115.
- [26] MarkLogic. *MarkLogic 5 Product Documentation*. 2012.
- [27] Keith Marzullo and Susan Owicki. “Maintaining the time in a distributed system”. *Proc. of PODC*. 1983, pp. 295–305.
- [28] Sergey Melnik et al. “Dremel: Interactive Analysis of Web-Scale Datasets”. *Proc. of VLDB*. 2010, pp. 330–339.
- [29] D.L. Mills. *Time synchronization in DCNET hosts*. Internet Project Report IEN-173. COMSAT Laboratories, Feb. 1981.
- [30] Oracle. *Oracle Total Recall*. 2012.
- [31] Andrew Pavlo et al. “A comparison of approaches to large-scale data analysis”. *Proc. of SIGMOD*. 2009, pp. 165–178.
- [32] Daniel Peng and Frank Dabek. “Large-scale incremental processing using distributed transactions and notifications”. *Proc. of OSDI*. 2010, pp. 1–15.
- [33] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. “System level concurrency control for distributed database systems”. *ACM TODS* 3.2 (June 1978), pp. 178–198.
- [34] Alexander Shraer et al. “Dynamic Reconfiguration of Primary/Backup Clusters”. *Proc. of USENIX ATC*. 2012, pp. 425–438.
- [35] Jeff Shute et al. “F1 — The Fault-Tolerant Distributed RDBMS Supporting Google’s Ad Business”. *Proc. of SIGMOD*. May 2012, pp. 777–778.
- [36] Yair Sovran et al. “Transactional storage for geo-replicated systems”. *Proc. of SOSP*. 2011, pp. 385–400.
- [37] Michael Stonebraker. *Why Enterprises Are Uninterested in NoSQL*. 2010.
- [38] Michael Stonebraker. *Six SQL Urban Myths*. 2010.
- [39] Michael Stonebraker et al. “The end of an architectural era: (it’s time for a complete rewrite)”. *Proc. of VLDB*. 2007, pp. 1150–1160.
- [40] Alexander Thomson et al. “Calvin: Fast Distributed Transactions for Partitioned Database Systems”. *Proc. of SIGMOD*. 2012, pp. 1–12.
- [41] Ashish Thusoo et al. “Hive — A Petabyte Scale Data Warehouse Using Hadoop”. *Proc. of ICDE*. 2010, pp. 996–1005.
- [42] VoltDB. *VoltDB Resources*. 2012.

## A Paxos Leader-Lease Management

The simplest means to ensure the disjointness of Paxos-leader-lease intervals would be for a leader to issue a synchronous Paxos write of the lease interval, whenever it would be extended. A subsequent leader would read the interval and wait until that interval has passed.

TrueTime can be used to ensure disjointness without these extra log writes. The potential  $i$ th leader keeps a lower bound on the start of a lease vote from replica  $r$  as  $v_{i,r}^{\text{leader}} = \text{TT.now}().\text{earliest}$ , computed before  $e_{i,r}^{\text{send}}$  (defined as when the lease request is sent by the leader). Each replica  $r$  grants a lease at lease  $e_{i,r}^{\text{grant}}$ , which happens after  $e_{i,r}^{\text{receive}}$  (when the replica receives a lease request); the lease ends at  $t_{i,r}^{\text{end}} = \text{TT.now}().\text{latest} + 10$ , computed after  $e_{i,r}^{\text{receive}}$ . A replica  $r$  obeys the **single-vote** rule: it will not grant another lease vote until  $\text{TT.after}(t_{i,r}^{\text{end}})$  is true. To enforce this rule across different incarnations of  $r$ , Spanner logs a lease vote at the granting replica before granting the lease; this log write can be piggybacked upon existing Paxos-protocol log writes.

When the  $i$ th leader receives a quorum of votes (event  $e_i^{\text{quorum}}$ ), it computes its lease interval as  $\text{lease}_i = [\text{TT.now}().\text{latest}, \min_r(v_{i,r}^{\text{leader}}) + 10]$ . The lease is deemed to have expired at the leader when  $\text{TT.before}(\min_r(v_{i,r}^{\text{leader}}) + 10)$  is false. To prove disjointness, we make use of the fact that the  $i$ th and  $(i+1)$ th leaders must have one replica in common in their quorums. Call that replica  $r_0$ . Proof:

$$\begin{aligned}
 \text{lease}_i.\text{end} &= \min_r(v_{i,r}^{\text{leader}}) + 10 && (\text{by definition}) \\
 \min_r(v_{i,r}^{\text{leader}}) + 10 &\leq v_{i,r_0}^{\text{leader}} + 10 && (\text{min}) \\
 v_{i,r_0}^{\text{leader}} + 10 &\leq t_{\text{abs}}(e_{i,r_0}^{\text{send}}) + 10 && (\text{by definition}) \\
 t_{\text{abs}}(e_{i,r_0}^{\text{send}}) + 10 &\leq t_{\text{abs}}(e_{i,r_0}^{\text{receive}}) + 10 && (\text{causality}) \\
 t_{\text{abs}}(e_{i,r_0}^{\text{receive}}) + 10 &\leq t_{i,r_0}^{\text{end}} && (\text{by definition}) \\
 t_{i,r_0}^{\text{end}} < t_{\text{abs}}(e_{i+1,r_0}^{\text{grant}}) &&& (\text{single-vote}) \\
 t_{\text{abs}}(e_{i+1,r_0}^{\text{grant}}) &\leq t_{\text{abs}}(e_{i+1}^{\text{quorum}}) && (\text{causality}) \\
 t_{\text{abs}}(e_{i+1}^{\text{quorum}}) &\leq \text{lease}_{i+1}.\text{start} && (\text{by definition})
 \end{aligned}$$

# RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems

Yongqiang He <sup>#\\$1</sup>, Rubao Lee <sup>%2</sup>, Yin Huai <sup>%3</sup>, Zheng Shao <sup>#4</sup>, Namit Jain <sup>#5</sup>, Xiaodong Zhang <sup>%6</sup>, Zhiwei Xu <sup>\\$7</sup>

<sup>#</sup>*Facebook Data Infrastructure Team*

{<sup>1</sup>heyongqiang, <sup>4</sup>zshao, <sup>5</sup>njain}@fb.com

<sup>%</sup>*Department of Computer Science and Engineering, The Ohio State University*

{<sup>2</sup>liru, <sup>3</sup>huai, <sup>6</sup>zhang}@cse.ohio-state.edu

<sup>\\$</sup>*Institute of Computing Technology, Chinese Academy of Sciences*

<sup>7</sup>zxu@ict.ac.cn

**Abstract**—MapReduce-based data warehouse systems are playing important roles of supporting big data analytics to understand quickly the dynamics of user behavior trends and their needs in typical Web service providers and social network sites (e.g., Facebook). In such a system, the data placement structure is a critical factor that can affect the warehouse performance in a fundamental way. Based on our observations and analysis of Facebook production systems, we have characterized four requirements for the data placement structure: (1) fast data loading, (2) fast query processing, (3) highly efficient storage space utilization, and (4) strong adaptivity to highly dynamic workload patterns. We have examined three commonly accepted data placement structures in conventional databases, namely row-stores, column-stores, and hybrid-stores in the context of large data analysis using MapReduce. We show that they are not very suitable for big data processing in distributed systems. In this paper, we present a big data placement structure called RCFile (Record Columnar File) and its implementation in the Hadoop system. With intensive experiments, we show the effectiveness of RCFile in satisfying the four requirements. RCFile has been chosen in Facebook data warehouse system as the default option. It has also been adopted by Hive and Pig, the two most widely used data analysis systems developed in Facebook and Yahoo!

## I. INTRODUCTION

We have entered an era of data explosion, where many data sets being processed and analyzed are called “big data”. Big data not only requires a huge amount of storage, but also demands new data management on large distributed systems because conventional database systems have difficulty to manage big data. One important and emerging application of big data happens in social networks on the Internet, where billions of people all over the world connect and the number of users along with their various activities is growing rapidly. For example, the number of registered users in Facebook, the largest social network in the world has been over 500 million [1]. One critical task in Facebook is to understand quickly the dynamics of user behavior trends and user needs based on big data sets recording busy user activities.

The MapReduce framework [2] and its open-source implementation Hadoop [3] provide a scalable and fault-tolerant infrastructure for big data analysis on large clusters. Furthermore, MapReduce-based data warehouse systems have been successfully built in major Web service providers and social

network Websites, and are playing critical roles for executing various daily operations including Web click-stream analysis, advertisement analysis, data mining applications, and many others. Two widely used Hadoop-based warehouse systems are Hive [4][5] in Facebook and Pig [6] in Yahoo!

These MapReduce-based warehouse systems cannot directly control storage disks in clusters. Instead, they have to utilize the cluster-level distributed file system (e.g. HDFS, the Hadoop Distributed File System) to store a huge amount of table data. Therefore, a serious challenge in building such a system is to find an efficient data placement structure that determines how to organize table data in the underlying HDFS. Being a critical factor that can affect warehouse performance in a fundamental way, such a data placement structure must be well optimized to meet the big data processing requirements and to efficiently leverage merits in a MapReduce environment.

### A. Big Data Processing Requirements

Based on our analysis on Facebook systems and huge user data sets, we have summarized the following four critical requirements for a data placement structure in a MapReduce environment.

- 1) *Fast data loading.* Loading data quickly is critical for the Facebook production data warehouse. On average, more than 20TB data are pushed into a Facebook data warehouse every day. Thus, it is highly desirable to reduce data loading time, since network and disk traffic during data loading will interfere with normal query executions.
- 2) *Fast query processing.* Many queries are response-time critical in order to satisfy the requirements of both real-time Website requests and heavy workloads of decision supporting queries submitted by highly-concurrent users. This requires that the underlying data placement structure retain the high speed for query processing as the amount of queries rapidly increases.
- 3) *Highly efficient storage space utilization.* Rapidly growing user activities have constantly demanded scalable storage capacity and computing power. Limited disk

- space demands that data storage be well-managed, in practice, to address the issues on how to place data in disks so that space utilization is maximized.
- 4) *Strong adaptivity to highly dynamic workload patterns.* Data sets are analyzed by different application users for different purposes in many different ways [7]. Some data analytics are routine processes that are executed periodically in a static mode, while some are ad-hoc queries issued from internal platforms. Most workloads do not follow any regular patterns, which demand the underlying system be highly adaptive to unexpected dynamics in data processing with limited storage space, instead of being specific to certain workload patterns.

### B. Data Placement for MapReduce

A critical challenge in designing and implementing an efficient data placement structure for a MapReduce-based data warehouse system is to address the above four requirements considering unique features in a MapReduce computing environment. In conventional database systems, three data placement structures have been widely studied, which are:

- 1) *horizontal row-store structure* ([8]),
- 2) *vertical column-store structure* ([9][10][11][12]), and
- 3) *hybrid PAX store structure* ([13][14]).

Each of these structures has its advantages considering one of the above requirements. However, simply porting these database-oriented structures into a MapReduce-based data warehouse system cannot fully satisfy all four requirements.

We here briefly summarize major limitations of these structures for big data, and will provide a detailed evaluation on the three structures in Section II. First, row-store cannot support fast query processing because it cannot skip unnecessary column reads when a query only requires only a few columns from a wide table with many columns [10]. Second, column-store can often cause high record reconstruction overhead with expensive network transfers in a cluster. This is because, with column-store, HDFS cannot guarantee that all fields in the same record are stored in the same cluster node. Although pre-grouping multiple columns together can reduce the overhead, it does not have a strong adaptivity to respond highly dynamic workload patterns. Third, with the goal of optimizing CPU cache performance, the hybrid PAX structure that uses column-store inside each disk page cannot help improve the I/O performance [15][12] for analytics of big data.

In this paper, we present our data placement structure, called RCFFile (*Record Columnar File*), and its implementation in Hadoop. We highlight the RCFFile structure as follows.

- 1) A table stored in RCFFile is first horizontally partitioned into multiple *row groups*. Then, each row group is vertically partitioned so that each column is stored independently.
- 2) RCFFile utilizes a column-wise data compression within each row group, and provides a *lazy decompression* technique to avoid unnecessary column decompression during query execution.

- 3) RCFFile allows a flexible row group size. A default size is given considering both data compression performance and query execution performance. RCFFile also allows users to select the row group size for a given table.

We have implemented RCFFile in Hadoop. As to be discussed and evaluated in this paper, RCFFile can satisfy all the above four requirements, since it not only retains the merits of both the row-store and the column-store, and also has added PAX's missing role of I/O performance improvement. After extensive evaluation in production systems, RCFFile has been chosen in the Facebook Hive data warehouse system as the default option. Besides, RCFFile has also been adopted in the Yahoo Pig system, and is being considered in other MapReduce-based data processing systems.

The rest of the paper is organized as follows. We present a detailed analysis of existing three data placement structures in Section II. We present the design, implementation, and several critical issues of our solution RCFFile in Section III. In Section IV, we introduce the API of RCFFile. Performance evaluation is presented in Section V. We overview other related work in Section VI, and conclude the paper in Section VII.

## II. MERITS AND LIMITATIONS OF EXISTING DATA PLACEMENT STRUCTURES

In this section, we will introduce three existing data placement structures, and discuss why they may not be suitable to a Hadoop-based data warehouse system.

### A. Horizontal Row-store

The row-store structure dominates in conventional one-sizes-fits-all database systems [16]. With this structure, relational records are organized in an N-ary storage model [8]. All fields of one record are padded one by one in the order of their occurrences. Records are placed contiguously in a disk page. Figure 1 gives an example to show how a table is placed by the row-store structure in an HDFS block.

The major weaknesses of row-store for read-only data warehouse systems have been intensively discussed. First, row-store cannot provide fast query processing due to unnecessary column reads if only a subset of columns in a table are needed in a query. Second, it is not easy for row-store to achieve a high data compression ratio (and thus a high storage space utilization) due to mixed columns with different data domains

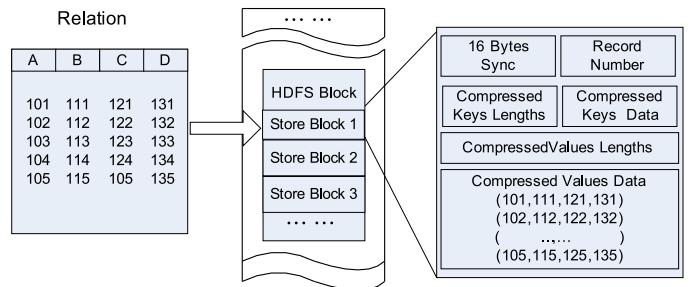


Fig. 1: An example of row-store in an HDFS block.

[10][11]. Although prior work [17][18] has shown that row-store with careful *entropy coding* and the utilization of *column correlations* can have a better data compression ratio than that of column-store, it can cause a high data decompression overhead by complex data storage implementations.

The major advantage of row-store for a Hadoop-based system is that it has fast data loading and strong adaptive ability to dynamic workloads. This is because row-store guarantees that all fields in the same record is located in the same cluster node since they are in the same HDFS block.

### B. Vertical Column-store

The vertical store scheme is based on a column-oriented store model for read-optimized data warehouse systems. In a vertical storage, a relation is vertically partitioned into several sub-relations. Basically, there are two schemes of vertical stores. One scheme is to put each column in one sub-relation, such as the Decomposition Storage Model (DSM) [9], MonetDB [19], and an experimental system in [15]. Another scheme is to organize all the columns of a relation into different column groups, and usually allow column overlapping among multiple column groups, such as C-store [10] and Yahoo Zebra [20]. In this paper, we call the first scheme *the column-store*, and the second one *the column-group*. Notice that, for column-group, how data is organized (row-oriented or column-oriented) in a group is dependent on system implementations. In C-store, a column group uses the column-store model, i.e., each column is stored individually. However, to accelerate a record reconstruction activity, all columns in a column group must be ordered in the same way. In Zebra used for the Pig System, a column group is actually row-oriented to reduce the overhead of a record reconstruction in a MapReduce environment.

Figure 2 shows an example on how a table is stored by column-group on HDFS. In this example, column *A* and column *B* are stored in the same column group, while column *C* and column *D* are stored in two independent column groups.

Column-store can avoid reading unnecessary columns during a query execution, and can easily achieve a high compression ratio by compressing each column within the same data domain. However, it cannot provide fast query processing in Hadoop-based systems due to high overhead of a tuple reconstruction. Column-store cannot guarantee that all fields in the same record are located in the same cluster node. For instance, in the example in Figure 2, the four fields of a record are stored in three HDFS blocks that can be located in different nodes. Therefore, a record reconstruction will cause a large amount of data transfers via networks among multiple cluster nodes. As introduced in the original MapReduce paper [2], excessive network transfers in a MapReduce cluster can always be a bottleneck source, which should be avoided if possible.

Since a column group is equivalent to a materialized view, it can avoid the overhead of a record reconstruction [10]. However, it cannot satisfy the requirement of quickly adapting dynamic workloads, unless all column groups have been created with the pre-knowledge of possible queries.

Otherwise, for a query that needs a non-existing combination of columns, a record reconstruction is still required to use two or more existing column groups. Furthermore, column-group can also create redundant column data storage due to the column overlap among multiple groups. This would under-utilize storage space.

### C. Hybrid Store: PAX

PAX [13] and its improvement in Data Morphing [21] use a hybrid placement structure aiming at improving CPU cache performance. For a record with multiple fields from different columns, instead of putting these fields into different disk pages, PAX puts them in a single disk page to save additional operations for record reconstructions. Within each disk page, PAX uses a mini-page to store all fields belonging to each column, and uses a page header to store pointers to mini-pages.

Like row-store, PAX has a strong adaptive ability to various dynamic query workloads. However, since it was mainly proposed for performance improvement of CPU cache utilization for data sets loaded in main memory, PAX cannot directly satisfy the requirements of both high storage space utilization and fast query processing speed on large distributed systems for the following three reasons.

- 1) PAX is not associated with data compression, which is not necessary for cache optimization, but very important for large data processing systems. It does provide an opportunity to do column-wise data compression [13].
- 2) PAX cannot improve I/O performance because it does not change the actual content of a page [15][12]. This

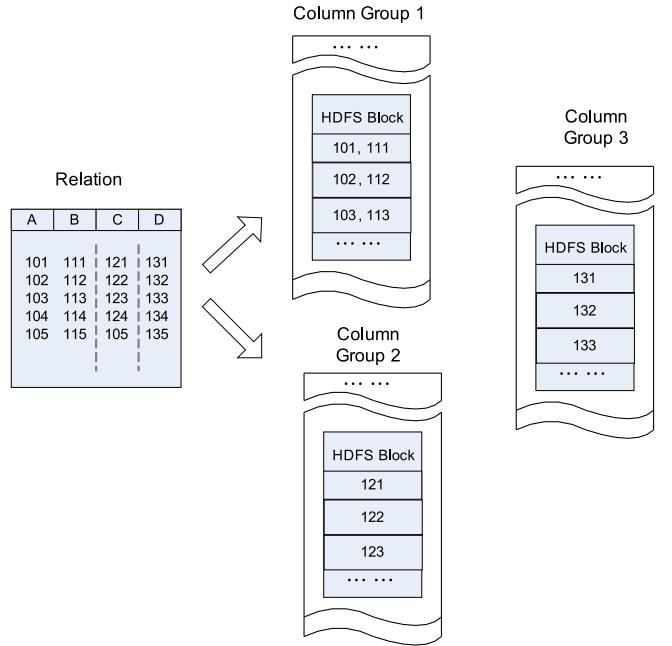


Fig. 2: An example of column-group in an HDFS block. The four columns are stored into three column groups, since column *A* and *B* are grouped in the first column group.

- limitation would not help our goal of fast query processing for a huge amount of disk scans on massively growing data sets.
- 3) Limited by the page-level data manipulation inside a traditional DBMS engine, PAX uses a fixed page as the basic unit of data record organization. With such a fixed size, PAX would not efficiently store data sets with a highly-diverse range of data resource types of different sizes in large data processing systems, such as the one in Facebook.

### III. THE DESIGN AND IMPLEMENTATION OF RCFILE

In this section, we present RCFfile (Record Columnar File), a data placement structure designed for MapReduce-based data warehouse systems, such as Hive. RCFfile applies the concept of “*first horizontally-partition, then vertically-partition*” from PAX. It combines the advantages of both row-store and column-store. First, as row-store, RCFfile guarantees that data in the same row are located in the same node, thus it has low cost of tuple reconstruction. Second, as column-store, RCFfile can exploit a column-wise data compression and skip unnecessary column reads.

#### A. Data Layout and Compression

RCFfile is designed and implemented on top of the Hadoop Distributed File System (HDFS). As demonstrated in the example shown in Figure 3, RCFfile has the following data layout to store a table:

- 1) According to the HDFS structure, a table can have multiple HDFS blocks.
- 2) In each HDFS block, RCFfile organizes records with the basic unit of a *row group*. That is to say, all the records stored in an HDFS block are partitioned into row groups. For a table, all row groups have the same size. Depending on the row group size and the HDFS block size, an HDFS block can have only one or multiple row groups.

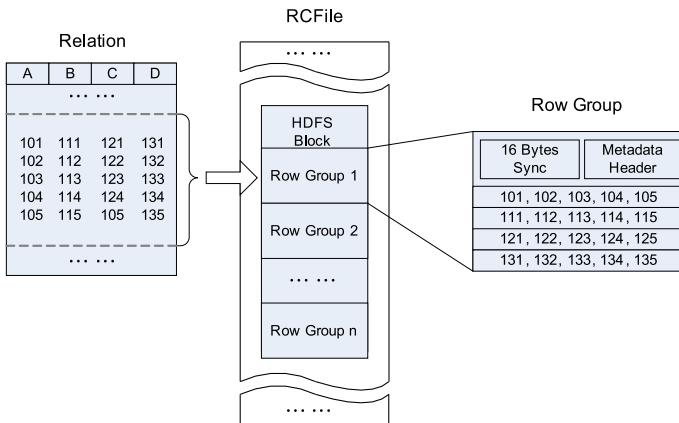


Fig. 3: An example to demonstrate the data layout of RCFfile in an HDFS block.

- 3) A row group contains three sections. The first section is a sync marker that is placed in the beginning of the row group. The sync marker is mainly used to separate two continuous row groups in an HDFS block. The second section is a metadata header for the row group. The metadata header stores the information items on how many records are in this row group, how many bytes are in each column, and how many bytes are in each field in a column. The third section is the table data section that is actually a column-store. In this section, all the fields in the same column are stored continuously together. For example, as shown in Figure 3, the section first stores all fields in column A, and then all fields in column B, and so on.

We now introduce how data is compressed in RCFfile. In each row group, the metadata header section and the table data section are compressed independently as follows.

- First, for the whole metadata header section, RCFfile uses the RLE (Run Length Encoding) algorithm to compress data. Since all the values of the field lengths in the same column are continuously stored in this section, the RLE algorithm can find long runs of repeated data values, especially for fixed field lengths.
- Second, the table data section is not compressed as a whole unit. Rather, each column is independently compressed with the Gzip compression algorithm. RCFfile uses the heavy-weight Gzip algorithm in order to get better compression ratios than other light-weight algorithms. For example, the RLE algorithm is not used since the column data is not already sorted. In addition, due to the lazy decompression technology to be discussed next, RCFfile does not need to decompress all the columns when processing a row group. Thus, the relatively high decompression overhead of the Gzip algorithm can be reduced.

Though currently RCFfile uses the same algorithm for all columns in the table data section, it allows us to use different algorithms to compress different columns. One future work related to the RCFfile project is to automatically select the best compression algorithm for each column according to its data type and data distribution.

#### B. Data Appending

RCFfile does not allow arbitrary data writing operations. Only an appending interface is provided for data writing in RCFfile because the underlying HDFS currently only supports data writes to the end of a file. The method of data appending in RCFfile is summarized as follows.

- 1) RCFfile creates and maintains an in-memory *column holder* for each column. When a record is appended, all its fields will be scattered, and each field will be appended into its corresponding column holder. In addition, RCFfile will record corresponding metadata of each field in the metadata header.
- 2) RCFfile provides two parameters to control how many records can be buffered in memory before they are

- flushed into the disk. One parameter is the limit of the number of records, and the other parameter is the limit of the size of the memory buffer.
- 3) RCFFile first compresses the metadata header and stores it in the disk. Then it compresses each column holder separately, and flushes compressed column holders into one row group in the underlying file system.

### C. Data Reads and Lazy Decompression

Under the MapReduce framework, a mapper is started for an HDFS block. The mapper will sequentially process each row group in the HDFS block.

When processing a row group, RCFFile does not need to fully read the whole content of the row group into memory. Rather, it only reads the metadata header and the needed columns in the row group for a given query. Thus, it can skip unnecessary columns and gain the I/O advantages of column-store. For instance, suppose we have a table with four columns  $tbl(c_1, c_2, c_3, c_4)$ , and we have a query “*SELECT c<sub>1</sub> FROM tbl WHERE c<sub>4</sub> = 1*”. Then, in each row group, RCFFile only reads the content of column  $c_1$  and  $c_4$ .

After the metadata header and data of needed columns have been loaded into memory, they are all in the compressed format and thus need to be decompressed. The metadata header is always decompressed and held in memory until RCFFile processes the next row group. However, RCFFile does not decompress all the loaded columns. Instead, it uses a lazy decompression technique.

Lazy decompression means that a column will not be decompressed in memory until RCFFile has determined that the data in the column will be really useful for query execution. Lazy decompression is extremely useful due to the existence of various where conditions in a query. If a where condition cannot be satisfied by all the records in a row group, then RCFFile does not decompress the columns that do not occur in the where condition. For example, in the above query, column  $c_4$  in any row group must be decompressed. However, for a row group, if no field of column  $c_4$  in the group has the value 1, then it is unnecessary to decompress column  $c_1$  in the group.

### D. Row Group Size

I/O performance is a major concern of RCFFile. Therefore, RCFFile needs to use a large and flexible row group size. There are two considerations to determine the row group size:

- 1) A large row group size can have better data compression efficiency than that of a small one. However, according to our observations of daily applications in Facebook, when the row group size reaches a threshold, increasing the row group size cannot further improve compression ratio with the Gzip algorithm.
- 2) A large row group size may have lower read performance than that of a small size because a large size can decrease the performance benefits of lazy decompression. Furthermore, a large row group size would have a higher memory usage than a small size, and would affect executions of other co-running MapReduce jobs.

*1) Compression efficiency:* In order to demonstrate how different row group sizes can affect the compression ratio, we have conducted an experiment with a portion of data from a table in Facebook. We examined the table sizes using different row group sizes (from 16KB to 32MB), and the results are shown in Figure 4.

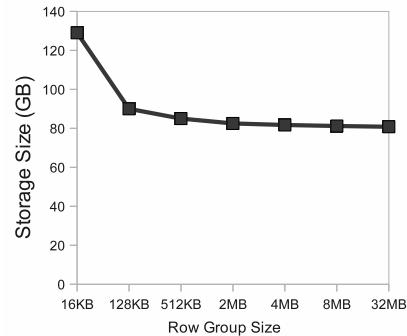


Fig. 4: Data storage size with different row group sizes in RCFFile.

We can see that a large row group size can certainly improve the data compression efficiency and decrease the storage size. Therefore, we could not use a small row group size with the requirement of reducing storage space. However, as shown in Figure 4, when the row group size are larger than 4MB, the compressed data size in the storage almost becomes a constant.

*2) The relationship between row group size and lazy decompression:* Though a large row group size helps decrease the storage size for a table, it may hurt the data read performance, since a small row group size is a better choice to gain the performance advantage of lazy decompression than a large row group size.

We present an example to demonstrate the relationship between row group size and lazy decompression. Consider a query “*SELECT a,b,c,d FROM tbl WHERE e = 1*”. We assume that in an HDFS block, only one record of the table can satisfy the condition of  $e = 1$ . We now consider an extreme case where the row group size is as large as possible so that the HDFS block contains only one row group. In this case, RCFFile will decompress column  $e$  first to check the where condition, and find that the condition can be satisfied in this row group. Therefore, RCFFile will also decompress all the other four columns ( $a, b, c, d$ ) since they are needed by the query.

However, since only one record in the HDFS block is actually useful, i.e., the one with  $e = 1$ , it is not necessary to decompress the other data items of columns ( $a, b, c, d$ ) except this record. If we use a small row group size, then there is only one useful row group that contains the record with  $e = 1$ , and all other row groups are useless for this query. Therefore, only in the useful row group, all the five columns ( $a, b, c, d, e$ ) need to be decompressed. In other useless row groups, only column  $e$  needs to be decompressed, and the other four columns will not be decompressed, since the where condition cannot be satisfied. Thus, the query execution time can be decreased.

Users should choose the row group size to consider both

the storage space and query execution requirements. Currently, RCFfile adapted in Facebook uses 4MB as the default row group size. The RCFfile library provides a parameter to allow users to select a size for their own table.

#### IV. THE API OF RCFFILE

RCFfile can be used in two ways. First, it can be used as the input or output of a MapReduce program, since it implements the standard interfaces of `InputFormat`, `RecordReader` and `OutputFormat`. This is the way how RCFfile is used in Hive. Second, it can also be directly accessed through its interfaces exposed by `RCFfile.Reader` and `RCFfile.Writer`.

##### A. Usage in a MapReduce Program

The implementations of `InputFormat`, `RecordReader` and `OutputFormat` in RCFfile are `RCFfileInputFormat`, `RCFfileRecordReader`, and `RCFfileOutputFormat`, respectively. When using RCFfile in a MapReduce program via these interfaces, there are two major configurations.

First, when using RCFfile to read a table, `RCFfileRecordReader` needs to know which columns of the table are needed. The following code example demonstrates how to set such information. After the setting, the configuration of `hive.io.file.readcolumn.ids` will be (2,3).

```
1 ArrayList<Integer> readCols = new ArrayList<Integer>();
2 readCols.add(Integer.valueOf(2));
3 readCols.add(Integer.valueOf(3));
4 ColumnProjectionUtils.setReadColumnIDs(conf, readCols);
```

Second, when using RCFfile as output, the number of columns needs to be specified in `RCFfileOutputFormat`. The following code example shows the methods to set and get the number of columns. The call of `RCFfileOutputFormat.setColumnNumber` will set the configuration value of `hive.io.rcfile.column.number.conf`. `RCFfileOutputFormat.getColumnNumber` can be used to get the number of columns.

```
1 RCFfileOutputFormat.setColumnNumber(conf, 8);
2 RCFfileOutputFormat.getColumnNumber(conf);
```

##### B. RCFfile Reader and Writer

RCFfile also provides Reader and Writer to allow applications to use RCFfile in their own ways. RCFfile Writer is simple since it only supports data appending.

The interfaces of RCFfile Reader are categorized into three sets: common utilities, row-wise reading, and column-wise reading. Table I summarizes these three categories. The following code example outlines how to use row-wise reading methods of RCFfile. Lines 1 to 6 define several critical variables. Lines 8 to 9 show the method to set needed columns. In this example, columns 2 and 3 are used. Line 10 uses method `setReadColumnIDs` to pass the information of needed columns to RCFfile. A RCFfile Reader is created at

---

```
1 ArrayList<Integer> readCols;
2 LongWritable rowID;
3 BytesRefArrayWritable cols;
4 FileSystem fs;
5 Path file;
6 Configuration conf;
7
8 readCols.add(2);
9 readCols.add(3);
10 setReadColumnIDs(conf, readCols);
11
12 Reader reader = new Reader(fs, file, conf);
13
14 while (reader.next(rowID))
15     reader.getCurrentRow(cols);
16
17 reader.close();
```

---

line 12, and then the while loop (lines 14-15) is used to read all the records one by one via the `getCurrentRow` method.

Notice that calling `getCurrentRow` does not mean that all the fields in the row have been decompressed. Actually, according to lazy decompression, a column will not be decompressed until one of its field is being serialized. The following code example shows the serialization of a field in column i. If this column has not been decompressed. This procedure will trigger lazy decompression. In the code, variable `standardWritableData` is the serialized field.

---

```
1 row = serDe.deserialize(cols);
2 oi = serDe.getObjectInspector();
3 fieldRefs = oi.getAllStructFieldRefs();
4 fieldData = oi.getStructFieldData(row, fieldRefs.get(i));
5 standardWritableData =
6     ObjectInspectorUtils.copyToStandardObject(
7         fieldData,
8         fieldRefs.get(i).getFieldObjectInspector(),
9         ObjectInspectorCopyOption.WRITABLE);
```

---

TABLE I: Interfaces of RCFfile.Reader

(a) Common Utilities

Method	Action
Reader()	Create RCFfile reader
getPosition()	Get the current byte offset from the beginning
close()	Close the reader

(b) Row-wise reading

Method	Action
next()	Return if there is more data available. If yes, advance the current reader pointer to the next record
getCurrentRow()	Return the current row

(c) Column-wise reading

Method	Action
getColumn()	Fetch an array of a specific column data of the current row group
nextColumnsBatch()	Return if there are more row groups available. If yes, discard all bytes of the current row group and advance the current reader pointer to the next row group

## V. PERFORMANCE EVALUATION

We have conducted experiments with RCFfile and Hive at Facebook to demonstrate the effectiveness of RCFfile. The experiments were done in a Facebook cluster with 40 nodes. Most nodes are equipped with an Intel Xeon CPU with 8 cores, 32GB main memory, and 12 1TB disks. The operating system is Linux (kernel version 2.6). In our experiments, we have used the Hadoop 0.20.1, and the most recent version of Hive.

We have conducted two groups of experiments. In the first group, we have compared RCFfile with row-store, column-store, and column-group. In the second group, we have evaluated the performance of RCFfile from the perspectives of both storage space and query execution, by using different row group sizes and different workloads.

### A. RCFfile versus Other Structures

In this group of experiments, we configured Hive to use several different data placement structures. Row-store is implemented as shown in Figure 1. For both column-store and column-group, we have used the Zebra library [20], which is widely used in the Pig system [6]. Zebra implements both column-store and column-group. To conduct the experiments, we have integrated the Zebra library into Hive with necessary modifications.

Our goal in this group of experiments is to demonstrate the effectiveness of RCFfile in three aspects:

- 1) data storage space,
- 2) data loading time, and
- 3) query execution time.

Our workload is from the the benchmark proposed by Palvo et al. [22]. This benchmark has been widely-used to evaluate different large-scale data processing platforms [23].

1) *Storage Space*: We selected the *USERVISITS* table from the benchmark, and generated the data set whose size is about 120GB. The generated data is all in plain text, and we loaded it into Hive using different data placement structures. During loading, data is compressed by the Gzip algorithm for each structure.

The table has nine columns: *sourceIP*, *destURL*, *visitDate*, *adRevenue*, *userAgent*, *countryCode*, *languageCode*, *searchWord*, and *duration*. Most of them are of string type. By column-group in Zebra, *sourceIP* and *adRevenue* are located in the same column group, in order to optimize the queries in the benchmark.

Figure 5 shows the storage space sizes required by the raw data and by several data placement structures. We can see that data compression can significantly reduce the storage space, and different data placement structures show different compression efficiencies as follows.

- Row-store has the worst compression efficiency compared with column-store, column-group, and RCFfile. This is expected because that a column-wise data compression is better than a row-wise data compression with mixed data domains.

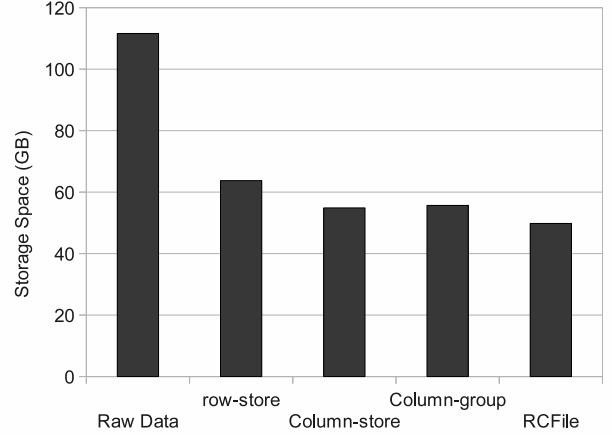


Fig. 5: Sizes (GB) of raw data and loaded tables with different data placement in Hive.

- RCFfile can reduce even more space than column-store does. This is because the implementation of Zebra stores column metadata and real column data together, so that it cannot compress them separately. Recall the RCFfile data layout in Section III-A, RCFfile uses two sections to store the real data of each column and the metadata about this column (mainly the length of each cell), and compress the two sections independently. Thus, RCFfile can have better data compression efficiency than that of Zebra.

2) *Data Loading Time*: For a data placement structure, data loading time (the time required by loading the raw data into the data warehouse) is an important factor for daily operations in Facebook. Reducing this time is critical since Facebook has to load about 20TB data into the production warehouse everyday.

We have recorded the data loading times for the *USERVISITS* table in the above experiment. The results are shown in Figure 6. We have the following observations:

- Among all cases, row-store always has the smallest data loading time. This is because it has the minimum overhead to re-organize records in the raw text file.
- Column-store and column-group have significantly long loading times than both row-store and RCFfile. This is because each record in the raw data file will be written to multiple HDFS blocks for different columns (or column groups). Since these multiple blocks are not in the same cluster node, data loading will cause much more network overhead than using other structures.
- RCFfile is slightly slower than row-store with a comparable performance in practice. This reflects the small overhead of RCFfile since it only needs to re-organize records inside each row group whose size is significantly smaller than the file size.

3) *Query execution time*: In this experiment, we executed two queries on the *RANKING* table from the benchmark. The table has three columns: *pageRank*, *pageURL*, and *avgDuration*. Both of column *pageRank* and *avgDuration* are of integer type, and *pageURL* is of string type. In Zebra, when using column-group, we organized *pageRank* and *pageURL* in the

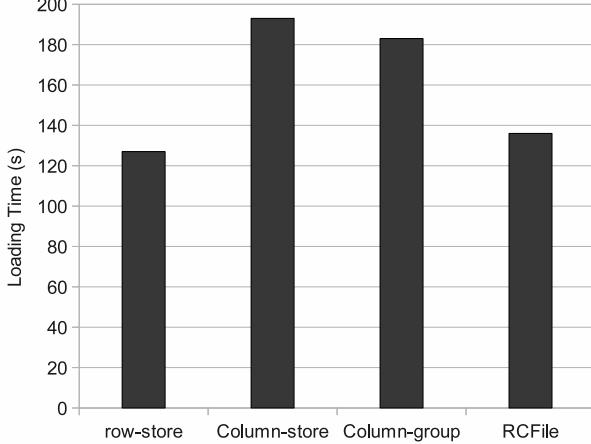


Fig. 6: Data loading times.

same column group. When using column-store, each of the three columns is stored independently. The queries are shown as follows.

Q1: `SELECT pagerank, pageurl FROM RANKING WHERE pagerank > 400;`

Q2: `SELECT pagerank, pageurl FROM RANKING WHERE pagerank < 400;`

In order to evaluate the performance of lazy decompression of RCFfile, the two queries were designed to have different selectivities according to their where conditions. It is about 5% for (*pagerank* > 400), and about 95% for (*pagerank* < 400).

Figure 7a and 7b show the execution times of the two queries with the four data placement structures. We have two major observations.

- For Q1, RCFfile outperforms the other three structures significantly. This is because the lazy decompression technique in RCFfile can accelerate the query execution with a low query selectivity.
- For Q2, column-group has the fastest query execution speed since the high selectivity of this query makes lazy decompression useless in this case. However, RCFfile still outperforms column-store. Note that the performance advantage of column-group is not free. It highly relies on pre-defined column combinations before query executions.

4) *Summary:* We have compared RCFfile with other data placement structures in three aspects of storage space, data loading time and query execution time. We show that each structure has its own merits for only one aspect. In contrast, our solution RCFfile, which adopts advantages of other structures, is the best choice in almost all the cases.

#### B. RCFfile with Different Row Group Sizes

In this group of experiments, we have examined how the row group size can affect data storage space and query execution time. We used two different workloads. The first workload

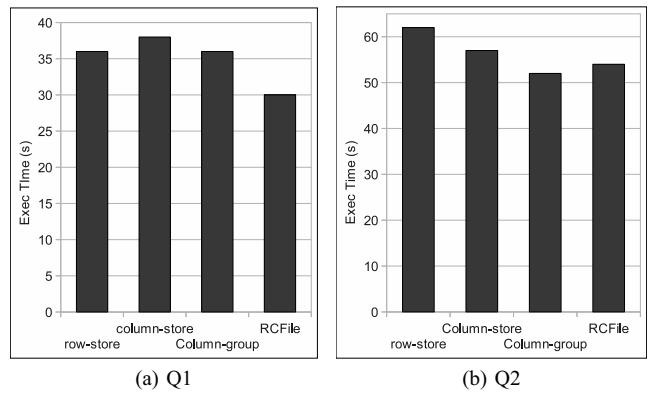


Fig. 7: Query execution times.

is the industry standard TPC-H benchmark for warehousing system evaluations. The second workload is generated by daily operations for advertisement business in Facebook.

1) *TPC-H workload:* We selected Q1 and Q6 from the benchmark as our queries. Both the two queries execute aggregations on the largest table *LINEITEM*. In our experiment, we examined the storage space for this table and query execution times with three different row group sizes (8KB, the default 4MB, and 16MB). The *LINEITEM* has 16 columns, and the two queries only use a small number of columns, and never touch the largest column *l\_comments*. In addition, we have also tested row-store structure for performance comparisons.

Figure 8 (a) shows the storage space for different configurations. We can see that with a small 8KB row group size, RCFfile needs more storage space than row-store. However, when increasing the row group size to 4MB and 16MB, RCFfile can significantly decrease storage space compared with row-store. In addition, in this Figure, we cannot see a big gap between the cases of 4MB and 16MB. This means that increasing row group size after a threshold would not help improve data compression efficiency significantly.

Figure 8 (b) and (c) show the query execution times of Q1 and Q6, respectively. We can see that RCFfile, no matter with what row group size, can significantly outperform row-store. This reflects the advantage of RCFfile over row-store, which skips unnecessary columns as column-store structure does.

Among the three row group sizes, the middle value (4MB) achieves the best performance. As shown in Figure 8 (a), since the 4MB row group size can decrease table size much more effectively than a small size of 8KB, it can significantly reduce the amount of disk accesses and thus accelerate query execution. It is also not surprisingly to find that the large 16MB row group size is slower than 4MB for both queries. First, as shown in Figure 8 (a), the two sizes (16MB and 4MB) have almost the same compression efficiency. Thus, the large row group size cannot gain further I/O space advantage. Second, according to the current RCFfile's implementation, it has more overhead to manage a large row group that must be decompressed and held in memory. Furthermore, a large row group can also decrease the advantage of lazy

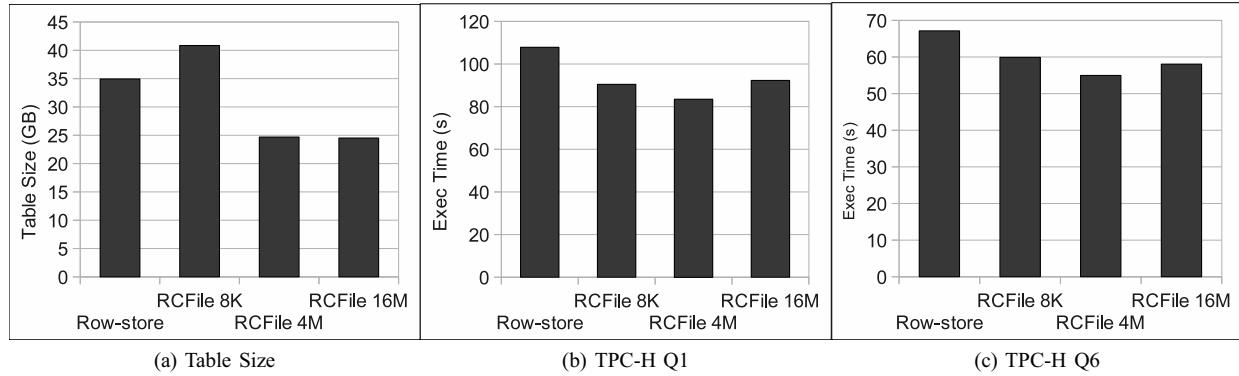


Fig. 8: The TPC-H workloads.

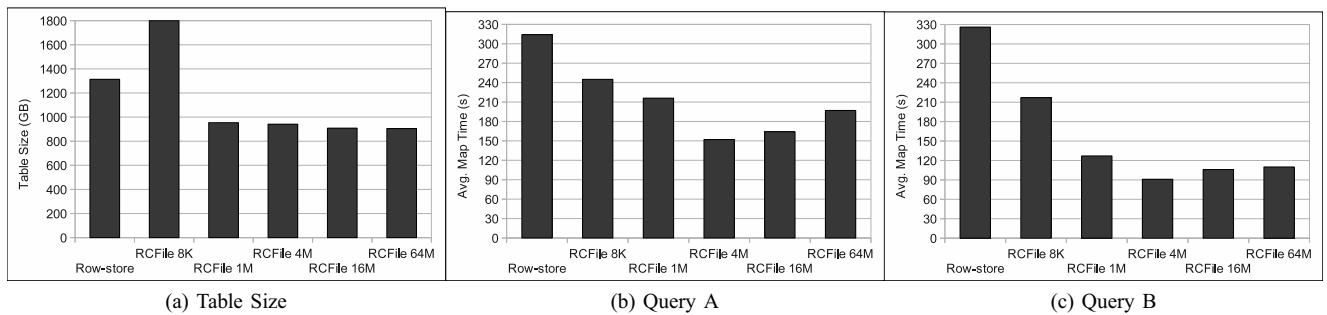


Fig. 9: The Facebook workload.

decompression, and cause unnecessary data decompression as we have discussed in Section III-D.

2) *Facebook workload*: We have further used a Facebook workload to examine how the row group size can affect RCFfile's performance. In this experiment, we examined different row group sizes (8KB, 1MB, 4MB, 16MB, and 64MB). The experiment was conducted on the table to store the advertisement click-stream in Facebook. The table (namely *adclicks*) is very wide with 38 columns. In this experiment, we used the trace collected in one day, and its size by row-store with compression is about 1.3TB. We used two queries as follows.

Query A: `SELECT adid, userid FROM adclicks;`

Query B: `SELECT adid, userid FROM adclicks WHERE userid="X";`

Figure 9 (a) shows the table sizes. Figure 9 (b) and (c) show the average mapper time of the MapReduce job for the query execution of Query A and Query B. The mapper time reflects the performance of the underlying RCFfile structure. From the viewpoints of both data compression and query execution times, these results are consistent with the previously presented TPC-H results. In summary, all these results show that a range of from 4MB (the default value) to 16MB is an effective selection as the row group size.

In addition, when comparing the query execution times of Query A and Query B in Figure 9 (b) and (c), we can have two

distinct observations. First, for row-store, the average mapper time of Query B is even longer than that of Query A. This is expected since Query B has a where condition that causes more computations. Second, however, for RCFfile with the row group size, the average mapper time of Query B is significantly shorter than that of Query A. This reflects the performance benefit of lazy decompression of RCFfile. Due to the existence of a where condition in Query B, a lazy decompression can avoid to decompress unnecessary data and thus improve query execution performance.

## VI. OTHER RELATED WORK

We have introduced and evaluated the row-store, the column-store/column-group, and the hybrid PAX store implemented in conventional database systems in Section II. Detailed comparison, analysis, and improvement of these three structures in data compression and query performance can be found in [17][15][24][18][12][11].

In the context of MapReduce-based data warehouse systems, Zebra [20] was the first effort to utilize the column-store/column-group in the Yahoo Pig system on top of Hadoop. As we have discussed in Section II and evaluated in Section V, Zebra's performance is highly dependent on how column groups have been pre-defined. The most recently related work to RCFfile is a data storage component in the *Cheetah* system [25]. Like RCFfile, the Cheetah system also first horizontally partitions a table into small units (each unit is called a *cell*),

and then vertically stores and compresses each column independently. However, Cheetah will further use Gzip to compress the whole cell. Thus, during query execution, Cheetah has to read from the storage and decompress the whole cell before processing any data in a cell. Compared to Cheetah, RCFFile can skip unnecessary column reads by independent column compression, and avoid unnecessary column decompression by the lazy decomposition technique.

Other related work includes the Google Bigtable system [26] and its open-source implementation Hbase [27] built on Hadoop. The major difference between RCFFile and Bigtable/Hbase is that RCFFile serves as a storage structure for the almost read-only data warehouse system, while Bigtable/Hbase is mainly a low-level key-value store for both read- and write-intensive applications.

## VII. CONCLUSION

A fast and space-efficient data placement structure is very important to big data analytics in large-scale distributed systems. According to our analysis on Facebook production systems, big data analytics in a MapReduce-based data warehouse has four critical requirements to the design and implementation of a data placement structure, namely 1) fast data loading, 2) fast query processing, 3) highly efficient storage space utilization, and 4) strong adaptivity to highly dynamic workload patterns. Our solution RCFFile is designed to meet all the four goals, and has been implemented on top of Hadoop. First, RCFFile has comparable data loading speed and workload adaptivity with the row-store. Second, RCFFile is read-optimized by avoiding unnecessary column reads during table scans. It outperforms the other structures in most of cases. Third, RCFFile uses column-wise compression and thus provides efficient storage space utilization.

RCFFile has been integrated into Hive, and plays an important role in daily Facebook operations. It is now used as the default storage structure option for internal data processing systems at Facebook. In order to improve storage space utilization, all the recent data generated by various Facebook applications since 2010 have been stored in the RCFFile structure, and the Facebook data infrastructure team is currently working to transform existing data sets stored in the row-store structure at Facebook into the RCFFile format.

RCFFile has been adopted in data processing systems beyond the scope of Facebook. An integration of RCFFile to Pig is being developed by Yahoo! RCFFile is used in another Hadoop-based data management system called Howl (<http://wiki.apache.org/pig/Howl>). In addition, according to communications in the Hive development community, RCFFile has been successfully integrated into other MapReduce-based data analytics platforms. We believe that RCFFile will continue to play its important role as a data placement standard for big data analytics in the MapReduce environment.

## VIII. ACKNOWLEDGMENTS

We thank Joydeep Sen Sarma for his initial thought about RCFFile. We appreciate Bill Bynum for his careful reading of

the paper. We thank anonymous reviewers for their feedback to improve the readability of the paper. This work is supported in part by China Basic Research Program (2011CB302500, 2011CB302800), the Co-building Program of Beijing Municipal Education Commission, and the US National Science Foundation under grants CCF072380 and CCF0913050.

## REFERENCES

- [1] <http://www.facebook.com/press/info.php?statistics>.
- [2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
- [3] <http://hadoop.apache.org/>.
- [4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a MapReduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.
- [5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.
- [6] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanan, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a highlevel dataflow system on top of MapReduce: The Pig experience," *PVLDB*, vol. 2, no. 2, pp. 1414–1425, 2009.
- [7] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *SIGMOD Conference*, 2010, pp. 1013–1020.
- [8] R. Ramakrishnan and J. Gehrke, "Database management systems," McGraw-Hill, 2003.
- [9] G. P. Copeland and S. Khoshafian, "A decomposition storage model," in *SIGMOD Conference*, 1985, pp. 268–279.
- [10] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented dbms," in *VLDB*, 2005, pp. 553–564.
- [11] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *SIGMOD Conference*, 2008.
- [12] A. L. Holloway and D. J. DeWitt, "Read-optimized databases, in depth," *PVLDB*, vol. 1, no. 1, pp. 502–513, 2008.
- [13] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB*, 2001, pp. 169–180.
- [14] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query processing techniques for solid state drives," in *SIGMOD Conference*, 2009, pp. 59–72.
- [15] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in *VLDB*, 2006, pp. 487–498.
- [16] M. Stonebraker and U. Çetintemel, "One size fits all: An idea whose time has come and gone (abstract)," in *ICDE*, 2005, pp. 2–11.
- [17] V. Raman and G. Swart, "How to wring a table dry: Entropy compression of relations and querying of compressed relations," in *VLDB*, 2006.
- [18] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *ICDE*, 2008, pp. 60–69.
- [19] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999.
- [20] <http://wiki.apache.org/pig/zebra>.
- [21] R. A. Hankins and J. M. Patel, "Data morphing: An adaptive, cache-conscious storage technique," in *VLDB*, 2003, pp. 417–428.
- [22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD Conference*, 2009, pp. 165–178.
- [23] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.
- [24] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt, "How to barter bits for chronons: compression and bandwidth trade offs for database scans," in *SIGMOD Conference*, 2007, pp. 389–400.
- [25] S. Chen, "Cheetah: A high performance, custom data warehouse on top of mapreduce," *PVLDB*, vol. 3, no. 2, pp. 1459–1468, 2010.
- [26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006, pp. 205–218.
- [27] <http://hbase.apache.org>.

---

# What is Data Science?

**The future belongs to the companies  
and people that turn data into products**



An O'Reilly Radar Report

By Mike Loukides

---

# Be at the forefront of the data revolution.

**February 28–March 1, 2012**  
**Santa Clara, CA**



**Strata offers the nuts-and-bolts of building a data-driven business.**

- See the latest tools and technologies you need to make data work
- Find new ways to leverage data across industries and disciplines
- Understand the career opportunities for data professionals
- Tracks include: Data Science, Business & Industry, Visualization & Interface, Hadoop & Big Data, Policy & Privacy, and Domain Data

Strata Conference is for developers, data scientists, data analysts, and other data professionals.

Registration is now open at [strataconf.com](http://strataconf.com)

Save 20% with code **REPORT20**

O'REILLY®

O'REILLY®

**Strata**  
Making Data Work

---

## Contents

Where data comes from.....	3
Working with data at scale.....	5
Making data tell its story .....	7
Data scientists.....	8

---

# What is Data Science?

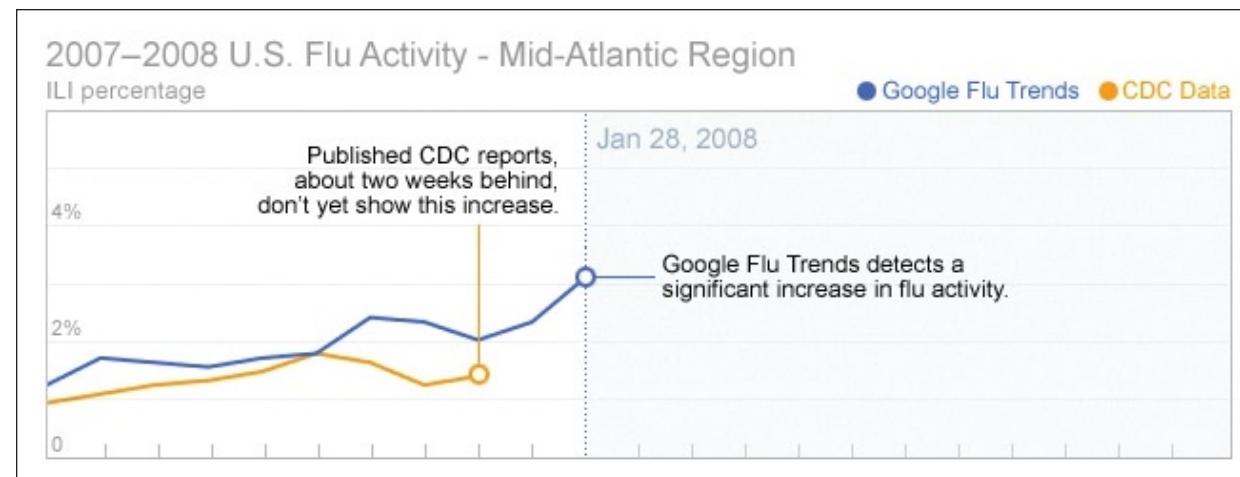
The Web is full of “data-driven apps.” Almost any e-commerce application is a data-driven application. There’s a database behind a web front end, and middleware that talks to a number of other databases and data services (credit card processing companies, banks, and so on). But merely using data isn’t really what we mean by “data science.” A data application acquires its value from the data itself, and creates more data as a result. It’s not just an application with data; it’s a data product. Data science enables the creation of data products.

One of the earlier data products on the Web was the Cddb database. The developers of Cddb realized that any CD had a unique signature, based on the exact length (in samples) of each track on the CD. Gracenote built a database of track lengths, and coupled it to a database of album metadata (track titles, artists, album titles). If you’ve ever used iTunes to rip a CD, you’ve taken advantage of this database. Before it does anything else, iTunes reads the length of every track, sends it to Cddb, and gets back the track titles. If you have a CD that’s not in the database (including a CD you’ve made yourself), you can create an entry for an unknown album. While this sounds simple enough, it’s revolutionary: Cddb views music as data, not as audio, and creates new value in doing so. Their business is fundamentally different from selling music, sharing music, or analyzing musical tastes (though these can also be “data products”). Cddb arises entirely from viewing a musical problem as a data problem.

Google is a master at creating data products. Here are a few examples:

- Google’s breakthrough was realizing that a search engine could use input other than the text on the page. Google’s PageRank algorithm was among the first to use data outside of the page itself, in particular, the number of links pointing to a page. Tracking links made Google searches much more useful, and PageRank has been a key ingredient to the company’s success.
- Spell checking isn’t a terribly difficult problem, but by suggesting corrections to misspelled searches, and observing what the user clicks in response, Google made it much more accurate. They’ve built a dictionary of common misspellings, their corrections, and the contexts in which they occur.
- Speech recognition has always been a hard problem, and it remains difficult. But Google has made huge strides by using the voice data they’ve collected, and has been able to integrate voice search into their core search engine.
- During the Swine Flu epidemic of 2009, Google was able to track the progress of the epidemic by following searches for flu-related topics.

## Flu trends



Google was able to spot trends in the Swine Flu epidemic roughly two weeks before the Center for Disease Control by analyzing searches that people were making in different regions of the country.

Google isn't the only company that knows how to use data. Facebook and LinkedIn use patterns of friendship relationships to suggest other people you may know, or should know, with sometimes frightening accuracy. Amazon saves your searches, correlates what you search for with what other users search for, and uses it to create surprisingly appropriate recommendations. These recommendations are "data products" that help to drive Amazon's more traditional retail business. They come about because Amazon understands that a book isn't just a book, a camera isn't just a camera, and a customer isn't just a customer; customers generate a trail of "data exhaust" that can be mined and put to use, and a camera is a cloud of data that can be correlated with the customers' behavior, the data they leave every time they visit the site.

The thread that ties most of these applications together is that data collected from users provides added value. Whether that data is search terms, voice samples, or product reviews, the users are in a feedback loop in which they contribute to the products they use. That's the beginning of data science.

In the last few years, there has been an explosion in the amount of data that's available. Whether we're talking about web server logs, tweet streams, online transaction records, "citizen science," data from sensors, government data, or some other source, the problem isn't finding data, it's figuring out what to do with it. And it's not just companies using their own data, or the data contributed by their users. It's increasingly common to mashup data from a number of sources. *Data Mashups in R* analyzes mortgage foreclosures in Philadelphia County by taking a public report from the county sheriff's office, extracting addresses and using Yahoo! to convert the addresses to latitude and longitude, then using the geographical data to place the foreclosures on a map (another data source), and group them by neighborhood, valuation, neighborhood per-capita income, and other socio-economic factors.

The question facing every company today, every startup, every non-profit, every project site that wants to attract a community, is how to use data effectively—not just their own data, but all the data that's available and relevant. Using data effectively requires something different from traditional statistics, where actuaries in business

suits perform arcane but fairly well-defined kinds of analysis. What differentiates data science from statistics is that data science is a holistic approach. We're increasingly finding data in the wild, and data scientists are involved with gathering data, massaging it into a tractable form, making it tell its story, and presenting that story to others.

To get a sense for what skills are required, let's look at the data life cycle: where it comes from, how you use it, and where it goes.

## Where data comes from

Data is everywhere: your government, your web server, your business partners, even your body. While we aren't drowning in a sea of data, we're finding that almost everything can (or has) been instrumented. At O'Reilly, we frequently combine publishing industry data from Nielsen BookScan with our own sales data, publicly available Amazon data, and even job data to see what's happening in the publishing industry. Sites like Infochimps and Factual provide access to many large datasets, including climate data, MySpace activity streams, and game logs from sporting events. Factual enlists users to update and improve its datasets, which cover topics as diverse as endocrinologists to hiking trails.

Much of the data we currently work with is the direct consequence of Web 2.0, and of Moore's Law applied to data. The Web has people spending more time online, and leaving a trail of data wherever they go. Mobile applications leave an even richer data trail, since many of them are annotated with geolocation, or involve video or audio, all of which can be mined. Point-of-sale devices and frequent-shopper's cards make it possible to capture all of your retail transactions, not just the ones you make online. All of this data would be useless if we couldn't store it, and that's where Moore's Law comes in. Since the early '80s, processor speed has increased from 10 MHz to 3.6 GHz—an increase of 360 (not counting increases in word length and number of cores). But we've seen much bigger increases in storage capacity, on every level. RAM

has moved from \$1,000/MB to roughly \$25/GB—a price reduction of about 40000, to say nothing of the reduction in size and increase in speed. Hitachi made the first gigabyte disk drives in 1982, weighing in at roughly 250 pounds; now terabyte drives are consumer equipment, and a 32 GB microSD card weighs about half a gram. Whether you look at bits per gram, bits per dollar, or raw capacity, storage has more than kept pace with the increase of CPU speed.



One of the first commercial disk drives from IBM. It has a 5 MB capacity and it's stored in a cabinet roughly the size of a luxury refrigerator. In contrast, a 32 GB microSD card measures around 5/8 x 3/8 inch and weighs about 0.5 gram.

(Photo: Mike Loukides. Disk drive on display at IBM Almaden Research)

---

The importance of Moore's law as applied to data isn't just geek pyrotechnics. Data expands to fill the space you have to store it. The more storage is available, the more data you will find to put into it. The data exhaust you leave behind whenever you surf the Web, friend someone on Facebook, or make a purchase in your local supermarket, is all carefully collected and analyzed. Increased storage capacity demands increased sophistication in the analysis and use of that data. That's the foundation of data science.

So, how do we make that data useful? The first step of any data analysis project is "data conditioning," or getting data into a state where it's usable. We are seeing more data in formats that are easier to consume: Atom data feeds, web services, microformats, and other newer technologies provide data in formats that's directly machine-consumable. But old-style screen scraping hasn't died, and isn't going to die. Many sources of "wild data" are extremely messy. They aren't well-behaved XML files with all the metadata nicely in place. The foreclosure data used in *Data Mashups in R* was posted on a public website by the Philadelphia county sheriff's office. This data was presented as an HTML file that was probably generated automatically from a spreadsheet. If you've ever seen the HTML that's generated by Excel, you know that's going to be fun to process.

Data conditioning can involve cleaning up messy HTML with tools like BeautifulSoup, natural language processing to parse plain text in English and other languages, or even getting humans to do the dirty work. You're likely to be dealing with an array of data sources, all in different forms. It would be nice if there was a standard set of tools to do the job, but there isn't. To do data conditioning, you have to be ready for whatever comes, and be willing to use anything from ancient Unix utilities such as *awk* to XML parsers and machine learning libraries. Scripting languages, such as Perl and Python, are essential.

Once you've parsed the data, you can start thinking about the quality of your data. Data is frequently missing or incongruous. If data is missing, do you simply ignore

the missing points? That isn't always possible. If data is incongruous, do you decide that something is wrong with badly behaved data (after all, equipment fails), or that the incongruous data is telling its own story, which may be more interesting? It's reported that the discovery of ozone layer depletion was delayed because automated data collection tools discarded readings that were too low<sup>1</sup>. In data science, what you have is frequently all you're going to get. It's usually impossible to get "better" data, and you have no alternative but to work with the data at hand.

If the problem involves human language, understanding the data adds another dimension to the problem. Roger Magoulas, who runs the data analysis group at O'Reilly, was recently searching a database for Apple job listings requiring geolocation skills. While that sounds like a simple task, the trick was disambiguating "Apple" from many job postings in the growing Apple industry. To do it well you need to understand the grammatical structure of a job posting; you need to be able to parse the English. And that problem is showing up more and more frequently. Try using Google Trends to figure out what's happening with the Cassandra database or the Python language, and you'll get a sense of the problem. Google has indexed many, many websites about large snakes. Disambiguation is never an easy task, but tools like the Natural Language Toolkit library can make it simpler.

When natural language processing fails, you can replace artificial intelligence with human intelligence. That's where services like Amazon's Mechanical Turk come in. If you can split your task up into a large number of subtasks that are easily described, you can use Mechanical Turk's marketplace for cheap labor. For example, if you're looking at job listings, and want to know which originated with Apple, you can have real people do the classification for roughly \$0.01 each. If you have already reduced the set to 10,000 postings with the word "Apple," paying humans \$0.01 to classify them only costs \$100.

## Working with data at scale

We've all heard a lot about "big data," but "big" is really a red herring. Oil companies, telecommunications companies, and other data-centric industries have had huge datasets for a long time. And as storage capacity continues to expand, today's "big" is certainly tomorrow's "medium" and next week's "small." The most meaningful definition I've heard: "big data" is when the size of the data itself becomes part of the problem. We're discussing data problems ranging from gigabytes to petabytes of data. At some point, traditional techniques for working with data run out of steam.

What are we trying to do with data that's different? According to Jeff Hammerbacher<sup>2</sup> (@hackingdata), we're trying to build information platforms or dataspaces. Information platforms are similar to traditional data warehouses, but different. They expose rich APIs, and are designed for exploring and understanding the data rather than for traditional analysis and reporting. They accept all data formats, including the most messy, and their schemas evolve as the understanding of the data changes.

Most of the organizations that have built data platforms have found it necessary to go beyond the relational database model. Traditional relational database systems stop being effective at this scale. Managing sharding and replication across a horde of database servers is difficult and slow. The need to define a schema in advance conflicts with reality of multiple, unstructured data sources, in which you may not know what's important until after you've analyzed the data. Relational databases are designed for consistency, to support complex transactions that can easily be rolled back if any one of a complex set of operations fails. While rock-solid consistency is crucial to many applications, it's not really necessary for the kind of analysis we're discussing here. Do you really care if you have 1,010 or 1,012 Twitter followers? Precision has an allure, but in most data-driven applications outside of finance, that allure is deceptive. Most data analysis is comparative: if you're asking whether sales to Northern Europe are increasing faster than sales to Southern Europe, you aren't concerned about the difference between 5.92 percent annual growth and 5.93 percent.

To store huge datasets effectively, we've seen a new breed of databases appear. These are frequently called NoSQL databases, or Non-Relational databases, though neither term is very useful. They group together fundamentally dissimilar products by telling you what they aren't. Many of these databases are the logical descendants of Google's BigTable and Amazon's Dynamo, and are designed to be distributed across many nodes, to provide "eventual consistency" but not absolute consistency, and to have very flexible schema. While there are two dozen or so products available (almost all of them open source), a few leaders have established themselves:

- **Cassandra:** Developed at Facebook, in production use at Twitter, Rackspace, Reddit, and other large sites. Cassandra is designed for high performance, reliability, and automatic replication. It has a very flexible data model. A new startup, Riptano, provides commercial support.
- **HBase:** Part of the Apache Hadoop project, and modelled on Google's BigTable. Suitable for extremely large databases (billions of rows, millions of columns), distributed across thousands of nodes. Along with Hadoop, commercial support is provided by Cloudera.

Storing data is only part of building a data platform, though. Data is only useful if you can do something with it, and enormous datasets present computational problems. Google popularized the MapReduce approach, which is basically a divide-and-conquer strategy for distributing an extremely large problem across an extremely large computing cluster. In the "map" stage, a programming task is divided into a number of identical subtasks, which are then distributed across many processors; the intermediate results are then combined by a single reduce task. In hindsight, MapReduce seems like an obvious solution to Google's biggest problem, creating large searches. It's easy to distribute a search across thousands of processors, and then combine the results into a single set of answers. What's less obvious is that MapReduce has proven to be widely applicable to many large data problems, ranging from search to machine learning.

---

The most popular open source implementation of MapReduce is the Hadoop project. Yahoo!'s claim that they had built the world's largest production Hadoop application, with 10,000 cores running Linux, brought it onto center stage. Many of the key Hadoop developers have found a home at Cloudera, which provides commercial support. Amazon's Elastic MapReduce makes it much easier to put Hadoop to work without investing in racks of Linux machines, by providing preconfigured Hadoop images for its EC2 clusters. You can allocate and de-allocate processors as needed, paying only for the time you use them.

Hadoop goes far beyond a simple MapReduce implementation (of which there are several); it's the key component of a data platform. It incorporates HDFS, a distributed filesystem designed for the performance and reliability requirements of huge datasets; the HBase database; Hive, which lets developers explore Hadoop datasets using SQL-like queries; a high-level dataflow language called Pig; and other components. If anything can be called a one-stop information platform, Hadoop is it.

Hadoop has been instrumental in enabling "agile" data analysis. In software development, "agile practices" are associated with faster product cycles, closer interaction between developers and consumers, and testing. Traditional data analysis has been hampered by extremely long turn-around times. If you start a calculation, it might not finish for hours, or even days. But Hadoop (and particularly Elastic MapReduce) make it easy to build clusters that can perform computations on large datasets quickly. Faster computations make it easier to test different assumptions, different datasets, and different algorithms. It's easier to consult with clients to figure out whether you're asking the right questions, and it's possible to pursue intriguing possibilities that you'd otherwise have to drop for lack of time.

Hadoop is essentially a batch system, but Hadoop Online Prototype (HOP) is an experimental project that enables stream processing. Hadoop processes data as it arrives, and delivers intermediate results in (near) real-time. Near real-time data analysis enables features like trending topics on sites like Twitter. These features only

require soft real-time; reports on trending topics don't require millisecond accuracy. As with the number of followers on Twitter, a "trending topics" report only needs to be current to within five minutes—or even an hour. According to Hilary Mason (@hmason), data scientist at bitly, it's possible to precompute much of the calculation, then use one of the experiments in real-time MapReduce to get presentable results.

Machine learning is another essential tool for the data scientist. We now expect web and mobile applications to incorporate recommendation engines, and building a recommendation engine is a quintessential artificial intelligence problem. You don't have to look at many modern web applications to see classification, error detection, image matching (behind Google Goggles and SnapTell) and even face detection—an ill-advised mobile application lets you take someone's picture with a cell phone, and look up that person's identity using photos available online. Andrew Ng's Machine Learning course at <http://www.youtube.com/watch?v=UzxYlbK2c7E> is one

of the most popular courses in computer science at Stanford, with hundreds of students.

There are many libraries available for machine learning: PyBrain in Python, Elefant, Weka in Java, and Mahout (coupled to Hadoop). Google has just announced their Prediction API, which exposes their machine learning algorithms for public use via a RESTful interface. For computer vision, the OpenCV library is a de-facto standard.

Mechanical Turk is also an important part of the toolbox. Machine learning almost always requires a "training set," or a significant body of known data with which to develop and tune the application. The Turk is an excellent way to develop training sets. Once you've collected your training data (perhaps a large collection of public photos from Twitter), you can have humans classify them inexpensively—possibly sorting them into categories, possibly drawing circles around faces, cars, or whatever interests you. It's an excellent way to classify a few thousand data points at a cost of a few cents each. Even a relatively large job only costs a few hundred dollars.

While I haven't stressed traditional statistics, building statistical models plays an important role in any data analysis. According to Mike Driscoll (@dataspora), statistics is the "grammar of data science." It is crucial to "making data speak coherently." We've all heard the joke that eating pickles causes death, because everyone who dies has eaten pickles. That joke doesn't work if you understand what correlation means. More to the point, it's easy to notice that one advertisement for *R in a Nutshell* generated 2 percent more conversions than another. But it takes statistics to know whether this difference is significant, or just a random fluctuation. Data science isn't just about the existence of data, or making guesses about what that data might mean; it's about testing hypotheses and making sure that the conclusions you're drawing from the data are valid. Statistics plays a role in everything from traditional business intelligence (BI) to understanding how Google's ad auctions work. Statistics has become a basic skill. It isn't superseded by newer techniques from machine learning and other disciplines; it complements them.

While there are many commercial statistical packages, the open source R language—and its comprehensive package library, CRAN—is an essential tool. Although R is an odd and quirky language, particularly to someone with a background in computer science, it comes close to providing "one-stop shopping" for most statistical work. It has excellent graphics facilities; CRAN includes parsers for many kinds of data; and newer extensions extend R into distributed computing. If there's a single tool that provides an end-to-end solution for statistics work, R is it.

### Making data tell its story

A picture may or may not be worth a thousand words, but a picture is certainly worth a thousand numbers. The problem with most data analysis algorithms is that they generate a set of numbers. To understand what the numbers mean, the stories they are really telling, you need to generate a graph. Edward Tufte's *Visual Display of Quantitative Information* is the classic for data visualization, and a foundational text for anyone practicing data science.

But that's not really what concerns us here. Visualization is crucial to each stage of the data scientist. According to Martin Wattenberg (@wattenberg, founder of Flowing Media), visualization is key to data conditioning: if you want to find out just how bad your data is, try plotting it. Visualization is also frequently the first step in analysis. Hilary Mason says that when she gets a new data set, she starts by making a dozen or more scatter plots, trying to get a sense of what might be interesting. Once you've gotten some hints at what the data might be saying, you can follow it up with more detailed analysis.

There are many packages for plotting and presenting data. GnuPlot is very effective; R incorporates a fairly comprehensive graphics package; Casey Reas' and Ben Fry's Processing is the state of the art, particularly if you need to create animations that show how things change over time. At IBM's Many Eyes, many of the visualizations are full-fledged interactive applications.

Nathan Yau's FlowingData blog is a great place to look for creative visualizations. One of my favorites is the animation of the growth of Walmart over time <http://flowingdata.com/2010/04/07/watching-the-growth-of-walmart-now-with-100-more-sams-club/>. And this is one place where "art" comes in: not just the aesthetics of the visualization itself, but how you understand it. Does it look like the spread of cancer throughout a body? Or the spread of a flu virus through a population? Making data tell its story isn't just a matter of presenting results; it involves making connections, then going back to other data sources to verify them. Does a successful retail chain spread like an epidemic, and if so, does that give us new insights into how economies work? That's not a question we could even have asked a few years ago. There was insufficient computing power, the data was all locked up in proprietary sources, and the tools for working with the data were insufficient. It's the kind of question we now ask routinely.

## Data scientists

Data science requires skills ranging from traditional computer science to mathematics to art. Describing the data science group he put together at Facebook (possibly the first data science group at a consumer-oriented web property), Jeff Hammerbacher said:

*...on any given day, a team member could author a multistage processing pipeline in Python, design a hypothesis test, perform a regression analysis over data samples with R, design and implement an algorithm for some data-intensive product or service in Hadoop, or communicate the results of our analyses to other members of the organization<sup>3</sup>*

Where do you find the people this versatile? According to DJ Patil, chief scientist at LinkedIn (@dpatil), the best data scientists tend to be “hard scientists,” particularly physicists, rather than computer science majors. Physicists have a strong mathematical background, computing skills, and come from a discipline in which survival depends on getting the most from the data. They have to think about the big picture, the big problem. When you’ve just spent a lot of grant money generating data, you can’t just throw the data out if it isn’t as clean as you’d like. You have to make it tell its story. You need some creativity for when the story the data is telling isn’t what you think it’s telling.

Scientists also know how to break large problems up into smaller problems. Patil described the process of creating the group recommendation feature at LinkedIn. It would have been easy to turn this into a high-ceremony development project that would take thousands of hours of developer time, plus thousands of hours of computing time to do massive correlations across LinkedIn’s membership. But the process worked quite differently: it started out with a relatively small, simple program that looked at members’ profiles and made recommendations accordingly. Asking things like, did you go to Cornell? Then you might like to join the Cornell Alumni group. It then branched out incrementally. In addition to looking at

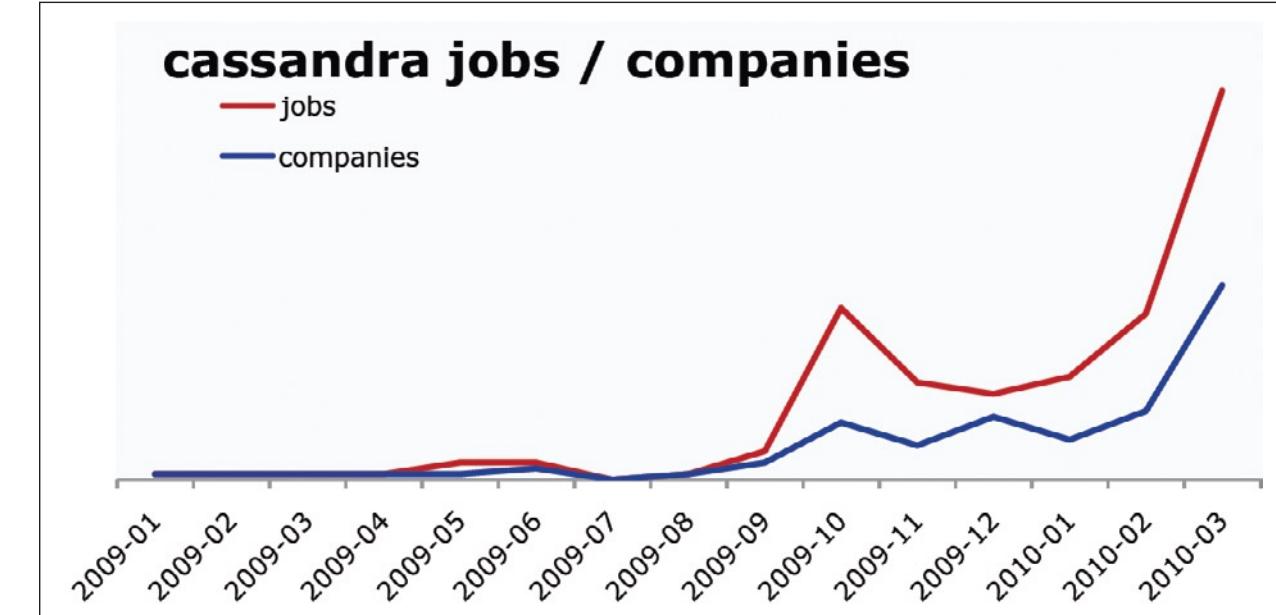
profiles, LinkedIn’s data scientists started looking at events that members attended. Then at books members had in their libraries. The result was a valuable data product that analyzed a huge database—but it was never conceived as such. It started small, and added value iteratively. It was an agile, flexible process that built toward its goal incrementally, rather than tackling a huge mountain of data all at once.

This is the heart of what Patil calls “data jiu-jitsu”—using smaller auxiliary problems to solve a large, difficult problem that appears intractable. CDDB is a great example of data jiu-jitsu: identifying music by analyzing an audio stream directly is a very difficult problem (though not unsolvable—see midomi, for example). But the CDDB staff used data creatively to solve a much more tractable problem that gave them the same result. Computing a signature based on track lengths, and then looking up that signature in a database, is trivially simple.

Entrepreneurship is another piece of the puzzle. Patil’s first flippant answer to “what kind of person are you looking for when you hire a data scientist?” was “someone you would start a company with.” That’s an important insight: we’re entering the era of products that are built on data. We don’t yet know what those products are, but we do know that the winners will be the people, and the companies, that find those products. Hilary Mason came to the same conclusion. Her job as scientist at bit.ly is really to investigate the data that bit.ly is generating, and find out how to build interesting products from it. No one in the nascent data industry is trying to build the 2012 Nissan Stanza or Office 2015; they’re all trying to find new products. In addition to being physicists, mathematicians, programmers, and artists, they’re entrepreneurs.

Data scientists combine entrepreneurship with patience, the willingness to build data products incrementally, the ability to explore, and the ability to iterate over a solution. They are inherently interdisciplinary. They can tackle all aspects of a problem, from initial data collection and data conditioning to drawing conclusions. They can

## Hiring trends for data science



It's not easy to get a handle on jobs in data science. However, data from O'Reilly Research shows a steady year-over-year increase in Hadoop and Cassandra job listings, which are good proxies for the “data science” market as a whole. This graph shows the increase in Cassandra jobs, and the companies listing Cassandra positions, over time.

think outside the box to come up with new ways to view the problem, or to work with very broadly defined problems: “here’s a lot of data, what can you make from it?”

The future belongs to the companies who figure out how to collect and use data successfully. Google, Amazon, Facebook, and LinkedIn have all tapped into their datastreams and made that the core of their success. They were the vanguard, but newer companies like bit.ly are following their path. Whether it’s mining your personal biology, building maps from the shared experi-

ence of millions of travellers, or studying the URLs that people pass to others, the next generation of successful businesses will be built around data. The part of Hal Varian’s quote that nobody remembers says it all:

**The ability to take data—to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it—that’s going to be a hugely important skill in the next decades.**

Data is indeed the new Intel Inside.

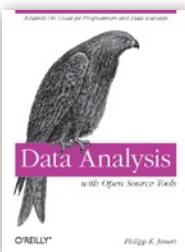
<sup>1</sup> The NASA article denies this, but also says that in 1984, they decided that the low values (which went back to the ’70s) were “real.” Whether humans or software decided to ignore anomalous data, it appears that data was ignored.

<sup>2</sup> Information Platforms as Dataspaces, by Jeff Hammerbacher (in *Beautiful Data*)

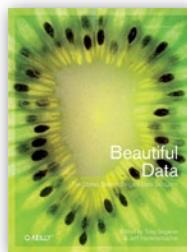
<sup>3</sup> Information Platforms as Dataspaces, by Jeff Hammerbacher (in *Beautiful Data*)

---

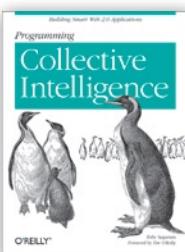
O'Reilly publications related to data science—  
found at [oreilly.com](http://oreilly.com)

**Data Analysis with Open Source Tools**

This book shows you how to think about data and the results you want to achieve with it.

**Beautiful Data**

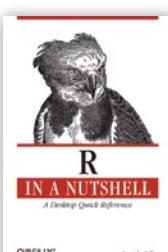
Learn from the best data practitioners in the field about how wide-ranging—and beautiful—working with data can be.

**Programming Collective Intelligence**

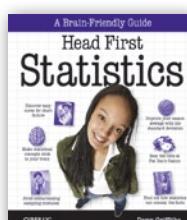
Learn how to build web applications that mine the data created by people on the Internet.

**Beautiful Visualization**

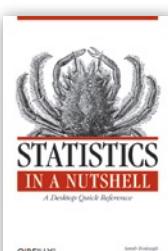
This book demonstrates why visualizations are beautiful not only for their aesthetic design, but also for elegant layers of detail.

**R in a Nutshell**

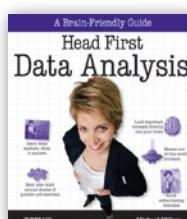
A quick and practical reference to learn what is becoming the standard for developing statistical software.

**Head First Statistics**

This book teaches statistics through puzzles, stories, visual aids, and real-world examples.

**Statistics in a Nutshell**

An introduction and reference for anyone with no previous background in statistics.

**Head First Data Analysis**

Learn how to collect your data, sort the distractions from the truth, and find meaningful patterns.

# The Dangers of Replication and a Solution

Jim Gray

Pat Helland

Patrick O'Neil (UMB)

Dennis Shasha (NYU)

May 1996

Technical Report

MSR-TR-96-17

Microsoft Research  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA 98052

This paper appeared in the proceedings of the 1996 ACM SIGMOD Conference at Montreal, pp. 173-182

**ACM:**

Copyright ©1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

# The Dangers of Replication and a Solution

Jim Gray (Gray@Microsoft.com)  
Pat Helland (PHelland@Microsoft.com)  
Patrick O’Neil (POneil@cs.UMB.edu)  
Dennis Shasha (Shasha@cs.NYU.edu)

**Abstract:** *Update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up: a ten-fold increase in nodes and traffic gives a thousand fold increase in deadlocks or reconciliations. Master copy replication (primary copy) schemes reduce this problem. A simple analytic model demonstrates these results. A new two-tier replication algorithm is proposed that allows mobile (disconnected) applications to propose tentative update transactions that are later applied to a master copy. Commutative update transactions avoid the instability of other replication schemes.*

## 1. Introduction

Data is replicated at multiple network nodes for performance and availability. **Eager replication** keeps all replicas exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction. Eager replication gives serializable execution – there are no concurrency anomalies. But, eager replication reduces update performance and increases transaction response times because extra updates and messages are added to the transaction.

Eager replication is not an option for mobile applications where most nodes are normally disconnected. Mobile applications require **lazy replication** algorithms that asynchronously propagate replica updates to other nodes after the updating transaction commits. Some continuously connected systems use lazy replication to improve response time.

Lazy replication also has shortcomings, the most serious being stale data versions. When two transactions read and write data concurrently, one transaction’s updates should be serialized after the other’s. This avoids concurrency anomalies. Eager replication typically uses a locking scheme to detect and regulate concurrent execution. Lazy replication schemes typically use a multi-version concurrency control scheme to detect non-serializable behavior [Bernstein, Hadzilacos, Goodman], [Berenson, et. al.]. Most multi-version isolation schemes provide the transaction with the most recent committed value. Lazy replication may allow a transaction to see a very old committed value. Committed updates to a local value may be “in transit” to this node if the update strategy is “lazy”.

---

Permission to make digital/hard copy of part or all of this material is granted provided that copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication, and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, to republish, requires a fee and/or specific permission.

Eager replication delays or aborts an uncommitted transaction if committing it would violate serialization. Lazy replication has a more difficult task because some replica updates have already been committed when the serialization problem is first detected. There is usually no automatic way to reverse the committed replica updates, rather a program or person must *reconcile* conflicting transactions.

To make this tangible, consider a joint checking account you share with your spouse. Suppose it has \$1,000 in it. This account is replicated in three places: your checkbook, your spouse’s checkbook, and the bank’s ledger.

Eager replication assures that all three books have the same account balance. It prevents you and your spouse from writing checks totaling more than \$1,000. If you try to overdraw your account, the transaction will fail.

Lazy replication allows both you and your spouse to write checks totaling \$1,000 for a total of \$2,000 in withdrawals. When these checks arrived at the bank, or when you communicated with your spouse, someone or something reconciles the transactions that used the virtual \$1,000.

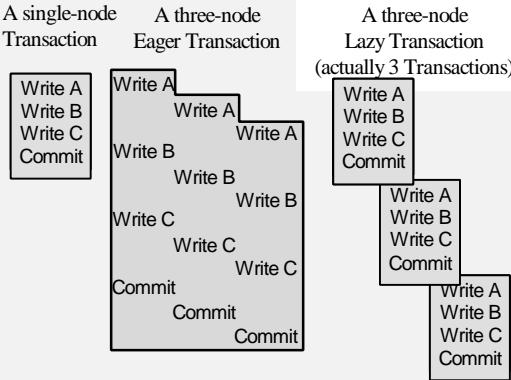
It would be nice to automate this reconciliation. The bank does that by rejecting updates that cause an overdraft. This is a master replication scheme: the bank has the master copy and only the bank’s updates really count. Unfortunately, this works only for the bank. You, your spouse, and your creditors are likely to spend considerable time reconciling the “extra” thousand dollars worth of transactions. In the meantime, your books will be inconsistent with the bank’s books. That makes it difficult for you to perform further banking operations.

The database for a checking account is a single number, and a log of updates to that number. It is the simplest database. In reality, databases are more complex and the serialization issues are more subtle.

*The theme of this paper is that update-anywhere-anytime-anyway replication is unstable.*

1. *If the number of checkbooks per account increases by a factor of ten, the deadlock or reconciliation rates rises by a factor of a thousand.*
2. *Disconnected operation and message delays mean lazy replication has more frequent reconciliation.*

**Figure 1:** When replicated, a simple single-node transaction may apply its updates remotely either as part of the same transaction (*eager*) or as separate transactions (*lazy*). In either case, if data is replicated at  $N$  nodes, the transaction does  $N$  times as much work



Simple replication works well at low loads and with a few nodes. This creates a *scaleup pitfall*. A prototype system demonstrates well. Only a few transactions deadlock or need reconciliation when running on two connected nodes. But the system behaves very differently when the application is scaled up to a large number of nodes, or when nodes are disconnected more often, or when message propagation delays are longer. Such systems have higher transaction rates. Suddenly, the deadlock and reconciliation rate is astronomically higher (cubic growth is predicted by the model). The database at each node diverges further and further from the others as reconciliation fails. Each reconciliation failure implies differences among nodes. Soon, the system suffers *system delusion* — the database is inconsistent and there is no obvious way to repair it [Gray & Reuter, pp. 149-150].

This is a bleak picture, but probably accurate. Simple replication (transactional update-anywhere-anytime-anyway) cannot be made to work with global serializability.

In outline, the paper gives a simple model of replication and a closed-form average-case analysis for the probability of waits, deadlocks, and reconciliations. For simplicity, the model ignores many issues that would make the predicted behavior even worse. In particular, it ignores the message propagation delays needed to broadcast replica updates. It ignores “true” serialization, and assumes a weak multi-version form of committed-read serialization (no read locks) [Berenson]. The paper then considers object master replication. Unrestricted lazy master replication has many of the instability problems of eager and group replication.

A restricted form of replication avoids these problems: *two-tier replication* has *base nodes* that are always connected, and *mobile nodes* that are usually disconnected.

1. Mobile nodes propose tentative update transactions to objects owned by other nodes. Each mobile node keeps two object versions: a local version and a best known master version.

2. Mobile nodes occasionally connect to base nodes and propose tentative update transactions to a master node. These proposed transactions are re-executed and may succeed or be rejected. To improve the chances of success, tentative transactions are designed to commute with other transactions. After exchanges the mobile node’s database is synchronized with the base nodes. Rejected tentative transactions are reconciled by the mobile node owner who generated the transaction.

Our analysis shows that this scheme supports lazy replication and mobile computing but avoids system delusion: tentative updates may be rejected but the base database state remains consistent.

## 2. Replication Models

Figure 1 shows two ways to propagate updates to replicas:

1. **Eager:** Updates are applied to all replicas of an object as part of the original transaction.
2. **Lazy:** One replica is updated by the originating transaction. Updates to other replicas propagate asynchronously, typically as a separate transaction for each node.

**Figure 2:** Updates may be controlled in two ways. Either all updates emanate from a master copy of the object, or updates may emanate from any. Group ownership has many more chances for conflicting updates.

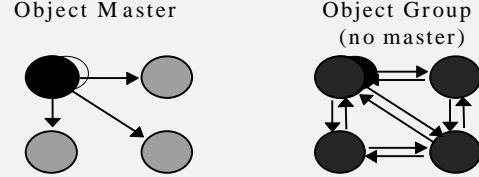


Figure 2 shows two ways to regulate replica updates:

1. **Group:** Any node with a copy of a data item can update it. This is often called *update anywhere*.
2. **Master:** Each object has a master node. Only the master can update the *primary copy* of the object. All other replicas are read-only. Other nodes wanting to update the object request the master do the update.

**Table 1:** A taxonomy of replication strategies contrasting propagation strategy (eager or lazy) with the ownership strategy (master or group).

Propagation vs. Ownership	Lazy	Eager
Group	N transactions N object owners	one transaction N object owners
Master	N transactions one object owner	one transaction one object owner
Two Tier	N+1 transactions, one object owner tentative local updates, eager base updates	

**Table 2. Variables used in the model and analysis**

<i>DB_Size</i>	number of distinct objects in the database
<i>Nodes</i>	number of nodes; each node replicates all objects
<i>Transactions</i>	number of concurrent transactions at a node. This is a derived value.
<i>TPS</i>	number of transactions per second originating at this node.
<i>Actions</i>	number of updates in a transaction
<i>Action_Time</i>	time to perform an action
<i>Time_Between_Disconnects</i>	mean time between network disconnect of a node.
<i>Disconnected_time</i>	mean time node is disconnected from network
<i>Message_Delay</i>	time between update of an object and update of a replica (ignored)
<i>Message_cpu</i>	processing and transmission time needed to send a replication message or apply a replica update (ignored)

The analysis below indicates that group and lazy replication are more prone to serializability violations than master and eager replication

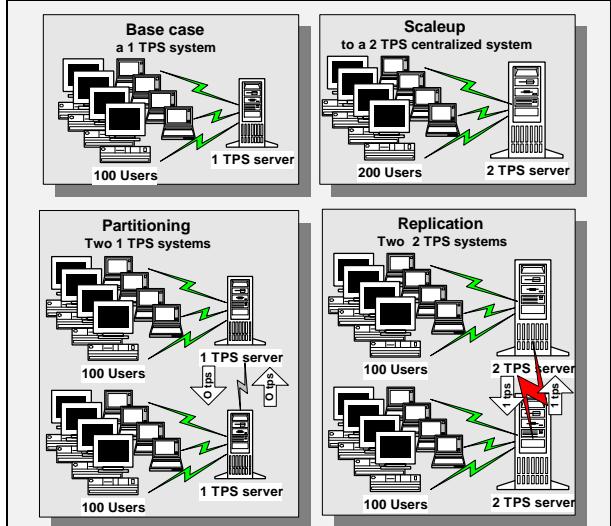
The model assumes the database consists of a fixed set of objects. There are a fixed number of nodes, each storing a replica of all objects. Each node originates a fixed number of transactions per second. Each transaction updates a fixed number of objects. Access to objects is equi-probable (there are no hotspots). Inserts and deletes are modeled as updates. Reads are ignored. Replica update requests have a transmit delay and also require processing by the sender and receiver. These delays and extra processing are ignored; only the work of sequentially updating the replicas at each node is modeled. Some nodes are mobile and disconnected most of the time. When first connected, a mobile node sends and receives deferred replica updates. Table 2 lists the model parameters.

One can imagine many variations of this model. Applying eager updates in parallel comes to mind. Each design alternative gives slightly different results. The design here roughly characterizes the basic alternatives. We believe obvious variations will not substantially change the results here.

Each node generates *TPS* transactions per second. Each transaction involves a fixed number of actions. Each action requires a fixed time to execute. So, a transaction's duration is *Actions*  $\times$  *Action\_Time*. Given these two observations, the number of concurrent transactions originating at a node is:

$$\text{Transactions} = \text{TPS} \times \text{Actions} \times \text{Action_Time} \quad (1)$$

A more careful analysis would consider that fact that, as system load and contention rises, the time to complete an action increases. In a scaleable server system, this *time-dilation* is a second-order effect and is ignored here.



**Figure 3:** Systems can grow by (1) *scaleup*: buying a bigger machine, (2) *partitioning*: dividing the work between two machines, or (3) *replication*: placing the data at two machines and having each machine keep the data current. This simple idea is key to understanding the  $N^2$  growth. Notice that each of the replicated servers at the lower right of the illustration is performing 2 TPS and the aggregate rate is 4 TPS. Doubling the users increased the total workload by a factor of four. Read-only transactions need not generate any additional load on remote nodes.

In a system of  $N$  nodes,  $N$  times as many transactions will be originating per second. Since each update transaction must replicate its updates to the other  $(N-1)$  nodes, it is easy to see that the transaction size for eager systems grows by a factor of  $N$  and the node update rate grows by  $N^2$ . In lazy systems, each *user* update transaction generates  $N-1$  lazy replica updates, so there are  $N$  times as many concurrent transactions, and the node update rate is  $N^2$  higher. This non-linear growth in node update rates leads to unstable behavior as the system is scaled up.

### 3. Eager Replication

Eager replication updates all replicas when a transaction updates any instance of the object. There are no serialization anomalies (inconsistencies) and no need for reconciliation in eager systems. Locking detects potential anomalies and converts them to waits or deadlocks.

With eager replication, reads at connected nodes give current data. Reads at disconnected nodes may give stale (out of date) data. Simple eager replication systems prohibit updates if any node is disconnected. For high availability, eager replication systems allow updates among members of the quorum or cluster [Gifford], [Garcia-Molina]. When a node joins the quorum, the quorum sends the new node all replica updates since the node was

disconnected. We assume here that a quorum or fault tolerance scheme is used to improve update availability.

Even if all the nodes are connected all the time, updates may fail due to deadlocks that prevent serialization errors. The following simple analysis derives the wait and deadlock rates of an eager replication system. We start with wait and deadlock rates for a single-node system.

In a single-node system the “other” transactions have about  $\frac{\text{Transactions? Actions}}{2}$  resources locked (each is about half way

complete). Since objects are chosen uniformly from the database, the chance that a request by one transaction will request a resource locked by any other transaction is:  $\frac{2}{\text{DB\_size}}$ . A transaction makes  $\text{Actions}$  such re-

quests, so the chance that it will wait sometime in its lifetime is approximately [Gray et. al.], [Gray & Reuter pp. 428]:

$$PW = \frac{1}{2} \left( 1 - \frac{\text{Transactions? Actions}}{2 \cdot \text{DB\_size}} \right)^{\text{Actions}} \approx \frac{\text{Transactions? Actions}^2}{2 \cdot \text{DB\_Size}} \quad (2)$$

A deadlock consists of a cycle of transactions waiting for one another. The probability a transaction forms a cycle of length two is  $PW^2$  divided by the number of transactions. Cycles of length  $j$  are proportional to  $PW^j$  and so are even less likely if  $PW \ll 1$ . Applying equation (1), the probability that the transaction deadlocks is approximately:

$$PD = \frac{PW^2}{\text{Transactions}} \cdot \frac{\text{Transactions? Actions}^4}{4 \cdot \text{DB\_Size}^2} \approx \frac{TPS \cdot \text{Action\_Time? Actions}^5}{4 \cdot \text{DB\_Size}^2} \quad (3)$$

Equation (3) gives the deadlock hazard for a transaction. The deadlock rate for a transaction is the probability it deadlocks in the next second. That is PD divided by the transaction lifetime ( $\text{Actions} \times \text{Action\_Time}$ ).

$$\text{Trans\_Deadlock\_rate} = \frac{TPS \cdot \text{Actions}^4}{4 \cdot \text{DB\_Size}^2} \quad (4)$$

Since the node runs  $\text{Transactions}$  concurrent transactions, the deadlock rate for the whole node is higher. Multiplying equation (4) and equation (1), the node deadlock rate is:

$$\text{Node\_Deadlock\_Rate} = \frac{TPS^2 \cdot \text{Action\_Time} \cdot \text{Actions}^5}{4 \cdot \text{DB\_Size}^2} \quad (5)$$

Suppose now that several such systems are replicated using eager replication — the updates are done immediately as in Figure 1. Each node will initiate its local load of  $TPS$  transactions per second<sup>1</sup>. The transaction size, duration, and aggregate transaction rate for eager systems is:

$$\begin{aligned} \text{Transaction\_Size} &= \text{Actions} \times \text{Nodes} \\ \text{Transaction\_Duration} &= \text{Actions} \times \text{Nodes} \times \text{Action\_Time} \\ \text{Total\_TPS} &= TPS \times \text{Nodes} \end{aligned} \quad (6)$$

<sup>1</sup> The assumption that transaction arrival rate per node stays constant as nodes are replicated assumes that nodes are lightly loaded. As the replication workload increases, the nodes must grow processing and IO power to handle the increased load. Growing power at an  $N^2$  rate is problematic.

Each node is now doing its own work and also applying the updates generated by other nodes. So each update transaction actually performs many more actions ( $\text{Nodes} \times \text{Actions}$ ) and so has a much longer lifetime — indeed it takes at least  $\text{Nodes}$  times longer<sup>2</sup>. As a result the total number of transactions in the system rises quadratically with the number of nodes:

$$\text{Total\_Transactions} = TPS \times \text{Actions} \times \text{Action\_Time} \times \text{Nodes}^2 \quad (7)$$

This rise in active transactions is due to eager transactions taking  $N$ -Times longer and due to lazy updates generating  $N$ -times more transactions. The action rate also rises very fast with  $N$ . Each node generates work for all other nodes. The eager work rate, measured in actions per second is:

$$\begin{aligned} \text{Action\_Rate} &= \text{Total\_TPS} \times \text{Transaction\_Size} \\ &= TPS \times \text{Actions} \times \text{Nodes}^2 \end{aligned} \quad (8)$$

It is surprising that the action rate and the number of active transactions is the same for eager and lazy systems. Eager systems have fewer-longer transactions. Lazy systems have more and shorter transactions. So, although equations (6) are different for lazy systems, equations (7) and (8) apply to both eager and lazy systems.

Ignoring message handling, the probability a transaction waits can be computed using the argument for equation (2). The transaction makes  $\text{Actions}$  requests while the other  $\text{Total\_Transactions}$  have  $\text{Actions}/2$  objects locked. The result is approximately:

$$\begin{aligned} PW_{eager} &= \frac{\text{Total\_Transactions} \cdot \text{Actions}}{2 \cdot \text{DB\_Size}} \\ &\approx \frac{TPS \cdot \text{Action\_Time} \cdot \text{Actions}^3 \cdot \text{Nodes}^2}{2 \cdot \text{DB\_Size}} \end{aligned} \quad (9)$$

This is the probability that one transaction waits. The wait rate (waits per second) for the entire system is computed as:

$$\begin{aligned} \text{Total\_Eager\_Wait\_Rate} &= \frac{PW_{eager}}{\text{Transaction\_Duration}} \cdot \text{Total\_Transactions} \\ &\approx \frac{TPS^2 \cdot \text{Action\_Time} \cdot (\text{Actions} \cdot \text{Nodes})^3}{2 \cdot \text{DB\_Size}} \end{aligned} \quad (10)$$

As with equation (4), The probability that a particular transaction deadlocks is approximately:

<sup>2</sup> An alternate model has eager actions broadcast the update to all replicas in one instant. The replicas are updated in parallel and the elapsed time for each action is constant (independent of  $N$ ). In our model, we attempt to capture message handing costs by serializing the individual updates. If one follows this model, then the processing at each node rises quadratically, but the number of concurrent transactions stays constant with scaleup. This model avoids the polynomial explosion of waits and deadlocks if the total TPS rate is held constant.

$$PD\_eager? \frac{\frac{Total\_Transactions ? Actions^4}{4 ? DB\_Size^2}}{\frac{TPS ? Action\_Time ? Actions^5 ? Nodes^2}{4 ? DB\_Size^2}} \quad (11)$$

The equation for a single-transaction deadlock implies the total deadlock rate. Using the arguments for equations (4) and (5), and using equations (7) and (11):

*Total\_Eager\_Deadlock\_Rate*

$$\frac{? Total\_Transactions ? \frac{PD\_eager}{Transaction\_Duration}}{? \frac{TPS^2 ? Action\_Time ? Actions^5 ? Nodes^3}{4 ? DB\_Size^2}} \quad (12)$$

If message delays were added to the model, then each transaction would last much longer, would hold resources much longer, and so would be more likely to collide with other transactions. Equation (12) also ignores the “second order” effect of two transactions racing to update the same object at the same time (it does not distinguish between *Master* and *Group* replication). If  $DB\_Size >> Node$ , such conflicts will be rare.

This analysis points to some serious problems with eager replication. Deadlocks rise as the third power of the number of nodes in the network, and the fifth power of the transaction size. Going from one-node to ten nodes increases the deadlock rate a thousand fold. A ten-fold increase in the transaction size increases the deadlock rate by a factor of 100,000.

To ameliorate this, one might imagine that the database size grows with the number of nodes (as in the checkbook example earlier, or in the TPC-A, TPC-B, and TPC-C benchmarks). More nodes, and more transactions mean more data. With a scaled up database size, equation (12) becomes:

*Eager\_Deadlock\_Rate\_Scaled\_DB*

$$\frac{? TPS^2 ? Action\_Time ? Actions^5 ? Nodes}{4 ? DB\_Size^2} \quad (13)$$

Now a ten-fold growth in the number of nodes creates *only* a ten-fold growth in the deadlock rate. This is still an unstable situation, but it is a big improvement over equation (12).

Having a master for each object helps eager replication avoid deadlocks. Suppose each object has an owner node. Updates go to this node first and are then applied to the replicas. If, each transaction updated a single replica, the object-master approach would eliminate all deadlocks.

In summary, eager replication has two major problems:

1. Mobile nodes cannot use an eager scheme when disconnected.
2. The probability of deadlocks, and consequently failed transactions rises very quickly with transaction size and

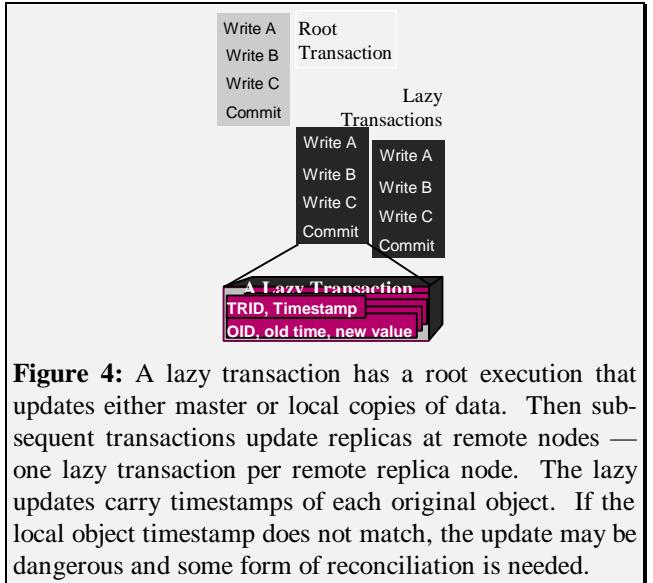
with the number of nodes. A ten-fold increase in nodes gives a thousand-fold increase in failed transactions (deadlocks).

We see no solution to this problem. If replica updates were done concurrently, the action time would not increase with  $N$  then the growth rate would *only* be quadratic.

## 4. Lazy Group Replication

Lazy group replication allows any node to update any local data. When the transaction commits, a transaction is sent to every other node to apply the root transaction’s updates to the replicas at the destination node (see Figure 4). It is possible for two nodes to update the same object and race each other to install their updates at other nodes. The replication mechanism must detect this and reconcile the two transactions so that their updates are not lost.

Timestamps are commonly used to detect and reconcile lazy-group transactional updates. Each object carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old object timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica’s timestamp and the update’s old timestamp are equal. If so, the update is safe. The local replica’s timestamp advances to the new transaction’s timestamp and the object value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be “dangerous”. In such cases, the node rejects the incoming transaction and submits it for *reconciliation*.



**Figure 4:** A lazy transaction has a root execution that updates either master or local copies of data. Then subsequent transactions update replicas at remote nodes — one lazy transaction per remote replica node. The lazy updates carry timestamps of each original object. If the local object timestamp does not match, the update may be dangerous and some form of reconciliation is needed.

Transactions that would wait in an eager replication system face reconciliation in a lazy-group replication sys-

tem. Waits are much more frequent than deadlocks because it takes two waits to make a deadlock. Indeed, if waits are a rare event, then deadlocks are very rare (*rare*<sup>2</sup>). Eager replication waits cause delays while deadlocks create application faults. With lazy replication, the much more frequent waits are what determines the reconciliation frequency. So, the system-wide lazy-group reconciliation rate follows the transaction wait rate equation (Equation 10):

*Lazy\_Group\_Reconciliation\_Rate*

$$? \frac{TPS^2 ? Action\_Time ? (Actions ? Nodes)^3}{2 ? DB\_Size} \quad (14)$$

As with eager replication, if message propagation times were added, the reconciliation rate would rise. Still, having the reconciliation rate rise by a factor of a thousand when the system scales up by a factor of ten is frightening.

The really bad case arises in mobile computing. Suppose that the typical node is disconnected most of the time. The node accepts and applies transactions for a day. Then, at night it connects and downloads them to the rest of the network. At that time it also accepts replica updates. It is as though the message propagation time was 24 hours.

If any two transactions at any two different nodes update the same data during the disconnection period, then they will need reconciliation. What is the chance of two disconnected transactions colliding during the *Disconnected\_Time*?

If each node updates a small fraction of the database each day then the number of distinct *outbound* pending object updates at reconnect is approximately:

$$Outbound\_Updates ? Disconnect\_Time ? TPS ? Actions \quad (15)$$

Each of these updates applies to all the replicas of an object. The pending *inbound updates* for this node from the rest of the network is approximately (*Nodes*-1) times larger than this.

*Inbound\_Updates*

$$? ?Nodes ? 1? ? Disconnect\_Time? TPS ? Actions \quad (16)$$

If the inbound and outbound sets overlap, then reconciliation is needed. The chance of an object being in both sets is approximately:

*P(collision)*

$$? \frac{Inbound\_Updates ? Outbound\_Updates}{DB\_Size} \quad (17)$$

$$? \frac{Nodes ? (Disconnect\_Time ? TPS ? Actions)^2}{DB\_Size}$$

Equation (17) is the chance one node needs reconciliation during the *Disconnect\_Time* cycle. The rate for all nodes is:

$$\begin{aligned} & Lazy\_Group\_Reconciliation\_Rate ? \\ & P(collision) ? \frac{Nodes}{Disconnect\_Time} \\ & ? \frac{Disconnect\_Time ? TPS ? Actions ? Nodes^2}{DB\_Size} \end{aligned} \quad (18)$$

The quadratic nature of this equation suggests that a system that performs well on a few nodes with simple transactions may become unstable as the system scales up.

## 5. Lazy Master Replication

Master replication assigns an owner to each object. The owner stores the object's correct current value. Updates are first done by the owner and then propagated to other replicas. Different objects may have different owners.

When a transaction wants to update an object, it sends an RPC (remote procedure call) to the node owning the object. To get serializability, a read action should send read-lock RPCs to the masters of any objects it reads.

To simplify the analysis, we assume the node originating the transaction broadcasts the replica updates to all the slave replicas after the master transaction commits. The originating node sends one slave transaction to each slave node (as in Figure 1). Slave updates are timestamped to assure that all the replicas converge to the same final state. If the record timestamp is newer than a replica update timestamp, the update is “stale” and can be ignored. Alternatively, each master node sends replica updates to slaves in sequential commit order.

Lazy-Master replication is not appropriate for mobile applications. A node wanting to update an object must be connected to the object owner and participate in an atomic transaction with the owner.

As with eager systems, lazy-master systems have no reconciliation failures; rather, conflicts are resolved by waiting or deadlock. Ignoring message delays, the deadlock rate for a lazy-master replication system is similar to a single node system with much higher transaction rates. Lazy master transactions operate on master copies of objects. But, because there are *Nodes* times more users, there are *Nodes* times as many concurrent master transactions and approximately *Nodes*<sup>2</sup> times as many replica update transactions. The replica update transactions do not really matter, they are background housekeeping transactions. They can abort and restart without affecting the user. So the main issue is how frequently the master transactions deadlock. Using the logic of equation (5), the deadlock rate is approximated by:

$$Lazy\_Master\_Deadlock\_Rate? \frac{(TPS ? Nodes)^2 ? Action\_Time ? Actions^5}{4 ? DB\_Size^2} \quad (19)$$

This is better behavior than lazy-group replication. Lazy-master replication sends fewer messages during the base transaction and so completes more quickly. Nevertheless, all of these replication schemes have troubling deadlock or reconciliation rates as they grow to many nodes.

In summary, lazy-master replication requires contact with object masters and so is not useable by mobile applications. Lazy-master replication is slightly less deadlock prone than eager-group replication primarily because the transactions have shorter duration.

## 6. Non-Transactional Replication Schemes

The equations in the previous sections are facts of nature — they help explain another fact of nature. They show why there are no high-update-traffic replicated databases with globally serializable transactions.

Certainly, there are replicated databases: bibles, phone books, check books, mail systems, name servers, and so on. But updates to these databases are managed in interesting ways — typically in a lazy-master way. Further, updates are not record-value oriented; rather, updates are expressed as transactional transformations such as “Debit the account by \$50” instead of “change account from \$200 to \$150”.

One strategy is to abandon serializability for the *convergence property*: if no new transactions arrive, and if all the nodes are connected together, they will all converge to the same replicated state after exchanging replica updates. The resulting state contains the committed appends, and the most recent replacements, but updates may be lost.

Lotus Notes gives a good example of convergence [Kawell]. Notes is a lazy group replication design (update anywhere, anytime, anyhow). Notes provides convergence rather than an ACID transaction execution model. The database state may not reflect any particular serial execution, but all the states will be identical. As explained below, timestamp schemes have the lost-update problem.

Lotus Notes achieves convergence by offering lazy-group replication at the transaction level. It provides two forms of update transaction:

1. **Append** adds data to a Notes file. Every appended note has a timestamp. Notes are stored in timestamp order. If all nodes are in contact with all others, then they will all converge on the same state.
2. **Timestamped replace a value** replaces a value with a newer value. If the current value of the object already has a timestamp greater than this update’s timestamp, the incoming update is discarded.

If convergence were the only goal, the timestamp method would be sufficient. But, the timestamp scheme may lose the effects of some transactions because it just applies the most recent updates. Applying a timestamp scheme to the check-book example, if there are two concurrent updates to a check-

book balance, the highest timestamp value wins and the other update is discarded as a “stale” value. Concurrency control theory calls this the *lost update problem*. Timestamp schemes are vulnerable to lost updates.

Convergence is desirable, but the converged state should reflect the effects of all committed transactions. In general this is not possible unless global serialization techniques are used.

In certain cases transactions can be designed to commute, so that the database ends up in the same state no matter what transaction execution order is chosen. Timestamped Append is a kind of commutative update but there are others (e.g., adding and subtracting constants from an integer value). It would be possible for Notes to support a third form of transaction:

3. **Commutative updates** that are incremental transformations of a value that can be applied in any order.

Lotus Notes, the Internet name service, mail systems, Microsoft Access, and many other applications use some of these techniques to achieve convergence and avoid delusion.

Microsoft Access offers convergence as follows. It has a single design master node that controls all schema updates to a replicated database. It offers update-anywhere for record instances. Each node keeps a version vector with each replicated record. These version vectors are exchanged on demand or periodically. The most recent update wins each pairwise exchange. Rejected updates are reported [Hammond].

The examples contrast with a simple update-anywhere-anytime-anyhow lazy-group replication offered by some systems. If the transaction profiles are not constrained, lazy-group schemes suffer from unstable reconciliation described in earlier sections. Such systems degenerate into system delusion as they scale up.

Lazy group replication schemes are emerging with specialized reconciliation rules. Oracle 7 provides a choice of twelve reconciliation rules to merge conflicting updates [Oracle]. In addition, users can program their own reconciliation rules. These rules give priority certain sites, or time priority, or value priority, or they merge commutative updates. The rules make some transactions commutative. A similar, transaction-level approach is followed in the two-tier scheme described next.

## 7. Two-Tier Replication

An ideal replication scheme would achieve four goals:

**Availability and scalability:** Provide high availability and scalability through replication, while avoiding instability.

**Mobility:** Allow mobile nodes to read and update the database while disconnected from the network.

**Serializability:** Provide single-copy serializable transaction execution.

**Convergence:** Provide convergence to avoid system delusion.

The safest transactional replication schemes, (ones that avoid system delusion) are the eager systems and lazy master systems. They have no reconciliation problems (they have no reconciliation). But these systems have other problems. As shown earlier:

1. Mastered objects cannot accept updates if the master node is not accessible. This makes it difficult to use master replication for mobile applications.
2. Master systems are unstable under increasing load. Deadlocks rise quickly as nodes are added.
3. Only eager systems and lazy master (where reads go to the master) give ACID serializability.

Circumventing these problems requires changing the way the system is used. We believe a scaleable replication system must function more like the check books, phone books, Lotus Notes, Access, and other replication systems we see about us.

Lazy-group replication systems are prone to reconciliation problems as they scale up. Manually reconciling conflicting transactions is unworkable. One approach is to *undo* all the work of any transaction that needs reconciliation — backing out all the updates of the transaction. This makes transactions atomic, consistent, and isolated, but not durable — or at least not durable until the updates are propagated to each node. In such a lazy group system, every transaction is tentative until all its replica updates have been propagated. If some mobile replica node is disconnected for a very long time, all transactions will be tentative until the missing node reconnects. So, an undo-oriented lazy-group replication scheme is untenable for mobile applications.

The solution seems to require a modified mastered replication scheme. To avoid reconciliation, each object is mastered by a node — much as the bank owns your checking account and your mail server owns your mailbox. Mobile agents can make tentative updates, then connect to the base nodes and immediately learn if the tentative update is acceptable.

The **two-tier replication** scheme begins by assuming there are two kinds of nodes:

**Mobile nodes** are disconnected much of the time. They store a replica of the database and may originate tentative transactions. A mobile node may be the master of some data items.

**Base nodes** are always connected. They store a replica of the database. Most items are mastered at base nodes.

Replicated data items have two versions at mobile nodes:

**Master Version:** The most recent value received from the object master. The version at the object master is *the* master version, but disconnected or lazy replica nodes may have older versions.

**Tentative Version:** The local object may be updated by tentative transactions. The most recent value due to local updates is maintained as a tentative value.

Similarly, there are two kinds of transactions:

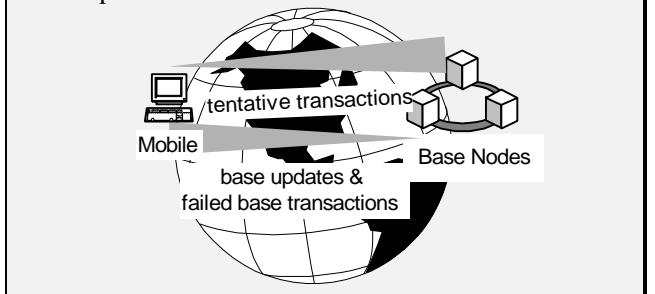
**Base Transaction:** Base transactions work only on master data, and they produce new master data. They involve at most one connected-mobile node and may involve several base nodes.

**Tentative Transaction:** Tentative transactions work on local tentative data. They produce new tentative versions. They also produce a base transaction to be run at a later time on the base nodes.

Tentative transactions must follow a **scope rule**: they may involve objects mastered on base nodes and mastered at the mobile node originating the transaction (call this the transaction's *scope*). The idea is that the mobile node and all the base nodes will be in contact when the tentative transaction is processed as a “real” base transaction — so the real transaction will be able to read the master copy of each item in the scope.

Local transactions that read and write *only* local data can be designed in any way you like. They cannot read-or write any tentative data because that would make them tentative.

**Figure 5:** The two-tier-replication scheme. Base nodes store replicas of the database. Each object is mastered at some node. Mobile nodes store a replica of the database, but are usually disconnected. Mobile nodes accumulate tentative transactions that run against the tentative database stored at the node. Tentative transactions are reprocessed as base transactions when the mobile node reconnects to the base. Tentative transactions may fail when reprocessed.



The base transaction generated by a tentative transaction may fail or it may produce different results. The base transaction has an **acceptance criterion**: a test the resulting outputs must pass for the slightly different base transaction results to be acceptable. To give some sample acceptance criteria:

- ?? The bank balance must not go negative.
- ?? The price quote can not exceed the tentative quote.
- ?? The seats must be aisle seats.

If a tentative transaction fails, the originating node and person who generated the transaction are informed it failed and why it failed. Acceptance failure is equivalent to the reconciliation mechanism of the lazy-group replication schemes. The differences are (1) the master database is always converged — there is no system delusion, and (2) the originating node need only contact a base node in order to discover if a tentative transaction is acceptable.

To continue the checking account analogy, the bank's version of the account is the master version. In writing checks, you and your spouse are creating tentative transactions which result in tentative versions of the account. The bank runs a base transaction when it clears the check. If you contact your bank and it clears the check, then you know the tentative transaction is a real transaction.

Consider the two-tier replication scheme's behavior during connected operation. In this environment, a two-tier system operates much like a lazy-master system with the additional restriction that no transaction can update data mastered at more than one mobile node. This restriction is not really needed in the connected case.

Now consider the disconnected case. Imagine that a mobile node disconnected a day ago. It has a copy of the base data as of yesterday. It has generated tentative transactions on that base data and on the local data mastered by the mobile node. These transactions generated tentative data versions at the mobile node. If the mobile node queries this data it sees the tentative values. For example, if it updated documents, produced contracts, and sent mail messages, those tentative updates are all visible at the mobile node.

When a mobile node connects to a base node, the mobile node:

1. Discards its tentative object versions since they will soon be refreshed from the masters,
2. Sends replica updates for any objects mastered at the mobile node to the base node “hosting” the mobile node,
3. Sends all its tentative transactions (and all their input parameters) to the base node to be executed in the order in which they committed on the mobile node,
4. Accepts replica updates from the base node (this is standard lazy-master replication), and
5. Accepts notice of the success or failure of each tentative transaction.

The “host” base node is the other tier of the two tiers. When contacted by a mobile node, the host base node:

1. Sends delayed replica update transactions to the mobile node.
2. Accepts delayed update transactions for mobile-mastered objects from the mobile node.
3. Accepts the list of tentative transactions, their input messages, and their acceptance criteria. Reruns each tentative transaction in the order it committed on the mobile node. During this reprocessing, the base transaction reads and writes object master copies using a lazy-master execution model. The scope-rule assures that the base transaction only accesses data mastered by the originating mobile node and base nodes. So master copies of all data in the transaction's scope are available to the base transaction. If the base transaction fails its acceptance criteria, the base transaction is aborted and a diagnostic message is returned to the mobile node. If the acceptance criteria requires the base and tentative transaction have identical outputs, then subsequent transactions reading tentative results written by T will fail too. On the other hand, weaker acceptance criteria are possible.
4. After the base node commits a base transaction, it propagates the lazy replica updates as transactions sent to all the other replica nodes. This is standard lazy-master.
5. When all the tentative transactions have been reprocessed as base transactions, the mobile node's state is converged with the base state.

The key properties of the two-tier replication scheme are:

1. Mobile nodes may make tentative database updates.
2. Base transactions execute with single-copy serializability so the master base system state is the result of a serializable execution.
3. A transaction becomes durable when the base transaction completes.
4. Replicas at all connected nodes converge to the base system state.
5. If all transactions commute, there are no reconciliations.

This comes close to meeting the four goals outlined at the start of this section.

When executing a base transaction, the two-tier scheme is a lazy-master scheme. So, the deadlock rate for base transactions is given by equation (19). This is still an  $N^2$  deadlock rate. If a base transaction deadlocks, it is resubmitted and reprocessed until it succeeds, much as the replica update transactions are resubmitted in case of deadlock.

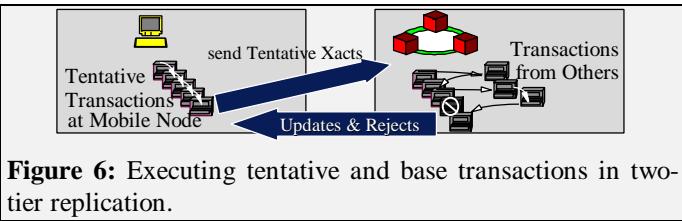
The reconciliation rate for base transactions will be zero if all the transactions commute. The reconciliation rate

is driven by the rate at which the base transactions fail their acceptance criteria.

Processing the base transaction may produce results different from the tentative results. This is acceptable for some applications. It is fine if the checking account balance is different when the transaction is reprocessed. Other transactions from other nodes may have affected the account while the mobile node was disconnected. But, there are cases where the changes may not be acceptable. If the price of an item has increased by a large amount, if the item is out of stock, or if aisle seats are no longer available, then the salesman's price or delivery quote must be reconciled with the customer.

These acceptance criteria are application specific. The replication system can do no more than detect that there is a difference between the tentative and base transaction. This is probably too pessimistic a test. So, the replication system will simply run the tentative transaction. If the tentative transaction completes successfully and passes the acceptance test, then the replication system assumes all is well and propagates the replica updates as usual.

Users are aware that all updates are tentative until the transaction becomes a base transaction. If the base transaction fails, the user may have to revise and resubmit a transaction. The programmer must design the transactions to be commutative and to have acceptance criteria to detect whether the tentative transaction agrees with the base transaction effects.



**Figure 6:** Executing tentative and base transactions in two-tier replication.

Thinking again of the checkbook example of an earlier section. The check is in fact a tentative update being sent to the bank. The bank either honors the check or rejects it. Analogous mechanisms are found in forms flow systems ranging from tax filing, applying for a job, or subscribing to a magazine. It is an approach widely used in human commerce.

This approach is similar to, but more general than the Data Cycle architecture [Herman] which has a single master node for all objects.

The approach can be used to obtain pure serializability if the base transaction only reads and writes master objects (current versions).

## 8. Summary

Replicating data at many nodes and letting anyone update the data is problematic. Security is one issue, performance is another. When the standard transaction model is applied to a replicated database, the size of each transaction rises by the degree of replication. This, combined with higher transaction rates means dramatically higher deadlock rates.

It might seem at first that a lazy replication scheme will solve this problem. Unfortunately, lazy-group replication just converts waits and deadlocks into reconciliations. Lazy-master replication has slightly better behavior than eager-master replication. Both suffer from dramatically increased deadlock as the replication degree rises. None of the master schemes allow mobile computers to update the database while disconnected from the system.

The solution appears to be to use semantic tricks (timestamps, and commutative transactions), combined with a two-tier replication scheme. Two-tier replication supports mobile nodes and combines the benefits of an eager-master-replication scheme and a local update scheme.

## **9. Acknowledgments**

Tanj (John G.) Bennett of Microsoft and Alex Thomasian of IBM gave some very helpful advice on an earlier version of this paper. The anonymous referees made several helpful suggestions to improve the presentation. Dwight Joe pointed out a mistake in the published version of equation 19.

## **10. References**

- Bernstein, P.A., V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison Wesley, Reading MA., 1987.
- Berenson, H., Bernstein, P.A., Gray, J., Jim Melton, J., O'Neil, E., O'Neil, P., "A Critique of ANSI SQL Isolation Levels," Proc. ACM SIGMOD 95, pp. 1-10, San Jose CA, June 1995.
- Garcia Molina, H. "Performance of Update Algorithms for Replicated Data in a Distributed Database," TR STAN-CS-79-744, CS Dept., Stanford U., Stanford, CA., June 1979.
- Garcia Molina, H., Barbara, D., "How to Assign Votes in a Distributed System," J. ACM, 32(4). Pp. 841-860, October, 1985.
- Gifford, D. K., "Weighted Voting for Replicated Data," Proc. ACM SIGOPS SOSP, pp: 150-159, Pacific Grove, CA, December 1979.
- Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, CA. 1993.
- Gray, J., Homan, P, Korth, H., Obermarck, R., "A Strawman Analysis of the Probability of Deadlock," IBM RJ 2131, IBM Research, San Jose, CA., 1981.
- Hammond, Brad, "Wingman, A Replication Service for Microsoft Access and Visual Basic", Microsoft White Paper,  
bradha@microsoft.com
- Herman, G., Gopal, G, Lee, K., Weinrib, A., "The Datacycle Architecture for Very High Throughput Database Systems," Proc. ACM SIGMOD, San Francisco, CA. May 1987.
- Kawell, L., Beckhardt, S., Halvorsen, T., Raymond Ozzie, R., Greif, I., "Replicated Document Management in a Group Communication System," Proc. Second Conference on Computer Supported Cooperative Work, Sept. 1988.
- Oracle, "Oracle7 Server Distributed Systems: Replicated Data," Oracle part number A21903.March 1994, Oracle, Redwood Shores, CA. Or <http://www.oracle.com/products/oracle7/server/whitepapers/replication/html/index>



## Data clustering: 50 years beyond K-means <sup>☆</sup>

Anil K. Jain \*

*Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan 48824, USA  
 Department of Brain and Cognitive Engineering, Korea University, Anam-dong, Seoul, 136-713, Korea*

### ARTICLE INFO

#### Article history:

Available online 9 September 2009

#### Keywords:

Data clustering  
 User's dilemma  
 Historical developments  
 Perspectives on clustering  
 King-Sun Fu prize

### ABSTRACT

Organizing data into sensible groupings is one of the most fundamental modes of understanding and learning. As an example, a common scheme of scientific classification puts organisms into a system of ranked taxa: domain, kingdom, phylum, class, etc. Cluster analysis is the formal study of methods and algorithms for grouping, or clustering, objects according to measured or perceived intrinsic characteristics or similarity. Cluster analysis does not use category labels that tag objects with prior identifiers, i.e., class labels. The absence of category information distinguishes data clustering (unsupervised learning) from classification or discriminant analysis (supervised learning). The aim of clustering is to find structure in data and is therefore exploratory in nature. Clustering has a long and rich history in a variety of scientific fields. One of the most popular and simple clustering algorithms, K-means, was first published in 1955. In spite of the fact that K-means was proposed over 50 years ago and thousands of clustering algorithms have been published since then, K-means is still widely used. This speaks to the difficulty in designing a general purpose clustering algorithm and the ill-posed problem of clustering. We provide a brief overview of clustering, summarize well known clustering methods, discuss the major challenges and key issues in designing clustering algorithms, and point out some of the emerging and useful research directions, including semi-supervised clustering, ensemble clustering, simultaneous feature selection during data clustering, and large scale data clustering.

© 2009 Elsevier B.V. All rights reserved.

### 1. Introduction

Advances in sensing and storage technology and dramatic growth in applications such as Internet search, digital imaging, and video surveillance have created many high-volume, high-dimensional data sets. It is estimated that the digital universe consumed approximately 281 exabytes in 2007, and it is projected to be 10 times that size by 2011 (1 exabyte is  $\sim 10^{18}$  bytes or 1,000,000 terabytes) (Gantz, 2008). Most of the data is stored digitally in electronic media, thus providing huge potential for the development of automatic data analysis, classification, and retrieval techniques. In addition to the growth in the amount of data, the variety of available data (text, image, and video) has also increased. Inexpensive digital and video cameras have made available huge archives of images and videos. The prevalence of RFID tags or transponders due to their low cost and small size has resulted in the deployment of millions of sensors that transmit data regularly. E-mails, blogs, transaction data, and billions of Web pages create terabytes of new data every day. Many of these data

streams are unstructured, adding to the difficulty in analyzing them.

The increase in both the volume and the variety of data requires advances in methodology to automatically understand, process, and summarize the data. Data analysis techniques can be broadly classified into two major types (Tukey, 1977): (i) *exploratory* or descriptive, meaning that the investigator does not have pre-specified models or hypotheses but wants to understand the general characteristics or structure of the high-dimensional data, and (ii) *confirmatory* or inferential, meaning that the investigator wants to confirm the validity of a hypothesis/model or a set of assumptions given the available data. Many statistical techniques have been proposed to analyze the data, such as analysis of variance, linear regression, discriminant analysis, canonical correlation analysis, multi-dimensional scaling, factor analysis, principal component analysis, and cluster analysis to name a few. A useful overview is given in (Tabachnick and Fidell, 2007).

In pattern recognition, data analysis is concerned with predictive modeling: given some training data, we want to predict the behavior of the unseen test data. This task is also referred to as *learning*. Often, a clear distinction is made between learning problems that are (i) supervised (classification) or (ii) unsupervised (clustering), the first involving only *labeled data* (training patterns with known category labels) while the latter involving only *unlabeled data* (Duda et al., 2001). Clustering is a more difficult and

\* This paper is based on the King-Sun Fu Prize lecture delivered at the 19th International Conference on Pattern Recognition (ICPR), Tempe, FL, December 8, 2008.

\* Tel.: +1 517 355 9282; fax: +1 517 432 1061.

E-mail address: [jain@cse.msu.edu](mailto:jain@cse.msu.edu)

challenging problem than classification. There is a growing interest in a hybrid setting, called *semi-supervised learning* (Chapelle et al., 2006); in semi-supervised classification, the labels of only a small portion of the training data set are available. The unlabeled data, instead of being discarded, are also used in the learning process. In semi-supervised clustering, instead of specifying the class labels, pair-wise constraints are specified, which is a *weaker* way of encoding the prior knowledge. A pair-wise *must-link* constraint corresponds to the requirement that two objects should be assigned the same cluster label, whereas the cluster labels of two objects participating in a *cannot-link* constraint should be different. Constraints can be particularly beneficial in data clustering (Lange et al., 2005; Basu et al., 2008), where precise definitions of underlying clusters are absent. In the search for good models, one would like to include all the available information, no matter whether it is unlabeled data, data with constraints, or labeled data. Fig. 1 illustrates this spectrum of different types of learning problems of interest in pattern recognition and machine learning.

## 2. Data clustering

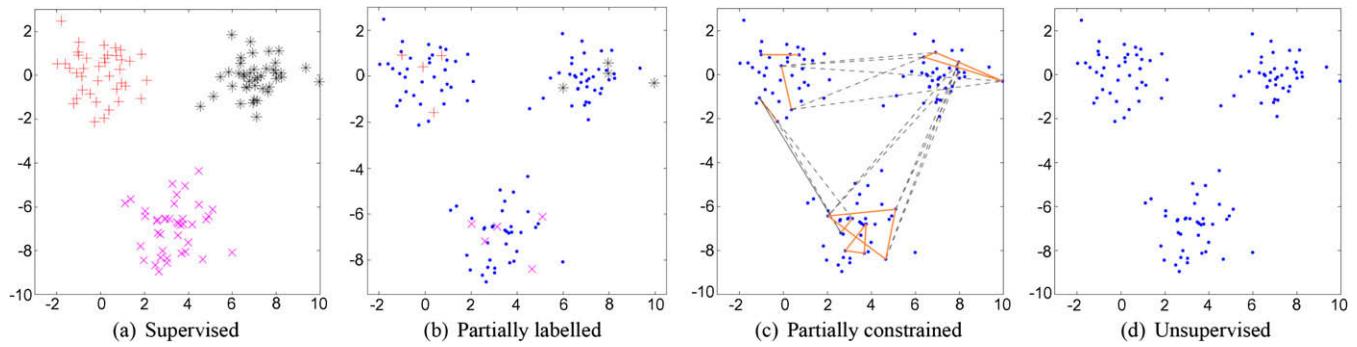
The goal of data clustering, also known as cluster analysis, is to discover the *natural* grouping(s) of a set of patterns, points, or objects. Webster (Merriam-Webster Online Dictionary, 2008) defines cluster analysis as “a statistical classification technique for discovering whether the individuals of a population fall into different groups by making quantitative comparisons of multiple character-

istics.” An example of clustering is shown in Fig. 2. The objective is to develop an automatic algorithm that will discover the natural groupings (Fig. 2b) in the unlabeled data (Fig. 2a).

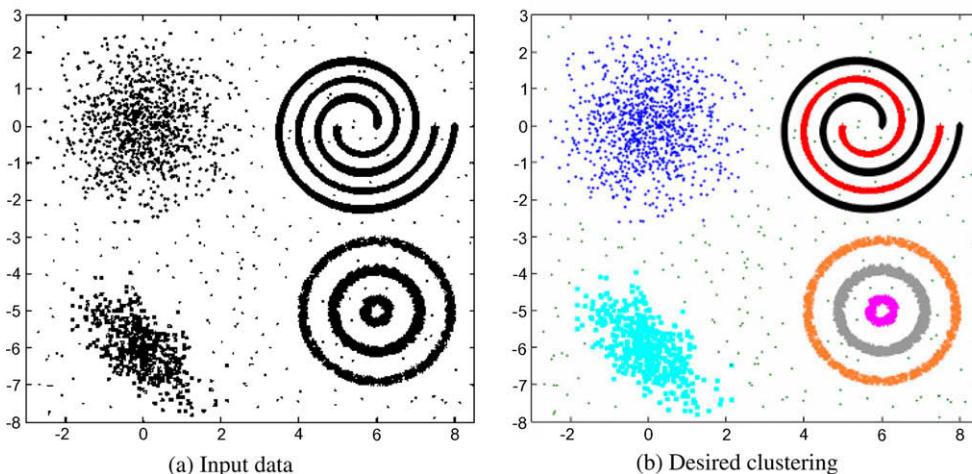
An operational definition of clustering can be stated as follows: Given a *representation* of  $n$  objects, find  $K$  groups based on a measure of *similarity* such that the similarities between objects in the same group are high while the similarities between objects in different groups are low. But, what is the notion of similarity? What is the definition of a cluster? Fig. 2 shows that clusters can differ in terms of their *shape*, *size*, and *density*. The presence of noise in the data makes the detection of the clusters even more difficult. An ideal cluster can be defined as a set of points that is *compact* and *isolated*. In reality, a cluster is a subjective entity that is in the eye of the beholder and whose significance and interpretation requires domain knowledge. But, while humans are excellent cluster seekers in two and possibly three dimensions, we need automatic algorithms for high-dimensional data. It is this challenge along with the unknown number of clusters for the given data that has resulted in thousands of clustering algorithms that have been published and that continue to appear.

### 2.1. Why clustering?

Cluster analysis is prevalent in any discipline that involves analysis of multivariate data. A search via Google Scholar (2009) found 1660 entries with the words *data clustering* that appeared in 2007



**Fig. 1.** Learning problems: dots correspond to points without any labels. Points with labels are denoted by plus signs, asterisks, and crosses. In (c), the must-link and cannot-link constraints are denoted by solid and dashed lines, respectively (figure taken from Lange et al. (2005)).



**Fig. 2.** Diversity of clusters. The seven clusters in (a) (denoted by seven different colors in 1(b)) differ in shape, size, and density. Although these clusters are apparent to a data analyst, none of the available clustering algorithms can detect all these clusters.

alone. This vast literature speaks to the importance of clustering in data analysis. It is difficult to exhaustively list the numerous scientific fields and applications that have utilized clustering techniques as well as the thousands of published algorithms. Image segmentation, an important problem in computer vision, can be formulated as a clustering problem (Jain and Flynn, 1996; Frigui and Krishnapuram, 1999; Shi and Malik, 2000). Documents can be clustered (Iwayama and Tokunaga, 1995) to generate topical hierarchies for efficient information access (Sahami, 1998) or retrieval (Bhatia and Deogun, 1998). Clustering is also used to group customers into different types for efficient marketing (Arabie and Hubert, 1994), to group services delivery engagements for workforce management and planning (Hu et al., 2007) as well as to study genome data (Baldi and Hatfield, 2002) in biology.

Data clustering has been used for the following three main purposes.

- *Underlying structure*: to gain insight into data, generate hypotheses, detect anomalies, and identify salient features.
- *Natural classification*: to identify the degree of similarity among forms or organisms (phylogenetic relationship).
- *Compression*: as a method for organizing the data and summarizing it through cluster prototypes.

An example of class discovery is shown in Fig. 3. Here, clustering was used to discover subclasses in an online handwritten character recognition application (Connell and Jain, 2002). Different users write the same digits in different ways, thereby increasing the within-class variance. Clustering the training patterns from a class can discover new subclasses, called the lexemes in handwritten characters. Instead of using a single model for each character, multiple models based on the number of subclasses are used to improve the recognition accuracy (see Fig. 3).

Given the large number of Web pages on the Internet, most search queries typically result in an extremely large number of hits. This creates the need for search results to be organized. Search engines like Clusty ([www.clusty.org](http://www.clusty.org)) cluster the search results and present them in a more organized way to the user.

## 2.2. Historical developments

The development of clustering methodology has been a truly interdisciplinary endeavor. Taxonomists, social scientists, psychologists, biologists, statisticians, mathematicians, engineers, computer scientists, medical researchers, and others who collect and process real data have all contributed to clustering methodology. According to JSTOR (2009), data clustering first appeared in the title of a 1954 article dealing with anthropological data. Data clustering is also known as Q-analysis, typology, clumping, and taxonomy

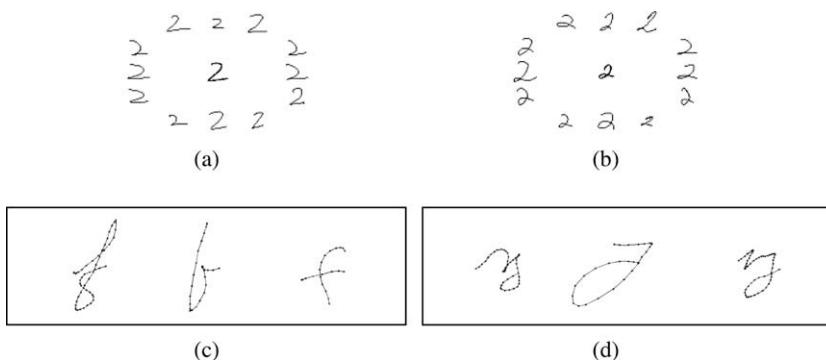
(Jain and Dubes, 1988) depending on the field where it is applied. There are several books published on data clustering; classic ones are by Sokal and Sneath (1963), Anderberg (1973), Hartigan (1975), Jain and Dubes (1988), and Duda et al. (2001). Clustering algorithms have also been extensively studied in data mining (see books by Han and Kamber (2000) and Tan et al. (2005) and machine learning (Bishop, 2006).

Clustering algorithms can be broadly divided into two groups: *hierarchical* and *partitional*. Hierarchical clustering algorithms recursively find nested clusters either in agglomerative mode (starting with each data point in its own cluster and merging the most similar pair of clusters successively to form a cluster hierarchy) or in divisive (top-down) mode (starting with all the data points in one cluster and recursively dividing each cluster into smaller clusters). Compared to hierarchical clustering algorithms, partitional clustering algorithms find all the clusters simultaneously as a partition of the data and do not impose a hierarchical structure. Input to a hierarchical algorithm is an  $n \times n$  similarity matrix, where  $n$  is the number of objects to be clustered. On the other hand, a partitional algorithm can use either an  $n \times d$  pattern matrix, where  $n$  objects are embedded in a  $d$ -dimensional feature space, or an  $n \times n$  similarity matrix. Note that a similarity matrix can be easily derived from a pattern matrix, but ordination methods such as multi-dimensional scaling (MDS) are needed to derive a pattern matrix from a similarity matrix.

The most well-known hierarchical algorithms are single-link and complete-link; the most popular and the simplest partitional algorithm is K-means. Since partitional algorithms are preferred in pattern recognition due to the nature of available data, our coverage here is focused on these algorithms. K-means has a rich and diverse history as it was independently discovered in different scientific fields by Steinhaus (1956), Lloyd (proposed in 1957, published in 1982), Ball and Hall (1965), and MacQueen (1967). Even though K-means was first proposed over 50 years ago, it is still one of the most widely used algorithms for clustering. Ease of implementation, simplicity, efficiency, and empirical success are the main reasons for its popularity. Below we will first summarize the development in K-means, and then discuss the major approaches that have been developed for data clustering.

## 2.3. K-means algorithm

Let  $X = \{x_i\}$ ,  $i = 1, \dots, n$  be the set of  $n$   $d$ -dimensional points to be clustered into a set of  $K$  clusters,  $C = \{c_k, k = 1, \dots, K\}$ . K-means algorithm finds a partition such that the squared error between the empirical mean of a cluster and the points in the cluster is minimized. Let  $\mu_k$  be the mean of cluster  $c_k$ . The squared error between  $\mu_k$  and the points in cluster  $c_k$  is defined as



**Fig. 3.** Finding subclasses using data clustering. (a) and (b) show two different ways of writing the digit 2; (c) three different subclasses for the character 'f'; (d) three different subclasses for the letter 'y'.

$$J(c_k) = \sum_{x_i \in c_k} \|x_i - \mu_k\|^2.$$

The goal of K-means is to minimize the sum of the squared error over all  $K$  clusters,

$$J(C) = \sum_{k=1}^K \sum_{x_i \in c_k} \|x_i - \mu_k\|^2.$$

Minimizing this objective function is known to be an NP-hard problem (even for  $K = 2$ ) (Drineas et al., 1999). Thus K-means, which is a greedy algorithm, can only converge to a local minimum, even though recent study has shown with a large probability K-means could converge to the global optimum when clusters are well separated (Meila, 2006). K-means starts with an initial partition with  $K$  clusters and assign patterns to clusters so as to reduce the squared error. Since the squared error always decreases with an increase in the number of clusters  $K$  (with  $J(C) = 0$  when  $K = n$ ), it can be minimized only for a fixed number of clusters. The main steps of K-means algorithm are as follows (Jain and Dubes, 1988):

1. Select an initial partition with  $K$  clusters; repeat steps 2 and 3 until cluster membership stabilizes.
2. Generate a new partition by assigning each pattern to its closest cluster center.
3. Compute new cluster centers.

**Fig. 4** shows an illustration of the K-means algorithm on a 2-dimensional dataset with three clusters.

#### 2.4. Parameters of K-means

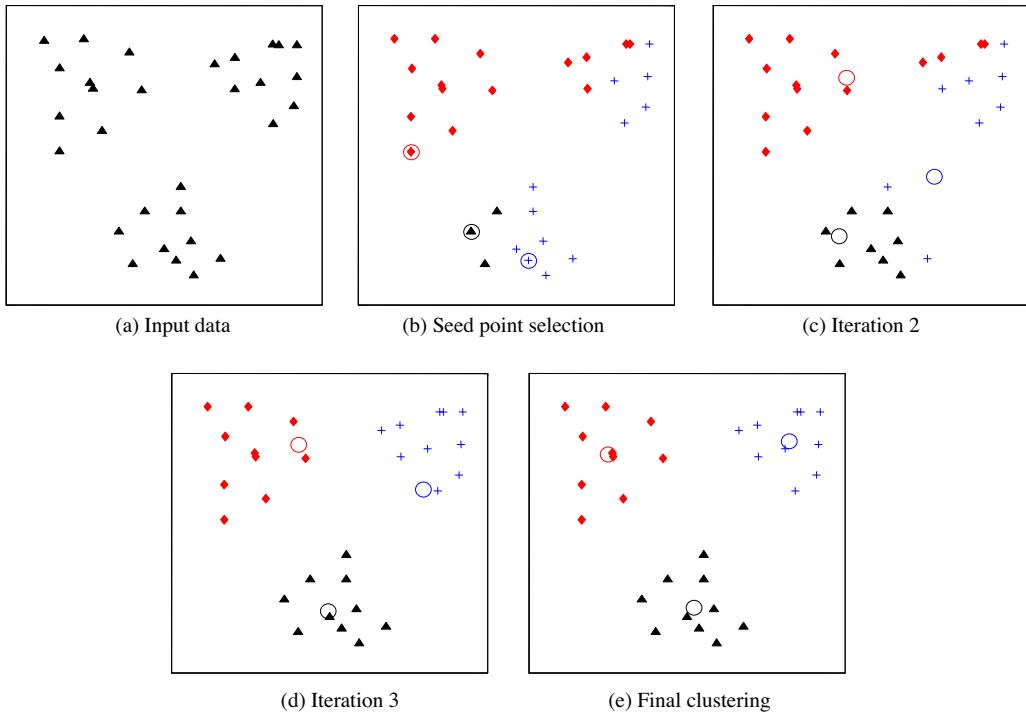
The K-means algorithm requires three user-specified parameters: number of clusters  $K$ , cluster initialization, and distance metric. The most critical choice is  $K$ . While no perfect mathematical criterion exists, a number of heuristics (see (Tibshirani et al.,

2001), and discussion therein) are available for choosing  $K$ . Typically, K-means is run independently for different values of  $K$  and the partition that appears the most meaningful to the domain expert is selected. Different initializations can lead to different final clustering because K-means only converges to local minima. One way to overcome the local minima is to run the K-means algorithm, for a given  $K$ , with multiple different initial partitions and choose the partition with the smallest squared error.

K-means is typically used with the Euclidean metric for computing the distance between points and cluster centers. As a result, K-means finds spherical or ball-shaped clusters in data. K-means with Mahalanobis distance metric has been used to detect hyperellipsoidal clusters (Mao and Jain, 1996), but this comes at the expense of higher computational cost. A variant of K-means using the Itakura–Saito distance has been used for vector quantization in speech processing (Linde et al., 1980) and K-means with  $L_1$  distance was proposed in (Kashima et al., 2008). Banerjee et al. (2004) exploits the family of Bregman distances for K-means.

#### 2.5. Extensions of K-means

The basic K-means algorithm has been extended in many different ways. Some of these extensions deal with additional heuristics involving the minimum cluster size and merging and splitting clusters. Two well-known variants of K-means in pattern recognition literature are ISODATA Ball and Hall (1965) and FORGY Forgé (1965). In K-means, each data point is assigned to a single cluster (called *hard assignment*). Fuzzy  $c$ -means, proposed by Dunn (1973) and later improved by Bezdek (1981), is an extension of K-means where each data point can be a member of multiple clusters with a membership value (*soft assignment*). A good overview of fuzzy set based clustering is available in (Backer, 1978). Data reduction by replacing group examples with their centroids before clustering them was used to speed up K-means and fuzzy C-means in (Eschrich et al., 2003). Some of the other significant modifica-



**Fig. 4.** Illustration of K-means algorithm. (a) Two-dimensional input data with three clusters; (b) three seed points selected as cluster centers and initial assignment of the data points to clusters; (c) and (d) intermediate iterations updating cluster labels and their centers; (e) final clustering obtained by K-means algorithm at convergence.

tions are summarized below. Steinbach et al. (2000) proposed a hierarchical divisive version of K-means, called *bisecting K-means*, that recursively partitions the data into two clusters at each step. In (Pelleg and Moore, 1999), *kd-tree* is used to efficiently identify the closest cluster centers for all the data points, a key step in K-means. Bradley et al. (1998) presented a fast scalable and single-pass version of K-means that does not require all the data to be fit in the memory at the same time. *X-means* (Pelleg and Moore, 2000) automatically finds  $K$  by optimizing a criterion such as Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC). In *K-medoid* (Kaufman and Rousseeuw, 2005), clusters are represented using the median of the data instead of the mean. *Kernel K-means* (Scholkopf et al., 1998) was proposed to detect arbitrary shaped clusters, with an appropriate choice of the kernel similarity function. Note that all these extensions introduce some additional algorithmic parameters that must be specified by the user.

## 2.6. Major approaches to clustering

As mentioned before, thousands of clustering algorithms have been proposed in the literature in many different scientific disciplines. This makes it extremely difficult to review all the published approaches. Nevertheless, clustering methods differ in the choice of the objective function, probabilistic generative models, and heuristics. We will briefly review some of the major approaches.

Clusters can be defined as high density regions in the feature space separated by low density regions. Algorithms following this notion of clusters directly search for connected dense regions in the feature space. Different algorithms use different definitions of connectedness. The Jarvis–Patrick algorithm defines the similarity between a pair of points as the number of common neighbors they share, where neighbors are the points present in a region of pre-specified radius around the point (Frank and Todeschini, 1994). Ester et al. (1996) proposed the DBSCAN clustering algorithm, which is similar to the Jarvis–Patrick algorithm. It directly searches for connected dense regions in the feature space by estimating the density using the Parzen window method. The performance of the Jarvis–Patrick algorithm and DBSCAN depend on two parameters: neighborhood size in terms of distance, and the minimum number of points in a neighborhood for its inclusion in a cluster. In addition, a number of probabilistic models have been developed for data clustering that model the density function by a probabilistic mixture model. These approaches assume that the data is generated from a mixture distribution, where each cluster is described by one or more mixture components (McLachlan and Basford, 1987). The EM algorithm (Dempster et al., 1977) is often used to infer the parameters in mixture models. Several Bayesian approaches have been developed to improve the mixture models for data clustering, including Latent Dirichlet Allocation (LDA) (Blei et al., 2003), Pachinko Allocation model (Li and McCallum, 2006) and undirected graphical model for data clustering (Welling et al., 2005).

While the density based methods, particularly the non-parametric density based approaches, are attractive because of their inherent ability to deal with arbitrary shaped clusters, they have limitations in handling high-dimensional data. When the data is high-dimensional, the feature space is usually sparse, making it difficult to distinguish high-density regions from low-density regions. Subspace clustering algorithms overcome this limitation by finding clusters embedded in low-dimensional subspaces of the given high-dimensional data. CLIQUE (Agrawal et al., 1998) is a scalable clustering algorithm designed to find subspaces in the data with high-density clusters. Because it estimates the density

only in a low dimensional subspace, CLIQUE does not suffer from the problem of high dimensionality.

Graph theoretic clustering, sometimes referred to as spectral clustering, represents the data points as nodes in a weighted graph. The edges connecting the nodes are weighted by their pair-wise similarity. The central idea is to partition the nodes into two subsets  $A$  and  $B$  such that the cut size, i.e., the sum of the weights assigned to the edges connecting between nodes in  $A$  and  $B$ , is minimized. Initial algorithms solved this problem using the minimum cut algorithm, which often results in clusters of imbalanced sizes. A cluster size (number of data points in a cluster) constraint was later adopted by the ratio cut algorithm (Hagen and Kahng, 1992). An efficient approximate graph-cut based clustering algorithm with cluster size (volume of the cluster, or sum of edge weights within a cluster) constraint, called Normalized Cut, was first proposed by Shi and Malik (2000). Its multi-class version was proposed by Yu and Shi (2003). Meila and Shi (2001) presented a Markov Random Walk view of spectral clustering and proposed the Modified Normalized Cut (MNCut) algorithm that can handle an arbitrary number of clusters. Another variant of spectral clustering algorithm was proposed by Ng et al. (2001), where a new data representation is derived from the normalized eigenvectors of a kernel matrix. Laplacian Eigenmap (Belkin and Niyogi, 2002) is another spectral clustering method that derives the data representation based on the eigenvectors of the graph Laplacian. Hofmann and Buhmann (1997) proposed a deterministic annealing algorithm for clustering data represented using proximity measures between the data objects. Pavan and Pelillo (2007) formulate the pair-wise clustering problem by relating clusters to maximal dominant sets (Motzkin and Straus, 1965), which are a continuous generalization of cliques in a graph.

Several clustering algorithms have an information theoretic formulation. For example, the minimum entropy method presented in Roberts et al. (2001) assumes that the data is generated using a mixture model and each cluster is modeled using a semi-parametric probability density. The parameters are estimated by maximizing the KL-divergence between the unconditional density and the conditional density of a data points conditioned over the cluster. This minimizes the overlap between the conditional and unconditional densities, thereby separating the clusters from each other. In other words, this formulation results in an approach that minimizes the expected entropy of the partitions over the observed data. The information bottleneck method (Tishby et al., 1999) was proposed as a generalization to the rate-distortion theory and adopts a lossy data compression view. In simple words, given a joint distribution over two random variables, information bottleneck compresses one of the variables while retaining the maximum amount of mutual information with respect to the other variable. An application of this to document clustering is shown in (Slonim and Tishby, 2000) where the two random variables are words and documents. The words are clustered first, such that the mutual information with respect to documents is maximally retained, and using the clustered words, the documents are clustered such that the mutual information between clustered words and clustered documents is maximally retained.

## 3. User's dilemma

In spite of the prevalence of such a large number of clustering algorithms, and their success in a number of different application domains, clustering remains a difficult problem. This can be attributed to the inherent vagueness in the definition of a cluster, and the difficulty in defining an appropriate similarity measure and objective function.

The following fundamental challenges associated with clustering were highlighted in (Jain and Dubes, 1988), which are relevant even to this day.

- (a) What is a cluster?
- (b) What features should be used?
- (c) Should the data be normalized?
- (d) Does the data contain any outliers?
- (e) How do we define the pair-wise similarity?
- (f) How many clusters are present in the data?
- (g) Which clustering method should be used?
- (h) Does the data have any clustering tendency?
- (i) Are the discovered clusters and partition valid?

We will highlight and illustrate some of these challenges below.

### 3.1. Data representation

*Data representation* is one of the most important factors that influence the performance of the clustering algorithm. If the representation (choice of features) is good, the clusters are likely to be compact and isolated and even a simple clustering algorithm such as K-means will find them. Unfortunately, there is no universally good representation; the choice of representation must be guided by the domain knowledge. Fig. 5a shows a dataset where K-means fails to partition it into the two “natural” clusters. The partition obtained by K-means is shown by a dashed line in Fig. 5a. However, when the same data points in (a) are represented using the top two eigenvectors of the RBF similarity matrix computed from the data in Fig. 5b, they become well separated, making it trivial for K-means to cluster the data (Ng et al., 2001).

### 3.2. Purpose of grouping

The representation of the data is closely tied with the purpose of grouping. The representation must go hand in hand with the end goal of the user. An example dataset of 16 animals represented using 13 Boolean features was used in (Pampalk et al., 2003) to demonstrate how the representation affects the grouping. The animals are represented using 13 Boolean features related to their appearance and activity. When a large weight is placed on the appearance features compared to the activity features, the animals were clustered into *mammals* vs. *birds*. On the other hand, a large weight on the activity features clustered the dataset into *predators*

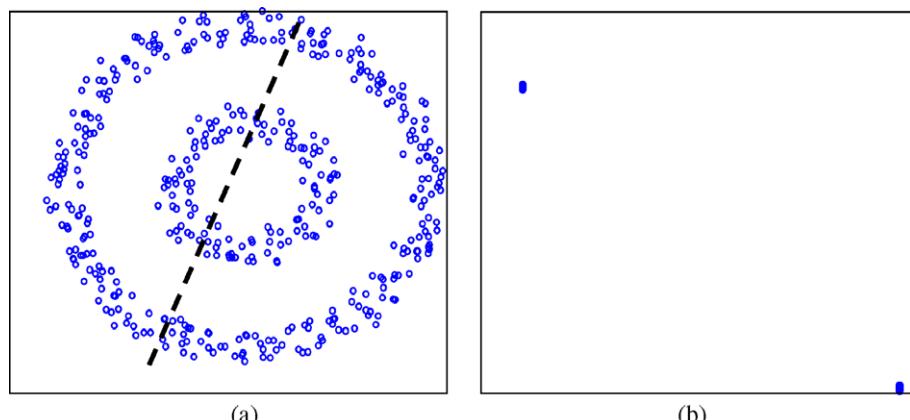
vs. *non-predators*. Both these partitionings shown in Fig. 6 are equally valid, and they uncover meaningful structures in the data. It is up to the user to carefully choose his representation to obtain a desired clustering.

### 3.3. Number of clusters

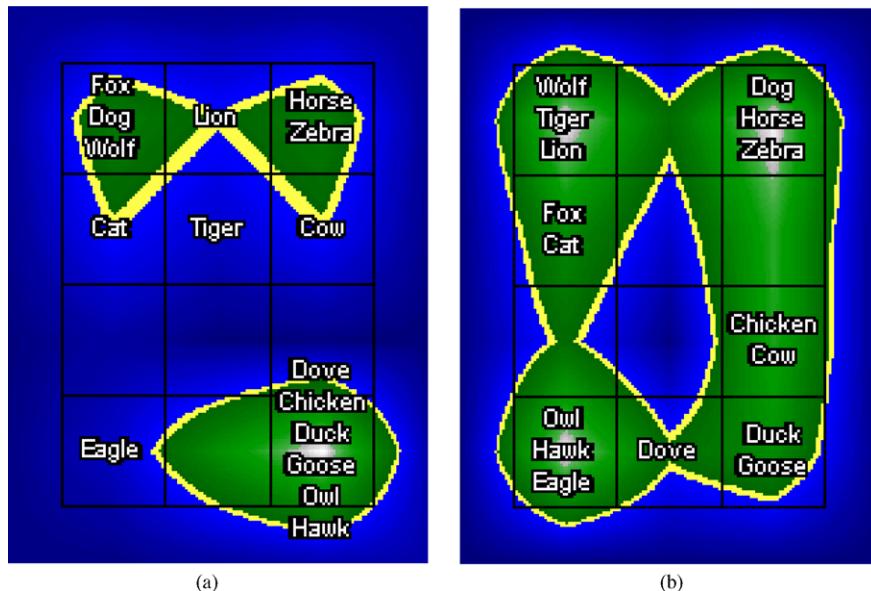
Automatically determining the number of clusters has been one of the most difficult problems in data clustering. Most methods for automatically determining the number of clusters cast it into the problem of model selection. Usually, clustering algorithms are run with different values of  $K$ ; the best value of  $K$  is then chosen based on a predefined criterion. Figueiredo and Jain (2002) used the minimum message length (MML) criteria (Wallace and Boulton, 1968; Wallace and Freeman, 1987) in conjunction with the Gaussian mixture model (GMM) to estimate  $K$ . Their approach starts with a large number of clusters, and gradually merges the clusters if this leads to a decrease in the MML criterion. A related approach but using the principle of Minimum Description Length (MDL) was used in (Hansen and Yu, 2001) for selecting the number of clusters. The other criteria for selecting the number of clusters are the Bayes Information Criterion (BIC) and Akaike Information Criterion (AIC). Gap statistics (Tibshirani et al., 2001) is another commonly used approach for deciding the number of clusters. The key assumption is that when dividing data into an optimal number of clusters, the resulting partition is most resilient to the random perturbations. The Dirichlet Process (DP) (Ferguson, 1973; Rasmussen, 2000) introduces a non-parametric prior for the number of clusters. It is often used by probabilistic models to derive a posterior distribution for the number of clusters, from which the most likely number of clusters can be computed. In spite of these objective criteria, it is not easy to decide which value of  $K$  leads to more meaningful clusters. Fig. 7a shows a two-dimensional synthetic dataset generated from a mixture of six Gaussian components. The true labels of the points are shown in Fig. 7e. When a mixture of Gaussians is fit to the data with 2, 5, and 6 components, shown in Fig. 7b-d, respectively, each one of them seems to be a reasonable fit.

### 3.4. Cluster validity

Clustering algorithms tend to find clusters in the data irrespective of whether or not any clusters are present. Fig. 8a shows a dataset with no *natural* clustering; the points here were generated



**Fig. 5.** Importance of a good representation. (a) “Two rings” dataset where K-means fails to find the two “natural” clusters; the dashed line shows the linear cluster separation boundary obtained by running K-means with  $K = 2$ . (b) a new representation of the data in (a) based on the top 2 eigenvectors of the graph Laplacian of the data, computed using an RBF kernel; K-means now can easily detect the two clusters.

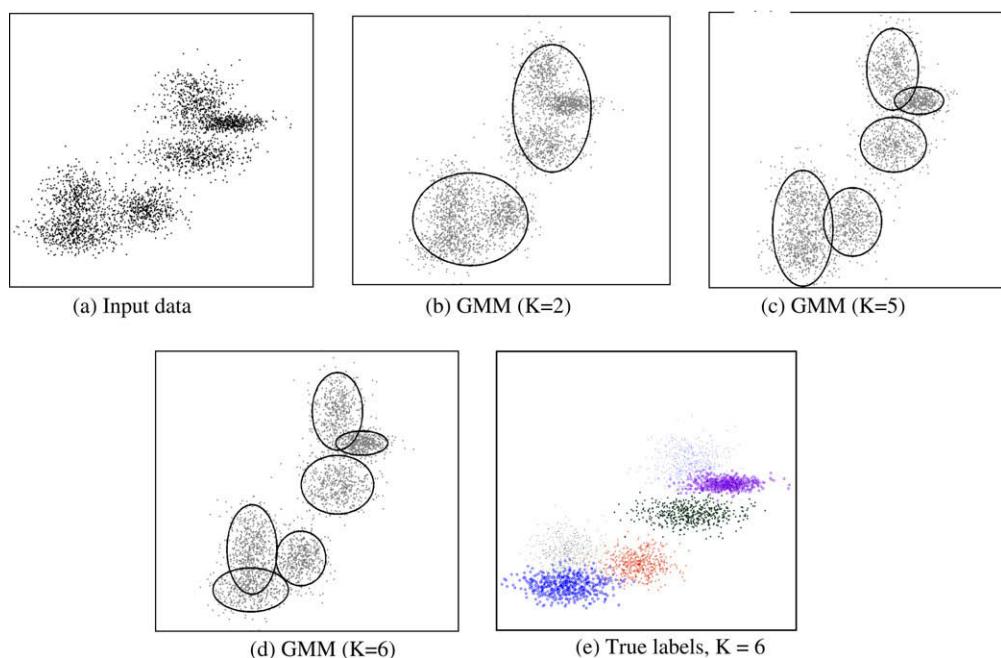


**Fig. 6.** Different weights on features result in different partitioning of the data. Sixteen animals are represented based on 13 Boolean features related to appearance and activity. (a) Partitioning with large weights assigned to the appearance based features; (b) a partitioning with large weights assigned to the activity features. The figures in (a) and (b) are excerpted from Pampalk et al. (2003), and are known as “heat maps” where the colors represent the density of samples at a location; the warmer the color, the larger the density.

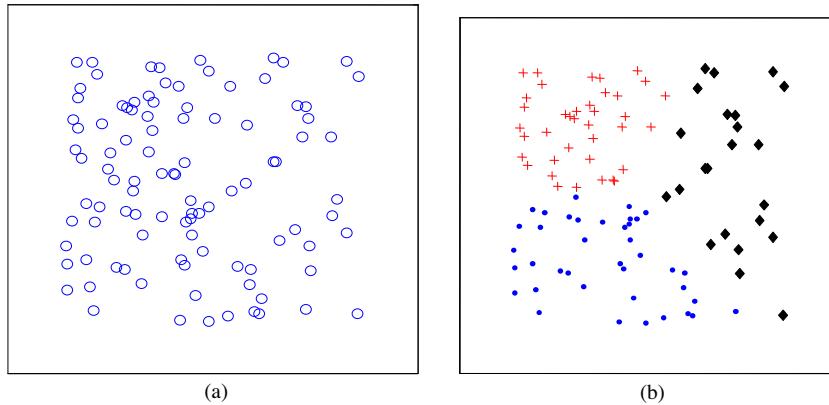
uniformly in a unit square. However, when the K-means algorithm is run on this data with  $K = 3$ , three clusters are identified as shown in Fig. 8b! Cluster validity refers to formal procedures that evaluate the results of cluster analysis in a quantitative and objective fashion (Jain and Dubes, 1988). In fact, even before a clustering algorithm is applied to the data, the user should determine if the data even has a clustering tendency (Smith and Jain, 1984).

Cluster validity indices can be defined based on three different criteria: *internal*, *relative*, and *external* (Jain and Dubes, 1988). Indices based on *internal criteria* assess the fit between the structure imposed by the clustering algorithm (clustering) and the data

using the data alone. Indices based on *relative criteria* compare multiple structures (generated by different algorithms, for example) and decide which of them is better in some sense. *External indices* measure the performance by matching cluster structure to the a priori information, namely the “true” class labels (often referred to as ground truth). Typically, clustering results are evaluated using the external criterion, but if the true labels are available, why even bother with clustering? The notion of *cluster stability* (Lange et al., 2004) is appealing as an internal stability measure. Cluster stability is measured as the amount of variation in the clustering solution over different subsamples drawn from



**Fig. 7.** Automatic selection of number of clusters,  $K$ . (a) Input data generated from a mixture of six Gaussian distributions; (b)–(d) Gaussian mixture model (GMM) fit to the data with 2, 5, and 6 components, respectively; and (e) true labels of the data.



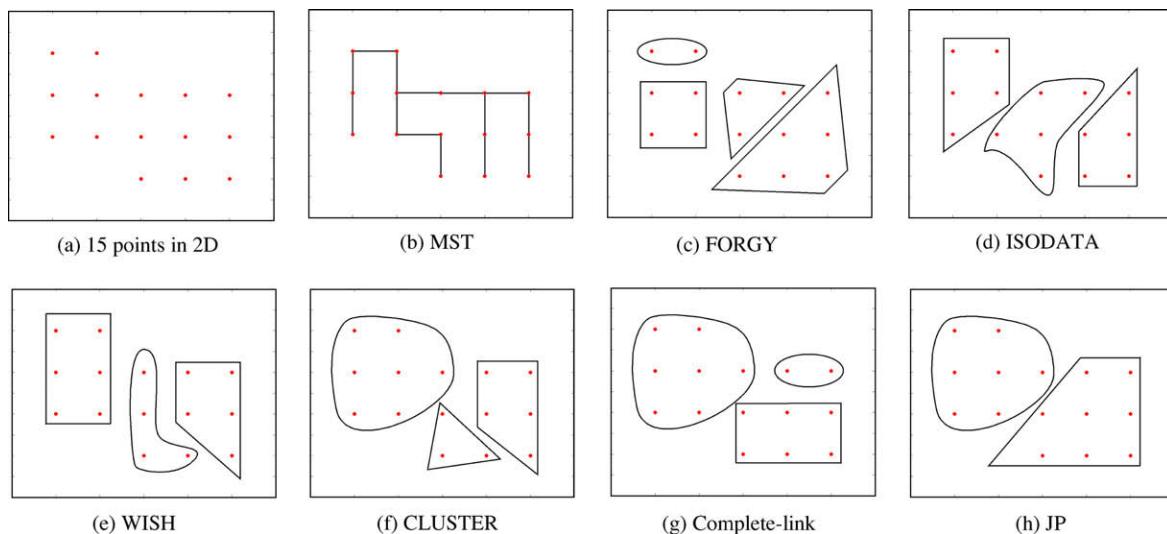
**Fig. 8.** Cluster validity. (a) A dataset with no “natural” clustering; (b) K-means partition with  $K = 3$ .

the input data. Different measures of variation can be used to obtain different stability measures. In (Lange et al., 2004), a supervised classifier is trained on one of the subsamples of the data, by using the cluster labels obtained by clustering the subsample, as the *true* labels. The performance of this classifier on the testing subset(s) indicates the stability of the clustering algorithm. In model based algorithms (e.g., centroid based representation of clusters in K-means, or Gaussian mixture models), the distance between the models found for different subsamples can be used to measure the stability (von Luxburg and David, 2005). Shamir and Tishby (2008) define stability as the generalization ability of a clustering algorithm (in PAC-Bayesian sense). They argue that since many algorithms can be shown to be asymptotically stable, the rate at which the asymptotic stability is reached with respect to the number of samples is a more useful measure of cluster stability. Cross-validation is a widely used evaluation method in supervised learning. It has been adapted to unsupervised learning by replacing the notation of “prediction accuracy” with a different validity measure. For example, given the mixture models obtained from the data in one fold, the likelihood of the data in the other folds serves as an indication of the algorithm’s performance, and can be used to determine the number of clusters  $K$ .

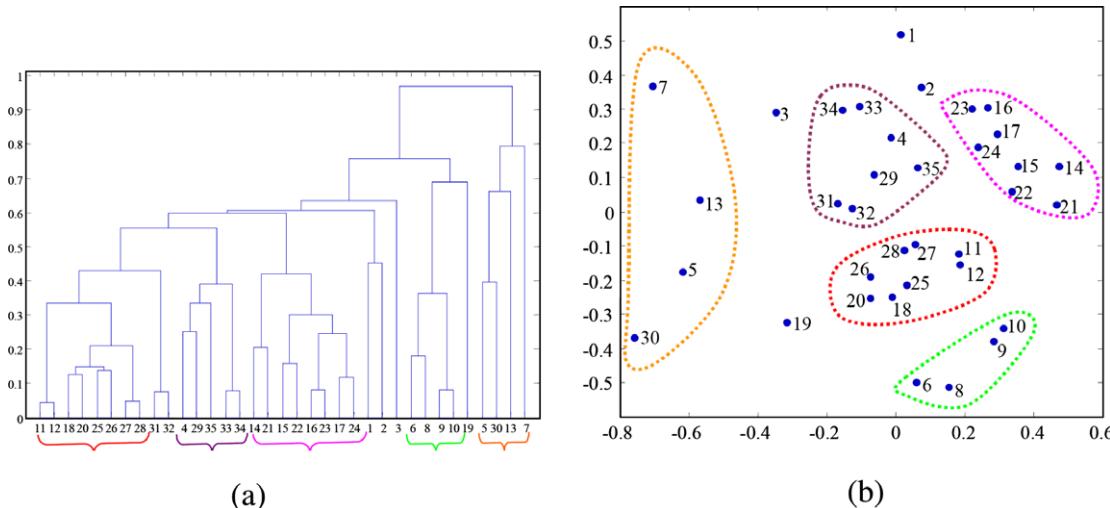
### 3.5. Comparing clustering algorithms

Different clustering algorithms often result in entirely different partitions even on the same data. In Fig. 9, seven different algorithms were applied to cluster the 15 two-dimensional points. FORGY, ISODATA, CLUSTER, and WISH are partitional algorithms that minimize the squared error criterion (they are variants of the basic K-means algorithm). Of the remaining three algorithms, MST (minimum spanning tree) can be viewed as a single-link hierarchical algorithm, and JP is a nearest neighbor clustering algorithm. Note that a hierarchical algorithm can be used to generate a partition by specifying a threshold on the similarity. It is evident that none of the clustering is superior to the other, but some are similar to the other.

An interesting question is to identify algorithms that generate similar partitions irrespective of the data. In other words, can we cluster the clustering algorithms? Jain et al. (2004) clustered 35 different clustering algorithms into 5 groups based on their partitions on 12 different datasets. The similarity between the clustering algorithms is measured as the averaged similarity between the partitions obtained on the 12 datasets. The similarity between a pair of partitions is measured using the Adjusted Rand Index (ARI). A hierarchical clustering of the 35 clustering algorithms is



**Fig. 9.** Several clusterings of fifteen patterns in two dimensions: (a) fifteen patterns; (b) minimum spanning tree of the fifteen patterns; (c) clusters from FORGY; (d) clusters from ISODATA; (e) clusters from WISH; (f) clusters from CLUSTER; (g) clusters from complete-link hierarchical clustering; and (h) clusters from Jarvis-Patrick clustering algorithm. (Figure reproduced from Dubes and Jain (1976)).



**Fig. 10.** Clustering of clustering algorithms. (a) Hierarchical clustering of 35 different algorithms; (b) Sammon's mapping of the 35 algorithms into a two-dimensional space, with the clusters highlighted for visualization. The algorithms in the group (4, 29, 31–35) correspond to K-means, spectral clustering, Gaussian mixture models, and Ward's linkage. The algorithms in group (6, 8–10) correspond to CHAMELEON algorithm with different objective functions.

shown in Fig. 10a. It is not surprising to see that the related algorithms are clustered together. For a visualization of the similarity between the algorithms, the 35 algorithms are also embedded in a two-dimensional space; this is achieved by applying the Sammon's projection algorithm (Sammon, 1969) to the  $35 \times 35$  similarity matrix. Fig. 10b shows that all the CHAMELEON variations (6, 8–10) are clustered into a single cluster. This plot suggests that the clustering algorithms following the same clustering strategy result in similar clustering in spite of minor variations in the parameters or objective functions involved. In (Meila, 2003), a different metric in the space of clusterings, termed Variation of Information, was proposed. It measures the similarity between two clustering algorithms by the amount of information lost or gained when choosing one clustering over the other.

Clustering algorithms can also be compared at the theoretical level based on their objective functions. In order to perform such a comparison, a distinction should be made between a *clustering method* and a *clustering algorithm* (Jain and Dubes, 1988). A clustering method is a general strategy employed to solve a clustering problem. A clustering algorithm, on the other hand, is simply an instance of a method. For instance, minimizing the squared error is a clustering method, and there are many different clustering algorithms, including K-means, that implement the minimum squared error method. Some equivalence relationships even between different clustering methods have been shown. For example, Dhillon et al. (2004) show that spectral methods and kernel K-means are equivalent; for a choice of kernel in spectral clustering, there exists a kernel for which the objective functions of kernel K-means and spectral clustering are the same. The equivalence between non-negative matrix factorization for clustering and kernel K-means algorithm is shown in (Ding et al., 2005). All these methods are directly related to the analysis of eigenvectors of the similarity matrix.

The above discussion underscores one of the important facts about clustering; *there is no best clustering algorithm*. Each clustering algorithm imposes a structure on the data either explicitly or implicitly. When there is a good match between the model and the data, good partitions are obtained. Since the structure of the data is not known a priori, one needs to try competing and diverse approaches to determine an appropriate algorithm for the clustering task at hand. This idea of no best clustering algorithm is partially captured by the impossibility theorem (Kleinberg, 2002), which states that no single clustering algorithm simultaneously satisfies a set of basic axioms of data clustering.

### 3.6. Admissibility analysis of clustering algorithms

Fisher and vanNess (1971) formally analyzed clustering algorithms with the objective of comparing them and providing guidance in choosing a clustering procedure. They defined a set of *admissibility criteria* for clustering algorithms that test the sensitivity of clustering algorithms with respect to the changes that do not alter the essential structure of the data. A clustering is called *A-admissible* if it satisfies criterion A. Example criteria include *convex*, *point and cluster proportion*, *cluster omission*, and *monotone*. They are briefly described below.

- *Convex*: A clustering algorithm is *convex-admissible* if it results in a clustering where the convex hulls of clusters do not intersect.
- *Cluster proportion*: A clustering algorithm is *cluster-proportion admissible* if the cluster boundaries do not alter even if some of the clusters are duplicated an arbitrary number of times.
- *Cluster omission*: A clustering algorithm is *omission-admissible* if by removing one of the clusters from the data and re-running the algorithm, the clustering on the remaining  $K-1$  clusters is identical to the one obtained on them with  $K$  clusters.
- *Monotone*: A clustering algorithm is *monotone-admissible* if the clustering results do not change when a monotone transformation is applied to the elements of the similarity matrix.

Fisher and Van Ness proved that one cannot construct algorithms that satisfy certain admissibility criteria. For example, if an algorithm is monotone-admissible, it cannot be a hierarchical clustering algorithm.

Kleinberg (2002) addressed a similar problem, where he defined three criteria:

- *Scale invariance*: An arbitrary scaling of the similarity metric must not change the clustering results.
- *Richness*: The clustering algorithm must be able to achieve all possible partitions on the data.
- *Consistency*: By shrinking within-cluster distances and stretching between-cluster distances, the clustering results must not change.

Kleinberg also provides results similar to that of (Fisher and vanNess, 1971), showing that it is impossible to construct an algorithm that satisfies all these properties, hence the title of his paper

“An Impossibility Theorem for Clustering”. Further discussions in (Kleinberg, 2002) reveal that a clustering algorithm can indeed be designed by relaxing the definition of *satisfying* a criterion to *nearly-satisfying* the criterion. While the set of axioms defined here are reasonable to a large extent, they are in no way the only possible set of axioms, and hence the results must be interpreted accordingly (Ben-David and Ackerman, 2008).

#### 4. Trends in data clustering

Information explosion is not only creating large amounts of data but also a diverse set of data, both *structured* and *unstructured*. *Unstructured data* is a collection of objects that do not follow a specific format. For example, images, text, audio, video, etc. On the other hand, in *structured data*, there are semantic relationships within each object that are important. Most clustering approaches ignore the structure in the objects to be clustered and use a feature vector based representation for both structured and unstructured data. The traditional view of data partitioning based on vector-based feature representation does not always serve as an adequate framework. Examples include objects represented using sets of points (Lowe, 2004), consumer purchase records (Guha et al., 2000), data collected from questionnaires and rankings (Critchlow, 1985), social networks (Wasserman and Faust, 1994), and data streams (Guha et al., 2003b). Models and algorithms are being developed to process huge volumes of heterogeneous data. A brief summary of some of the recent trends in data clustering is presented below.

##### 4.1. Clustering ensembles

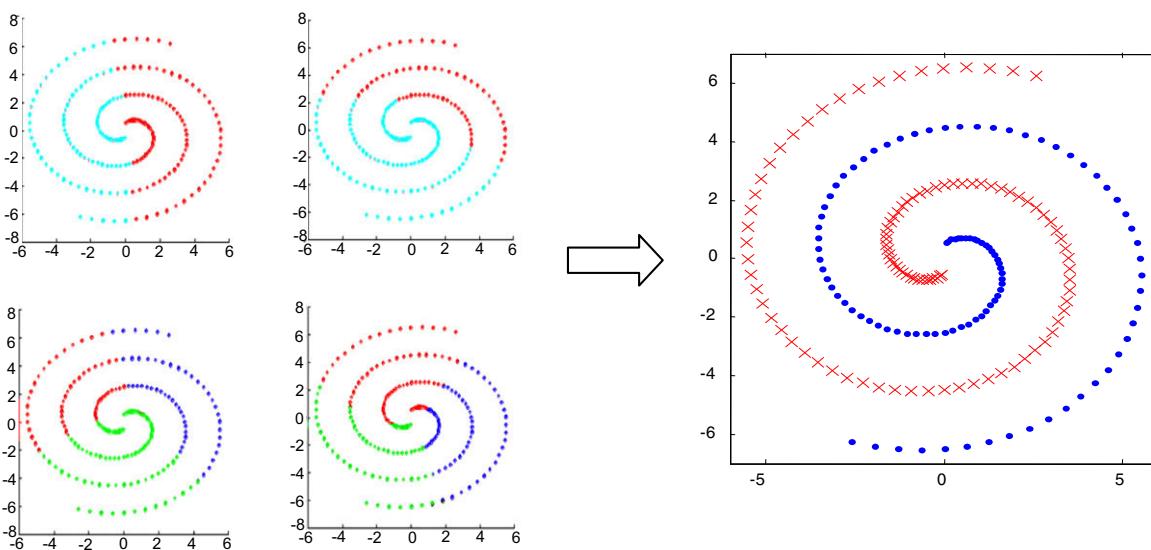
The success of ensemble methods for supervised learning has motivated the development of ensemble methods for unsupervised learning (Fred and Jain, 2002). The basic idea is that by taking *multiple looks* at the same data, one can generate multiple partitions (*clustering ensemble*) of the same data. By combining the resulting partitions, it is possible to obtain a good data partitioning even when the clusters are not compact and well separated. Fred and Jain used this approach by taking an ensemble of partitions obtained by K-means; the ensemble was obtained by changing the value of K and using random cluster initializations. These parti-

tions were then combined using a co-occurrence matrix that resulted in a good separation of the clusters. An example of a clustering ensemble is shown in Fig. 11 where a “two-spiral” dataset is used to demonstrate its effectiveness. K-means is run multiple, say N, times with varying values of the number of clusters K. The new similarity between a pair of points is defined as the number of times the two points co-occur in the same cluster in N runs of K-means. The final clustering is obtained by clustering the data based on the new pair-wise similarity. Strehl and Ghosh (2003) proposed several probabilistic models for integrating multiple partitions. More recent work on cluster ensembles can be found in (Hore et al., 2009a).

There are many different ways of generating a clustering ensemble and then combining the partitions. For example, multiple data partitions can be generated by: (i) applying different clustering algorithms, (ii) applying the same clustering algorithm with different values of parameters or initializations, and (iii) combining of different data representations (feature spaces) and clustering algorithms. The evidence accumulation step that combines the information provided by the different partitions can be viewed as learning the similarity measure among the data points.

##### 4.2. Semi-supervised clustering

Clustering is inherently an ill-posed problem where the goal is to partition the data into some unknown number of clusters based on intrinsic information alone. The data-driven nature of clustering makes it very difficult to design clustering algorithms that will correctly find clusters in the given data. Any external or *side information* available along with the  $n \times d$  pattern matrix or the  $n \times n$  similarity matrix can be extremely useful in finding a good partition of data. Clustering algorithms that utilize such side information are said to be operating in a *semi-supervised mode* (Chapelle et al., 2006). There are two open questions: (i) how should the side information be specified? and (ii) how is it obtained in practice? One of the most common methods of specifying the side information is in the form of pair-wise constraints. A *must-link constraint* specifies that the point pair connected by the constraint belong to the same cluster. On the other hand, a *cannot-link constraint* specifies that the point pair connected by the constraint do not belong to the same cluster. It is generally assumed that the con-



**Fig. 11.** Clustering ensembles. Multiple runs of K-means are used to learn the pair-wise similarity using the “co-occurrence” of points in clusters. This similarity can be used to detect arbitrary shaped clusters.

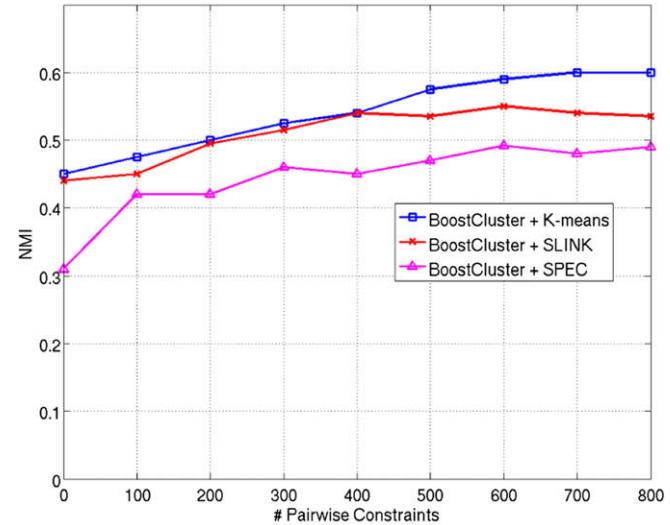
straints are provided by the domain expert. There is limited work on automatically deriving constraints from the data. Some attempts to derive constraints from domain ontology and other external sources into clustering algorithms include the usage of WordNet ontology, gene ontology, Wikipedia, etc. to guide clustering solutions. However, these are mostly feature constraints and not constraints on the instances (Hotho et al., 2003; Liu et al., 2004; Banerjee et al., 2007b). Other approaches for including side information include (i) “seeding”, where some labeled data is used along with large amount of unlabeled data for better clustering (Basu et al., 2002) and (ii) methods that allow encouraging or discouraging some links (Law et al., 2005; Figueiredo et al., 2006).

**Fig. 12** illustrates the semi-supervised learning in an image segmentation application (Lange et al., 2005). The textured image to be segmented (clustered) is shown in **Fig. 12a**. In addition to the image, a set of user-specified pair-wise constraints on the pixel labels are also provided. **Fig. 12b** shows the clustering obtained when no constraints are used, while **Fig. 12c** shows improved clustering with the use of constraints. In both the cases, the number of clusters was assumed to be known ( $K = 5$ ).

Most approaches (Bar-Hillel et al., 2003; Basu et al., 2004; Chapelle et al., 2006; Lu and Leen, 2007) to semi-supervised clustering modify the objective function of existing clustering algorithms to incorporate the pair-wise constraints. It is desirable to have an approach to semi-supervised clustering that can improve the performance of an already existing clustering algorithm without modifying it. *BoostCluster* (Liu et al., 2007) adopts this philosophy and follows a boosting framework to improve the performance of any given clustering algorithm using pair-wise constraints. It iteratively modifies the input to the clustering algorithm by generating new data representations (transforming the  $n \times n$  similarity matrix) such that the pair-wise constraints are satisfied while also maintaining the integrity of the clustering output. **Fig. 13** shows the performance of BoostCluster evaluated on handwritten digit database in the UCI repository (Blake, 1998) with 4000 points in 256-dimensional feature space. BoostCluster is able to improve the performance of all the three commonly used clustering algorithms, K-means, single-link, and Spectral clustering as pair-wise constraints are added to the data. Only must-link constraints are specified here and the number of true clusters is assumed to be known ( $K = 10$ ).

#### 4.3. Large-scale clustering

Large-scale data clustering addresses the challenge of clustering millions of data points that are represented in thousands of features. **Table 1** shows a few examples of real-world applications

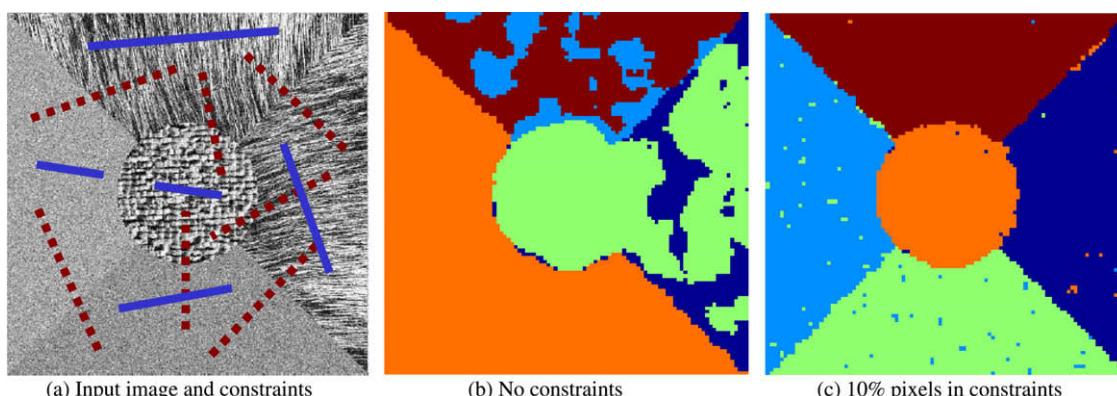


**Fig. 13.** Performance of BoostCluster (measured using Normalized Mutual Information (NMI)) as the number of pair-wise constraints is increased. The three plots correspond to boosted performance of K-means, Single-Link (SLINK), and Spectral clustering (SPEC).

for large-scale data clustering. Below, we review the application of large-scale data clustering to content-based image retrieval.

The goal of Content Based Image Retrieval (CBIR) is to retrieve visually similar images to a given query image. Although the topic has been studied for the past 15 years or so, there has been only limited success. Most early work on CBIR was based on computing color, shape, and texture based features and using them to define a similarity between the images. A 2008 survey on CBIR highlights the different approaches used for CBIR through time (Datta et al., 2008). Recent approaches for CBIR use key point based features. For example, SIFT (Lowe, 2004) descriptors can be used to represent the images (see **Fig. 14**). However, once the size of the image database increases (~10 million), and assuming 10 ms to compute the matching score between an image pair, a linear search would take approximately 30 h to answer one query. This clearly is unacceptable.

On the other hand, text retrieval applications are much faster. It takes about one-tenth of a second to search 10 billion documents in Google. A novel approach for image retrieval is to convert the problem into a text retrieval problem. The key points from all the images are first clustered into a large number of clusters (which is usually much less than the number of key points



**Fig. 12.** Semi-supervised learning. (a) Input image consisting of five homogeneous textured regions; examples of must-link (solid blue lines) and must not link (broken red lines) constraints between pixels to be clustered are specified. (b) 5-Cluster solution (segmentation) without constraints. (c) Improved clustering (with five clusters) with 10% of the data points included in the pair-wise constraints (Lange et al., 2005).

**Table 1**

Example applications of large-scale data clustering.

Application	Description	# Objects	# Features
Document clustering	Group documents of similar topics (Andrews et al., 2007)	$10^6$	$10^4$
Gene clustering	Group genes with similar expression levels (Lukashin et al., 2003)	$10^5$	$10^2$
Content-based image retrieval	Quantize low-level image features (Philbin et al., 2007)	$10^9$	$10^2$
Clustering of earth science data	Derive climate indices (Steinbach et al., 2003)	$10^5$	$10^2$

themselves). These are called *visual words*. An image is then represented by a histogram of visual words, i.e., the number of key-points from the image that are in each word or each cluster. By representing each image by a histogram of visual words, we can then cast the problem of image search into a problem of text retrieval and exploit text search engines for efficient image retrieval. One of the major challenges in quantizing key points is the number of objects to be clustered. For a collection of 1000 images with an average of 1000 key points and target number of 5000 visual words, it requires clustering  $10^6$  objects into 5000 clusters.

A large number of clustering algorithms have been developed to efficiently handle large-size data sets. Most of these studies can be classified into four categories:

- *Efficient Nearest Neighbor (NN) Search*: One of the basic operations in any data clustering algorithm is to decide the cluster membership of each data point, which requires NN search. Algorithms for efficient NN search are either tree-based (e.g. kd-tree (Moore, 1998; Muja and Lowe, 2009)) or random projection based (e.g., Locality Sensitive Hash (Buhler, 2001)).
- *Data summarization*: The objective here is to improve the clustering efficiency by first summarizing a large data set into a relatively small subset, and then applying the clustering algorithms

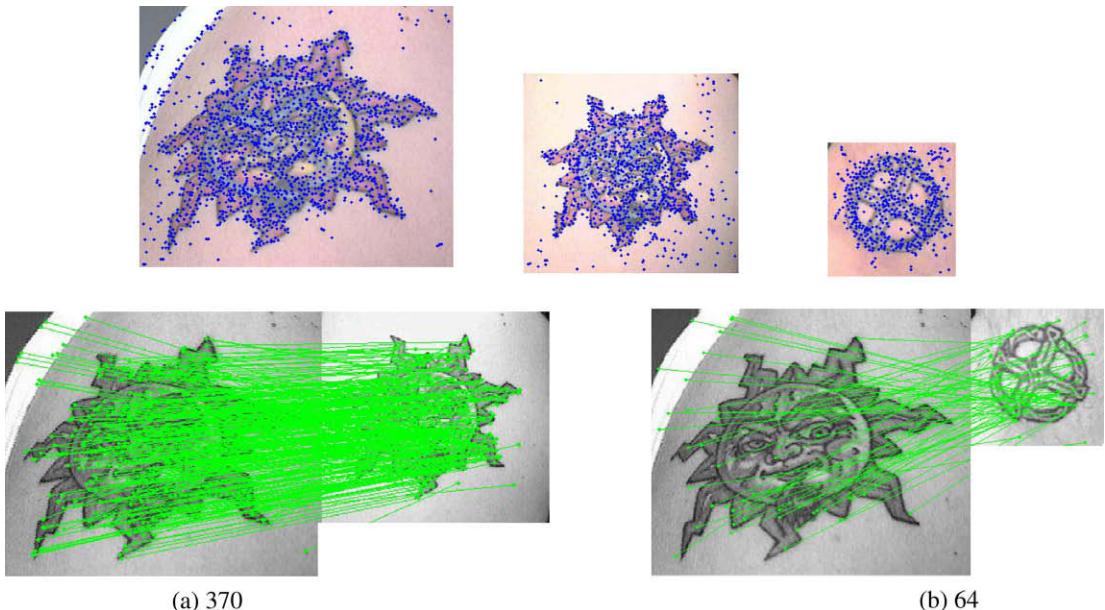
to the summarized data set. Example algorithms include BIRCH (Zhang et al., 1996), divide-and-conquer (Steinbach et al., 2000), coresets K-means (Har-peled and Mazumdar, 2004), and coarsening methods (Karypis and Kumar, 1995).

- *Distributed computing*: Approaches in this category (Dhillon and Modha, 1999) divide each step of a data clustering algorithm into a number of procedures that can be computed independently. These independent computational procedures will then be carried out in parallel by different processors to reduce the overall computation time.
- *Incremental clustering*: These algorithms, for example (Bradley et al., 1998) are designed to operate in a single pass over data points to improve the efficiency of data clustering. This is in contrast to most clustering algorithms that require multiple passes over data points before identifying the cluster centers. COBWEB is a popular hierarchical clustering algorithm that does a single pass through the available data and arranges it into a classification tree incrementally (Fisher, 1987).
- *Sampling-based methods*: Algorithms like CURE (Guha et al., 1998; Kollios et al., 2003) subsample a large dataset selectively, and perform clustering over the smaller set, which is later transferred to the larger dataset.

#### 4.4. Multi-way clustering

Objects or entities to be clustered are often formed by a combination of related heterogeneous components. For example, a document is made of words, title, authors, citations, etc. While objects can be converted into a pooled feature vector of its components prior to clustering, it is not a natural representation of the objects and may result in poor clustering performance.

Co-clustering (Hartigan, 1972; Mirkin, 1996) aims to cluster both features and instances of the data (or both rows and columns of the  $n \times d$  pattern matrix) simultaneously to identify the subset of features where the resulting clusters are meaningful according to certain evaluation criterion. This problem was first studied under the name *direct clustering* by Hartigan (1972). It is also called *bi-dimensional clustering* (Cheng et al., 2000), *double clustering*,



**Fig. 14.** Three tattoo images represented using SIFT key points. (a) A pair of similar images has 370 matching key points; (b) a pair of different images has 64 matching key points. The green lines show the matching key-points between the images (Lee et al., 2008).

*coupled clustering*, or *bimodal clustering*. This notion is also related to subspace clustering where all the clusters are identified in a common subspace. Co-clustering is most popular in the field of bioinformatics, especially in gene clustering, and has also been successfully applied to document clustering (Slonim and Tishby, 2000; Dhillon et al., 2003).

The co-clustering framework was extended to *multi-way clustering* in (Bekkerman et al., 2005) to cluster a set of objects by simultaneously clustering their heterogeneous components. Indeed, the problem is much more challenging because different pairs of components may participate in different types of similarity relationships. In addition, some relations may involve more than two components. Banerjee et al. (2007a) present a family of multi-way clustering schemes that is applicable to a class of loss functions known as Bregman divergences. Sindhwan et al. (2008) apply semi-supervised learning in the co-clustering framework.

#### 4.5. Heterogeneous data

In traditional pattern recognition settings, a feature vector consists of measurements of different properties of an object. This representation of objects is not a natural representation for several types of data. *Heterogeneous data* refers to the data where the objects may not be naturally represented using a fixed length feature vector.

*Rank data:* Consider a dataset generated by ranking of a set of  $n$  movies by different people; only some of the  $n$  objects are ranked. The task is to cluster the users whose rankings are similar and also to identify the ‘representative rankings’ of each group (Mallows, 1957; Critchlow, 1985; Busse et al., 2007).

*Dynamic data:* Dynamic data, as opposed to static data, can change over the course of time e.g., blogs, Web pages, etc. As the data gets modified, clustering must be updated accordingly. A *data stream* is a kind of dynamic data that is transient in nature, and cannot be stored on a disk. Examples include network packets received by a router and stock market, retail chain, or credit card transaction streams. Characteristics of the data streams include their high volume and potentially unbounded size, sequential access, and dynamically evolving nature. This imposes additional requirements to traditional clustering algorithms to rapidly process and summarize the massive amount of continuously arriving data. It also requires the ability to adapt to changes in the data distribution, the ability to detect emerging clusters and distinguish them from outliers in the data, and the ability to merge old clusters or discard expired ones. All of these requirements make data stream clustering a significant challenge since they are expected to be single-pass algorithms (Guha et al., 2003b). Because of the high-speed processing requirements, many of the data stream clustering methods (Guha et al., 2003a; Aggarwal et al., 2003; Cao et al., 2006; Hore et al., 2009b) are extensions of simple algorithms such as K-means, K-medoid, fuzzy c-means, or density-based clustering, modified to work in a data stream environment setting.

*Graph data:* Several objects, such as chemical compounds, protein structures, etc. can be represented most naturally as graphs. Many of the initial efforts in graph clustering focused on extracting graph features to allow existing clustering algorithms to be applied to the graph feature vectors (Tsuda and Kudo, 2006). The features can be extracted based on patterns such as frequent subgraphs, shortest paths, cycles, and tree-based patterns. With the emergence of kernel learning, there have been growing efforts to develop kernel functions that are more suited for graph-based data (Kashima et al., 2003). One way to determine the similarity between graphs is by aligning their corresponding adjacency matrix representations (Umeyama, 1988).

*Relational data:* Another area that has attracted considerable interest is clustering relational (network) data. Unlike the cluster-

ing of graph data, where the objective is to partition a collection of graphs into disjoint groups, the task here is to partition a large graph (i.e., network) into cohesive subgraphs based on their link structure and node attributes. The problem becomes even more complicated when the links (which represent relations between objects) are allowed to have diverse types. One of the key issues is to define an appropriate clustering criterion for relational data. A general probabilistic model for relational data was first proposed in (Taskar et al., 2001), where different related entities are modeled as distributions conditioned on each other. Newman’s modularity function (Newman and Girvan, 2004; Newman, 2006) is a widely-used criterion for finding community structures in networks, but the measure considers only the link structure and ignores attribute similarities. A spectral relaxation to Newman and Girvan’s objective function (Newman and Girvan, 2004) for network graph clustering is presented in (White and Smyth, 2005). Since real networks are often dynamic, another issue is to model the evolutionary behavior of networks, taking into account changes in the group membership and other characteristic features (Backstrom et al., 2006).

## 5. Summary

Organizing data into sensible groupings arises naturally in many scientific fields. It is, therefore, not surprising to see the continued popularity of data clustering. It is important to remember that cluster analysis is an exploratory tool; the output of clustering algorithms only suggest hypotheses. While numerous clustering algorithms have been published and new ones continue to appear, there is no single clustering algorithm that has been shown to dominate other algorithms across all application domains. Most algorithms, including the simple K-means, are admissible algorithms. With the emergence of new applications, it has become increasingly clear that the task of seeking the best clustering principle might indeed be futile. As an example, consider the application domain of enterprise knowledge management. Given the same set of document corpus, different user groups (e.g., legal, marketing, management, etc.) may be interested in generating partitions of documents based on their respective needs. A clustering method that satisfies the requirements for one group of users may not satisfy the requirements of another. As mentioned earlier, “clustering is in the eye of the beholder” – so indeed data clustering must involve the user or application needs.

Clustering has numerous success stories in data analysis. In spite of this, machine learning and pattern recognition communities need to address a number of issues to improve our understanding of data clustering. Below is a list of problems and research directions that are worth focusing in this regard.

- (a) There needs to be a suite of benchmark data (with ground truth) available for the research community to test and evaluate clustering methods. The benchmark should include data sets from various domains (documents, images, time series, customer transactions, biological sequences, social networks, etc.). Benchmark should also include both static and dynamic data (the latter would be useful in analyzing clusters that change over time), quantitative and/or qualitative attributes, linked and non-linked objects, etc. Though the idea of providing a benchmark data is not new (e.g., UCI ML and KDD repository), current benchmarks are limited to small, static data sets.
- (b) We need to achieve a tighter integration between clustering algorithms and the application needs. For example, some applications may require generating only a few cohesive clusters (less cohesive clusters can be ignored), while others

- may require the best partition of the entire data. In most applications, it may not necessarily be the best clustering algorithm that really matters. Rather, it is more crucial to choose the right feature extraction method that identifies the underlying clustering structure of the data.
- (c) Regardless of the principle (or objective), most clustering methods are eventually cast into combinatorial optimization problems that aim to find the partitioning of data that optimizes the objective. As a result, computational issue becomes critical when the application involves large-scale data. For instance, finding the global optimal solution for K-means is NP-hard. Hence, it is important to choose clustering principles that lead to computationally efficient solutions.
- (d) A fundamental issue related to clustering is its stability or consistency. A good clustering principle should result in a data partitioning that is stable with respect to perturbations in the data. We need to develop clustering methods that lead to stable solutions.
- (e) Choose clustering principles according to their satisfiability of the stated axioms. Despite Kleinberg's impossibility theorem, several studies have shown that it can be overcome by relaxing some of the axioms. Thus, maybe one way to evaluate a clustering principle is to determine to what degree it satisfies the axioms.
- (f) Given the inherent difficulty of clustering, it makes more sense to develop semi-supervised clustering techniques in which the labeled data and (user specified) pair-wise constraints can be used to decide both (i) data representation and (ii) appropriate objective function for data clustering.

## Acknowledgements

I would like to acknowledge the National Science Foundation and the Office of Naval research for supporting my research in data clustering, dimensionality reduction, classification, and semi-supervised learning. I am grateful to Rong Jin, Pang-Ning Tan, and Pavan Mallapragada for helping me prepare the King-Sun Fu lecture as well as this manuscript. I have learned much from and enjoyed my fruitful collaborations in data clustering with Eric Backer, Joachim Buhmann, (late) Richard Dubes, Mario Figueiredo, Patrick Flynn, Ana Fred, Martin Law, J.C. Mao, M. Narasimha Murty, Steve Smith, and Alexander Topchy. Joydeep Ghosh, Larry Hall, Jianying Hu, Mario Figueiredo, and Ana Fred provided many useful suggestions to improve the quality of this paper.

## References

- Aggarwal, Charu C., Han, Jiawei, Wang, Jianyong, Yu, Philip S., 2003. A framework for clustering evolving data streams. In: Proc. 29th Internat. Conf. on Very Large Data Bases, pp. 81–92.
- Agrawal, Rakesh, Gehrke, Johannes, Gunopulos, Dimitrios, Raghavan, Prabhakar, 1998. Automatic subspace clustering of high dimensional data for data mining applications. In: Proc. ACM SIGMOD, pp. 94–105.
- Anderberg, M.R., 1973. Cluster Analysis for Applications. Academic Press.
- Andrews, Nicholas O., Fox, Edward A., 2007. Recent developments in document clustering. Technical report TR-07-35. Department of Computer Science, Virginia Tech.
- Arabie, P., Hubert, L., 1994. Cluster analysis in marketing research. In: Advanced Methods in Marketing Research. Blackwell, Oxford, pp. 160–189.
- Backer, Eric, 1978. Cluster Analysis by Optimal Decomposition of Induced Fuzzy Sets. Delft University Press.
- Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X., 2006. Group formation in large social networks: Membership, growth, and evolution. In: Proc. 12th KDD.
- Baldi, P., Hatfield, G., 2002. DNA Microarrays and Gene Expression. Cambridge University Press.
- Ball, G., Hall, D., 1965. ISODATA, a novel method of data analysis and pattern classification. Technical report NTIS AD 699616. Stanford Research Institute, Stanford, CA.
- Banerjee, Arindam, Merugu, Srujan, Dhillon, Inderjit, Ghosh, Joydeep, 2004. Clustering with bregman divergences. *J. Machine Learn. Res.*, 234–245.
- Banerjee, Arindam, Basu, Sugato, Merugu, Srujan, 2007a. Multi-way clustering on relation graphs. In: Proc. 7th SIAM Internat. Conf. on Data Mining.
- Banerjee, S., Ramanathan, K., Gupta, A., 2007b. Clustering short texts using Wikipedia. In: Proc. SIGIR.
- Bar-Hillel, Aaron, Hertz, T., Shental, Noam, Weinshall, Daphna, 2003. Learning distance functions using equivalence relations. In: Proc. 20th Internat. Conf. on Machine Learning, pp. 11–18.
- Basu, Sugato, Banerjee, Arindam, Mooney, Raymond, 2002. Semi-supervised clustering by seeding. In: Proc. 19th Internat. Conf. on Machine Learning.
- Basu, Sugato, Bilenko, Mikhail, Mooney, Raymond J., 2004. A probabilistic framework for semi-supervised clustering. In: Proc. 10th KDD, pp. 59–68.
- Basu, Sugato, Davidson, Ian, Wagstaff, Kiri (Eds.), 2008. Constrained Clustering: Advances in Algorithms, Theory and Applications. Data Mining and Knowledge Discovery, vol. 3. Chapman & Hall/CRC.
- Bekkerman, Ron, El-Yaniv, Ran, McCallum, Andrew, 2005. Multi-way distributional clustering via pairwise interactions. In: Proc. 22nd Internat. Conf. Machine Learning, pp. 41–48.
- Belkin, Mikhail, Niyogi, Partha, 2002. Laplacian eigenmaps and spectral techniques for embedding and clustering. *Advances in Neural Information Processing Systems*, vol. 14, pp. 585–591.
- Ben-David, S., Ackerman, M., 2008. Measures of clustering quality: A working set of axioms for clustering. *Advances in Neural Information Processing Systems*.
- Bezdek, J.C., 1981. Pattern Recognition with Fuzzy Objective Function Algorithms. Plenum Press.
- Bhatia, S., Deogun, J., 1998. Conceptual clustering in information retrieval. *IEEE Trans. Systems Man Cybernet.* 28 (B), 427–436.
- Bishop, Christopher M., 2006. Pattern Recognition and Machine Learning. Springer.
- Blake, Merz C.J., 1998. UCI repository of machine learning databases.
- Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent dirichlet allocation. *J. Machine Learn. Res.* 3, 993–1022.
- Bradley, P.S., Fayyad, U., Reina, C., 1998. Scaling clustering algorithms to large databases. In: Proc. 4th KDD.
- Buhler, J., 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17 (5), 419–428.
- Busse, Ludwig M., Orbanz, Peter, Buhmann, Joachim M., 2007. Cluster analysis of heterogeneous rank data. In: Proc. 24th Internat. Conf. on Machine Learning, pp. 113–120.
- Cao, F., Ester, M., Qian, W., Zhou, A., 2006. Density-based clustering over an evolving data stream with noise. In: Proc. SIAM Conf. Data Mining.
- Chapelle, O., Schölkopf, B., Zien, A. (Eds.), 2006. Semi-Supervised Learning. MIT Press, Cambridge, MA.
- Cheng, Yizong, Church, George M., 2000. Bioclustering of expression data. In: Proc. Eighth Internat. Conf. on Intelligent Systems for Molecular Biology, AAAI Press, pp. 93–103.
- Connell, S.D., Jain, A.K., 2002. Writer adaptation for online handwriting recognition. *IEEE Trans. Pattern Anal. Machine Intell.* 24 (3), 329–346.
- Critchlow, D., 1985. Metric Methods for Analyzing Partially Ranked Data. Springer.
- Datta, Ritendra, Joshi, Dhiraj, Li, Jia, Wang, James Z., 2008. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys* 40 (2) (Article 5).
- Dempster, A.P., Laird, N.M., Rubin, D.B., 1977. Maximum likelihood from incomplete data via the EM algorithm. *J. Roy. Statist. Soc.* 39, 1–38.
- Dhillon, I., Modha, D., 1999. A data-clustering algorithm on distributed memory multiprocessors. In: Proc. KDD'99 Workshop on High Performance Knowledge Discovery, pp. 245–260.
- Dhillon, Inderjit S., Mallela, Subramanyam, Guyon, Isabelle, Elisseeff, André, 2003. A divisive information-theoretic feature clustering algorithm for text classification. *J. Machine Learn. Res.* 3, 2003.
- Dhillon, Inderjit S., Guan, Yuqiang, Kulis, Brian, 2004. Kernel  $k$ -means: Spectral clustering and normalized cuts. In: Proc. 10th KDD, pp. 551–556.
- Ding, Chris, He, Xiaofeng, Simon, Horst D., 2005. On the equivalence of nonnegative matrix factorization and spectral clustering. In: Proc. SIAM Internat. Conf. on Data Mining, pp. 606–610.
- Drineas, P., Frieze, A., Kannan, R., Vempala, S., Vinay, V., 1999. Clustering large graphs via the singular value decomposition. *Machine Learn.* 56 (1–3), 9–33.
- Dubes, Richard C., Jain, Anil K., 1976. Clustering techniques: User's dilemma. *Pattern Recognition*, 247–260.
- Duda, R., Hart, P., Stork, D., 2001. Pattern Classification, second ed. John Wiley and Sons, New York.
- Dunn, J.C., 1973. A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters. *J. Cybernet.* 3, 32–57.
- Eschrich, S., Ke, Jingwei, Hall, L.O., Goldgof, D.B., 2003. Fast accurate fuzzy clustering through data reduction. *IEEE Trans. Fuzzy Systems* 11 (2), 262–270.
- Ester, Martin, Peter Kriegel, Hans, S., Jörg, Xu, Xiaowei, 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. 2nd KDD, AAAI Press.
- Ferguson, Thomas S., 1973. A Bayesian analysis of some nonparametric problems. *Ann. Statist.* 1, 209–230.
- Figueiredo, Mario, Jain, Anil K., 2002. Unsupervised learning of finite mixture models. *IEEE Trans. Pattern Anal. Machine Intell.* 24 (3), 381–396.
- Figueiredo, M.A.T., Chang, D.S., Murino, V., 2006. Clustering under prior knowledge with application to image segmentation. *Adv. Neural Inform. Process. Systems* 19, 401–408.

- Fisher, Douglas H., 1987. Knowledge acquisition via incremental conceptual clustering. *Machine Learn.*, 139–172.
- Fisher, L., vanNess, J., 1971. Admissible clustering procedures. *Biometrika*.
- Forgy, E.W., 1965. Cluster analysis of multivariate data: Efficiency vs. interpretability of classifications. *Biometrics* 21, 768–769.
- Frank, Ildiko E., Todeschini, Roberto, 1994. *Data Analysis Handbook*. Elsevier Science Inc., pp. 227–228.
- Fred, A., Jain, A.K., 2002. Data clustering using evidence accumulation. In: Proc. Internat. Conf. Pattern Recognition (ICPR).
- Frigui, H., Krishnapuram, R., 1999. A robust competitive clustering algorithm with applications in computer vision. *IEEE Trans. Pattern Anal. Machine Intell.* 21, 450–465.
- Gantz, John F., 2008. The diverse and exploding digital universe. Available online at: <<http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf>>.
- Google Scholar, 2009 (February). Google Scholar. <<http://scholar.google.com>>.
- Guha, Sudipto, Rastogi, Rajeev, Shim, Kyuseok, 1998. CURE: An efficient clustering algorithm for large databases. In: Proc. ICDM, pp. 73–84.
- Guha, Sudipto, Rastogi, Rajeev, Shim, Kyuseok, 2000. Rock: A robust clustering algorithm for categorical attributes. *Inform. Systems* 25 (5), 345–366.
- Guha, Sudipto, Meyerson, A., Mishra, Nina, Motwani, Rajeev, O'Callaghan, L., 2003a. Clustering data streams: Theory and practice. *Trans. Knowledge Discovery Eng.*
- Guha, Sudipto, Mishra, Nina, Motwani, Rajeev, 2003b. Clustering data streams. *IEEE Trans. Knowledge Data Eng.* 15 (3), 515–528.
- Hagen, L., Kahng, A.B., 1992. New spectral methods for ratio cut partitioning and clustering. *IEEE Trans. Comput.-Aid. Des. Integrated Circuits Systems* 11 (9), 1074–1085.
- Han, Jiawei, Kamber, Micheline, 2000. *Data Mining: Concepts and Techniques*. Morgan Kaufmann.
- Hansen, Mark H., Yu, Bin, 2001. Model selection and the principle of minimum description length. *J. Amer. Statist. Assoc.* 96 (454), 746–774.
- Har-peled, Sariel, Mazumdar, Soham, 2004. Coresets for  $k$ -means and  $k$ -median clustering and their applications. In: Proc. 36th Annu. ACM Sympos. Theory Comput., pp. 291–300.
- Hartigan, J.A., 1972. Direct clustering of a data matrix. *J. Amer. Statist. Assoc.* 67 (337), 123–132.
- Hartigan, J.A., 1975. *Clustering Algorithms*. John Wiley and Sons.
- Hofmann, T., Buhmann, J.M., 1997. Pairwise data clustering by deterministic annealing. *IEEE Trans. Pattern Anal. Machine Intell.* 19 (1), 1–14.
- Hore, Prodip, Hall, Lawrence O., Goldgof, Dmitry B., 2009a. A scalable framework for cluster ensembles. *Pattern Recognition* 42 (5), 676–688.
- Hore, Prodip, Hall, Lawrence O., Goldgof, Dmitry B., Gu, Yuhua, Maudsley, Andrew A., Darkazanli, Ammar, 2009b. A scalable framework for segmenting magnetic resonance images. *J. Signal Process. Systems* 54 (1–3), 183–203.
- Hotho, A., Staab, S., Stumme, G., 2003. Ontologies to improve text document clustering. In: Proc. of the ICDM.
- Hu, J., Ray, B.K., Singh, M., 2007. Statistical methods for automated generation of service engagement staffing plans. *IBM J. Res. Dev.* 51 (3), 281–293.
- Iwayama, M., Tokunaga, T., 1995. Cluster-based text categorization: A comparison of category search strategies. In: Proc. 18th ACM Internat. Conf. on Research and Development in Information Retrieval, pp. 273–281.
- Jain, Anil K., Dubes, Richard C., 1988. *Algorithms for Clustering Data*. Prentice Hall.
- Jain, Anil K., Flynn, P., 1996. Image segmentation using clustering. In: *Advances in Image Understanding*. IEEE Computer Society Press, pp. 65–83.
- Jain, A.K., Topchy, A., Law, M.H.C., Buhmann, J.M., 2004. Landscape of clustering algorithms. In: Proc. Internat. Conf. on Pattern Recognition, vol. 1, pp. 260–263.
- JSTOR, 2009. JSTOR. <<http://www.jstor.org>>.
- Karypis, George, Kumar, Vipin, 1995. A fast and high quality multilevel scheme for partitioning irregular graphs. In: Proc. Internat. Conf. on Parallel Processing, pp. 113–122.
- Kashima, H., Tsuda, K., Inokuchi, A., 2003. Marginalized Kernels between labeled graphs. In: Proc. 20th Internat. Conf. on Machine Learning, pp. 321–328.
- Kashima, H., Hu, J., Ray, B., Singh, M., 2008. K-means clustering of proportional data using L1 distance. In: Proc. Internat. Conf. on Pattern Recognition, pp. 1–4.
- Kaufman, Leonard, Rousseeuw, Peter J., 2005. *Finding groups in data: An introduction to cluster analysis*. Wiley series in Probability and Statistics.
- Kleinberg, Jon, 2002. An impossibility theorem for clustering. In: NIPS 15, pp. 463–470.
- Kollios, G., Gunopulos, D., Koudas, N., Berchtold, S., 2003. Efficient biased sampling for approximate clustering and outlier detection in large data sets. *IEEE Trans. Knowledge Data Eng.* 15 (5), 1170–1187.
- Lange, Tilman, Roth, Volker, Braun, Mikio L., Buhmann, Joachim M., 2004. Stability-based validation of clustering solutions. *Neural Comput.* 16 (6), 1299–1323.
- Lange, T., Law, M.H., Jain, A.K., Buhmann, J., 2005. Learning with constrained and unlabelled data. *IEEE Comput. Soc. Conf. Comput. Vision Pattern Recognition* 1, 730–737.
- Law, Martin, Topchy, Alexander, Jain, A.K., 2005. Model-based clustering with probabilistic constraints. In: Proc. SIAM Conf. on Data Mining, pp. 641–645.
- Lee, Jung-Eun, Jain, Anil K., Jin, Rong, 2008. Scars, marks and tattoos (SMT): Soft biometric for suspect and victim identification. In: Proceedings of the Biometric Symposium.
- Li, W., McCallum, A., 2006. Pachinko allocation: Dag-structured mixture models of topic correlations. In: Proc. 23rd Internat. Conf. on Machine Learning, pp. 577–584.
- Linde, Y., Buzo, A., Gray, R., 1980. An algorithm for vector quantizer design. *IEEE Trans. Comm.* 28, 84–94.
- Liu, J., Wang, W., Yang, J., 2004. A framework for ontology-driven subspace clustering. In: Proc. KDD.
- Liu, Yi, Jin, Rong, Jain, A.K., 2007. Boostcluster: Boosting clustering by pairwise constraints. In: Proc. 13th KDD, pp. 450–459.
- Lloyd, S., 1982. Least squares quantization in PCM. *IEEE Trans. Inform. Theory* 28, 129–137. Originally as an unpublished Bell laboratories Technical Note (1957).
- Lowe, David G., 2004. Distinctive image features from scale-invariant keypoints. *Internat. J. Comput. Vision* 60 (2), 91–110.
- Lu, Zhengdong, Leen, Todd K., 2007. Penalized probabilistic clustering. *Neural Comput.* 19 (6), 1528–1567.
- Lukashin, A.V., Lukashev, M.E., Fuchs, R., 2003. Topology of gene expression networks as revealed by data mining and modeling. *Bioinformatics* 19 (15), 1909–1916.
- MacQueen, J., 1967. Some methods for classification and analysis of multivariate observations. In: Fifth Berkeley Symposium on Mathematics. Statistics and Probability. University of California Press, pp. 281–297.
- Mallows, C.L., 1957. Non-null ranking models. *Biometrika* 44, 114–130.
- Mao, J., Jain, A.K., 1996. A self-organizing network for hyper-ellipsoidal clustering (HEC). *IEEE Trans. Neural Networks* 7 (January), 16–29.
- McLachlan, G.I., Basford, K.E., 1987. *Mixture Models: Inference and Applications to Clustering*. Marcel Dekker.
- Meila, Marina, 2003. Comparing clusterings by the variation of information. In: COLT, pp. 173–187.
- Meila, Marina, 2006. The uniqueness of a good optimum for  $k$ -means. In: Proc. 23rd Internat. Conf. Machine Learning, pp. 625–632.
- Meila, Marina, Shi, Jianbo, 2001. A random walks view of spectral segmentation. In: Proc. AISTATAS.
- Merriam-Webster Online Dictionary, 2008. Cluster analysis. <<http://www.merriam-webster-online.com>>.
- Mirkin, Boris, 1996. *Mathematical Classification and Clustering*. Kluwer Academic Publishers.
- Moore, Andrew W., 1998. Very fast EM-based mixture model clustering using multiresolution kd-trees. In: NIPS, pp. 543–549.
- Motzkin, T.S., Straus, E.G., 1965. Maxima for graphs and a new proof of a theorem of Turan. *Canadian J. Math.* 17, 533–540.
- Muja, M., Lowe, D.G., 2009. Fast approximate nearest neighbors with automatic algorithm configuration. In: Proc. Internat. Conf. on Computer Vision Theory and Applications (VISAPP'09).
- Newman, M.E.J., 2006. Modularity and community structure in networks. In: Proc. National Academy of Sciences, USA.
- Newman, M., Girvan, M., 2004. Finding and evaluating community structure in networks. *Phys. Rev. E* 69 (026113), 3.
- Ng, Andrew Y., Jordan, Michael I., Weiss, Yair, 2001. On spectral clustering: Analysis and an algorithm. *Adv. Neural Inform. Process. Systems*, vol. 14. MIT Press, pp. 849–856.
- Pampalk, Elias, Dixon, Simon, Widmer, Gerhard, 2003. On the evaluation of perceptual similarity measures for music. In: Proc. Sixth Internat. Conf. on Digital Audio Effects (DAFx-03), pp. 7–12.
- Pavan, Massimiliano, Pelillo, Marcello, 2007. Dominant sets and pairwise clustering. *IEEE Trans. Pattern Anal. Machine Intell.* 29 (1), 167–172.
- Pelleg, Dan, Moore, Andrew, 1999. Accelerating exact  $k$ -means algorithms with geometric reasoning. In: Chaudhuri Surajit, Madigan David (Eds.), Proc. Fifth Internat. Conf. on Knowledge Discovery in Databases, AAAI Press, pp. 277–281.
- Pelleg, Dan, Moore, Andrew, 2000. X-means: Extending  $k$ -means with efficient estimation of the number of clusters. In: Proc. Seventeenth Internat. Conf. on Machine Learning, pp. 727–734.
- Philbin, J., Chum, O., Isard, M., Sivic, J., Zisserman, A., 2007. Object retrieval with large vocabularies and fast spatial matching. In: Proc. IEEE Conf. on Computer Vision and Pattern Recognition.
- Rasmussen, Carl, 2000. The infinite gaussian mixture model. *Adv. Neural Inform. Process. Systems* 12, 554–560.
- Roberts Stephen J., Holmes, Christopher, Denison, Dave, 2001. Minimum-entropy data clustering using reversible jump Markov chain Monte Carlo. In: Proc. Internat. Conf. Artificial Neural Networks, pp. 103–110.
- Sahami, Mehran, 1998. Using Machine Learning to Improve Information Access. Ph.D. Thesis, Computer Science Department, Stanford University.
- Sammon Jr., J.W., 1969. A nonlinear mapping for data structure analysis. *IEEE Trans. Comput.* 18, 401–409.
- Scholkopf, Bernhard, Smola, Alexander, Muller, Klaus-Robert, 1998. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Comput.* 10 (5), 1299–1319.
- Shamir, Ohad, Tishby, Naftali, 2008. Cluster stability for finite samples. *Adv. Neural Inform. Process. Systems* 20, 1297–1304.
- Shi, Jianbo, Malik, Jitendra, 2000. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Machine Intell.* 22, 888–905.
- Sindhwani, V., Hu, J., Mojsilovic, A., 2008. Regularized co-clustering with dual supervision. In: Advances in Neural Information Processing Systems.
- Slonim, Noam, Tishby, Naftali, 2000. Document clustering using word clusters via the information bottleneck method. In: ACM SIGIR 2000, pp. 208–215.
- Smith, Stephen P., Jain, Anil K., 1984. Testing for uniformity in multidimensional data. *IEEE Trans. Pattern Anal. Machine Intell.* 6 (1), 73–81.
- Sokal, Robert R., Sneath, Peter H.A., 1963. *Principles of Numerical Taxonomy*. W.H. Freeman, San Francisco.
- Steinbach, M., Karypis, G., Kumar, V., 2000. A comparison of document clustering techniques. In: KDD Workshop on Text Mining.

- Steinbach, Michael, Tan, Pang-Ning, Kumar, Vipin, Klooster, Steve, Potter, Christopher, 2003. Discovery of climate indices using clustering. In: Proc. Ninth ACM SIGKDD Internat. Conf. on Knowledge Discovery and Data Mining.
- Steinhaus, H., 1956. Sur la division des corp materiels en parties. Bull. Acad. Polon. Sci. IV (C1.III), 801–804.
- Strehl, Alexander, Ghosh, Joydeep, 2003. Cluster ensembles – A knowledge reuse framework for combining multiple partitions. *J. Machine Learn. Res.* 3, 583–617.
- Tabachnick, B.G., Fidell, L.S., 2007. Using Multivariate Statistics, fifth ed. Allyn and Bacon, Boston.
- Tan, Pang-Ning, Steinbach, Michael, Kumar, Vipin, 2005. Introduction to Data Mining, first ed. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.
- Taskar, B., Segal, E., Koller, D., 2001. Probabilistic clustering in relational data. In: Proc. Seventeenth Internat. Joint Conf. on Artificial Intelligence (IJCAI), pp. 870–887.
- Tibshirani, R., Walther, G., Hastie, T., 2001. Estimating the number of clusters in a data set via the gap statistic. *J. Roy. Statist. Soc. B*, 411–423.
- Tishby, Naftali, Pereira, Fernando C., Bialek, William, 1999. The information bottleneck method. In: Proc. 37th Allerton Conf. on Communication, Control and Computing, pp. 368–377.
- Tsuda, Koji, Kudo, Taku, 2006. Clustering graphs by weighted substructure mining. In: Proc. 23rd Internat. Conf. on Machine Learning, pp. 953–960.
- Tukey, John Wilder, 1977. Exploratory Data Analysis. Addison-Wesley.
- Umeyama, S., 1988. An eigen decomposition approach to weighted graph matching problems. *IEEE Trans. Pattern Anal. Machine Intell.* 10 (5), 695–703.
- von Luxburg, U., David, Ben S., 2005. Towards a statistical theory of clustering. In: Pascal Workshop on Statistics and Optimization of Clustering.
- Wallace, C.S., Boulton, D.M., 1968. An information measure for classification. *Comput. J.* 11, 185–195.
- Wallace, C.S., Freeman, P.R., 1987. Estimation and inference by compact coding (with discussions). *JRSSB* 49, 240–251.
- Wasserman, S., Faust, K., 1994. Social Network Analysis: Methods and Applications. Cambridge University Press.
- Welling, M., Rosen-Zvi, M., Hinton, G., 2005. Exponential family harmoniums with an application to information retrieval. *Adv. Neural Inform. Process. Systems* 17, 1481–1488.
- White, Scott, Smyth, Padhraic, 2005. A spectral clustering approach to finding communities in graph. In: Proc. SIAM Data Mining.
- Yu, Stella X., Shi, Jianbo, 2003. Multiclass spectral clustering. In: Proc. Internat. Conf. on Computer Vision, pp. 313–319.
- Zhang, Tian, Ramakrishnan, Raghu, Livny, Miron, 1996. BIRCH: An efficient data clustering method for very large databases. In: Proc. 1996 ACM SIGMOD Internat. Conf. on Management of data, vol. 25, pp. 103–114.

# Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds

Ripal Nathuji and Aman Kansal

Microsoft Research  
Redmond, WA 98052

{ripaln, kansal}@microsoft.com

Alireza Ghaffarkhah

University of New Mexico  
Albuquerque, NM 87131  
alinem@ece.unm.edu

## Abstract

Cloud computing offers users the ability to access large pools of computational and storage resources on demand. Multiple commercial clouds already allow businesses to replace, or supplement, privately owned IT assets, alleviating them from the burden of managing and maintaining these facilities. However, there are issues that must be addressed before this vision of utility computing can be fully realized. In existing systems, customers are charged based upon the amount of resources used or reserved, but no guarantees are made regarding the application level performance or quality-of-service (QoS) that the given resources will provide. As cloud providers continue to utilize virtualization technologies in their systems, this can become problematic. In particular, the consolidation of multiple customer applications onto multicore servers introduces performance interference between collocated workloads, significantly impacting application QoS. To address this challenge, we advocate that the cloud should transparently provision additional resources as necessary to achieve the performance that customers would have realized if they were running in isolation. Accordingly, we have developed Q-Clouds, a QoS-aware control framework that tunes resource allocations to mitigate performance interference effects. Q-Clouds uses online feedback to build a multi-input multi-output (MIMO) model that captures performance interference interactions, and uses it to perform closed loop resource management. In addition, we utilize this functionality to allow applications to specify multiple levels of QoS as application Q-states. For such applications, Q-Clouds dynamically provisions underutilized resources to enable elevated QoS levels, thereby improving system efficiency. Experimental evaluations of our solution

using benchmark applications illustrate the benefits: performance interference is mitigated completely when feasible, and system utilization is improved by up to 35% using Q-states.

**Categories and Subject Descriptors** C.0 [General]: System architectures; C.4 [Performance of Systems]: Modeling techniques; D.4.8 [Operating Systems]: Performance—Modeling and prediction; K.6.4 [Management of Computing and Information Systems]: System Management

**General Terms** Design, Management, Performance

**Keywords** Virtualization, Cloud computing, Resource management

## 1. Introduction

Cloud computing is rapidly gaining prominence, as evidenced by the deployment and growth of commercial cloud platforms [2, 9, 18]. The existence of these services enables businesses to replace, or dynamically supplement, their own IT infrastructures with large pools of computational and storage resources that are available on demand. While these consolidated infrastructures provide significant scale and efficiency advantages, there are still issues which prevent their widespread adoption. In particular, an important problem that remains to be effectively addressed is how to manage the quality-of-service (QoS) experienced by customers that share cloud resources.

Today, cloud providers charge customers based upon usage or reservation of datacenter resources (CPU hours, storage capacity, network bandwidth, etc), and service level agreements (SLAs) are typically based on resource availability. For example, a cloud provider may make guarantees in terms of system uptime and I/O request reliability. However, current systems do not make any application layer quality-of-service (QoS) guarantees. In environments where a given resource usage directly translates to application QoS, this is reasonable since customers deterministically receive levels of QoS based upon their purchased resource capacities. In shared cloud infrastructures, though, this is often not the case. Cloud platforms routinely employ virtualiza-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys'10, April 13–16, 2010, Paris, France.  
Copyright © 2010 ACM 978-1-60558-577-2/10/04... \$10.00

tion [3, 32, 35] to encapsulate workloads in virtual machines (VMs) and consolidate them on multicore servers. Virtualization helps enable cohosting of independent workloads by providing fault isolation, thereby preventing failures in one application from propagating to others. However, virtualization does not guarantee performance isolation between VMs [16]. The resultant performance interference between consolidated workloads obfuscates the relationship between resource allocations and application QoS.

For customers, performance interference implies that paying for a quantity of resources does not equate to a desired level of QoS. For example, an application using one core of a multicore processor may experience significantly reduced performance when another application simultaneously runs on an adjacent core, due to an increased miss rate in the last level cache (LLC) [8, 13, 39]. One approach to deal with this indeterminism is to improve virtualization technologies, through better resource partitioning in both hardware and software, and remove performance interference altogether. A benefit of this approach is that the cloud can continue to be agnostic of application QoS and maintain simple resource capacity based provisioning and billing. However, perfect resource partitioning can be difficult and costly to implement, and even if accomplished, may result in inefficient use of resources [36].

To overcome the challenges imposed by performance interference effects, we advocate an alternative approach: QoS-aware clouds that actively compensate for performance interference using closed loop resource management. We present *Q-Clouds*, a QoS-aware control theoretic management framework for multicore cloud servers. Q-Cloud servers manage interference among consolidated VMs by dynamically adapting resource allocations to applications based upon workload SLAs. The SLAs are defined in terms of application specific performance metrics, and online adaptation is achieved through simple semantic-less [14] feedback signals. *Q-Clouds ensures that the performance experienced by applications is the same as they would have achieved if there was no performance interference.*

The Q-Clouds system makes multiple contributions. First, Q-Clouds employs application feedback to build multi-input multi-output (MIMO) models that capture interference relationships between applications. An advantage of the approach is that the system does not need to determine the underlying sources of interference. The MIMO model is used in a closed loop controller to tune resource allocations and achieve specified performance levels for each VM. Our second contribution builds on the Q-Clouds performance driven resource management to increase resource utilizations. In particular, applications may specify multiple QoS levels denoted as Q-states. Here, the lowest Q-state is the minimal performance that an application requires, but higher Q-states may be defined by the customer when they are willing to pay for higher levels of QoS. Q-Clouds uses this infor-

mation to provision underutilized resources to applications when possible, while still guaranteeing that accompanying performance interference effects do not cause collocated applications to experience reduced QoS. We evaluate Q-Clouds on Intel Nehalem based multicore hardware with the Hyper-V virtualization platform. Our experiments with benchmark workloads, which exhibit performance degradations of up to 31% due to interference, show that Q-Clouds is able to completely mitigate the interference effects when sufficient resources are available. Additionally, system utilization is improved by up to 35% using application Q-states.

## 2. The Need for QoS-Aware Clouds

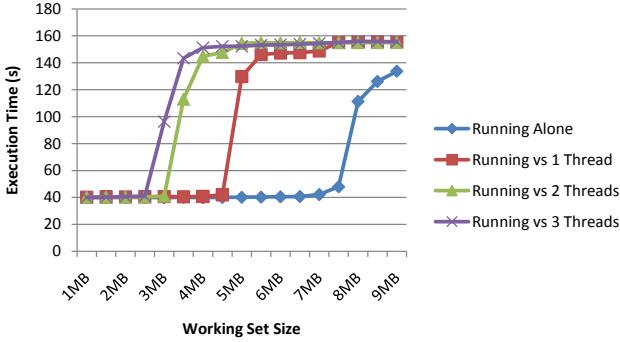
Managing application performance and QoS remains a key challenge for cloud infrastructures. The fundamental problem that arises in these environments is that *application performance can change due to the existence of other virtual machines* on a shared server [16, 31]. Moreover, the presence of other applications cannot be controlled by the customer. As a result, cloud promises on resource availability and capacity do not guarantee application performance. In this section we highlight the performance interference problem and discuss why adopting resource partitioning approaches to mitigate such effects are undesirable for cloud scenarios. We argue that interference should instead be addressed proactively with performance SLAs, as proposed in Q-Clouds.

### 2.1 Performance Interference with Consolidated VMs

If each customer application were allowed to run on dedicated compute, network, and storage resources, there would be no interference. In that situation, ignoring heterogeneous hardware [22], resource level SLAs in effect provide application QoS guarantees. However, the key advantages of cloud platforms come from efficient resource sharing and scaling. The nature of resource sharing is governed by two technology trends: virtualization and multicore processing. Virtualization [3, 32, 35] is employed for fault isolation and improved manageability, while multicore designs allow processors to continue leveraging Moore’s Law for performance benefits. However, since each hosted workload may not be parallelized or scale-up to make use of all cores within a server, it becomes imperative to consolidate multiple workloads for efficient hardware utilization. The workloads, encapsulated in VMs, are then allocated “virtual” processors (VPs) that are backed by individual cores, or fractions of cores.

While virtualization helps to isolate VMs with respect to faults and security [20], it does not provide perfect performance isolation. For example, consolidating VMs onto a multicore package that incorporates a shared last level cache (LLC) creates an opportunity for the VMs to interfere with each other. We experimentally illustrate this problem using data collected on a quad-core Nehalem processor with an 8MB LLC. To localize the effects of cache interference,

we first disable hardware prefetching mechanisms. We then measure the performance, in terms of execution time, of a simple synthetic CPU bound microbenchmark running in a VM. The microbenchmark is written such that the application iterates over a specified working set size multiple times until it has accessed a constant amount of data (2GB) which is significantly larger than the working set size. Therefore, as the working set size increases, the number of total iterations decreases to maintain a constant amount of work. Figure 1 provides the data obtained from this experiment.



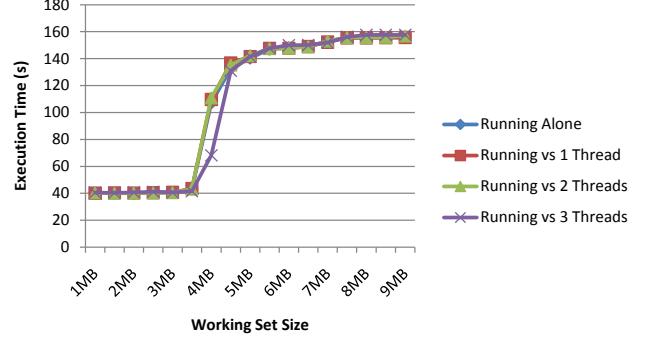
**Figure 1.** Performance impact of cache interference on consolidated VMs.

As illustrated in the figure, when a VM is running by itself, its performance degrades once the working set becomes larger than the LLC since it is effectively memory bound. We compare this performance to the case where the single VM is consolidated with a second VM running synthetic memory bound threads. The number of collocated threads varies from one to three, giving a total of up to four active threads, equal to the number of physical cores. Depending on the working set size, the performance of our first VM with a resource guarantee of one core, is significantly impacted, by up to 380%, due to LLC sharing.

## 2.2 Resource Partitioning to Mitigate Interference

For cloud systems, performance variations for the same resource usage, as seen in Figure 1, are not acceptable. One approach to handle performance interference is to better enforce resource partitioning in the shared system. For instance, the LLC can be effectively partitioned among VMs using page coloring [37] to avoid interference due to LLC sharing. Figure 2 illustrates the effects of page coloring in our previous experiment. Here, each of the two VMs are guaranteed their own 4MB partition of the cache. As expected, the existence of additional threads in the other VM no longer causes performance interference.

Based upon the page coloring results in Figure 2, resource partitioning may seem like a promising direction to solve the performance interference problem. However, adopting this approach is not plausible for two reasons. First, there is a cost in terms of system complexity when implementing partitioning mechanisms such as page coloring. Moreover, there



**Figure 2.** Mitigating LLC interference with VM page coloring.

are several dimensions of performance interference such as shared I/O and memory bandwidths [16], placing additional requirements on both hardware and software for effective partitioning across all of them. Secondly, even if the complexity of partitioning mechanisms is acceptable, it is likely that partitioning would lead to inefficient use of resources. For example, without dynamic page recoloring, in our example we would have allocated all of our LLC to the two VMs. This could prevent us from consolidating additional VMs, causing inefficient processor usage. Even with hardware support for cache partitioning, there would be inefficiencies in cache utilization due to the fact that lines rarely used by one core could not be used by another due to the strict partitioning [36]. Finally, in some cases, technologies like HyperThreading [30] that are designed to increase resource efficiency naturally add interference in the system. Thus, perfect performance isolation with partitioning may neither be practical due to complexity nor efficient in terms of resource utilization. Therefore, we must employ an alternative approach that ensures applications experience the QoS expected from purchased cloud resources.

## 2.3 Managing Clouds with Performance SLAs

When performance interference effects are present, the relationship between the amount of resource guaranteed and the application QoS achieved is broken. As a result, there are various scenarios that can be considered in terms of the metric used to charge the customer and the metric that is guaranteed. Figure 3 lays out four possibilities, based on whether the cloud *charges by* resources or application performance and whether the cloud *guarantees* resource or application performance. We discuss the key aspects of each of these below.

**Scenario 1:** Our first scenario is agnostic of application performance. Guarantees and prices are based solely on resource usage. This approach is simple to implement, and widely adopted in current cloud implementations. However, due to performance interference, this method is not sufficient for customers since the existence of other workloads, outside of their control, can cause significant performance vari-

		Guarantee metric	
		Resource capacity	Application performance
Pricing metric	Resource capacity	Scenario 1	Scenario 3
	Application Performance	Scenario 2	Scenario 4

**Figure 3.** Pricing and guarantee scenarios for clouds.

ations. Indeed, with this scenario the cloud provider benefits if interference prone workloads get placed together as then the application performance is minimized and the customer is forced to pay for more resources to meet performance SLAs.

**Scenario 2:** Following from scenario 1, in the second case the cloud still guarantees resource capacities but the price charged to the customer is based on the actual performance that is experienced. Compared to the first scenario, here if the customer experiences reduced performance because of interference, their payment to the cloud provider is adjusted accordingly. This case is still detrimental to the customer since there is no guarantee of performance. Moreover, in this case the cloud is also impacted negatively since the revenue generated by contracting resources is not deterministic.

**Scenario 3:** In this scenario, the cloud guarantees that the performance SLAs of applications are met. The customer is charged, however, based upon the amount of resources required to meet the SLA. This variation on scenario 1 therefore provides the customer with the benefit of performance guarantees, but still has the drawback that the customer is inevitably responsible for the costs of additional resources that are required to meet the performance SLA when there is interference amongst consolidated VMs.

**Scenario 4:** Our final case combines the benefits of scenarios 2 and 3. The cloud manages applications in a manner that performance guarantees are observed. Moreover, the customer is charged in terms of that level of performance, even if the cloud must tune resource allocations due to performance interference. The interesting issue, then, is how the charge for a level of performance is determined. Recalling that resource usage dictates application performance in the absence of interference, pricing can be based upon the resources required to meet a level of QoS when an application is running in an isolated manner. Then, if interference does occur, the cloud tunes resource allocations so that the desired

performance is still achieved without affecting the charge to the customer. This provides the best experience for the customer, and the cloud is motivated to optimize resource usage and minimize interference, leading to the most efficient configurations. It is clear, then, that this is the best alternative amongst our four scenarios. Realizing this case, however, requires integrating QoS-aware resource management into cloud computing systems.

## 2.4 Enabling QoS-Aware Clouds

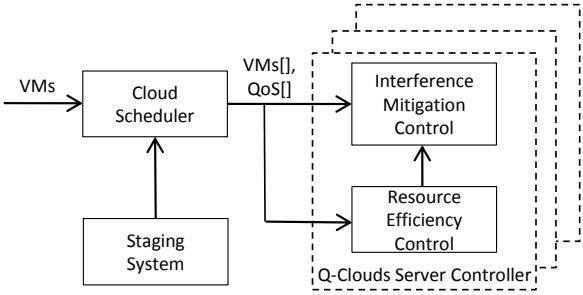
The Q-Clouds system is designed to support QoS-aware management of cloud hosted applications under performance interference effects. In particular, we are concerned with platform-level interference issues. From a cloud management perspective, there are two implications of interference that motivate the functional requirements of Q-Clouds. First, when placing VMs to maximize resource utilization and efficiency, it is necessary to understand the resource usage and requirements of the encapsulated applications. This information allows packing algorithms to deploy VMs in a manner that requires the least number of physical servers to be kept online. Such placement decisions may result in SLA violations if workloads need more resources than expected due to interference with co-hosted VMs. In the absence of a reliable mechanism that can compute the extent of interference for any possible VM placement, a practical approach towards handling this issue is for the VM deployment agent to maintain a resource “head room” when deploying VMs. The head room allows a Q-Clouds enabled server to dynamically tune resource allocations. An interference predictor could be used to estimate headroom allocations [16], or an empirically determined value can be used. *The first requirement of Q-Clouds is then to repurpose unallocated resources from the head-room, when necessary due to interference, to maintain performance SLAs.*

A second implication of performance interference is that, due to large variations in the extent of interference, the assigned head rooms will by nature be conservative. Therefore it is likely that the overall system will go underutilized. From an efficiency perspective, this is undesirable. One way to address this is by allowing applications to use the excess resources from the left over head rooms. In a cloud environment, this is only beneficial if the excess resources do indeed increase the application performance for some applications, the respective customers are willing to pay for the additional performance, and if any resulting interference effects do not cause other applications to drop below their SLAs. We enable such resource allocations through application Q-states, where customers can specify additional discrete levels of QoS that would be desirable, and for each state, the additional cost they would be willing to incur. *The second requirement of Q-Clouds is to utilize application Q-states to increase system utilizations when there is remaining head room.*

### 3. Q-Clouds Management Architecture

#### 3.1 System Overview

Figure 4 presents an architectural overview of the management components used in Q-clouds, which we discuss next. Methods for the consolidation policies used by the cloud scheduler are assumed available from prior work [4, 11, 15], and are summarized only briefly. The focus of this paper is on the platform level mechanisms required in the subsequent online control. However, we discuss all of these elements to provide a holistic view of Q-Clouds enabled datacenters.



**Figure 4.** VM Deployment and QoS management with Q-Clouds.

**Cloud Scheduler:** The *cloud scheduler* determines the placement of VMs submitted by customers on cloud servers. The cloud scheduler makes this decision based upon the resource requirements of workloads, as well as any other constraints that must be satisfied for security or reliability [29]. In order to determine resource requirements, VMs are first profiled on a *staging server* to determine the amount of resources needed to attain a desired level of QoS in an interference-free environment. The resource capacity determined here dictates what the VM owner will pay, regardless of whether additional resources must be provisioned at runtime due to performance interference. Once the resource requirement information is available, previously proposed solutions to the consolidation problem, such as those based on bin-packing [4, 11, 15] and those used in commercial clouds, may be employed. We assume any such solution is used, but with one variation: during VM placement, the cloud scheduler leaves a prescribed amount of unused resources on each server for use in subsequent online control. We refer to this unused capacity as “head-room” which, for example, may be determined empirically based on typical excesses used in interference mitigation. Upon completion, the cloud scheduler sends each Q-Clouds server a set of VMs along with their associated interference-free QoS and resource requirements.

**Interference Mitigation Control:** The head-room on each server is used to allocate additional resources to impacted VMs to bring their performance to the same level as what would have been observed without interference. In particular, we incorporate a closed loop *interference mitigation control* component that helps realize Scenario 4 from Section 2.3 by tuning resource allocations to achieve the de-

sired performance for each VM. The controller operates on each server individually, where it employs a MIMO model to relate resource usage of all collocated VMs with their performance levels. We discuss the MIMO modeling and control approach in more detail in Section 3.2.

**Resource Efficiency Control:** Since the resource capacity used for interference mitigation is not known beforehand and the interference among VMs can vary, it is likely that the head-room allocated by the cloud scheduler ends up being conservative. The slack in resource can then be dynamically allocated to applications to achieve higher levels of QoS. In particular, we consider provisioning additional resources to applications that have multiple QoS levels defined via Q-states. We discuss Q-states in more detail in Section 3.3, but the goal of the *resource efficiency control* component is to determine when higher Q-states can be achieved, and to redefine target QoS points for interference mitigation control based upon the results of its optimization.

#### 3.2 MIMO Modeling and Control

The control methods utilized by Q-Clouds rely upon the availability of a model that describes the relationship between resource allocations and the QoS experienced by VMs. A key capability of the system is that it learns this model online. A requirement to do so, however, is access to both the system inputs and the outputs. The inputs are readily available as control actuators to the underlying system. Access to the outputs, VM performance, requires active feedback from applications. To meet this need, the Q-Clouds platform relays QoS information from each VM on the server to the platform controller. In particular, we expose a paravirtualized interface that allows performance information to be communicated between virtual machines and an agent in the management partition (e.g. Root for the Hyper-V platform, or Dom0 for Xen). A question, however, is what type of QoS data is required from each application. An artifact of cloud environments is that each application may have its own metric of performance. For example, compute applications may measure performance in millions of instructions per second (MIPS), while server workloads may use response time or request throughput. Our design does not use any semantic information regarding these performance metrics as the goal of Q-Clouds is simply to modify resource allocations so that the required level of QoS is met. This maps to a *tracking* control problem and it is sufficient for each application to provide its QoS information as a raw data value. This feedback is then used for two purposes: to build a system model and to drive online control of the platform.

In order to apply efficient resource allocation control, it is desirable for the system model to incorporate performance interference relationships between VMs that are consolidated onto a server. To meet this need, we adopt a multi-input, multi-output (MIMO) model approach which naturally captures performance interference interactions. Specif-

ically, we consider a discrete-time MIMO model of the platform with  $p$  inputs and  $q$  outputs in order to design a model predictive control framework for Q-Cloud servers. The inputs  $u_1[k], u_2[k], \dots, u_p[k]$  of the model are defined to be the actuators used by the platform controller to manage resource allocations at time step  $k$ . For example, the system may utilize virtual processor (VP) capping mechanisms [23] on each VM to throttle the processing resources an application can use. The outputs  $y_1[k], y_2[k], \dots, y_q[k]$  of the model are the predicted QoS values at time step  $k$ . Let us denote by  $\mathbf{u}[k] = [u_1[k] \ u_2[k] \ \dots \ u_p[k]]^T$  and  $\mathbf{y}[k] = [y_1[k] \ y_2[k] \ \dots \ y_q[k]]^T$  the stacked vectors of the model inputs and outputs respectively. The general MIMO model of the platform is then given by a set of nonlinear difference equations as shown in Equation 1, where  $\Phi()$  determines the outputs at time step  $k$  based upon previous outputs as well as current and previous inputs. The impact of previous outputs on the values at the current time step is determined by the parameter  $n$  and referred to as the *order of the system*. Similarly, the value  $m$  determines to what extent previous values of the input continue to impact the output at time step  $k$ . When  $n$  or  $m$  are nonzero, the current output depends on the history of prior inputs and outputs.

$$\mathbf{y}[k] = \Phi(\mathbf{y}[k-1], \dots, \mathbf{y}[k-n], \mathbf{u}[k], \dots, \mathbf{u}[k-m]) \quad (1)$$

In general, modeling of nonlinear dynamical system as depicted by Equation 1 is quite challenging. First, most nonlinear system identification techniques require significant computational complexity resulting in real-time implementation issues. Also, the large amount of learning data required for such modeling often makes the system identification of nonlinear dynamical systems impractical. However, in most applications, there exist simplified models that can capture the input-output interactions with a small amount of uncertainty. For example, static models are typically used for computing platforms experiencing very fast dynamics. In this case the platform model can be specified as  $\mathbf{y}[k] = \Phi(\mathbf{u}[k])$  with  $n = m = 0$ . We observed that in most of our experiments, a static approximation of the system model given by  $\mathbf{y}[k] = \mathbf{A}\mathbf{u}[k] + \mathbf{b}$  is precise enough when the range of inputs under control is small, as is the case for interference mitigation control. Adopting this approximation enables the use of fast learning algorithms such as Least Mean Squares (LMS) or Recursive Least Squares (RLS) for finding the model parameters  $\mathbf{A}$  and  $\mathbf{b}$ . Thus, for interference mitigation control, the model can be learned at run time as VMs get placed, allowing the controller to rapidly tune resource allocations as interference effects are observed.

It is important to note that the most accurate model in general can depend upon the workloads that are being considered, and the level of accuracy required may depend upon the type of control being applied. For example, if the control being applied requires capturing system dynamics and non-

linearities, neural networks might be applicable to better estimate the relationship in Equation 1 [24]. For our purposes, we evaluate and compare multiple cases in Section 5.1 to study these tradeoffs in our experiments. Once an appropriate MIMO model has been found, at any step we find optimal control inputs by solving a constrained optimization problem that leverages the predictive capabilities provided by the model. Taking the case where we model  $\mathbf{y}[k] = \mathbf{A}\mathbf{u}[k] + \mathbf{b}$ , Equation 2 provides an overview of the optimization based control, where  $\mathbf{u}^*$  represents the optimal set of inputs based upon  $\Lambda(\mathbf{y})$  and  $\Psi(\mathbf{y})$ . As denoted by  $\text{argmin}_{\mathbf{u}}$  in the equation, we attempt to find among the possible input vectors  $\mathbf{u}$ , the one which minimizes  $\Lambda(\mathbf{y})$ . For example,  $\Lambda(\mathbf{y})$  can quantify the deviation from the SLA outputs prescribed to the controller. The goal of the optimization is to find the minimum of this function where  $\mathbf{u}$  is in the space of allowable inputs  $\mathcal{U}$ , and constraints on the output represented by  $\Psi(\mathbf{y}) \leq 0$  are met. For example, where we want to meet some SLA outputs  $\mathbf{y}_{SLA}$ ,  $\Psi(\mathbf{y}) = \mathbf{y}_{SLA} - \mathbf{y}$ . In this manner,  $\Lambda(\mathbf{y})$  and  $\Psi(\mathbf{y})$  are chosen based upon the desired QoS metrics for each of the consolidated VMs.

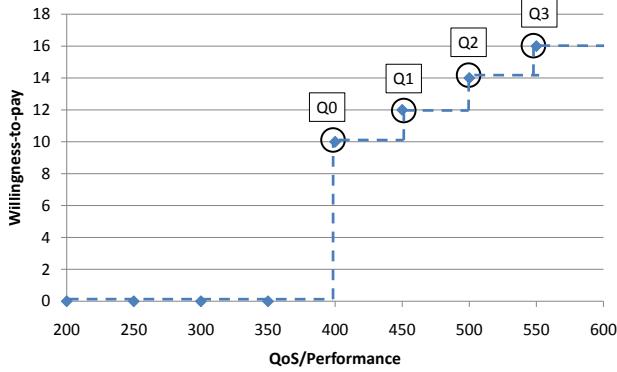
$$\begin{aligned} \mathbf{u}^* &= \text{argmin}_{\mathbf{u}} \Lambda(\mathbf{y}) = \text{argmin}_{\mathbf{u}} \Lambda(\mathbf{A}\mathbf{u} + \mathbf{b}) \\ \text{s.t.} \\ \mathbf{u} &\in \mathcal{U}, \\ \Psi(\mathbf{y}) &= \Psi(\mathbf{A}\mathbf{u} + \mathbf{b}) \leq 0 \end{aligned} \quad (2)$$

As summarized above, our control approach is based upon a MIMO model that is learned and adapted online, and then used as a prediction tool for optimization based control. A benefit of this approach is that the model can also be used to determine whether provisioning additional resources to an application results in an improvement to QoS, and subsequently if such over provisioning may negatively impact other workloads. This ability can be used to determine when idle resources can be used to allow VMs to execute at higher QoS levels as described next.

### 3.3 Improving Cloud Efficiency with Q-states

For management purposes, we assume a minimum performance SLA is specified by customers for a VM deployed in the cloud. For example, a customer may request a level of QoS that requires half a processing core for a VM when it runs in isolation. Q-Clouds must then ensure that the application achieves that QoS during consolidated execution, even if additional resources must be supplied because of performance interference. Let us denote this level of QoS as  $Q_0$ . For some customers, it may be that provisioning resources beyond those required to achieve  $Q_0$  may have additional value that they are willing to pay for. For example, certain computations may be able to vary their level of accuracy, where a base level of accuracy is denoted as  $Q_0$ , but the customer may be willing to pay for marginal improvements beyond this [14]. Q-clouds allows customers to define

additional Q-states that convey this desire so that servers can dynamically allot supplemental resources to applications, increasing the overall utilization and revenue of the cloud.



**Figure 5.** Example of application QoS and definition of Q-states and willingness-to-pay values.

The relationship between QoS and user benefit is often captured by continuous utility functions. Though continuous functions are convenient from a mathematical perspective, they are practically difficult for a user to define. Indeed, in the case where additional utility implies the willingness to pay extra for additional resources, it is likely that the mapping of utility to QoS may not even be continuous. Therefore, we support discrete Q-states, where we expect that it is much easier for customers to define a few levels of QoS,  $Q_x$ , in addition to  $Q_0$  along with the amount of money that they would be willing to pay if the level of QoS is achieved. Figure 5 provides an example of this idea.

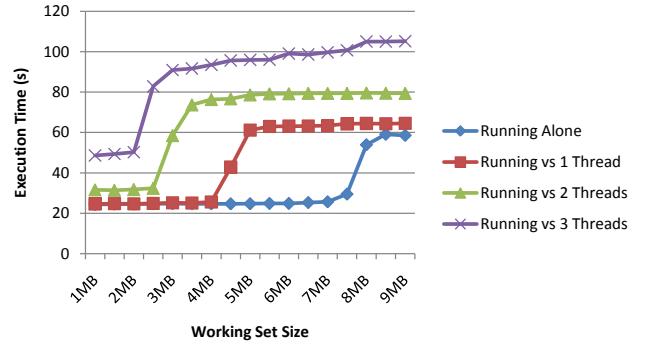
Given a set of Q-states for each VM, the platform controller can dynamically use up surplus head room on a server. In particular, the resource efficiency control component can perform an optimization to determine which, if any, VMs may be able to run at an elevated Q-state within the remaining resource capacity. At the same time, the controller must ensure that every VM still achieves its minimal QoS level of  $Q_0$ . To do this the controller again makes use of our MIMO model. In this manner, Q-states allows future capabilities for customers to autonomously bid on and purchase additional resources only when it is beneficial to them (i.e. they get a performance benefit that pushes them to the next Q-state), while not disturbing other co-hosted applications. While in this paper we consider the use of Q-states to improve resource utilization, in the future we hope to investigate how they can help realize market based allocation of stranded datacenter resources [5].

#### 4. Experimental Methodology

In this section we provide a brief overview of the experimental setup and implementation used to evaluate Q-Clouds enabled servers. We begin by defining the performance interference effects and workloads that we have chosen to consider. We then briefly describe our system implementation.

#### 4.1 Performance Interference Effects and Workloads

As highlighted in previous work, performance interference can occur for various reasons including interference in CPU caches, memory bandwidth, and I/O paths [13, 16, 19, 31]. The Q-Clouds management framework is designed to handle any of these effects, with the availability of appropriate control actuators. Our current system includes the ability to cap the VP utilizations of guest VMs, thereby limiting accessibility to CPU resources [23]. Therefore, we limit our evaluation to interference effects that can be directly affected by varying CPU resources. In this context, there are three points of possible interference. First, multiple memory intensive applications can affect each other's performance due to interference in their respective memory bandwidth availability [19]. Second, as shown in Figure 1, applications can contend for resources in the last level cache (LLC). A final interference point is created by the use of hardware prefetching in processors. Here, the prefetchers employ a policy to determine when to prefetch cache lines, and scale back dynamically based upon bus utilization, etc. Figure 6 illustrates the inclusion of this effect when we enable prefetching in the experiments for Figure 1. Here, we see that even when the measured thread has a working set of 1MB and is not affected by sharing the LLC, its execution time doubles when it is running against three other threads due to reduced prefetching.



**Figure 6.** Performance impact of cache and prefetch hardware interference on consolidated VMs.

Based upon these multiple points of interference, and the availability of the VP capping actuator, in this paper, we focus on CPU bound workloads. In the future we plan to include additional control actuators so that we can better address I/O bound applications as well. For our evaluation, we use a set of benchmarks from the SPEC CPU2006 suite. Prior work has grouped these workloads according to their sensitivity to characteristics of the memory hierarchy (cache size, etc) [17]. As in Figure 6, our experiments utilize four applications running on the same quad-core processor. Therefore, we choose five of the SPEC CPU2006 benchmarks that are identified as being sensitive, and use all possible combinations of four as experimental workload mixes, shown in Table 1.

**Table 1.** SPEC CPU2006 workload mixes.

Benchmark	Workload Mix				
	1	2	3	4	5
436.cactusADM	X	X	X	X	
437.leslie3d	X	X	X		X
459.GemsFDTD	X	X		X	X
470.lbm	X		X	X	X
471.omnetpp		X	X	X	X

## 4.2 System Implementation

We implement and evaluate the Q-Clouds system using a dual socket enterprise server. Each socket is provisioned with a quad core Nehalem processor and 18GB of memory, for a total of eight cores and 36GB of memory on the platform. The Nehalem processor incorporates a three level cache hierarchy, where each core has its own L1 (32KB) and L2 (256KB) caches, and there is a large shared 8MB L3. Since this configuration restricts the interference effects we consider to each package, in our experiments we run four VMs that are consolidated onto one package, and leave the other idle. Finally, we deploy the Hyper-V virtualization software on the server, based upon the Windows 2008 R2 operating system [35].

Hyper-V utilizes a Root partition, similar to Dom0 in Xen, that is able to perform privileged management operations on the platform. We implement a user space Root Agent that runs in the Root partition to drive the Q-Clouds resource management control and QoS monitoring. For resource monitoring, the agent can employ one of two methods. From our prior work, we have a simple paravirtualized interface that VM agents running in guests can use to convey QoS information over VMBus [23]. In addition, we have enabled the hypervisor to utilize performance counters in the hardware to monitor metrics on a per virtual processor (VP) basis. For example, we can monitor the number of instructions retired, cache accesses, cache misses, etc. The Root Agent can access this information from the hypervisor. In the context of this paper, since we focus on CPU bound applications whose performance QoS is based upon instructions executed, the Root Agent makes use of the hardware performance counter data to monitor the performance of each guest VM in terms of millions of instructions executed per second (MIPS).

As explained above, the Root Agent is able to monitor VM performance through the hypervisor. In addition, it has the ability to dynamically adjust CPU resource allocations provided to guest VMs by setting the associated VP utilization cap through the hypervisor. Based upon these inputs (VP caps) and outputs (guest QoS feedback), we implement our system controller in Matlab. In our lab setup, the Matlab component of the controller runs on a separate machine, and communicates with the RootAgent over the network. This setup is strictly due to ease of implementation, and there is

no restriction that the controller be developed in this manner for our system. In our case, the Matlab component of the controller periodically queries the Root Agent for updated QoS feedback, and then invokes new VP caps through the Root Agent when necessary. As described in Section 3.2, the controller implements two pieces of functionality: 1) It uses feedback data to develop MIMO models of the consolidated VMs. 2) It uses the model to solve a constrained optimization problem in order to meet the SLAs of the applications.

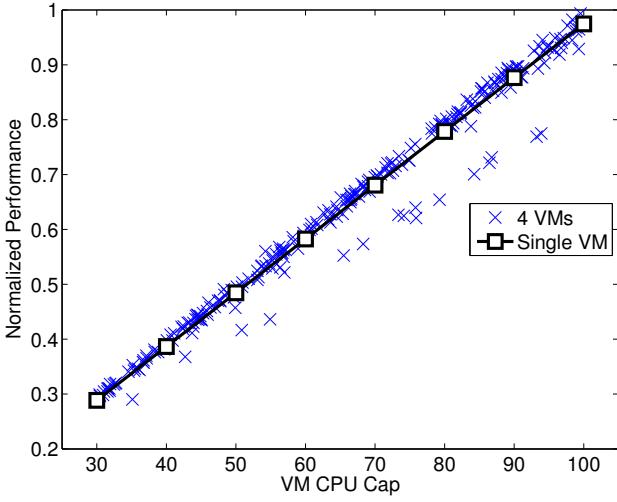
## 5. Evaluating Q-Clouds

### 5.1 Modeling VM Performance Interference

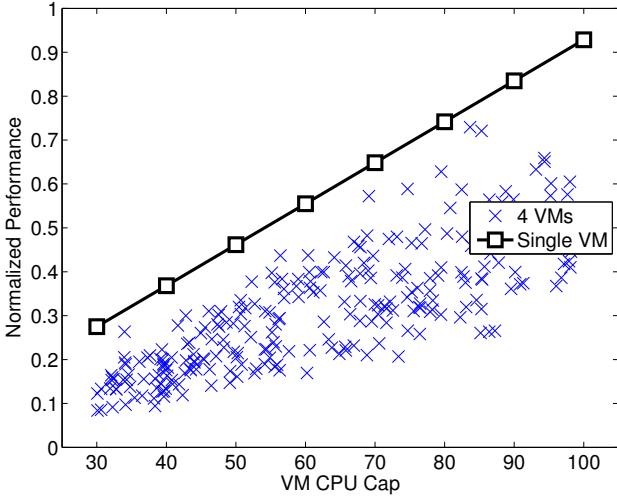
The feedback control implemented by Q-Cloud servers is predicated upon an available MIMO model that allows the controller to account for performance interference effects between consolidated applications. Figure 7 motivates the need for such a model. The figure illustrates the performance of our synthetic application from Figure 1 as a function of the CPU allocation (VP cap), comparing execution data when the application is running alone versus when it is consolidated with three other VMs running the same application for a total of four threads on a quad-core processor. Figure 7(a) illustrates the resulting performance when the working set size of the application is 1KB, and Figure 7(b) demonstrates how things change when the working set is made larger and interference begins to occur.

In both cases of Figure 7, we observe a linear relationship between performance and VP cap when there is a single VM running the application. We can compare this result against when we consolidate the remaining VMs and randomly vary the VP cap to all four. Both figures include the ensuing data points based upon the monitored performance of the first VM. As expected, when the working set is small, there is effectively no deviation from the original trend, and hence any performance model that captures the relationship between resource allocation and performance for the application by itself continues to hold under consolidation. However, in the case of the larger working set, we observe that the consolidated performance is strictly less than that achieved when the VM is running in isolation, and moreover, there is a wide variance. The latter can be attributed to the varying CPU allocations for the other three VMs, resulting in different amounts of interference. In general, the figure supports the fact that the system model used for control should be a function of all the workloads that a VM is consolidated with since performance interference interactions can vary.

We construct our system MIMO model using application performance feedback. In general, the interactions between applications can create nonlinear relationships. However, often you can model the nonlinear system as a linear system and it is accurate enough within a region of operation to perform control. The drawback, though, is that if you want to perform some optimized control over a larger region of the operating space (larger range of the controlled inputs), a lin-



(a) Application with 1KB Working Set Size

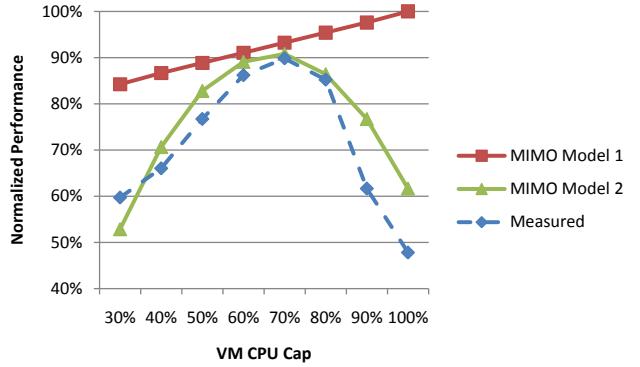


(b) Application with 3MB Working Set Size

**Figure 7.** Comparing the deviation from isolated performance as VMs are consolidated onto a multicore package.

ear model may not allow you to find a good solution. Figure 8 illustrates this tradeoff with synthetic workloads that stress interference effects. The figure compares two modeling approaches. The first, MIMO model 1, uses the linear model described in Section 3.2 to capture the relationship between performance of VMs as a function of all VP caps. MIMO model 2 uses a more complex approach where the deviation from the linear trend between VP cap and performance when there is no performance interference is modeled with a second order polynomial. Here, we find the least squares solution for the coefficients of the polynomial to solve for the model.

In Figure 8, we plot the normalized performance of the system, where we have four VMs each running our synthetic workload with a 3MB working set. In each of the data points, the same VP cap is applied to all VMs. We can see from the



**Figure 8.** Comparison of MIMO modeling approaches.

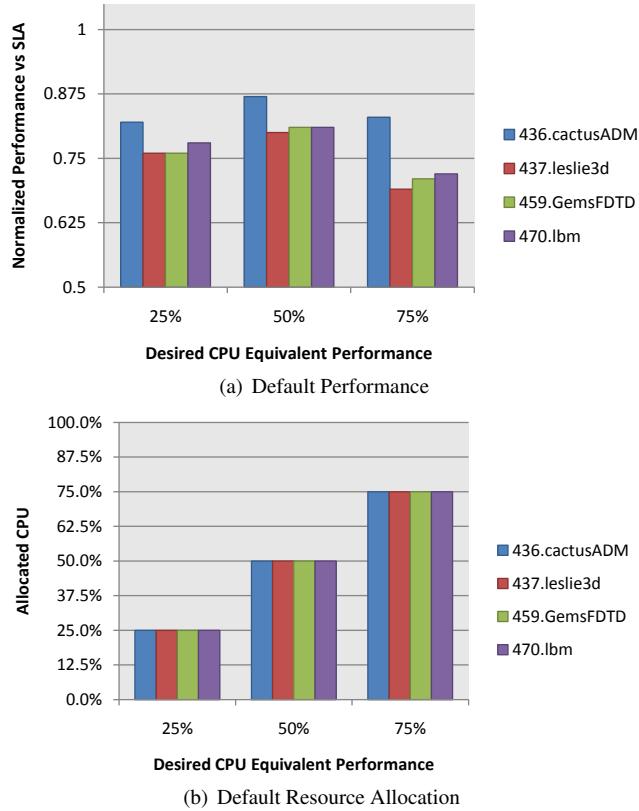
measured data (dashed curve in the figure) that the operating point that maximizes performance is around 65%. If we wanted to predict this operating point with the MIMO models, the linear model would pick a poor solution since it predicts maximum performance at the operating point of 100%. Our second model, on the other hand, is able to capture the trends over the larger operating space much more accurately. The drawback of MIMO model 2, however, is that it is more complex, especially as we scale the number of system inputs. It also requires a model learning phase, possibly performed during system staging, where the VP caps of the VMs can be varied randomly across the entire operating region to accurately capture the interference relationships across a wider range of controlled inputs. On the other hand, MIMO model 1 can be adapted dynamically using a small history of observed data obtained during online control.

We observe that MIMO model 2 is of most benefit when we need to predict across the larger operating space. Indeed, our evaluations of the models show that in general, they both have similar error on average (approximately 13%) when you only need to predict within a small region. Therefore, we employ the models selectively depending on our needs. MIMO model 1 is used for interference mitigation control to achieve a specified set of SLAs (QoS points). This model is constructed completely online, and does not require learning during staging. We show in Section 5.2 that this light-weight approach allows Q-Cloud servers to control the system to appropriately meet SLAs. MIMO model 2 is then employed for resource efficiency control where we'd like to allocate remaining system resources to VMs to achieve higher Q-states. The resource efficiency control component can use the second MIMO model to quickly determine what set of Q-states are plausible given performance interference effects.

## 5.2 Meeting QoS Requirements with Q-Clouds

In evaluating the ability of Q-Cloud servers to meet QoS requirements under performance interference effects, we must assume the amount of head room that is available during consolidated execution. As previously mentioned, in this paper our focus is on the platform component of the Q-Clouds

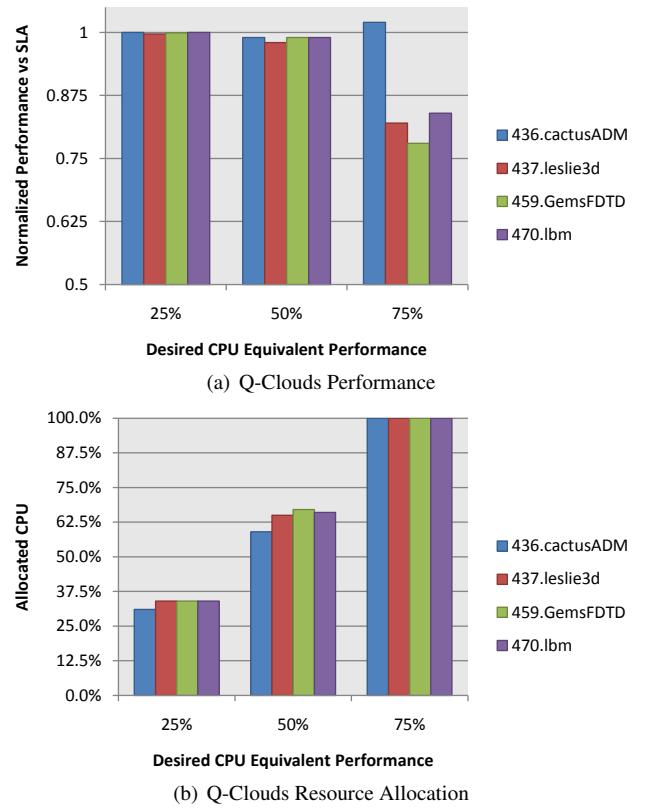
systems. Therefore, we assume different amounts of head room that may occur when VMs are deployed by the cloud scheduler. In particular, for each of the workload mixes in Table 1, we assume that all four VMs have SLAs that require 25%, 50%, or 75% of the CPU when there is no performance interference, resulting in head rooms of 75%, 50%, and 25% during consolidation. This gives us fifteen scenarios comprised of five workload mixes and three possible SLAs per mix. In our first set of experiments we deploy the four workloads in a mix, and define the desired SLAs based upon the performance experienced with the respective resource capacity when there is no interference. We compare against a default case where the system assumes that resources required to meet the SLA do not change as workloads are consolidated (i.e. QoS-unaware resource allocations).



**Figure 9.** Performance and resource allocations with a default QoS-unaware system for workload mix one.

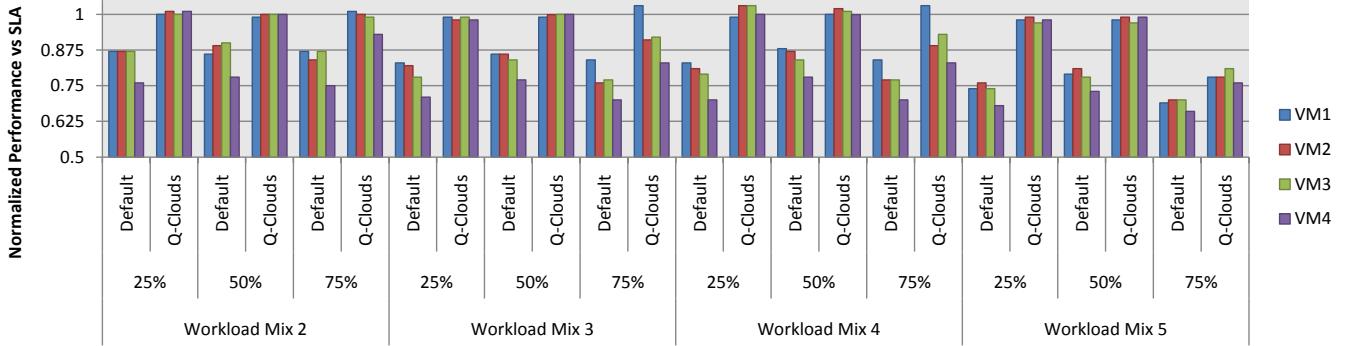
Figure 9 provides results from our default case with the first workload mix. We observe that with QoS-unaware management, applications are consistently impacted by up to 30%. While some applications are affected more heavily than others, there is significant degradation across all of them. This clearly demonstrates the need for the type of online management provided by Q-Clouds. Taking this experiment and enabling interference mitigation control, we obtain the application performance and resource allocation data

provided in Figure 10. Comparing Figures 9(a) and 10(a) we observe clearly that the Q-Clouds controller is able to use the MIMO model of the system to adapt resource allocations so that the desired performance is met, with the exception of the case where the desired performance is equivalent to 75% of the CPU. The reason for this can be observed in Figures 9(b) and 10(b). We see that when applications desire CPU equivalent performances of 25% and 50%, the Q-Clouds server allocates additional resources to the VMs in order to meet the desired QoS. However, in the case of 75%, the system allocates additional resources to the point that it is unable to provision additional capacity. Therefore, the inability of Q-Clouds to meet QoS requirements is due to the fact that there is not enough head room in the system for Q-Clouds to properly provision the VMs.



**Figure 10.** Performance and resource allocations with a Q-Clouds enabled system for workload mix one.

Figure 11 provides the performance data for the remainder of our workload mixes. The results are similar to our first mix, where Q-Clouds is able to utilize available resources from the consolidation step to provision additional resources to VMs so that QoS requirements can be met. Again, in the case where applications desire performance levels equivalent to 75% of a CPU, Q-Clouds is unable to meet the QoS for all workloads. Indeed, in some cases as in workload mix five, there is significant interference and none of the applica-



**Figure 11.** Performance comparisons of default and Q-Clouds for workload mixes two through five.

**Table 2.** Q-Clouds average CPU allocation after interference mitigation control.

Workload Mix	Default allocation of 25% CPU	Default allocation of 50% CPU	Default allocation of 75% CPU
1	33%	64%	100%
2	31%	58%	96%
3	32%	63%	100%
4	33%	63%	100%
5	36%	70%	100%

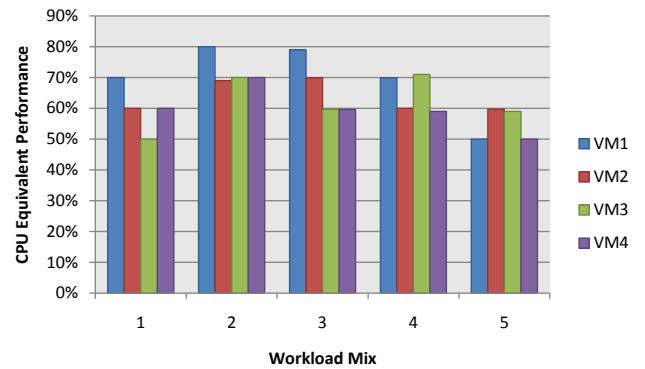
tions can meet their QoS. This is in spite of the fact that in this case, as shown in Table 2, Q-Clouds allocates the system fully to the applications.

Given the results in Figure 11 and Table 2, it is clear that provisioning VMs that require CPU equivalent performance of 75% does not leave ample head room in the system for Q-Cloud controllers to mitigate performance interference effects. On the other hand, allocations of 25% leave significant system resources unused, on average 67%. The intermediate case of 50% leaves between 30% and 42% of resources unused. As we show next, we can dynamically reprovision these resources to achieve higher QoS levels as defined by applications, thereby increasing resource efficiency and revenue for the cloud while providing better performance for customers.

### 5.3 Improving Cloud Utilization with Q-states

Building on the results in Section 5.2, we evaluate the inclusion of Q-states as described in Section 3.3. For our evaluation, we again take our five workload mixes, but assume in each case a head room of 50% where each VM requires performance equivalent to 50% of a CPU when there is no performance interference. By our definition of Q-states, these QoS points are Q0 states. We define three additional Q-states, Q1, Q2, and Q3, where for each VM these are defined to be CPU equivalent performance of 60%, 70%, and 80% respectively. We integrate MIMO model 2 in a global optimization step, where the Q-Cloud resource efficiency controller determines the set of Q-states that can be achieved. The idea is that it can use the MIMO model as a predictor to determine plausible operating points. It then defines the appropri-

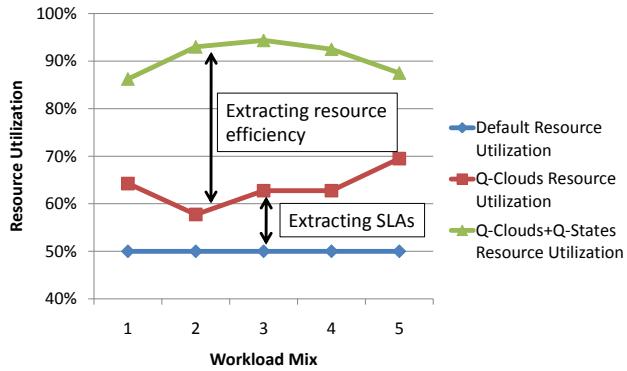
ate target performance points to the interference mitigation controller which from then on uses a dynamically adapted linear MIMO model 1 to steer the system towards the specified operating point.



**Figure 12.** QoS provided to applications with Q-states: Some VMs experience higher Q-states than Q0 (CPU equivalent performance of 50%).

Figure 12 provides the QoS levels provided by the Q-Cloud server. We observe that the system is indeed able to identify appropriate Q-states and achieve them. Depending on the workload mix, the system is able to provide various levels of QoS improvement. For example, in workload mix five, there is significant interference between applications, so two of them are limited to Q0 while the other two receive Q1. On the other hand, in workload mix two there is significant flexibility, and the system is able to provision VM1 with Q3 and the others with Q2. In general, the system is able to leverage the predictive capabilities of the more com-

plex MIMO model 2 to find an operating point that is reachable without overly straining the system. Though we do not consider the ensuing revenue generated by running at higher Q-states, the Q-Clouds controller can allow customers to bid with willingness to pay values and determine the optimal allocation in terms of revenue. Here we validate the benefits of the mechanism, and plan to further investigate its ability to enable dynamic market driven resource allocation for clouds [5].



**Figure 13.** Resource utilization comparisons between default systems, Q-Clouds servers, and Q-Clouds with Q-state enabled resource efficiency optimization.

As a final result, Figure 13 summarizes the system-level benefits of the Q-Clouds system. We observe the first step in resource utilization where the Q-Clouds platform controller increases resource allocations in order to meet the minimal Q0 level of QoS. Then, based upon application defined Q-states, the system tries to provision remaining resources to meet higher QoS levels, while still accounting for performance interference effects so that all other applications are still able to meet their SLA. We observe that the use of interference mitigation control to meet Q0 levels of performance uses up additional resources by up to 20%. In addition, employing resource efficiency control increases system utilization by another 35%. Overall, the average Q-Cloud server utilization is 91%. This illustrates how the combination of Q-states and closed loop control can significantly improve resource utilizations and efficiencies even in the presence of performance interference effects.

## 6. Related Work

Performance interference effects from shared resources can significantly impact the performance of co-located applications. Hardware extensions to enable improved resource management of shared processor resources such as caches have been considered extensively in previous work. For example, hardware techniques have been designed that dynamically partition cache resources based upon utility metrics [27], or integrate novel insertion policies to pseudo-partition caches [36]. Similarly, changes to the memory subsystem have been developed that can help manage in-

terference effects in memory bandwidth [19]. Mechanisms to enable improved monitoring of shared cache structures to detect and quantify interference have been proposed as well [38]. More sophisticated techniques build on these monitoring schemes to enable priority driven management of resources in hardware for improved QoS support [13]. Similar to operating system scheduling techniques for performance isolation [8, 39], we approach the problem of performance interference based upon existing hardware platform capabilities. Moreover, with Q-Clouds, our goal is to address multiple sources of performance interference by dynamically provisioning additional resources when necessary to meet QoS objectives of virtualized applications, without having to fully understand the details of where and how interference is occurring. However, the availability of additional hardware support for monitoring and actuation can help improve both modeling and control of the overall system, and would be invaluable for instantiations of Q-Clouds on future generations of hardware which incorporate these types of features.

A key benefit of virtualization is the flexibility it provides to easily manage applications. The disassociation between virtual and physical resources allows the management layer to transparently adapt resource allocations [21, 25, 26]. In addition, the integration of live migration technologies [6] introduces the ability to adaptively allocate VMs across distributed servers. This can be used to pack VMs in a manner that minimizes unused resources, where the allocator may even forecast future resource needs [4, 15]. More sophisticated schemes may consider the overheads and sequences of migrations necessary to achieve a desired cluster allocation [11]. However, awareness of performance interference effects between VMs is limited in these approaches. For example, prior attempts to intelligently place HPC workloads to avoid cache contention have been evaluated [31]. Approaches to predict performance interference [13, 16] can be used in conjunction with these solutions to prescribe a practical amount of excess resources on platforms. Once workloads have placed onto servers, Q-Clouds performs feedback driven control to provision resources using a MIMO model that captures interference effects from multiple points.

Feedback control methods have been applied for resource management in various environments. For example, controllers targeted for the specific case of web server applications have been developed [1]. Other instances have considered the use of control theoretic methods to concurrently manage application performance and power consumption [23, 34]. In the case of virtualized systems, prior work has evaluated the use of feedback control for resource allocation between tiers of multi-tier applications [25] to meet SLAs, where the approach can be extended to include multiple resources beyond just the CPU using MIMO models [26]. Recognizing the proliferation of controllers across the system, methods to coordinate distributed controllers

have also been investigated [28]. The control techniques used in our Q-Clouds system compliment these methods, while specifically addressing the issue of resource management under performance interference effects for cloud environments.

## 7. Conclusions and Future Work

Cloud computing is a rapidly emerging paradigm with significant promise to transform computing systems. Indeed, multiple commercial solutions are beginning to provide cloud services and resources [2, 9, 18]. Today, customers are charged based upon resource usage or reservation. However, the performance that an application will obtain from a given amount of resource can vary. A significant source of variation is from performance interference effects between virtualized applications that are consolidated onto multicore servers. It is clear that for clouds to gain significant traction, customers must have some guarantees that a quantity of resource will provide a deterministic level of performance. In this paper we present our Q-Clouds system that is designed to provide assurances that the performance experienced by applications is independent of whether it is consolidated with other workloads. Contributions of our system include the development and use of a MIMO model that captures interference effects to drive a closed loop resource management controller. The controller utilizes feedback from applications to meet specified performance levels for each VM. We build on this capability by introducing the notion of Q-states, where applications can specify additional levels of QoS beyond a minimum that they are willing to pay for. Q-Cloud servers can make use of these states to provision idle resources dynamically, thereby improving cloud efficiency and utilizations.

In this paper we have implemented and evaluated Q-Cloud servers based upon interference effects between CPU bound applications. Though this work provides a solid foundation, there are multiple challenging problems that are not addressed here which we hope to pursue in the future. First, we plan to investigate how Q-Clouds can be extended to better address interference in the I/O paths. As part of this, we will need to extend the platform virtualization infrastructure with appropriate control actuators. This will require accounting for evolving hardware mechanisms which improve platform level I/O virtualization [7], as well as how we might leverage exposed control and monitoring mechanisms from existing technologies that help enable performance isolation for storage resources that are shared amongst multiple virtualized hosts [10, 33]. In evaluating these enhancements, we would also exercise Q-Clouds with a more diverse set of workloads.

A second direction that was not discussed in this paper concerns issues around applications with phased behaviors. Depending upon the relative time constants between phased behavior and the windows across which SLAs are

defined and enforced, it may be necessary to manage a system in a phase-aware manner. In order to address this need, we can consider if and how prior work on phase monitoring and prediction can be applied to our control and modeling approaches [12]. A final direction of future work includes investigating the integration of performance interference aware control for dynamic workload placement using live migration techniques. Here we hope to better understand how different application mixes affect the overall benefits of our solution, and if workload heterogeneity can be exploited to make better use of cloud hardware under QoS constraints.

## References

- [1] T. Abdelzaher, K. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), 2002.
- [2] Amazon Elastic Compute Cloud. <http://aws.amazon.com/ec2>.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
- [4] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2007.
- [5] A. Byde, M. Salle, and C. Bartolini. Market-based resource allocation for utility data centers. Technical Report HPL-2003-188, HP Laboratories, September 2003.
- [6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.
- [7] Y. Dong, Z. Yu, and G. Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Proceedings of the First Workshop on I/O Virtualization (WIOV)*, December 2008.
- [8] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2007.
- [9] Google App Engine. <http://code.google.com/appengine>.
- [10] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2009.
- [11] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2009.

- [12] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2006.
- [13] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. Vm3: Measuring, modeling and managing vm shared resources. *Journal of Computer Networks*, 53(17), 2009.
- [14] A. Kansal, J. Liu, A. Singh, R. Nathuji, and T. Abdelzaher. Semantic-less coordination of power management and application performance. In *Workshop on Power Aware Computing and Systems (HotPower)*, October 2009.
- [15] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006.
- [16] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 200–209, April 2007.
- [17] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.
- [18] Microsoft Azure Services Platform. <http://www.microsoft.com/azure>.
- [19] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium*, 2007.
- [20] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2008.
- [21] R. Nathuji and K. Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.
- [22] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*, June 2007.
- [23] R. Nathuji, P. England, P. Sharma, and A. Singh. Feedback driven qos-aware power budgeting for virtualized servers. In *Proceedings of the Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID)*, April 2009.
- [24] M. Norgaard, O. Ravn, N. K. Poulsen, and L. K. Hansen. *Neural Networks for Modelling and Control of Dynamic Systems*. Springer, April 2003.
- [25] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the EuroSys Conference*, 2007.
- [26] P. Padala, K.-Y. Hou, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the EuroSys Conference*, 2009.
- [27] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2006.
- [28] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No “power” struggles: Coordinated multi-level power management for the data center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.
- [29] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2009.
- [30] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995.
- [31] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2008.
- [32] VMware ESX. <http://www.vmware.com/products/esx>.
- [33] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2007.
- [34] X. Wang and Y. Wang. Co-con: Coordinated control of power and application performance for virtualized server clusters. In *Proceedings of the 17th IEEE International Workshop on Quality of Service (IWQoS)*, Charleston, South Carolina, July 2009.
- [35] Windows Server 2008 R2 Hyper-V. <http://www.microsoft.com/hyper-v>.
- [36] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2009.
- [37] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the EuroSys Conference*, March 2009.
- [38] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2007.
- [39] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

# Lithium: Virtual Machine Storage for the Cloud

Jacob Gorm Hansen  
VMware  
Aarhus, Denmark  
jgorm@vmware.com

Eric Jul  
Bell Laboratories  
Alcatel-Lucent, Dublin, Ireland  
eric@cs.bell-labs.com

## ABSTRACT

To address the limitations of centralized shared storage for cloud computing, we are building Lithium, a distributed storage system designed specifically for virtualization workloads running in large-scale data centers and clouds. Lithium aims to be scalable, highly available, and compatible with commodity hardware and existing application software. The design of Lithium borrows ideas and techniques originating from research into Byzantine Fault Tolerance systems and popularized by distributed version control software, and demonstrates their practical applicability to the performance-sensitive problem of VM hosting. To our initial surprise, we have found that seemingly expensive techniques such as versioned storage and incremental hashing can lead to a system that is not only more robust to data corruption and host failures, but also often faster than naïve approaches and, for a relatively small cluster of just eight hosts, performs well compared with an enterprise-class Fibre Channel disk array.

## Categories and Subject Descriptors

D.4.3 [Operating Systems]: File Systems Management—*Distributed file systems*; H.3.4 [Information storage and retrieval]: Systems and Software—*Distributed Systems*; D.4.5 [Operating Systems]: Reliability—*Fault-tolerance*

## General Terms

Design, Experimentation, Performance, Reliability

## 1. INTRODUCTION

The emergence of cloud computing has motivated the creation of a new generation of distributed storage systems. These systems trade backwards-compatibility in exchange for better scalability and resiliency to disk and host failures. Instead of centralizing storage in a Storage Area Network (SAN), they use cheap directly attached storage and massive replication to keep data durable and available. The downside to these cloud-scale storage systems is their lack

of backwards-compatibility with legacy software written before the cloud-era. Virtual machine monitors like VMware's ESX platform excel at running both new and legacy software inside virtual machines (VMs), but have traditionally depended on centralized shared storage as to take advantage of virtualization features such as live VM migration [10, 32] and host fail-over [44]. Local storage is rarely used for hosting VMs because it ties them to specific hosts thereby making them vulnerable to single host failures.

Though SAN and NAS (Network Attached Storage) disk arrays often are veritable storage super computers, their scalability is fundamentally limited, and occasionally they become bottlenecks that limit the virtualization potential of certain applications. As new generations of processors and chipsets in compute nodes sport more cores and more memory, consolidation factors go up and the array's disks, service processors, and network links become increasingly congested. Already today, businesses running virtualized desktops must work around the "boot storm" that occurs every morning when hundreds of employees try to simultaneously power on their VM-hosted desktops. Because it is a single point of failure, the array must also be constructed out of highly reliable components, which makes it expensive compared to commodity hardware. When using shared storage, scalability comes at a high cost.

As a potential alternative to shared storage, we propose a distributed storage system that makes VM storage location-independent and exploits the local storage capacity of compute nodes, hereby increasing flexibility and lowering the cost of virtual machine-based cloud computing. Our definition of a cloud is not limited to a single data center, but covers the spectrum from smartphones and laptops running VMs to dedicated local infrastructure combined with shared resources rented from utility providers. Therefore, our proposed system supports both strict and eventual consistency models, to allow consistent periodic off-site backups, long-distance VM migration, and off-line use of VMs running on disconnected devices such as laptops and smartphones. Our contribution is that we demonstrate the feasibility of self-certifying and eventually consistent replicated storage for performance-sensitive virtualization workloads; workloads that previously depended on specialized shared-storage hardware. Practical innovations include a local storage engine optimized for strict and eventual consistency replication with built-in robustness to data corruption, disk and host failures, and a novel cryptographic locking mechanism that obviates the need for centralized or distributed lock management that could otherwise limit scalability.

## 1.1 Background

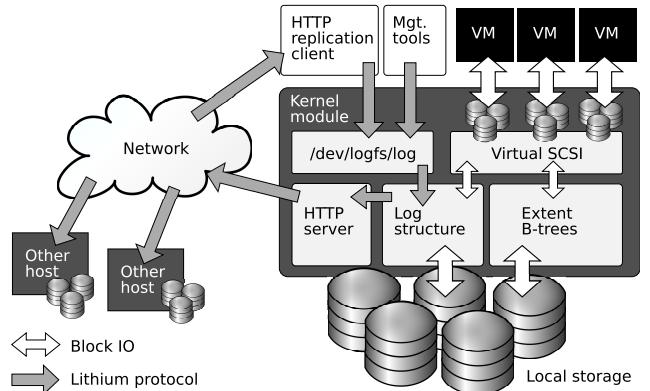
This work describes a distributed block storage system that provides a backwards-compatible SCSI disk emulation to a potentially large set of VMs running in a data center or cloud hosting facility. The aim is not to replace high-end storage arrays for applications that require the combined throughput of tens or hundreds of disk spindles, but rather to address the scalability and reliability needs of many smaller VMs that individually would run acceptably from one or a few local disk drives, but today require shared storage for taking advantage of features such as seamless VM migration. Examples include hosted virtual desktop VMs, web and email servers, and low-end online transaction processing (OLTP) applications. To allow VMs to survive disk or host failures, our proposed system uses peer-to-peer replication at the level of individual per-VM storage volumes. Though replication systems are well-studied and understood, the construction of a replication system for live VMs is complicated by practical issues such as disk scheduler non-determinism and disk or wire data corruption, both of which can result in silent replica divergence, and by performance interference from multiple workloads contending for local and remote disk heads, which can greatly inhibit individual VM storage throughput.

While a number of scalable storage systems have been proposed, most are only optimized for certain access patterns that do not fit the requirements of VM storage. The Google File System [16], for instance, is optimized for append-only workloads, and other systems only store simple key-value pairs of limited size [15, 42], or require objects to be immutable once inserted [37, 13, 3]. OceanStore’s Pond [35] and Antiquity [47] prototypes come close to meeting the requirements of VM storage, but Pond’s use of expensive erasure coding results in dismal write performance, and Antiquity’s *two-level naming* interface is not directly applicable to virtual disk emulation. Amazon’s Elastic Block Store [2] is a distributed storage system for VMs, but its durability guarantees, to the best of our knowledge, hinge on regular manual snapshot backups to Amazon S3, resulting in a trade-off between durability and performance. Section 5 surveys more related work, but based on the above, we believe that the design space for scalable distributed VM storage is still substantially uncharted and that a “blank slate” design approach is warranted.

## 2. DESIGN

Our proposed system, Lithium, is a scalable distributed storage system optimized for the characteristics of virtual machine IO: random and mostly exclusive block accesses to large and sparsely populated virtual disk files. For scalability reasons, Lithium does not provide a globally consistent POSIX name space, but names stored objects in a location-independent manner using globally unique incremental state hashes, similar to a key-value store but where each value is a large 64-bit address space. Lithium supports instant volume creation with lazy space allocation, instant creation of writable snapshots, and replication of volumes and snapshots with tunable consistency ranging from per-update synchronous replication to eventual consistency modes that allow VMs to remain available during periods of network congestion or disconnection.

Lithium’s underlying data format is self-certifying and self-



**Figure 1: Overview of the components of Lithium on each host.** VMs access the Lithium hypervisor kernel module through a virtual SCSI interface. The log-structured storage engine lives in the kernel, and policy-driven tools append to the log from user-space through the `/dev/logfs/log` device node.

sufficient, in the sense that data can be stored on unreliable hosts and disk drives without fear of silent data corruption, and all distributed aspects of the protocol are persisted as part of the core data format. This means that Lithium can function independently, without reliance on any centralized lock management or meta-data service, and can scale from just a couple of hosts to a large cluster. The design is compatible with commodity off-the-shelf server hardware and, thus does not require non-volatile RAM or other “magic” hardware.

Lithium consists of a kernel module for the VMware ESX hypervisor, and of a number of small user space tools that handle volume creation, branching, and replication. Figure 1 shows the main components installed on a Lithium-enabled host. All data is made persistent in a single on-disk log-structure that is indexed by multiple B-trees. The kernel module handles performance-critical network and local storage management, while user space processes that interface with the kernel module through a special control device node, are responsible for policy-driven tasks such as volume creation, branching, replication, membership management, and fail-over. This split responsibilities design allows for more rapid development of high-level policy code, while retaining performance on the critical data path.

In Lithium, data is stored in *volumes* – 64-bit block address spaces with physical disk drive semantics. However, a volume is also akin to a file in a traditional file system, in that space is allocated on demand, and that the volume can grow as large as the physical storage on which it resides. Volumes differ from files in that their names are 160-bit unique identifiers chosen at random from a single flat name space. A VM’s writes to a given volume are logged in the shared log-structure, and the location of the logical blocks written recorded in the volume’s B-tree. Each write in the log is prefixed by a commit header with a strong checksum, update ordering information, and the logical block addresses affected by the write. Writes can optionally be forwarded to other hosts for replication, and replica hosts maintain their own log and B-trees for each replicated volume, so that the replica hosts are ready to assume control over the volume, when necessary.

A new volume can be created on any machine by appending

an appropriate log entry to the local log through the control device node. The new volume is identified by a permanent base id (chosen at random upon creation) and a current version id. The version id is basically an incremental hash of all updates in the volume’s history, which allows for quick replica integrity verification by a simple comparison of version ids.

Other hosts can create replicas of a volume merely by connecting to a tiny HTTP server inside the Lithium kernel module. They can sync up from a given version id, and when completely synced, the HTTP connection switches over to synchronous replication. As long as the connection remains open, the volume replicas stay tightly synchronized. The use of HTTP affects only the initial connection setup and does not result in additional network round-trips or latencies. The host that created the volume starts out as the *primary* (or *owner*) for that volume. The primary serializes access, and is typically the host where the VM is running. Ownership can be transferred seamlessly to another replica, *e.g.*, when a VM migrates, which then becomes the new primary. The integrity and consistency of replicas is protected by a partially ordered data model known as *fork-consistency*, described next.

## 2.1 Fork-consistent Replication

Like a number of other recent research systems [26, 17, 47], updates in Lithium form a hash-chain, with individual updates uniquely identified and partially ordered using cryptographic hashes of their contexts and contents. This technique is often referred to as “fork-consistency” and also forms the basis of popular distributed source code versioning tools such as Mercurial [24]. Use of a partial ordering rather than a total ordering removes the need for a central server acting as a coordinator for update version numbers, and allows for simple distributed branching of storage objects, which in a storage system is useful for cloning and snapshotting of volumes. The use of strong checksums of both update contents and the accumulated state of the replication state machine allows for easy detection of replica divergence and integrity errors (“forks”). Each update has a unique id and a parent id, where the unique id is computed as a secure digest of the parent id concatenated with the contents of the current update, *i.e.*,  $id = h(parentid||updatecontents)$ , where  $h()$  is the secure digest implemented by a strong hash function (SHA-1 in our implementation). By having updates form a hash-chain with strong checksums, it becomes possible to replicate data objects onto untrusted and potentially Byzantine hardware; recent studies have found such Byzantine hardware surprisingly common [4]. Figure 2 shows how each volume is stored as a chain of updates where the chain is formed by backward references to parent updates. Fork-consistency allows a virtual disk volume to be mirrored anywhere, any number of times, and allows anyone to clone or snapshot a volume without coordinating with other hosts. Snapshots are first-class objects with their own unique base and version ids, and can be stored and accessed independently.

Lithium uses replication rather than erasure coding for redundancy. Erasure coding is an optimization that saves disk space and network bandwidth for replicas, but also complicates system design, multiplies the number of network and disk IOs needed to service a single client read, and can lead to new problems such as TCP throughput collapse [34]. Nev-

ertheless, if necessary, we expect that erasure codes or other bandwidth-saving measures can be added to Lithium, *e.g.*, merely by adding a user-space tool that splits or transforms the HTTP update stream before data gets relayed to other hosts.

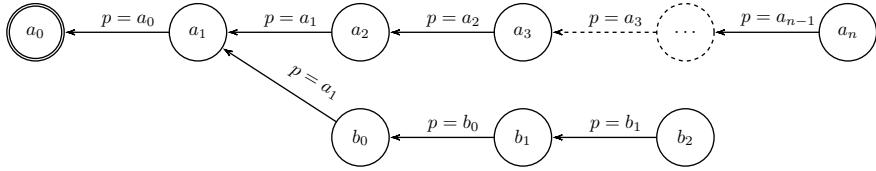
## 2.2 Replica Consistency

General state-machine replication systems have been studied extensively and the theory of their operation is well understood [40]. In practice, however, several complicating factors make building a well-performing and correct VM disk replication system less than straight-forward. Examples include parallel queued IOs that in combination with disk scheduler nondeterminism can result in replica divergence, and network and disk data corruption that if not checked will result in integrity errors propagating between hosts. Finally, the often massive sizes of the objects to be replicated makes full resynchronization of replicas, for instance to ensure consistency after a crash, impractical.

In a replication system that updates data in place, a classic two-phase commit (2PC) protocol [12] is necessary to avoid replica divergence. Updates are first logged out-of-place at all replicas, and when all have acknowledged receipt, the *head* node tells them to destructively commit. Unfortunately, 2PC protocols suffer a performance overhead from having to write everything twice. If 2PC is not used, a crash can result in not all replicas applying updates that were in flight at the time of the crash. Disk semantics require an update that has been acknowledged back to the application to be reflected onto the disk always, but updates may be on disk even if they have never been acknowledged. It is acceptable for the replicas to diverge, as long as the divergence is masked or repaired before data is returned to the application. In practice, this means that the cost of 2PC can be avoided as long as replicas are properly resynchronized after a crash. To avoid a full re-sync of replicas after a crash, previous disk replication systems, such as Petal [21], have used a dirty-bitmap of unstable disk regions where writes have recently been outstanding. The main problem with this approach is that it requires a bitmap per replica, which can be cumbersome, if the replication degree is high. Furthermore, this stateful approach requires persistent tracking of which replica was the primary at the time of the crash, to prevent an old replica overwriting a newer one.

In contrast, Lithium’s replication protocol is stateless and supports an unlimited amount of both lagging and synchronous replicas. While 3-way replication may provide sufficient reliability in most use cases, higher-degree replication may sometimes be desired, *e.g.*, for read-only shared system base disks used by many VMs. Bitmaps are impractical for high-degree replication. Instead, Lithium uses update logging to support Bayou-like eventual consistency [43].

The Lithium replication mechanism is similar to the first phase of 2PC in that updates are logged non-destructively. Once an update has been logged at a quorum of the replicas, the write IO is acknowledged back to the application VM. In eventual-consistency mode, IOs are acknowledged when they complete locally. Because updates are non-destructive, crashed or disconnected replicas can sync up by replaying of missing log updates from peers. Data lives permanently in the log, so the protocol does not require a separate commit-phase. Writes are issued and serialized at the primary replica, typically where the VM is running. We make the simplifying



**Figure 2: Partial update ordering and branches in a traditional fork-consistent system. The partial ordering is maintained through explicit “parent” back-references, and branches can be expressed by having multiple references to the same parent.**

assumption that writes always commit to the primary’s log, even when the VM tries to abort them or the drive returns a non-fatal error condition. We catch and attempt to repair drive errors (typically retry or busy errors) by quiescing IO to the drive and retrying the write, redirecting to another part of the disk, if necessary. This assumption allows us to issue IO locally at the primary and to the replicas in parallel. We also allow multiple IOs to be outstanding, which is a requirement for exploiting the full disk bandwidth. The strong checksum in our disk format commit headers ensures that incomplete updates can be detected and discarded. This means that we do not require a special “end-of-file” symbol: we can recover the end of the log after a crash simply by rolling forward from a checkpoint until we encounter an incomplete update. To avoid problems with “holes” in the log, we make sure always to acknowledge writes back to the VM in log order, even if reordered by the drive. For mutable volumes, reads are always handled at the primary, whereas immutable (snapshot) volumes can be read-shared across many hosts, *e.g.*, to provide scalable access to VM template images.

The on-disk format guarantees temporal ordering, so it is trivial to find the most recent out of a set of replicas, and because disk semantics as described above are sufficient, it is always safe to roll a group of replicas forward to the state of the most recent one. We use an access token, described in section 2.4, to ensure replica freshness and mutual exclusion with a single combined mechanism.

### 2.3 Log-structuring

On each host, Lithium exposes a single on-disk log-structure as a number of volumes. Each entry in the log updates either block data or the meta-data of a volume. The state of each volume equals the sum of its updates, and a volume can be recreated on another host simply by copying its update history. Each host maintains per-volume B-tree indexes to allow random accesses into each volume’s 64-bit block address space. Volumes can be created from scratch, or as branches from a base volume snapshot. In that case, multiple B-trees are chained, so that reads “fall through” empty regions in the children B-trees and are resolved recursively at the parent level. Branching a volume involves the creation of two new empty B-trees, corresponding to the left and right branches. Branching is a constant-time near-instant operation. Unwritten regions consume no B-tree or log space, so there is no need to pre-allocate space for volumes or to zero disk blocks in the background to prevent information leakage between VMs.

The disk log is the only authoritative data store used for holding hard state that is subject to replication. All other state is host-local soft-state that in the extreme can be repre-

ated from the log. As such, Lithium may be considered a log-structured file system (LFS) [36] or a scalable collection of logical disks [14]. Log-structuring is still regarded by many as a purely academic construct hampered by cleaning overheads and dismal sequential read performance. For replicated storage, however, log-structuring has a number of attractive properties:

**Temporal Order Preservation:** In a replication system, the main and overriding attraction of an LFS is that updates are non-destructive and temporally ordered. This greatly simplifies the implementation of an eventually consistent [43] replication system, for instance to support disconnected operation [19] or periodic asynchronous replication to off-site locations.

**Optimized for Random Writes:** Replication workloads are often write-biased. For example, a workload with a read-skewed 2:1 read-to-write ratio without replication is transformed to a write-skewed 3:2 write-to-read ratio by triple replication. Furthermore, parallel replication of multiple virtual disk update streams exhibits poor locality because each stream wants to position the disk head in its partition of the physical platter, but with a log-structured layout all replication writes can be made sequential, fully exploiting disk write bandwidth. Other methods, such as erasure-coding or de-duplication, can reduce the bandwidth needs, but do not by themselves remedy the disk head contention problems caused by many parallel replication streams. While it is true that sequential read performance may be negatively affected by log-structuring, large sequential IOs are rare when multiple VMs compete for disk bandwidth.

**Live Storage Migration:** Virtual machine migration [10, 32] enables automated load-balancing for better hardware utilization with less upfront planning. If the bindings between storage objects and hardware cannot change, hotspots and bottlenecks will develop. Just like with *live* migration of VMs, it is beneficial to keep storage objects available while they are being copied or migrated. A delta-oriented format such as the one used by Lithium simplifies object migration because changes made during migration can be trivially extracted from the source and appended to the destination log.

**Flash-compatible:** Whether or not all enterprise-storage is going to be Flash-based in the future is still a subject of debate [31], but some workloads clearly benefit from the access latencies offered by Flash SSDs. Because solid-state memories are likely to remain more expensive than disks, using them for backup replicas is generally considered a waste. A good compromise could be a hybrid approach where the primary workload uses Flash, and cheaper disk storage is used for backup replicas. However, to keep up with a Flash-based primary, the replicas will have to be write-

optimized, *i.e.*, log-structured. Because of the low access latencies and other inherent characteristics of Flash, log-structuring also benefits the Flash-based primary [5]. The downside to log-structuring is that it creates additional fragmentation, and that a level of indirection is needed to resolve logical block addresses into physical block addresses. Interestingly, when we started building Lithium, we did not expect log-structured logical disks to perform well enough for actual hosting of VMs, but expected them to run only as live write-only backups at replicas. However, we have found that the extra level of indirection rarely hurts performance, and now use the log format both for actual hosting of VMs and for their replicas.

## 2.4 Mutual Exclusion

We have described the fork-consistency model that uses cryptographic checksums to prevent silent replica divergence as a result of silent data corruption, and that allows branches of volumes to be created anywhere and treated as first-class replicated objects. We now describe our extensions to fork consistency that allow us to emulate shared-storage semantics across multiple volume replicas, and that allow us to deal with network and host failures. While such functionality could also have been provided through an external service, such as a distributed lock manager, a mechanism that is integrated with the core storage protocol is going to be more robust, because a single mechanism is responsible for ordering, replicating, and persisting both data and mutual exclusion meta-data updates. This is implemented as an extension to the fork-consistency protocol by adding the following functionality:

- A mutual exclusion *access token* constructed by augmenting the partial ordering of updates, using one-time secrets derived from a single master key referred to as the *secret view-stamp*.
- A fallback-mechanism that allows a majority quorum of the replicas to recover control from a failed primary by recovering the master key through secret sharing and a voting procedure.

### 2.4.1 Access Token

For emulating the semantics of a shared storage device, such as a SCSI disk connected to multiple hosts, fork-consistency alone is insufficient. VMs expect either “read-your-writes consistency”, where all writes are reflected in subsequent reads, or the weaker “session-consistency”, where durability of writes is not guaranteed across crashes or power losses (corresponding to a disk controller configured with write-back caching). In Lithium, the two modes correspond to either synchronous or asynchronous eventually-consistent replication. In the latter case, we ensure session consistency by restarting the VM after any loss of data.

Our SCSI emulation supports mutual exclusion through the `reserve` and `release` commands that lock and unlock a volume, and our update protocol implicitly reflects ownership information in a volume’s update history, so that replicas can verify that updates were created by a single exclusive owner. To ensure that this mechanism is not the weakest link of the protocol, we require the same level of integrity protection as with regular volume data. To this end, we introduce the notion of an *access token*, a one-time secret key that must be known to update a volume. The access token is constructed so that only an up-to-date replica can use it

to mutate its own copy of the volume. If used on the wrong volume, or on the wrong version of the correct volume, the token is useless.

Every time the primary replica creates an update, a new secret token is generated and stored locally in volatile memory. Only the current primary knows the current token, and the only way for volume ownership to change is by an explicit token exchange with another host. Because the token is a one-time password, the only host that is able to create valid updates for a given volume is the current primary. Replicas observe the update stream and can verify its integrity and correctness, but lack the necessary knowledge for updating the volume themselves. At any time, the only difference between the primary and the replicas is that only the primary knows the current access token.

The access token is constructed on top of fork-consistency by altering the partial ordering of the update hash chain. Instead of storing the clear text *id* of a version in the log entry, we include a randomly generated secret *view-stamp* *s* in the generation of the *id* for an update, so that

$$\begin{aligned} \text{instead of: } id &= h(\text{parentid} || \text{update}) \\ \text{we use: } id &= h(s || \text{parentid} || \text{update}) \end{aligned}$$

where *h* is our strong hash function. The secret view-stamp *s* is known only by the current primary. In the update header, instead of storing the clear text value of *id*, we store *h(id)* and only keep *id* in local system memory (because *h()* is a strong hash function, guessing *id* from *h(id)* is assumed impossible). Only in the successor entry’s *parentid* field do we store *id* in clear text. In other words, we change the pairwise ordering

$$\begin{aligned} \text{from: } a < b &\text{ iff } a.\text{id} = b.\text{parentid} \\ \text{to: } a < b &\text{ iff } a.\text{id} = h(b.\text{parentid}) \end{aligned}$$

where *<* is the direct predecessor relation. The difference here being that in order to name *a* as the predecessor to *b*, the host creating the update must know *a.id*, which only the primary replica does. Other hosts know *h(a.id)*, but not yet *a.id*. We refer to the most recent *id* as the *secret access token* for that volume. The randomly generated *s* makes access tokens unpredictable. While anyone can replicate a volume, only the primary, which knows the current access token *id*, can mutate it. Volume ownership can be passed to another host by exchange of the most recent access token. Assuming a correctly implemented token exchange protocol, replica divergence is now as improbable as inversion of *h()*. Figure 3 shows how updates are protected by one-time secret access tokens. The new model still supports decentralized branch creation, but branching is now explicit. When the parent for an update is stated as described above, the update will be applied to the existing branch. If it is stated as in the old model, *e.g.*, *b.parentid* = *a.id*, then a new branch will be created and the update applied there.

### 2.4.2 Automated Fail-over

A crash of the primary results in the loss of the access token, rendering the volume inaccessible to all. To allow access to be restored, we construct on top of the access token a view-change protocol that allows recovery of *s* by a remaining majority of nodes. A recovered *s* can be used to also recover *id*. To allow recovery of *s*, we treat it as a secret view-stamp [33], or epoch number, that stays constant as long as the same host is the primary for the volume, and we

use secret sharing to disperse slices of  $s$  across the remaining hosts. Instead of exchanging the access token directly, volume ownership is changed by appending a special *view-change* record to the log. The view-change record does not contain  $s$  but its public derivate  $h(s)$ , along with slices of  $s$ , encrypted under each replica’s host key. Thus  $h(s)$  is the *public view-stamp* identifying that view.

If a host suspects that the current primary has died, it generates a new random  $s'$  and proposes the corresponding view  $h(s')$  to the others. Similar to Paxos [20], conflicting proposals are resolved in favor of the largest view-stamp. Hosts cast their votes by returning their slices of  $s$ , and whoever collects enough slices to recover  $s$  wins and becomes the new primary. The new primary now knows both  $s$  and  $s'$  so it can reconstruct  $id$  using the recovered  $s$ , and append the view-change that changes the view to  $h(s')$ . When there is agreement on the new view, IO can commence. The former  $s$  can now be made public to revoke the view  $h(s)$  everywhere. If there are still replicas running in the former view  $h(s)$ , the eventual receipt of  $s$  will convince them that their views are stale and must be refreshed. Should the former primary still be alive and eventually learn  $s$ , it knows that it has to immediately discard of its copy of the VM and any changes made (if running in eventual consistency mode) after the point of divergence. Discarding and restarting the VM in this case ensures session-consistency.

The use of the secret token protects against replica divergence as a result of implementation or data corruption errors. Prevention against deliberate attacks would require signing of updates, which is straightforward to add. Using signatures alone is not enough, because a signature guarantees authenticity only, not freshness or mutual exclusion as provided by the access token. We have found that access tokens are a simple and practical alternative to centralized lock servers. During development, the mechanism has helped identify several implementation and protocol errors by turning replica divergence into a fail-stop condition.

### 3. IMPLEMENTATION DETAILS

Lithium treats each local storage device as a single log that holds a number of virtual volumes. Write operations translate directly into synchronous update records in the log, with the on-disk location of the update data being recorded in a B-tree index. The update may also propagate to other hosts that replicate the particular volume. Depending on configuration, the write is either acknowledged back to the VM immediately when the update is stable in the local log, or when all or a quorum of replicas have acknowledged the update. The first mode allows for disconnected operation with eventual consistency, while the second mode corresponds to synchronous replication.

When storing data on disk and when sending updates over the wire, the same format is used. A 512-byte log entry commit header describes the context of the update (its globally unique id and the name of its parent), the location of the update inside the virtual disk’s 64-bit address space, the length of the extent, and a strong checksum to protect against data corruption. The rest of the header space is occupied by a bit vector used for compressing away zero blocks, both to save disk space and network bandwidth (in practice, this often more than offsets the cost of commit headers), and to simplify log-compaction. For practical reasons, the on-disk log is divided into fixed size segments of 16MB each.

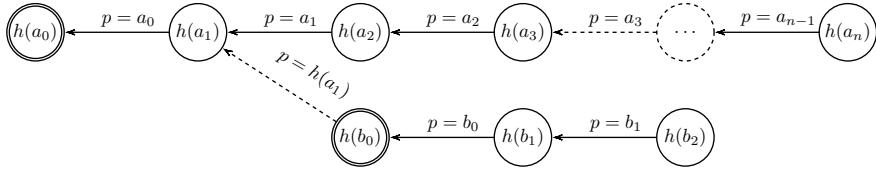
Workload	Type	Avg. len	Overhead
Windows XP (NTFS)	FS	29.9	0.26%
IOZone (XFS)	FS	33.8	0.23%
PostMark (XFS)	FS	61.8	0.13%
DVDStore2 (MySQL)	DB	24.0	0.33%
4096-byte random	Synthetic	8.0	1.0%
512-byte random	Synthetic	1.0	7.8%

Table 1: Extent Sizes and B-tree Memory Overhead

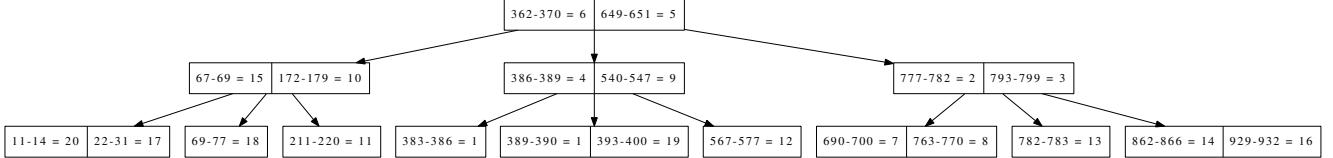
### 3.1 B-tree Indexes

Applications running on top of Lithium are unaware of the underlying log-structure of physical storage, so to support random access reads, Lithium needs to maintain an index that maps between the logical block addresses (LBAs) used by VMs, and the actual physical locations of the data on disk. This index could be implemented as a per-volume lookup table with an entry for each logical block number in the volume, but because each Lithium volume is a 64-bit block address space, a lookup table might become prohibitively large. Instead, Lithium uses a 64-bit extent indexed B-tree to track logical block locations in the log. The B-tree is a more complex data structure than a simple lookup table or a radix tree, but is also more flexible, and designed to perform well on disk. The B-tree index does not track individual block locations, but entire extents. If a VM writes a large file into a single contiguous area on disk, this will be reflected as just a single key in the B-tree. Figure 4 shows an example B-tree index with block ranges that correspond to different log entries. In the pathological worst case, the VM may write everything as tiny, random IOs, but we have found that on average the use of extents provides good compression. For instance, a base Windows XP install has an average extent length of 29.9 512-byte disk sectors. Each B-tree extent key is 20 bytes, and B-tree nodes are always at least half full. In this case the space overhead for the B-tree is at most  $\frac{20 \times 2}{512 \times 29.9} \approx 0.26\%$  of the disk space used by the VM. Table 1 lists the average extent sizes and resulting B-tree memory overheads we have encountered during development, along with the pathological worst cases of densely packed 4kB and 512B completely random writes. Guest OS disk schedulers attempt to merge adjacent IOs, which explains the large average extent sizes of the non-synthetic workloads. We demand-page B-tree nodes and cache them in main memory (using pseudo-LRU cache replacement), and as long as a similar percentage of the application’s disk working set fits in the cache, performance is largely unaffected by the additional level of indirection.

We carefully optimized the B-tree performance: Most IO operations are completely asynchronous, the tree has a large fan-out (more than a thousand keys per tree node), and updates are buffered and applied in large batches to amortize IO costs. The B-tree uses copy-on-write updates and is crash-consistent; it is only a cache of the authoritative state in the log: should it be damaged, it can always be recreated. The B-tree itself is not replicated, but if two hosts replicate the same volume, they will end up with similar B-trees. The B-trees are always kept up to date at all replicas, so control of a volume can migrate to another host instantly. Periodic checkpoints of B-trees and other soft state ensure constant time restart recovery.



**Figure 3: Update ordering and branches in Lithium.** In contrast to the original fork-consistent model in Figure 2, appending an update to an existing branch requires knowledge of its clear-text *id* to state it as a parent reference. Branching is an explicit but unprivileged operation.



**Figure 4: Simplified example of the B-tree used for mapping logical block offsets to version numbers in the log.** Block number 765 would resolve to revision number 8, and block 820 would return NULL, indication that the block were empty. Multiple trees can be chained together, to provide snapshot and cloning functionality.

### 3.2 Log Compaction

A local log compactor runs on each host in parallel with other workloads, and takes care of reclaiming free space as it shows up. Free space results from VMs that write the same logical blocks more than once, and the frequency with which this occurs is highly workload-dependent. Free space is tracked per log-segment in an in-memory priority queue, updated every time a logical block is overwritten in one of the volume B-trees, and persisted to disk as part of periodic checkpoints. When the topmost  $n$  segments in the queue have enough free space for a compaction to be worthwhile, the log compactor reads those  $n$  segments and writes out at most  $n - 1$  new compacted segments, consulting the relevant B-trees to detect unreferenced blocks along the way. Unreferenced blocks are compressed away using the zero block bit vector described previously. The log compactor can operate on arbitrary log segments in any order, but preserves the temporal ordering of updates as not to conflict with eventual consistency replication. It runs in parallel with other workloads and can be safely suspended at any time. In a real deployment, one would likely want to exploit diurnal workload patterns to run the compaction only when disks are idle, but in our experiments, we run the compactor as soon as enough free space is available for compaction to be worthwhile.

Due to compaction, a lagging replica that syncs up from a compacted version may not see the complete history of a volume, but when the replica has been brought up to the compacted version, it will have a complete and usable copy. Though we remove overwritten data blocks, we currently keep their empty update headers to allow lagging replicas to catch up from arbitrary versions. In the future, we plan to add simple checkpoints to our protocol to avoid storing update headers perpetually.

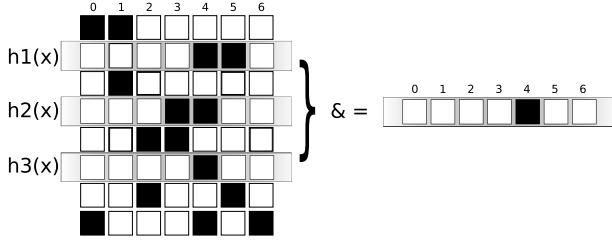
### 3.3 Locating Updates on Disk

The partial ordering of log updates has benefits with regards to cheap branches and simplifying exchange of updates. However, with 160 bits of address space and multiple terabytes of log space, finding the successor to an arbitrary

volume revision can be a challenge. This may be necessary when a lagging replica connects, because we only maintain soft state about replicas and their current versions, soft state that may be lost in a crash or forgotten due to a replica having been off-line. Unfortunately, the lack of locality in our version identifiers precludes the use of any dictionary data structure (*e.g.*, tree or hash table) that relies on key locality for update performance. On the other hand, arbitrary successor lookups are rare and out of the critical data path, so it is acceptable if they are much slower than updates. The brute force solution of an exhaustive scan of the disk surface is prohibitively expensive. Instead, we speed up the search by maintaining a Bloom filter [6] that describes the contents of each log segment on the disk. A Bloom filter is a simple bit vector indexed by  $k$  uniform hash functions. For a membership query to return a positive result, all bits corresponding to the results of applying the  $k$  hash functions to the key being looked up, must be set. There is a risk of false positives, but the value of  $k$  and the size of the bit vector  $m$  can be tuned to make the risk acceptably small for a given number of elements  $n$ . For our application, false positives do not affect correctness, but only performance.

A Bloom filter can compactly represent a set of revision identifier hashes, but can only answer membership queries, not point to the actual locations of data. We therefore keep a Bloom filter per 16MB log segment, and exhaustively scan all filters on a disk when looking for a specific version. If more than one filter claims to have the version, we perform a deep-scan of each corresponding log segment until we have an exact match. Each Bloom filter has  $m = 2^{16}$  bits (8,192 bytes) and is indexed by  $k = 10$  hash functions, corresponding to the ten 16-bit words of the 160-bit SHA-1 hash that is looked up. At the expected rate of up to  $n = 4,000$  log entries per segment, the filter has a false positive rate of  $(1 - (1 - \frac{1}{m})^{kn})^k \approx 0.04\%$ . An additional Bloom filter at the end of each segment, indexed by another set of hash functions, serves to square this probability, so that we rarely have to deep-scan segments in vain.

A one terabyte disk contains almost 60,000 16MB log segments, corresponding to about 480MB of Bloom filters. As



**Figure 5:** Combining  $k = 3$  entire rows of the inverted index to search multiple filters in parallel. A match is found in column 4, corresponding to log segment 4 on disk.

suming a disk read rate of 100MB/s, these can be scanned in less than 5s. However, we can do better if instead of storing each filter separately, we treat a set of filters as an *inverted index*, and store them as the columns of a bit-matrix. The columns are gradually populated in an in-memory buffer, which when full gets flushed to disk in a single write operation. Instead of reading the entire matrix for a lookup, we now only have to read the  $k$  rows corresponding to  $h_1(key), \dots, h_k(key)$ , as illustrated in Figure 5. With a 16MB matrix of 65,536 rows and 2,048 columns we are able to cover 2,048 log segments, corresponding to 32GB of physical log space. Our measurements show that, including the amortized disk write cost, we are able to insert one million keys into the inverted index per second, and that we are able to scan indexes covering between 700MB and 1TB of log space per second, with a single drive, depending on the degree to which filters have been filled (short-circuit optimization applies to the AND’ing of rows).

Multiple physical disks can be scanned in parallel to maintain scalability as disks are added. We can also increase the scanning speed by adding more columns to each index, at the cost of using more main memory for the current inverted index. For instance, a 32MB index will store 4,096 columns and double the scanning speed. As disks grow and memory becomes cheaper, the index size may be adjusted accordingly.

### 3.4 POSIX Interface

Lithium provides block-addressable object storage, but by design does not expose a global POSIX name space. Like shared memory, faithfully emulating POSIX across a network is challenging and introduces scalability bottlenecks that we wish to avoid. The VMs only expect a block-addressable virtual disk abstraction, but some of the tools that make up our data center offering need POSIX, *e.g.*, for configuration, lock, and VM-swap files. Each VM has a small collection of files that make up the runtime and configuration for that VM, and these files are expected to be kept group-consistent. Instead of rewriting our management tools, we use the VMware VMFS [45] cluster file system to provide a per-volume POSIX layer on top of Lithium. Because Lithium supports SCSI reservation semantics, VMFS runs over replicated Lithium volumes just as it would over shared storage. Popular VMware features like “VMotion” (live VM migration) and “High Availability” (distributed failure detection and automated fail-over) work on Lithium just like they would on shared storage. In a local network, the token exchange protocol described in section 2.4 is fast

enough to allow hundreds of control migrations per second, so VMFS running over a replicated Lithium volume feels very similar to VMFS running over shared storage.

## 4. EVALUATION

Our evaluation focuses on the performance of the prototype when configured for synchronous 2 and 3-way replication. To measure the performance of our prototype we ran the following IO benchmarks:

**PostMark** PostMark is a file-system benchmark that simulates a mail-server workload. In each VM we ran PostMark with 50,000 initial files and 100,000 transactions. PostMark primarily measures IO performance.

**DVDStore2** This online transaction processing benchmark simulates an Internet DVD store. We ran the benchmark with the default “small” dataset, using MySQL as database backend. Each benchmark ran for three minutes using default parameters. DVDStore2 measures a combination of CPU and IO performance.

Our test VMs ran Linux 2.6.25 with 256MB RAM and 8GB of virtual storage. Data volumes were stored either in Lithium’s log-structure, or separately in discrete 8GB disk partitions. When running Lithium, log compaction ran eagerly in parallel with the main workload, to simulate a disk full condition. Lithium was configured with 64MB of cache for its B-trees. Apart from the settings mentioned above, the benchmarks were run with default parameters.

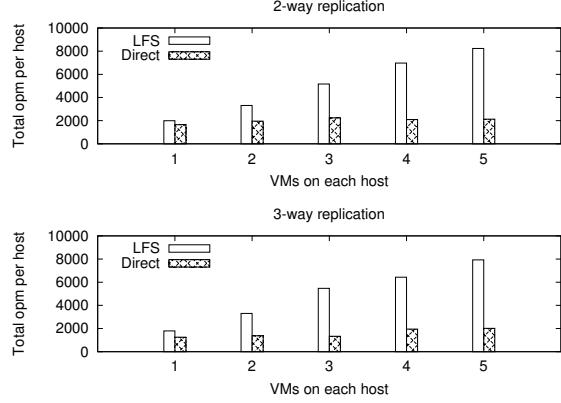
### 4.1 Replication Performance

Replication performance was one of the motivations for the choice of a write-optimized disk layout, because it had been our intuition that multiple parallel replication streams to different files on the same disk would result in poor throughput. When multiple VMs and replication streams share storage, most IO is going to be non-sequential. Apart from the caching that is already taking place inside the VMs, there is very little one can do to accelerate reads. Writes, however, can be accelerated through the use of log-structuring.

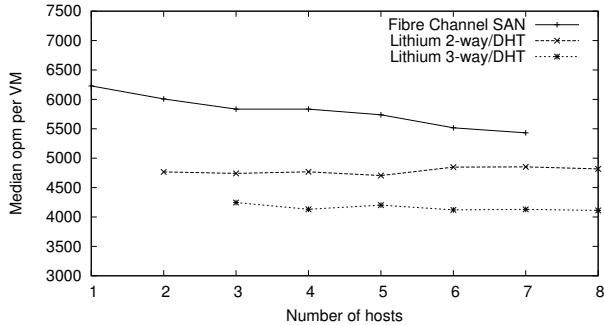
In this benchmark, we used three HP Proliant DL385 G2 hosts, each equipped with two dual-core 2.4GHz Opteron processors, 4GB RAM, and a single physical 2.5" 10K 148GB SAS disk, attached to a RAID controller without battery backed cache. We used these hosts, rather than the newer Dells described below, because they had more local storage available for VMs and replicas, and were more representative of commodity hardware. As a strawman, we modified the Lithium code to use a direct-access disk format with an 8GB *partition* of the disk assigned to each volume, instead of the normal case with a single shared log-structure. We ran the DVDStore2 benchmark, and used the total orders-per-minute number per host as the score. The results are shown in Figure 6, and clearly demonstrate the advantage of a write-optimized layout. Per-host throughput was 2–3 times higher when log-structuring was used instead of static partitioning.

### 4.2 Scalability

To test Lithium’s scalability in larger clusters, we obtained the use of eight Dell 2950 III servers, each with 16GB RAM, two quad-core 3GHz Xeon processors, and with three 32GB local SAS drives in RAID-5 configuration, using a Dell perc5/i



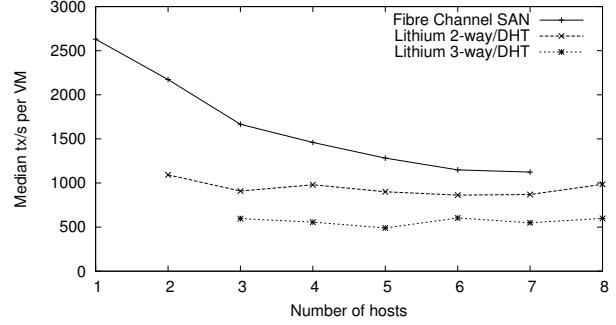
**Figure 6:** Performance of the DVD Store 2 OLTP benchmark in a two- and three-host fully replicated scenarios.



**Figure 7:** DVDStore2 Scalability. This graph shows the median orders per minute per VM, with 4 VMs per host for a varying number of hosts of Lithium in 2-way and 3-way replicated scenarios, and for a mid-range Fibre Channel storage array.

controller with 256MB of battery backed cache. Naturally this small cluster cannot be considered cloud-scale, but it was connected to a mid-range (US\$100,000 price range) Fibre Channel storage array, which provided a good reference for comparison against finely tuned enterprise hardware. The array was configured with 15 15k Fibre Channel disks in a RAID-5 configuration, presented as single LUN mounted at each ESX host by VMFS. The array was equipped with 8GB of battery backed cache, and connected to seven of the eight hosts through 4GB/s Fibre Channel links. Lithium was configured with either 2- or 3-way replication over single gigabit Ethernet links, in addition to the local RAID-5 data protection. Replicas were mapped to hosts probabilistically, using a simple distributed hash table (DHT) constructed for the purpose. In this setup, Lithium was able to tolerate three or five drive failures without data loss, where the array could only tolerate a single failure. Ideally, we would have reconfigured the local drives of the servers as RAID-0 for maximum performance, as Lithium provides its own redundancy. However, the hardware was made available to us on a temporary basis, so we were unable to alter the configuration in any way.

We ran the DVDStore2 benchmark in 4 VMs per host, and varied the number of hosts. As Figure 7 shows, a three-drive



**Figure 8:** PostMark Scalability. Median transactions per second per VM, with 4 VMs per host for a varying number of hosts of Lithium in 2-way and 3-way replicated scenarios, and for a mid-range Fibre Channel storage array.

RAID-5 per host is not enough to beat the high-performance array, but as more hosts are added, per-VM throughput remains constant for Lithium, whereas the array's performance is finite, resulting in degrading per-VM throughput as the cluster grows.

We also ran the PostMark benchmark (see Figure 8) in the same setting. PostMark is more IO intensive than DVD-Store2, and median VM performance drops faster on the storage array when hosts are added. Unfortunately, only seven hosts had working Fibre Channel links, but Lithium is likely to be as fast or faster than the array for a cluster of eight or more hosts.

Though our system introduces additional CPU overhead, *e.g.*, for update hashing, the benchmarks remained disk IO rather than CPU-bound. When we first designed the system, we measured the cost of hashing and found that a hand-optimized SHA-1 implementation, running on a single Intel Core-2 CPU core, could hash more than 270MB of data per second, almost enough to saturate the full write bandwidth of three SATA disk drives. In our view the features that the hashing enables more than warrant this CPU overhead.

As stated earlier, we did not expect Lithium to outperform finely tuned and expensive centralized systems in small-scale installations. We designed Lithium to have reasonable local performance, but first and foremost to provide incremental scalability and robustness against data corruption and disk and host failures. As the work progressed, we have been pleased to find that Lithium performance is on par with alternative approaches for local storage, and that its write-optimized disk layout shines for double and triple live replication. For small setups, the Fibre Channel array is still faster, but Lithium keeps scaling, does not have a single point of failure, and is able to tolerate both host and multiple drive failures.

## 5. RELATED WORK

Networked and distributed file systems have traditionally aimed at providing POSIX semantics over the network. Early examples such as NFS [39] and AFS [18] were centralized client-server designs, though HARP [22] used replication for availability and CODA [19] did support disconnected operation. FARsite [7] used replication and encryption of files and meta-data to tolerate Byzantine host failures, but as was the

case with CODA and AFS, provided only open-to-close data consistency, which is insufficient for VM hosting.

Lithium uses strong checksums and secret tokens to turn most Byzantine faults into fail-stop, but was not designed with Byzantine Fault Tolerance (BFT) as a goal. Fully BFT eventual consistency replication protocols have been proposed, and our storage engine design may be of relevance to developers of such systems. For instance, Zeno’s [41] need for non-volatile RAM for version numbers could likely be repaired with a Lithium-like disk format. Other systems such as Carbonite [9] focus on finding the right balance between durability and performance and are orthogonal to our work. Quorum systems built with secret sharing have been proposed for access control and signatures by Naor and Wool [30], but to our knowledge not previously been explored in the context of storage.

The Google File System was one of the first storage systems to operate at cloud-scale, but its single coordinator node became a problematic bottleneck as the system grew in size and had to support a more diverse set of applications [27]. Lithium tracks all meta-data with its stored objects and does not need a central coordinator node. Ceph [48] is a POSIX-compatible and fault-tolerant clustered file system, where files are mapped to nodes through consistent hashing. Recent versions of Ceph use *btrfs* in Linux for transactional log-structured file storage. Lithium is similar in that both use log-structuring for transactional storage, but Lithium differs in that it uses a common fork-consistent data format on disk and in the network protocol.

Boxwood [23] provides a foundation for building distributed storage systems that can be implemented as B-trees. Boxwood also provides a 2-way replication facility for variable-sized chunks used for holding the actual data pointed to by the B-tree, with a log of dirty regions used for reconciling replicas after a crash. Lithium uses B-trees internally, but focuses on providing a virtual block interface with disk semantics instead of B-tree transactions.

Parallax [28] is a snapshotting virtual volume manager for shared storage. Parallax uses 4kB block-indexed radix trees as the layer of indirection between virtual and physical disk blocks, where Lithium uses extent-indexed B-trees that point to log entries. The extent B-tree is a more dynamic data structure, because extents can vary in size from a single disk sector and up to several megabytes per write. A radix tree, on the other hand, introduces a trade-off between either very small update sizes (which results in the radix tree becoming very large and possibly limits the maximum amount of disk space than can be addressed) or larger updates that can result in read-modify-write overheads and false sharing. Parallax relies on an underlying shared storage abstraction, where Lithium emulates shared storage on top of replicated shared-nothing storage volumes.

The HP Federated Array of Bricks (FAB) [38] is a decentralized storage system that uses replication and erasure-coding, combined with voting to keep disk blocks available in spite of failures. FAB stores a timestamp with each 8MB disk block, and uses majority voting on the timestamps to ensure that all nodes return the same value for a given block (linearizability). Lithium does not apply updates in place, so linearizability and uniformity follow from the use of single primary and the temporal ordering of updates on disk. Olive [1] is an extension to FAB that supports distributed branching of disk volumes, which is complicated because

FAB applies updates in-place. Lithium’s branching functionality was simpler to implement, because updates are non-destructive.

Sun’s Zettabyte File System [8] (ZFS) is an advanced volume manager topped with a thin POSIX layer. ZFS uses Merkle hash trees to verify the integrity of disk blocks, and has good support for snapshots and for extracting deltas between them. Replication can be added by continuous snapshotting and shipping of deltas, but there is a small window where updates risk being lost if the primary host crashes. ZFS uses a fairly large minimum update size of 128kB, with smaller updates being emulated with a read-modify-write cycle. This can be expensive for VM workloads that frequently write less than 128kB, unless non-volatile RAM or other non-commodity hardware is used. In comparison, Lithium uses incremental hashes for integrity, supports single-block updates without read-modify-write, and, when configured for synchronous replication, has no window for data-loss.

Following Rosenblum and Ousterhout’s seminal LFS work, there has been a large amount of research in the area, including on how to reduce the log cleaning overheads, *e.g.*, through hole-plugging and free-block scheduling. Matthews surveys much of the design space in [25]. Others have explored hybrid designs [46], where log-structuring is used only for hot data, with cold data being migrated to more conventionally stored sections of the disks. The authors of that paper note that while seek times only improved 2.7 $\times$  between 1991 and 2002, disk bandwidth improved 10 $\times$ , making the benefits of log structuring larger now than at the time of the original LFS work. We view most of this work as orthogonal to our own, and note that replication may be the “killer app” that could finally make log-structuring mainstream.

## 6. FUTURE WORK

Flash in the form of SSDs and dedicated PCI cards is gaining popularity for IOPS-critical server workloads. We have performed simple experiments with Lithium running VMs off an SSD and using a SATA drive for storing replicas from other hosts. Going forward, we plan to explore this combination of Flash and mechanical drives further, and to experiment with faster network links, *e.g.*, 10G Ethernet with remote direct memory access (RDMA). In the version of Lithium described here, the VM always runs on a host with a full replica of its volume. We are in the process of adding remote access to replicas, so that VMs can run anywhere without having to wait for a replica being created first. We are also considering adding support for de-duplication at the extent level, leveraging previous VMware work [11]. Lithium uses simple majority quorums to prevent replica divergence. We plan to explore techniques described by Naor and Wieder [29]. They show how dynamic quorum system can be constructed that do not require a majority to operate.

## 7. CONCLUSION

Cloud computing promises to drive down the cost of computing by replacing few highly reliable, but costly, compute hosts with many cheap, but less reliable, ones. More hosts afford more redundancy, making individual hosts disposable, and system maintenance consists mainly of lazily replacing hardware when it fails. Virtual machines allow legacy soft-

ware to run unmodified in the cloud, but storage is often a limiting scalability factor.

In this paper, we have described Lithium, a fork-consistent replication system for virtual disks. Fork-consistency has previously been proposed for storing data on untrusted or Byzantine hosts, and forms the basis of popular distributed revision control systems. Our work shows that fork-consistent storage is viable even for demanding virtualized workloads such as file systems and online transaction processing. We address important practical issues, such as how to safely allow multiple outstanding IOs and how to augment the fork-consistency model with a novel cryptographic locking primitive to handle volume migration and fail-over. Furthermore, our system is able to emulate shared-storage SCSI reservation semantics and is compatible with clustered databases and file systems that use on-disk locks to coordinate access. We have shown how Lithium achieves substantial robustness both to data corruption and protocol implementation errors, and potentially unbounded scalability without bottlenecks or single points of failure. Furthermore, measurements from our prototype implementation show that Lithium is able to compete with an expensive Fibre Channel storage array on a small cluster of eight hosts, and is faster than traditional disk layouts for replication workloads.

## 8. ACKNOWLEDGEMENTS

The authors would like to thank Irfan Ahmad, Jørgen S. Hansen, Orran Krieger, Eno Thereska, George Coulouris, Eske Christiansen, Rene Schmidt, Steffen Grarup, Henning Schild, and the anonymous reviewers for their input and comments that helped shape and improve this paper.

## 9. REFERENCES

- [1] M. K. Aguilera, S. Spence, and A. Veitch. Olive: distributed point-in-time branching storage for real systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.
- [2] Amazon. Amazon Elastic Block Store (Amazon EBS). <http://aws.amazon.com/ebs>, 2009.
- [3] Amazon. Amazon Simple Storage Service (Amazon S3). <http://aws.amazon.com/S3>, 2009.
- [4] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *Trans. Storage*, 4(3):1–28, 2008.
- [5] G. Bartels and T. Mann. Cloudburst: A Compressing, Log-Structured Virtual Disk for Flash Memory. Technical Report 2001-001, Compaq Systems Research Center, February 2001.
- [6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.
- [7] W. J. Bolosky, J. R. Douceur, and J. Howell. The farsite project: a retrospective. *SIGOPS Oper. Syst. Rev.*, 41(2):17–26, 2007.
- [8] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The Zettabyte File System. Technical report, Sun Microsystems, 2003.
- [9] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.
- [10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Networked Systems Design and Implementation NSDI '05*, May 2005.
- [11] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proc. USENIX Annual Technical Conference*, San Diego, CA, 2009.
- [12] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. International Computer Science Series. Addison-Wesley, 4 edition, 2005.
- [13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.
- [14] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: a new approach to improving file systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 1993. ACM Press.
- [15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.
- [16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
- [17] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.
- [18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.
- [19] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.
- [20] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [21] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. *SIGOPS Oper. Syst. Rev.*, 30(5):84–92, 1996.
- [22] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp file system. *SIGOPS Oper. Syst. Rev.*, 25(5):226–238, 1991.
- [23] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04*:

- Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.
- [24] M. Mackall. Mercurial revision control system. <http://mercurial.selenic.com/>.
  - [25] J. N. Matthews. *Improving file system performance with adaptive methods*. PhD thesis, 1999.  
Co-Chair-Anderson, Thomas E. and  
Co-Chair-Hellerstein, Joseph M.
  - [26] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117, New York, NY, USA, 2002. ACM.
  - [27] M. K. McKusick and S. Quinlan. Gfs: Evolution on fast-forward. *Queue*, 7(7):10–20, 2009.
  - [28] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4):41–54, 2008.
  - [29] M. Naor and U. Wieder. Scalable and dynamic quorum systems. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 114–122, New York, NY, USA, 2003. ACM.
  - [30] M. Naor and A. Wool. Access control and signatures via quorum secret sharing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(9):909–922, Sep 1998.
  - [31] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158, New York, NY, USA, 2009. ACM.
  - [32] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.
  - [33] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA, 1988. ACM.
  - [34] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.
  - [35] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2003. USENIX Association.
  - [36] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
  - [37] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, 2001.
  - [38] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 48–58, New York, NY, USA, 2004. ACM.
  - [39] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.
  - [40] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
  - [41] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.
  - [42] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proc. USENIX Annual Technical Conference*, San Diego, CA, 2009.
  - [43] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.
  - [44] VMware. VMware High Availability. [http://www.vmware.com/pdf/ha\\_datasheet.pdf](http://www.vmware.com/pdf/ha_datasheet.pdf), 2009.
  - [45] VMware. VMware VMFS. [http://www.vmware.com/pdf/vmfs\\_datasheet.pdf](http://www.vmware.com/pdf/vmfs_datasheet.pdf), 2009.
  - [46] W. Wang, Y. Zhao, and R. Bunt. Hylog: A high performance approach to managing disk layout. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.
  - [47] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. In *EuroSys '07: Proceedings of the 2007 EuroSys conference*, pages 371–384, New York, NY, USA, 2007. ACM Press.
  - [48] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

---

# Bayesian Semi-supervised Learning with Graph Gaussian Processes

---

**Yin Cheng Ng<sup>1</sup>, Nicolò Colombo<sup>1</sup>, Ricardo Silva<sup>1,2</sup>**

<sup>1</sup>Statistical Science, University College London

<sup>2</sup>The Alan Turing Institute

{y.ng.12, nicolo.colombo, ricardo.silva}@ucl.ac.uk

## Abstract

We propose a data-efficient Gaussian process-based Bayesian approach to the semi-supervised learning problem on graphs. The proposed model shows extremely competitive performance when compared to the state-of-the-art graph neural networks on semi-supervised learning benchmark experiments, and outperforms the neural networks in active learning experiments where labels are scarce. Furthermore, the model does not require a validation data set for early stopping to control over-fitting. Our model can be viewed as an instance of empirical distribution regression weighted locally by network connectivity. We further motivate the intuitive construction of the model with a Bayesian linear model interpretation where the node features are filtered by an operator related to the graph Laplacian. The method can be easily implemented by adapting off-the-shelf scalable variational inference algorithms for Gaussian processes.

## 1 Introduction

Data sets with network and graph structures that describe the relationships between the data points (nodes) are abundant in the real world. Examples of such data sets include friendship graphs on social networks, citation networks of academic papers, web graphs and many others. The relational graphs often provide rich information in addition to the node features that can be exploited to build better predictive models of the node labels, which can be costly to collect. In scenarios where there are not enough resources to collect sufficient labels, it is important to design data-efficient models that can generalize well with few training labels. The class of learning problems where a relational graph of the data points is available is referred to as graph-based semi-supervised learning in the literature [7, 47].

Many of the successful graph-based semi-supervised learning models are based on graph Laplacian regularization or learning embeddings of the nodes. While these models have been widely adopted, their predictive performance leaves room for improvement. More recently, powerful graph neural networks that surpass Laplacian and embedding based methods in predictive performance have become popular. However, neural network models require relatively larger number of labels to prevent over-fitting and work well. We discuss the existing models for graph-based semi-supervised learning in detail in Section 4.

We propose a new Gaussian process model for graph-based semi-supervised learning problems that can generalize well with few labels, bridging the gap between the simpler models and the more data intensive graph neural networks. The proposed model is also competitive with graph neural networks in settings where there are sufficient labelled data. While posterior inference for the proposed model is intractable for classification problems, scalable variational inducing point approximation method for Gaussian processes can be directly applied to perform inference. Despite the potentially large number of inducing points that need to be optimized, the model is protected from over-fitting by the

variational lower bound, and does not require a validation data set for early stopping. We refer to the proposed model as the graph Gaussian process (GGP).

## 2 Background

In this section, we briefly review key concepts in Gaussian processes and the relevant variational approximation technique. Additionally, we review the graph Laplacian, which is relevant to the alternative view of the model that we describe in Section 3.1. This section also introduces the notation used across the paper.

### 2.1 Gaussian Processes

A Gaussian process  $f(\mathbf{x})$  (GP) is an infinite collection of random variables, of which any finite subset is jointly Gaussian distributed. Consequently, a GP is completely specified by its mean function  $m(\mathbf{x})$  and covariance kernel function  $k_\theta(\mathbf{x}, \mathbf{x}')$ , where  $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$  denote the possible inputs that index the GP and  $\theta$  is a set of hyper-parameters parameterizing the kernel function. We denote the GP as follows

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k_\theta(\mathbf{x}, \mathbf{x}')). \quad (1)$$

GPs are widely used as priors on functions in the Bayesian machine learning literatures because of their wide support, posterior consistency, tractable posterior in certain settings and many other good properties. Combined with a suitable likelihood function as specified in Equation 2, one can construct a regression or classification model that probabilistically accounts for uncertainties and control over-fitting through Bayesian smoothing. However, if the likelihood is non-Gaussian, such as in the case of classification, inferring the posterior process is analytically intractable and requires approximations. The GP is connected to the observed data via the likelihood function

$$y_n | f(\mathbf{x}_n) \sim p(y_n | f(\mathbf{x}_n)) \quad \forall n \in \{1, \dots, N\}. \quad (2)$$

The positive definite kernel function  $k_\theta(\mathbf{x}, \mathbf{x}') : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$  is a key component of GP that specifies the covariance of  $f(\mathbf{x})$  *a priori*. While  $k_\theta(\mathbf{x}, \mathbf{x}')$  is typically directly specified, any kernel function can be expressed as the inner product of features maps  $\langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle_{\mathcal{H}}$  in the Hilbert space  $\mathcal{H}$ . The dependency of the feature map on  $\theta$  is implicitly assumed for conciseness. The feature map  $\phi(\mathbf{x}) : \mathcal{X} \rightarrow \mathcal{H}$  projects  $\mathbf{x}$  into a typically high-dimensional (possibly infinite) feature space such that linear models in the feature space can model the target variable  $y$  effectively. Therefore, GP can equivalently be formulated as

$$f(\mathbf{x}) = \phi(\mathbf{x})^\top \mathbf{w}, \quad (3)$$

where  $\mathbf{w}$  is assigned a multivariate Gaussian prior distribution and marginalized. In this paper, we assume the index set to be  $\mathcal{X} = \mathbb{R}^{D \times 1}$  without loss of generality.

For a detailed review of the GP and the kernel functions, please refer to [45].

#### 2.1.1 Scalable Variational Inference for GP

Despite the flexibility of the GP prior, there are two major drawbacks that plague the model. First, if the likelihood function in Equation 2 is non-Gaussian, posterior inference cannot be computed analytically. Secondly, the computational complexity of the inference algorithm is  $O(N^3)$  where  $N$  is the number of training data points, rendering the model inapplicable to large data sets.

Fortunately, modern variational inference provides a solution to both problems by introducing a set of  $M$  inducing points  $\mathbf{Z} = [\mathbf{z}_1, \dots, \mathbf{z}_M]^\top$ , where  $\mathbf{z}_m \in \mathbb{R}^{D \times 1}$ . The inducing points, which are variational parameters, index a set of random variables  $\mathbf{u} = [f(\mathbf{z}_1), \dots, f(\mathbf{z}_M)]^\top$  that is a subset of the GP function  $f(\mathbf{x})$ . Through conditioning and assuming  $m(\mathbf{x})$  is zero, the conditional GP can be expressed as

$$f(\mathbf{x}) | \mathbf{u} \sim \mathcal{GP}(\mathbf{k}_{\mathbf{zx}}^\top \mathbf{K}_{\mathbf{zz}}^{-1} \mathbf{u}, k_\theta(\mathbf{x}, \mathbf{x}) - \mathbf{k}_{\mathbf{zx}}^\top \mathbf{K}_{\mathbf{zz}}^{-1} \mathbf{k}_{\mathbf{zx}}) \quad (4)$$

where  $\mathbf{k}_{\mathbf{zx}} = [k_\theta(\mathbf{z}_1, \mathbf{x}), \dots, k_\theta(\mathbf{z}_M, \mathbf{x})]$  and  $[\mathbf{K}_{\mathbf{zz}}]_{ij} = k_\theta(\mathbf{z}_i, \mathbf{z}_j)$ . Naturally,  $p(\mathbf{u}) = \mathcal{N}(\mathbf{0}, \mathbf{K}_{\mathbf{zz}})$ . The variational posterior distribution of  $\mathbf{u}$ ,  $q(\mathbf{u})$  is assumed to be a multivariate Gaussian distribution with mean  $\mathbf{m}$  and covariance matrix  $\mathbf{S}$ . Following the standard derivation of variational inference, the Evidence Lower Bound (ELBO) objective function is

$$\mathcal{L}(\theta, \mathbf{Z}, \mathbf{m}, \mathbf{S}) = \sum_{n=1}^N \mathbb{E}_{q(f(\mathbf{x}_n))} [\log p(y_n | f(\mathbf{x}_n))] - \text{KL}[q(\mathbf{u}) || p(\mathbf{u})]. \quad (5)$$

The variational distribution  $q(f(\mathbf{x}_n))$  can be easily derived from the conditional GP in Equation 4 and  $q(\mathbf{u})$ , and its expectation can be approximated effectively using 1-dimensional quadratures. We refer the readers to [30] for detailed derivations and results.

## 2.2 The Graph Laplacian

Given adjacency matrix  $\mathbf{A} \in \{0, 1\}^{N \times N}$  of an undirected binary graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  without self-loop, the corresponding graph Laplacian is defined as

$$\mathbf{L} = \mathbf{D} - \mathbf{A}, \quad (6)$$

where  $\mathbf{D}$  is the  $N \times N$  diagonal node degree matrix. The graph Laplacian can be viewed as an operator on the space of functions  $g : \mathcal{V} \rightarrow \mathbb{R}$  indexed by the graph's nodes such that

$$\mathbf{L}g(n) = \sum_{v \in Ne(n)} [g(n) - g(v)], \quad (7)$$

where  $Ne(n)$  is the set containing neighbours of node  $n$ . Intuitively, applying the Laplacian operator to the function  $g$  results in a function that quantifies the variability of  $g$  around the nodes in the graph.

The Laplacian's spectrum encodes the geometric properties of the graph that are useful in crafting graph filters and kernels [37, 43, 4, 9]. As the Laplacian matrix is real symmetric and diagonalizable, its eigen-decomposition exists. We denote the decomposition as

$$\mathbf{L} = \mathbf{U}\Lambda\mathbf{U}^T, \quad (8)$$

where the columns of  $\mathbf{U} \in \mathbb{R}^{N \times N}$  are the eigenfunctions of  $\mathbf{L}$  and the diagonal  $\Lambda \in \mathbb{R}^{N \times N}$  contains the corresponding eigenvalues. Therefore, the Laplacian operator can also be viewed as a filter on function  $g$  re-expressed using the eigenfunction basis. Regularization can be achieved by directly manipulating the eigenvalues of the system [39]. We refer the readers to [4, 37, 9] for comprehensive reviews of the graph Laplacian and its spectrum.

## 3 Graph Gaussian Processes

Given a data set of size  $N$  with  $D$ -dimensional features  $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N]^T$ , a symmetric binary adjacency matrix  $\mathbf{A} \in \{0, 1\}^{N \times N}$  that represents the relational graph of the data points and labels for a subset of the data points,  $\mathcal{Y}_O = [y_1, \dots, y_O]$ , with each  $y_i \in \{1, \dots, K\}$ , we seek to predict the unobserved labels of the remaining data points  $\mathcal{Y}_U = [y_{O+1}, \dots, y_N]$ . We denote the set of all labels as  $\mathbf{Y} = \mathcal{Y}_O \cup \mathcal{Y}_U$ .

The GGP specifies the conditional distribution  $p_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{A})$ , and predicts  $\mathcal{Y}_U$  via the predictive distribution  $p_\theta(\mathcal{Y}_U|\mathcal{Y}_O, \mathbf{X}, \mathbf{A})$ . The joint model is specified as the product of the conditionally independent likelihood  $p(y_n|h_n)$  and the GGP prior  $p_\theta(\mathbf{h}|\mathbf{X}, \mathbf{A})$  with hyper-parameters  $\theta$ . The latent likelihood parameter vector  $\mathbf{h} \in \mathbb{R}^{N \times 1}$  is defined in the next paragraph.

First, the model factorizes as

$$p_\theta(\mathbf{Y}, \mathbf{h}|\mathbf{X}, \mathbf{A}) = p_\theta(\mathbf{h}|\mathbf{X}, \mathbf{A}) \prod_{n=1}^N p(y_n|h_n), \quad (9)$$

where for the multi-class classification problem that we are interested in,  $p(y_n | h_n)$  is given by the robust-max likelihood [30, 16, 23, 21, 20].

Next, we construct the GGP prior from a Gaussian process distributed latent function  $f(\mathbf{x}) : \mathbb{R}^{D \times 1} \rightarrow \mathbb{R}$ ,  $f(\mathbf{x}) \sim \mathcal{GP}(\mathbf{0}, k_\theta(\mathbf{x}, \mathbf{x}'))$ , where the *key assumption* is that the likelihood parameter  $h_n$  for data point  $n$  is an average of the values of  $f$  over its 1-hop neighbourhood  $Ne(n)$  as given by  $\mathbf{A}$ :

$$h_n = \frac{f(\mathbf{x}_n) + \sum_{l \in Ne(n)} f(\mathbf{x}_l)}{1 + D_n} \quad (10)$$

where  $Ne(n) = \{l : l \in \{1, \dots, N\}, \mathbf{A}_{nl} = 1\}$ ,  $D_n = |Ne(n)|$ . We further motivate this key assumption in Section 3.1.

As  $f(\mathbf{x})$  has a zero mean function, the GGP prior can be succinctly expressed as a multivariate Gaussian random field

$$p_\theta(\mathbf{h}|\mathbf{X}, \mathbf{A}) = \mathcal{N}(\mathbf{0}, \mathbf{P}\mathbf{K}_{\mathbf{XX}}\mathbf{P}^T), \quad (11)$$

where  $\mathbf{P} = (\mathbf{I} + \mathbf{D})^{-1}(\mathbf{I} + \mathbf{A})$  and  $[\mathbf{K}_{\mathbf{XX}}]_{ij} = k_\theta(\mathbf{x}_i, \mathbf{x}_j)$ . A suitable kernel function  $k_\theta(\mathbf{x}_i, \mathbf{x}_j)$  for the task at hand can be chosen from the suite of well-studied existing kernels, such as those described in [13]. We refer to the chosen kernel function as the *base kernel* of the GGP. The  $\mathbf{P}$  matrix is sometimes known as the random-walk matrix in the literatures [9]. A graphical model representation of the proposed model is shown in Figure 1.

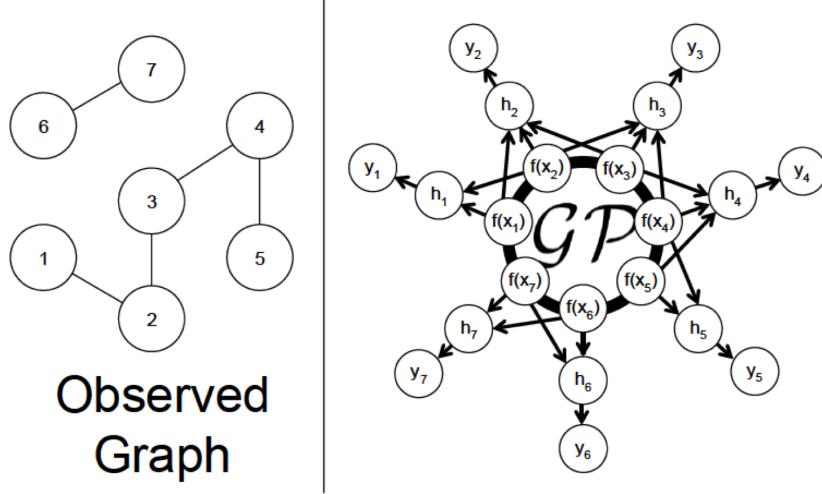


Figure 1: The figure depicts a relational graph (left) and the corresponding GGP represented as a graphical model (right). The thick circle represents a set of fully connected nodes.

The covariance structure specified in Equation 11 is equivalent to the pairwise covariance

$$\begin{aligned} Cov(h_m, h_n) &= \frac{1}{(1+D_m)(1+D_n)} \sum_{i \in \{m \cup Ne(m)\}} \sum_{j \in \{n \cup Ne(n)\}} k_\theta(\mathbf{x}_i, \mathbf{x}_j) \\ &= \langle \frac{1}{1+D_m} \sum_{i \in \{m \cup Ne(m)\}} \phi(\mathbf{x}_i), \frac{1}{1+D_n} \sum_{j \in \{n \cup Ne(n)\}} \phi(\mathbf{x}_j) \rangle_{\mathcal{H}} \end{aligned} \quad (12)$$

where  $\phi(\cdot)$  is the feature map that corresponds to the base kernel  $k_\theta(\cdot, \cdot)$ . Equation 12 can be viewed as the inner product between the empirical kernel mean embeddings that correspond to the bags of node features observed in the 1-hop neighborhood sub-graphs of node  $m$  and  $n$ , relating the proposed model to the Gaussian process distribution regression model presented in e.g. [15].

More specifically, we can view the GGP as a distribution classification model for the labelled bags of node features  $\{\{\mathbf{x}_i | i \in \{n \cup Ne(n)\}\}, y_n\}_{n=1}^O$ , such that the unobserved distribution  $P_n$  that generates  $\{\mathbf{x}_i | i \in \{n \cup Ne(n)\}\}$  is summarized by its empirical kernel mean embedding

$$\hat{\mu}_n = \frac{1}{1+D_n} \sum_{j \in \{n \cup Ne(n)\}} \phi(\mathbf{x}_j). \quad (13)$$

The prior on  $\mathbf{h}$  can equivalently be expressed as  $\mathbf{h} \sim \mathcal{GP}(0, \langle \hat{\mu}_m, \hat{\mu}_n \rangle_{\mathcal{H}})$ . For detailed reviews of the kernel mean embedding and distribution regression models, we refer the readers to [32] and [41] respectively.

One main assumption of the 1-hop neighbourhood averaging mechanism is homophily - i.e., nodes with similar covariates are more likely to form connections with each others [17]. The assumption allows us to approximately treat the node covariates from a 1-hop neighbourhood as samples drawn from the same data distribution, in order to model them using distribution regression. While it is perfectly reasonable to consider multi-hops neighbourhood averaging, the homophily assumption

starts to break down if we consider 2-hop neighbours which are not directly connected. Nevertheless, it is interesting to explore non-naive ways to account for multi-hop neighbours in the future, such as stacking 1-hop averaging graph GPs in a structure similar to that of the deep Gaussian processes [10, 34], or having multiple latent GPs for neighbours of different hops that are summed up in the likelihood functions.

### 3.1 An Alternative View of GGP

In this section, we present an alternative formulation of the GGP, which results in an intuitive interpretation of the model. The alternative formulation views the GGP as a Bayesian linear model on feature maps of the nodes that have been transformed by a function related to the graph Laplacian  $\mathbf{L}$ .

As we reviewed in Section 2.1, the kernel matrix  $\mathbf{K}_{\mathbf{X}\mathbf{X}}$  in Equation 11 can be written as the product of feature map matrix  $\Phi_{\mathbf{X}}\Phi_{\mathbf{X}}^T$  where row  $n$  of  $\Phi_{\mathbf{X}}$  corresponds to the feature maps of node  $n$ ,  $\phi(\mathbf{x}_n) = [\phi_{n1}, \dots, \phi_{nQ}]^T$ . Therefore, the covariance matrix in Equation 11,  $\mathbf{P}\Phi_{\mathbf{X}}\Phi_{\mathbf{X}}^T\mathbf{P}^T$ , can be viewed as the product of the transformed feature maps

$$\hat{\Phi}_{\mathbf{X}} = \mathbf{P}\Phi_{\mathbf{X}} = (\mathbf{I} + \mathbf{D})^{-1}\mathbf{D}\Phi_{\mathbf{X}} + (\mathbf{I} + \mathbf{D})^{-1}(\mathbf{I} - \mathbf{L})\Phi_{\mathbf{X}}. \quad (14)$$

where  $\mathbf{L}$  is the graph Laplacian matrix as defined in Equation 6. Isolating the transformed feature maps for node  $n$  (i.e., row  $n$  of  $\hat{\Phi}_{\mathbf{X}}$ ) gives

$$\hat{\phi}(\mathbf{x}_n) = \frac{D_n}{1 + D_n}\phi(\mathbf{x}_n) + \frac{1}{1 + D_n}[(\mathbf{I} - \mathbf{L})\Phi_{\mathbf{X}}]_n^T, \quad (15)$$

where  $D_n$  is the degree of node  $n$  and  $[\cdot]_n$  denotes row  $n$  of a matrix. The proposed GGP model is equivalent to a supervised Bayesian linear classification model with a feature pre-processing step that follows from the expression in Equation 15. For isolated nodes ( $D_n = 0$ ), the expression in Equation 15 leaves the node feature maps unchanged ( $\hat{\phi} = \phi$ ).

The  $(\mathbf{I} - \mathbf{L})$  term in Equation 15 can be viewed as a spectral filter  $\mathbf{U}(\mathbf{I} - \Lambda)\mathbf{U}^T$ , where  $\mathbf{U}$  and  $\Lambda$  are the eigenmatrix and eigenvalues of the Laplacian as defined in Section 2.2. For connected nodes, the expression results in new features that are weighted averages of the original features and features transformed by the spectral filter. The alternative formulation opens up opportunities to design other spectral filters with different regularization properties, such as those described in [39], that can replace the  $(\mathbf{I} - \mathbf{L})$  expression in Equation 15. We leave the exploration of this research direction to future work.

In addition, it is well-known that many graphs and networks observed in the real world follow the power-law node degree distributions [17], implying that there are a handful of nodes with very large degrees (known as hubs) and many with relatively small numbers of connections. The nodes with few connections (small  $D_n$ ) are likely to be connected to one of the handful of heavily connected nodes, and their transformed node feature maps are highly influenced by the features of the hub nodes. On the other hand, individual neighbours of the hub nodes have relatively small impact on the hub nodes because of the large number of neighbours that the hubs are connected to. This highlights the asymmetric outsize influence of hubs in the proposed GGP model, such that a mis-labelled hub node may result in a more significant drop in the model's accuracy compared to a mis-labelled node with much lower degree of connections.

### 3.2 Variational Inference with Inducing Points

Posterior inference for the GGP is analytically intractable because of the non-conjugate likelihood. We approximate the posterior of the GGP using a variational inference algorithm with inducing points similar to the inter-domain inference algorithm presented in [42]. Implementing the GGP with its variational inference algorithm amounts to implementing a new kernel function that follows Equation 12 in the GPflow Python package.<sup>1</sup>

We introduce a set of  $M$  inducing random variables  $\mathbf{u} = [f(\mathbf{z}_1), \dots, f(\mathbf{z}_M)]^T$  indexed by inducing points  $\{\mathbf{z}_m\}_{m=1}^M$  in the same domain as the GP function  $f(\mathbf{x}) \sim \mathcal{GP}(\mathbf{0}, k_\theta(\mathbf{x}, \mathbf{x}'))$ . As a result, the

---

<sup>1</sup><https://github.com/markvdw/GPflow-inter-domain>

inter-domain covariance between  $h_n$  and  $f(\mathbf{z}_m)$  is

$$Cov(h_n, f(\mathbf{z}_m)) = \frac{1}{D_n + 1} \left[ k_\theta(\mathbf{x}_n, \mathbf{z}_m) + \sum_{l \in Ne(n)} k_\theta(\mathbf{x}_l, \mathbf{z}_m) \right]. \quad (16)$$

Additionally, we introduce a multivariate Gaussian variational distribution  $q(\mathbf{u}) = \mathcal{N}(\mathbf{m}, \mathbf{S}\mathbf{S}^\top)$  for the inducing random variables with variational parameters  $\mathbf{m} \in \mathbb{R}^{M \times 1}$  and the lower triangular  $\mathbf{S} \in \mathbb{R}^{M \times M}$ . Through Gaussian conditioning,  $q(\mathbf{u})$  results in the variational Gaussian distribution  $q(\mathbf{h})$  that is of our interest. The variational parameters  $\mathbf{m}, \mathbf{S}, \{\mathbf{z}_m\}_{m=1}^M$  and the kernel hyperparameters  $\theta$  are then jointly fitted by maximizing the ELBO function in Equation 5.

### 3.3 Computational Complexity

The computational complexity of the inference algorithm is  $O(|\mathcal{Y}_o|M^2)$ . In the experiments, we chose  $M$  to be the number of labelled nodes in the graph  $|\mathcal{Y}_o|$ , which is small relative to the total number of nodes. Computing the covariance function in Equation 12 incurs a computational cost of  $O(D_{max}^2)$  per labelled node, where  $D_{max}$  is the maximum node degree. In practice, the computational cost of computing the covariance function is small because of the sparse property of graphs typically observed in the real-world [17].

## 4 Related Work

Graph-based learning problems have been studied extensively by researchers from both machine learning and signal processing communities, leading to many models and algorithms that are well-summarized in review papers [4, 35, 37].

Gaussian process-based models that operate on graphs have previously been developed in the closely related relational learning discipline, resulting in the mixed graph Gaussian process (XGP) [38] and relational Gaussian process (RGP) [8]. Additionally, the renowned Label Propagation (LP)[48] model can also be viewed as a GP with its covariance structure specified by the graph Laplacian matrix [49]. The GGP differs from the previously proposed GP models in that the local neighbourhood structures of the graph and the node features are directly used in the specification of the covariance function, resulting in a simple model that is highly effective.

Models based on Laplacian regularization that restrict the node labels to vary smoothly over graphs have also been proposed previously. The LP model can be viewed as an instance under this framework. Other Laplacian regularization based models include the deep semi-supervised embedding [44] and the manifold regularization [3] models. As shown in the experimental results in Table 1, the predictive performance of these models fall short of other more sophisticated models.

Additionally, models that extract embeddings of nodes and local sub-graphs which can be used for predictions have also been proposed by multiple authors. These models include DeepWalk [33], node2vec [19], planetoid [46] and many others. The proposed GGP is related to the embedding based models in that it can be viewed as a GP classifier that takes empirical kernel mean embeddings extracted from the 1-hop neighbourhood sub-graphs as inputs to predict node labels.

Finally, many geometric deep learning models that operate on graphs have been proposed and shown to be successful in graph-based semi-supervised learning problems. The earlier models including [26, 36, 18] are inspired by the recurrent neural networks. On the other hand, convolution neural networks that learn convolutional filters in the graph Laplacian spectral domain have been demonstrated to perform well. These models include the spectral CNN [5], DCNN [1], ChebNet [12] and GCN [25]. Neural networks that operate on the graph spectral domain are limited by the graph-specific Fourier basis. The more recently proposed MoNet [31] addressed the graph-specific limitation of spectral graph neural networks. The idea of filtering in graph spectral domain is a powerful one that has also been explored in the kernel literatures [39, 43]. We draw parallels between our proposed model and the spectral filtering approaches in Section 3.1, where we view the GGP as a standard GP classifier operating on feature maps that have been transformed through a filter that can be related to the graph spectral domain.

Our work has also been inspired by literatures in Gaussian processes that mix GPs via an additive function, such as [6, 14, 42].

## 5 Experiments

We present two sets of experiments to benchmark the predictive performance of the GGP against existing models under two different settings. In Section 5.1, we demonstrate that the GGP is a viable and extremely competitive alternative to the graph convolutional neural network (GCN) in settings where there are sufficient labelled data points. In Section 5.2, we test the models in an active learning experimental setup, and show that the GGP outperforms the baseline models when there are few training labels.

### 5.1 Semi-supervised Classification on Graphs

The semi-supervised classification experiments in this section exactly replicate the experimental setup in [25], where the GCN is known to perform well. The three benchmark data sets, as described in Table 2, are citation networks with bag-of-words (BOW) features, and the prediction targets are the topics of the scientific papers in the citation networks.

The experimental results are presented in Table 1, and show that the predictive performance of the proposed GGP is competitive with the GCN and MoNet [31] (another deep learning model), and superior to the other baseline models. While the GCN outperforms the proposed model by small margins on the test sets with 1,000 data points, it is important to note that the GCN had access to 500 additional labelled data points for early stopping. As the GGP does not require early stopping, the additional labelled data points can instead be directly used to train the model to significantly improve the predictive performance. To demonstrate this advantage, we report another set of results for a GGP trained using the 500 additional data points in Table 1, in the row labelled as ‘**GGP-X**’. The boost in the predictive performances shows that the GGP can better exploit the available labelled data to make predictions.

The GGP base kernel of choice is the 3rd degree polynomial kernel, which is known to work well with high-dimensional BOW features [45]. We re-weighted the BOW features using the popular term frequency-inverse document frequency (TFIDF) technique [40]. The variational parameters and the hyper-parameters were jointly optimized using the ADAM optimizer [24]. The baseline models that we compared to are the ones that were also presented and compared to in [25] and [31].

	<b>Cora</b>	<b>Citeseer</b>	<b>Pubmed</b>
GGP	80.9%	69.7%	77.1%
GGP-X	84.7%	75.6%	82.4%
GCN[25]	81.5%	70.3%	79.0%
DCNN[1]	76.8%	-	73.0%
MoNet[31]	81.7%	-	78.8%
DeepWalk[33]	67.2%	43.2%	65.3%
Planetoid[46]	75.7%	64.7%	77.2%
ICA[27]	75.1%	69.1%	73.9%
LP[48]	68.0%	45.3%	63.0%
SemiEmb[44]	59.0%	59.6%	71.1%
ManiReg[3]	59.5%	60.1%	70.7%

Table 1: This table shows the test classification accuracies of the semi-supervised learning experiments described in Section 5.1. The test sets consist of 1,000 data points. The GGP accuracies are averaged over 10 random restarts. The results for DCNN and MoNet are copied from [31] while the results for the other models are from [25]. Please refer to Section 5.1 for discussions of the results.

	<b>Type</b>	<b>N<sub>nodes</sub></b>	<b>N<sub>edges</sub></b>	<b>N<sub>label_cat.</sub></b>	<b>D<sub>features</sub></b>	<b>Label Rate</b>
<b>Cora</b>	Citation	2,708	5,429	7	1,433	0.052
<b>Citeseer</b>	Citation	3,327	4,732	6	3,703	0.036
<b>Pubmed</b>	Citation	19,717	44,338	3	500	0.003

Table 2: A summary of the benchmark data sets for the semi-supervised classification experiment.

## 5.2 Active Learning on Graphs

Active learning is a domain that faces the same challenges as semi-supervised learning where labels are scarce and expensive to obtain [47]. In active learning, a subset of unlabelled data points are selected sequentially to be queried according to an acquisition function, with the goal of maximizing the accuracy of the predictive model using significantly fewer labels than would be required if the labelled set were sampled uniformly at random [2]. A motivating example of this problem scenario is in the medical setting where the time of human experts is precious, and the machines must aim to make the best use of the time. Therefore, having a data efficient predictive model that can generalize well with few labels is of critical importance in addition to having a good acquisition function.

In this section, we leverage GGP as the semi-supervised classification model of active learner in graph-based active learning problem [47, 28, 11, 22, 29]. The GGP is paired with the proven  $\Sigma$ -optimal (SOPT) acquisition function to form an active learner [28]. The SOPT acquisition function is model agnostic in that it only requires the Laplacian matrix of the observed graph and the indices of the labelled nodes in order to identify the next node to query, such that the predictive accuracy of the active learner is maximally increased. The main goal of the active learning experiments is to demonstrate that the GGP can learn better than both the GCN and the Label Propagation model (LP) [48] with very few labelled data points.

Starting with only 1 randomly selected labelled data point (i.e., node), the active learner identifies the next data point to be labelled using the acquisition function. Once the label of the said data point is acquired, the classification model is retrained and its test accuracy is evaluated on the remaining unlabelled data points. In our experiments, the process is repeated until 50 labels are acquired. The experiments are also repeated with 10 different initial labelled data points. In addition to the SOPT acquisition function, we show the results of the same models paired with the random acquisition function (RAND) for comparisons.

The test accuracies with different numbers of labelled data points are presented as learning curves in Figure 2. In addition, we summarize the results numerically using the Area under the Learning Curve (ALC) metric in Table 3. The ALC is normalized to have a maximum value of 1, which corresponds to a hypothetical learner that can achieve 100% test accuracy with only 1 label. The results show that the proposed GGP model is indeed more data efficient than the baselines and can outperform both the GCN and the LP models when labelled data are scarce.

The benchmark data sets for the active learning experiments are the Cora and Citeseer data sets. However, due to technical restriction imposed by the SOPT acquisition function, only the largest connected sub-graph of the data set is used. The restriction reduces the number of nodes in the Cora and Citeseer data sets to 2,485 and 2,120 respectively. Both of the data sets were also used as benchmark data sets in [28].

We pre-process the BOW features with TFIDF and apply a linear kernel as the base kernel of the GGP. All parameters are jointly optimized using the ADAM optimizer. The GCN and LP models are trained using the settings recommended in [25] and [28] respectively.

	<b>Cora</b>	<b>Citeseer</b>
SOPT-GGP	$0.733 \pm 0.001$	$0.678 \pm 0.002$
SOPT-GCN	$0.706 \pm 0.001$	$0.675 \pm 0.002$
SOPT-LP	$0.672 \pm 0.001$	$0.638 \pm 0.001$
RAND-GGP	$0.575 \pm 0.007$	$0.557 \pm 0.008$
RAND-GCN	$0.584 \pm 0.011$	$0.533 \pm 0.008$
RAND-LP	$0.424 \pm 0.020$	$0.490 \pm 0.011$

Table 3: This table shows the Area under the Learning Curve (ALC) scores for the active learning experiments. ALC refers to the area under the learning curves shown in Figure 2 normalized to have a maximum value of 1. The ALCs are computed by averaging over 10 different initial data points. The results show that the GGP is able to generalize better with fewer labels compared to the baselines. ‘SOPT’ and ‘RAND’ refer to the acquisition functions used. Please refer to Section 5.2 for discussions of the results.

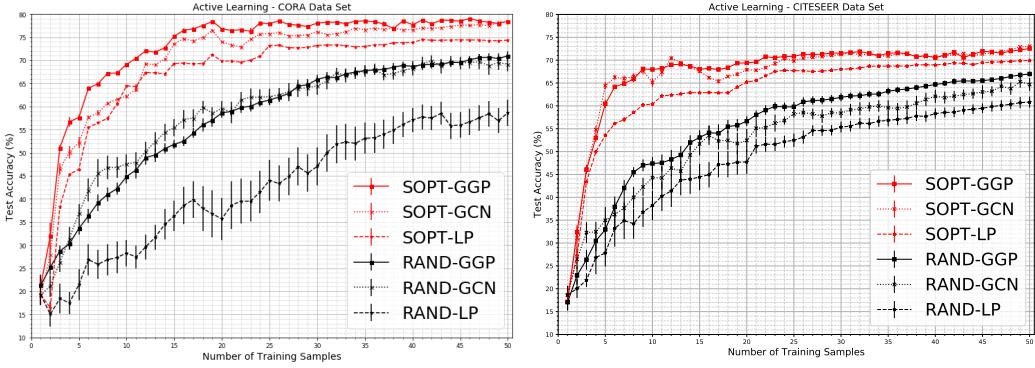


Figure 2: The sub-figures show the test accuracies from the active learning experiments (y-axis) for the Cora (left) and Citeseer (right) data sets with different number of labelled data points (x-axis). The results are averaged over 10 trials with different initial data points. SOPT and RAND refer to the acquisition functions described in Section 5.2. The smaller error bars of ‘RAND-GGP’ compared to those of ‘RAND-GCN’ demonstrate the relative robustness of the GGP models under random shuffling of data points in the training data set. The tiny error bars of the ‘SOPT-\*’ results show that the ‘SOPT’ acquisition function is insensitive to the randomly selected initial labelled data point. Please also refer to Table 3 for numerical summaries of the results.

## 6 Conclusion

We propose a Gaussian process model that is data-efficient for semi-supervised learning problems on graphs. In the experiments, we show that the proposed model is competitive with the state-of-the-art deep learning models, and outperforms when the number of labels is small. The proposed model is simple, effective and can leverage modern scalable variational inference algorithm for GP with minimal modification. In addition, the construction of our model is motivated by distribution regression using the empirical kernel mean embeddings, and can also be viewed under the framework of filtering in the graph spectrum. The spectral view offers a new potential research direction that can be explored in future work.

## Acknowledgements

This work was supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1.

## References

- [1] James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1993–2001, 2016.
- [2] Maria-Florina Balcan, Steve Hanneke, and Jennifer Wortman Vaughan. The true sample complexity of active learning. *Machine learning*, 80(2-3):111–139, 2010.
- [3] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of machine learning research*, 7(Nov):2399–2434, 2006.
- [4] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.
- [5] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.
- [6] M Yu Byron, John P Cunningham, Gopal Santhanam, Stephen I Ryu, Krishna V Shenoy, and Maneesh Sahani. Gaussian-process factor analysis for low-dimensional single-trial analysis of neural population activity. In *Advances in neural information processing systems*, pages 1881–1888, 2009.

- [7] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.
- [8] Wei Chu, Vikas Sindhwani, Zoubin Ghahramani, and S Sathiya Keerthi. Relational learning with gaussian processes. In *Advances in Neural Information Processing Systems*, pages 289–296, 2007.
- [9] Fan R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.
- [10] Andreas Damianou and Neil Lawrence. Deep gaussian processes. In *Artificial Intelligence and Statistics*, pages 207–215, 2013.
- [11] Gautam Dasarathy, Robert Nowak, and Xiaojin Zhu. S2: An efficient graph based active learning algorithm with application to nonparametric classification. In *Conference on Learning Theory*, pages 503–522, 2015.
- [12] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.
- [13] David Duvenaud. *Automatic model construction with Gaussian processes*. PhD thesis, University of Cambridge, 2014.
- [14] David K Duvenaud, Hannes Nickisch, and Carl E Rasmussen. Additive gaussian processes. In *Advances in neural information processing systems*, pages 226–234, 2011.
- [15] Seth R Flaxman, Yu-Xiang Wang, and Alexander J Smola. Who supported Obama in 2012?: Ecological inference through distribution regression. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 289–298. ACM, 2015.
- [16] Mark Girolami and Simon Rogers. Variational bayesian multinomial probit regression with gaussian process priors. *Neural Computation*, 18(8):1790–1817, 2006.
- [17] Anna Goldenberg, Alice X Zheng, Stephen E Fienberg, Edoardo M Airoldi, et al. A survey of statistical network models. *Foundations and Trends® in Machine Learning*, 2(2):129–233, 2010.
- [18] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 729–734. IEEE, 2005.
- [19] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.
- [20] James Hensman, Alexander G Matthews, Maurizio Filippone, and Zoubin Ghahramani. Mcmc for variationally sparse gaussian processes. In *Advances in Neural Information Processing Systems*, pages 1648–1656, 2015.
- [21] Daniel Hernández-Lobato, José M Hernández-Lobato, and Pierre Dupont. Robust multi-class gaussian process classification. In *Advances in neural information processing systems*, pages 280–288, 2011.
- [22] Kwang-Sung Jun and Robert Nowak. Graph-based active learning: A new look at expected error minimization. In *Signal and Information Processing (GlobalSIP), 2016 IEEE Global Conference on*, pages 1325–1329. IEEE, 2016.
- [23] Hyun-Chul Kim and Zoubin Ghahramani. Bayesian gaussian process classification with the em-ep algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1948–1959, 2006.
- [24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [25] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [26] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

- [27] Qing Lu and Lise Getoor. Link-based classification. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 496–503, 2003.
- [28] Yifei Ma, Roman Garnett, and Jeff Schneider.  $\sigma$ -optimality for active learning on gaussian random fields. In *Advances in Neural Information Processing Systems*, pages 2751–2759, 2013.
- [29] Oisin Mac Aodha, Neill D.F. Campbell, Jan Kautz, and Gabriel J. Brostow. Hierarchical Subquery Evaluation for Active Learning on a Graph. In *CVPR*, 2014.
- [30] A Matthews. *Scalable Gaussian process inference using variational methods*. PhD thesis, PhD thesis. University of Cambridge, 2016.
- [31] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodolà, Jan Svoboda, and Michael M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5425–5434. IEEE Computer Society, 2017.
- [32] Krikamol Muandet, Kenji Fukumizu, Bharath Sriperumbudur, Bernhard Schölkopf, et al. Kernel mean embedding of distributions: A review and beyond. *Foundations and Trends® in Machine Learning*, 10(1-2):1–141, 2017.
- [33] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.
- [34] Hugh Salimbeni and Marc Deisenroth. Doubly stochastic variational inference for deep gaussian processes. In *Advances in Neural Information Processing Systems*, pages 4588–4599, 2017.
- [35] Aliaksei Sandryhaila and Jose MF Moura. Big data analysis with signal processing on graphs: Representation and processing of massive data sets with irregular structure. *IEEE Signal Processing Magazine*, 31(5):80–90, 2014.
- [36] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.
- [37] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.
- [38] Ricardo Silva, Wei Chu, and Zoubin Ghahramani. Hidden common cause relations in relational learning. In *Advances in neural information processing systems*, pages 1345–1352, 2008.
- [39] Alexander J Smola and Risi Kondor. Kernels and regularization on graphs. In *Learning theory and kernel machines*, pages 144–158. Springer, 2003.
- [40] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.
- [41] Zoltán Szabó, Bharath K Sriperumbudur, Barnabás Póczos, and Arthur Gretton. Learning theory for distribution regression. *The Journal of Machine Learning Research*, 17(1):5272–5311, 2016.
- [42] Mark van der Wilk, Carl Edward Rasmussen, and James Hensman. Convolutional gaussian processes. In *Advances in Neural Information Processing Systems*, pages 2845–2854, 2017.
- [43] S Vichay N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.
- [44] Jason Weston, Frédéric Ratle, Hossein Mobahi, and Ronan Collobert. Deep learning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pages 639–655. Springer, 2012.
- [45] Christopher KI Williams and Carl Edward Rasmussen. Gaussian processes for machine learning. *the MIT Press*, 2(3):4, 2006.
- [46] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML’16, pages 40–48. JMLR.org, 2016.
- [47] Xiaojin Zhu. Semi-supervised learning literature survey. Technical Report 1530, Computer Sciences, University of Wisconsin-Madison, 2005.

- [48] Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pages 912–919, 2003.
- [49] Xiaojin Zhu, John D Lafferty, and Zoubin Ghahramani. Semi-supervised learning: From gaussian fields to gaussian processes. Technical report, Carnegie Mellon University, Computer Science Department, 2003.