Scalable SQL and NoSQL Data Stores

Rick Cattell

Originally published in 2010, last
revised December 2011

ABSTRACT

In this paper, we examine a number of SQL and so-called "NoSQL" data stores designed to scale simple OLTP-style application loads over many servers. Originally motivated by Web 2.0 applications, these systems are designed to scale to thousands or millions of users doing updates as well as reads, in contrast to traditional DBMSs and data warehouses. We contrast the new systems on their data model, consistency mechanisms, storage mechanisms, durability guarantees, availability, query support, and other dimensions. These systems typically sacrifice some of these dimensions, e.g. database-wide transaction consistency, in order to achieve others, e.g. higher availability and scalability.

Note: Bibliographic references for systems are not listed, but URLs for more information can be found in the System References table at the end of this paper.

Caveat: Statements in this paper are based on sources and documentation that may not be reliable, and the systems described are "moving targets," so some statements may be incorrect. Verify through other sources before depending on information here. Nevertheless, we hope this comprehensive survey is useful! Check for future corrections on the author's web site cattell.net/datastores.

Disclosure: The author is on the technical advisory board of Schooner Technologies and has a consulting business advising on scalable databases.

1. OVERVIEW

In recent years a number of new systems have been designed to provide good horizontal scalability for simple read/write database operations distributed over many servers. In contrast, traditional database products have comparatively little or no ability to scale horizontally on these applications. This paper examines and compares the various new systems.

Many of the new systems are referred to as "NoSQL" data stores. The definition of NoSQL, which stands for "Not Only SQL" or "Not Relational", is not entirely agreed upon. For the purposes of this paper, NoSQL systems generally have six key features:

1. the ability to horizontally scale "simple operation" throughput over many servers,
2. the ability to replicate and to distribute (partition) data over many servers,
3. a simple call level interface or protocol (in contrast to a SQL binding),
4. a weaker concurrency model than the ACID transactions of most relational (SQL) database systems,
5. efficient use of distributed indexes and RAM for data storage, and
6. the ability to dynamically add new attributes to data records.

The systems differ in other ways, and in this paper we contrast those differences. They range in functionality from the simplest distributed hashing, as supported by the popular memcached open source cache, to highly scalable partitioned tables, as supported by Google's BigTable [1]. In fact, BigTable, memcached, and Amazon's Dynamo [2] provided a "proof of concept" that inspired many of the data stores we describe here:

- Memcached demonstrated that in-memory indexes can be highly scalable, distributing and replicating objects over multiple nodes.
- Dynamo pioneered the idea of *eventual consistency* as a way to achieve higher availability and scalability: data fetched are not guaranteed to be up-to-date, but updates are guaranteed to be propagated to all nodes eventually.
- BigTable demonstrated that persistent record storage could be scaled to thousands of nodes, a feat that most of the other systems aspire to.

A key feature of NoSQL systems is "shared nothing" horizontal scaling – replicating and partitioning data over many servers. This allows them to support a large number of simple read/write operations per second. This simple operation load is traditionally called OLTP (online transaction processing), but it is also common in modern web applications

The NoSQL systems described here generally do not provide ACID transactional properties: updates are eventually propagated, but there are limited guarantees on the consistency of reads. Some authors suggest a "BASE" acronym in contrast to the "ACID" acronym:

- BASE = Basically Available, Soft state, Eventually consistent
- ACID = Atomicity, Consistency, Isolation, and Durability

The idea is that by giving up ACID constraints, one can achieve much higher performance and scalability.

However, the systems differ in how much they give up. For example, most of the systems call themselves "eventually consistent", meaning that updates are eventually propagated to all nodes, but many of them provide mechanisms for some degree of consistency, such as multi-version concurrency control (MVCC).

Proponents of NoSQL often cite Eric Brewer's CAP theorem [4], which states that a system can have only two out of three of the following properties: consistency, availability, and partition-tolerance. The NoSQL systems generally give up consistency. However, the trade-offs are complex, as we will see.

New relational DBMSs have also been introduced to provide better horizontal scaling for OLTP, when compared to traditional RDBMSs. After examining the NoSQL systems, we will look at these SQL systems and compare the strengths of the approaches. The SQL systems strive to provide horizontal scalability without abandoning SQL and ACID transactions. We will discuss the trade-offs here.

In this paper, we will refer to both the new SQL and NoSQL systems as *data stores*, since the term "database system" is widely used to refer to traditional DBMSs. However, we will still use the term "database" to refer to the stored data in these systems. All of the data stores have some administrative unit that you would call a database: data may be stored in one file, or in a directory, or via some other mechanism that defines the scope of data used by a group of applications. Each database is an island unto itself, even if the database is partitioned and distributed over multiple machines: there is no "federated database" concept in these systems (as with some relational and object-oriented databases), allowing multiple separately-administered databases to appear as one. Most of the systems allow horizontal partitioning of data, storing records on different servers according to some key; this is called "sharding". Some of the systems also allow vertical partitioning, where parts of a single record are stored on different servers.

1.1 Scope of this Paper

Before proceeding, some clarification is needed in defining "horizontal scalability" and "simple operations". These define the focus of this paper.

By "simple operations", we refer to key lookups, reads and writes of one record or a small number of records. This is in contrast to complex queries or joins, read-mostly access, or other application loads. With the advent of the web, especially Web 2.0 sites where millions of users may both read and write data, scalability for simple database operations has become more important. For example, applications may search and update multi-server databases of electronic mail, personal profiles, web postings, wikis, customer records, online dating records, classified ads, and many other kinds of data. These all generally fit the definition of "simple operation" applications: reading or writing a small number of related records in each operation.

The term "horizontal scalability" means the ability to distribute both the data and the load of these simple operations over many servers, with no RAM or disk shared among the servers. Horizontal scaling differs from "vertical" scaling, where a database system utilizes many cores and/or CPUs that share RAM and disks. Some of the systems we describe provide both vertical and horizontal scalability, and the effective use of multiple cores is important, but our main focus is on horizontal scalability, because the number of cores that can share memory is limited, and horizontal scaling generally proves less expensive, using commodity servers. Note that horizontal and vertical *partitioning* are not related to horizontal and vertical *scaling*, except that they are both useful for horizontal scaling.

1.2 Systems Beyond our Scope

Some authors have used a broad definition of NoSQL, including any database system that is not relational. Specifically, they include:

- *Graph database systems*: Neo4j and OrientDB provide efficient distributed storage and queries of a graph of nodes with references among them.
- *Object-oriented database systems*: Object-oriented DBMSs (e.g., Versant) also provide efficient distributed storage of a graph of objects, and materialize these objects as programming language objects.
- *Distributed object-oriented stores:* Very similar to object-oriented DBMSs, systems such as GemFire distribute object graphs in-memory on multiple servers.

These systems are a good choice for applications that must do fast and extensive reference-following, especially where data fits in memory. Programming language integration is also valuable. Unlike the NoSQL systems, these systems generally provide ACID transactions. Many of them provide horizontal scaling for reference-following and distributed query decomposition, as well. Due to space limitations, however, we have omitted these systems from our comparisons. The applications and the necessary optimizations for scaling for these systems differ from the systems we cover here, where key lookups and simple operations predominate over reference-following and complex object behavior. It is possible these systems can scale on simple operations as well, but that is a topic for a future paper, and proof through benchmarks.

Data warehousing database systems provide horizontal scaling, but are also beyond the scope of this paper. Data warehousing applications are different in important ways:

- They perform complex queries that collect and join information from many different tables.
- The ratio of reads to writes is high: that is, the database is read-only or read-mostly.

There are existing systems for data warehousing that scale well horizontally. Because the data is infrequently updated, it is possible to organize or replicate the database in ways that make scaling possible.

1.3 Data Model Terminology

Unlike relational (SQL) DBMSs, the terminology used by NoSQL data stores is often inconsistent. For the purposes of this paper, we need a consistent way to compare the data models and functionality.

All of the systems described here provide a way to store scalar values, like numbers and strings, as well as BLOBs. Some of them also provide a way to store more complex nested or reference values. The systems all store sets of attribute-value pairs, but use different data structures, specifically:

- A "tuple" is a row in a relational table, where attribute names are pre-defined in a schema, and the values must be scalar. The values are referenced by attribute name, as opposed to an array or list, where they are referenced by ordinal position.
- A "document" allows values to be nested documents or lists as well as scalar values, and the attribute names are dynamically defined for each document at runtime. A document differs from a tuple in that the attributes are not defined in a global schema, and this wider range of values are permitted.
- An "extensible record" is a hybrid between a tuple and a document, where families of attributes are defined in a schema, but new attributes can be added (within an attribute family) on a per-record basis. Attributes may be list-valued.
- An "object" is analogous to an object in programming languages, but without the procedural methods. Values may be references or nested objects.

1.4 Data Store Categories

In this paper, the data stores are grouped according to their data model:

- *Key-value Stores*: These systems store values and an index to find them, based on a programmer-defined key.
- *Document Stores*: These systems store documents, as just defined. The documents are indexed and a simple query mechanism is provided.
- *Extensible Record Stores*: These systems store extensible records that can be partitioned vertically and horizontally across nodes. Some papers call these "wide column stores".
- *Relational Databases*: These systems store (and index and query) tuples. The new RDBMSs that provide horizontal scaling are covered in this paper.

Data stores in these four categories are covered in the next four sections, respectively. We will then summarize and compare the systems.

2. KEY-VALUE STORES

The simplest data stores use a data model similar to the popular memcached distributed in-memory cache, with a single key-value index for all the data. We'll call these systems *key-value stores*. Unlike memcached, these systems generally provide a persistence mechanism and additional functionality as well: replication, versioning, locking, transactions, sorting, and/or other features. The client interface provides inserts, deletes, and index lookups. Like memcached, none of these systems offer secondary indices or keys.

2.1 Project Voldemort

Project Voldemort is an advanced key-value store, written in Java. It is open source, with substantial contributions from LinkedIn. Voldemort provides multi-version concurrency control (MVCC) for updates. It updates replicas asynchronously, so it does not guarantee consistent data. However, it can guarantee an up-to-date view if you read a majority of replicas.

Voldemort supports optimistic locking for consistent multi-record updates: if updates conflict with any other process, they can be backed out. Vector clocks, as used in Dynamo [3], provide an ordering on versions. You can also specify which version you want to update, for the put and delete operations.

Voldemort supports automatic sharding of data. *Consistent hashing* is used to distribute data around a ring of nodes: data hashed to node K is replicated on node K+1 … K+n where n is the desired number of extra copies (often n=1). Using good sharding technique, there should be many more "virtual" nodes than physical nodes (servers). Once data partitioning is set up, its operation is transparent. Nodes can be added or removed from a database cluster, and the system adapts automatically. Voldemort automatically detects and recovers failed nodes.

Voldemort can store data in RAM, but it also permits plugging in a storage engine. In particular, it supports a Berkeley DB and Random Access File storage engine. Voldemort supports lists and records in addition to simple scalar values.

2.2 Riak

Riak is written in Erlang. It was open-sourced by Basho in mid-2009. Basho alternately describes Riak as a "key-value store" and "document store". We will categorize it as an advanced key-value store here, because it lacks important features of document stores, but it (and Voldemort) have more functionality than the other key-value stores:

- Riak objects can be fetched and stored in JSON format, and thus can have multiple fields (like documents), and objects can be grouped into buckets, like the collections supported by document stores, with allowed/required fields defined on a per-bucket basis.
- Riak does not support indices on any fields except the primary key. The only thing you can do with the non-primary fields is fetch and store them as part of a JSON object. Riak lacks the query mechanisms of the document stores; the only lookup you can do is on primary key.

Riak supports replication of objects and sharding by hashing on the primary key. It allows replica values to be temporarily inconsistent. Consistency is tunable by specifying how many replicas (on different nodes) must respond for a successful read and how many must respond for a successful write. This is per-read and per-write, so different parts of an application can choose different trade-offs.

Like Voldemort, Riak uses a derivative of MVCC where vector clocks are assigned when values are updated. Vector clocks can be used to determine when objects are direct descendents of each other or a common parent, so Riak can often self-repair data that it discovers to be out of sync.

The Riak architecture is symmetric and simple. Like Voldemort, it uses consistent hashing. There is no distinguished node to track status of the system: the nodes use a gossip protocol to track who is alive and who has which data, and any node may service a client request. Riak also includes a map/reduce mechanism to split work over all the nodes in a cluster.

The client interface to Riak is based on RESTful HTTP requests. REST (REpresentational State Transfer) uses uniform, stateless, cacheable, client-server calls. There is also a programmatic interface for Erlang, Java, and other languages.

The storage part of Riak is "pluggable": the key-value pairs may be in memory, in ETS tables, in DETS tables, or in Osmos tables. ETS, DETS, and Osmos tables are all implemented in Erlang, with different performance and properties.

One unique feature of Riak is that it can store "links" between objects (documents), for example to link objects for authors to the objects for the books they wrote. Links reduce the need for secondary indices, but there is still no way to do range queries.

Here's an example of a Riak object described in JSON:

```
{
 "bucket":"customers",
 "key":"12345",
 "object":{
          "name":"Mr. Smith",
          "phone":"415-555-6524"  }
 "links":[
         ["sales","Mr. Salesguy","salesrep"],
         ["cust-orders","12345","orders"] ]
 "vclock":"opaque-riak-vclock",
 "lastmod":"Mon, 03 Aug 2009 18:49:42 GMT"
}
```

Note that the primary key is distinguished, while other fields are part of an "object" portion. Also note that the bucket, vector clock, and modification date is specified as part of the object, and links to other objects are supported.

2.3 Redis

The Redis key-value data store started as a one-person project but now has multiple contributors as BSD-licensed open source. It is written in C.

A Redis server is accessed by a wire protocol implemented in various client libraries (which must be updated when the protocol changes). The client side does the distributed hashing over servers. The servers store data in RAM, but data can be copied to disk for backup or system shutdown. System shutdown may be needed to add more nodes.

Like the other key-value stores, Redis implements insert, delete and lookup operations. Like Voldemort, it allows lists and sets to be associated with a key, not just a blob or string. It also includes list and set operations.

Redis does atomic updates by locking, and does asynchronous replication. It is reported to support about 100K gets/sets per second on an 8-core server.

2.4 Scalaris

Scalaris is functionally similar to Redis. It was written in Erlang at the Zuse Institute in Berlin, and is open source. In distributing data over nodes, it allows key ranges to be assigned to nodes, rather than simply hashing to nodes. This means that a query on a range of values does not need to go to every node, and it also may allow better load balancing, depending on key distribution.

Like the other key-value stores, it supports insert, delete, and lookup. It does replication synchronously (copies must be updated before the operation is complete) so data is guaranteed to be consistent. Scalaris also supports transactions with ACID properties on multiple objects. Data is stored in memory, but replication and recovery from node failures provides durability of the updates. Nevertheless, a multi-node power failure would cause disastrous loss of data, and the virtual memory limit sets a maximum database size.

Scalaris reads and writes must go to a majority of the replicas before an operation completes. Scalaris uses a ring of nodes, an unusual distribution and replication strategy that requires log(N) hops to read/write a key-value pair.

2.5 Tokyo Cabinet

Tokyo Cabinet / Tokyo Tyrant was a sourcefourge.net project, but is now licensed and maintained by FAL Labs. Tokyo Cabinet is the back-end server, Tokyo Tyrant is a client library for remote access. Both are written in C.

There are six different variations for the Tokyo Cabinet server: hash indexes in memory or on disk, B-trees in memory or on disk, fixed-size record tables, and variable-length record tables. The engines obviously differ in their performance characteristics, e.g. the fixed-length records allow quick lookups. There are slight variations on the API supported by these engines, but they all support common get/set/update operations. The documentation is a bit unclear, but they claim to support locking, ACID transactions, a binary array data type, and more complex update operations to atomically update a number or concatenate to a string. They support asynchronous replication with dual master or master/slave. Recovery of a failed node is manual, and there is no automatic sharding.

2.6 Memcached, Membrain, and Membase

The memcached open-source distributed in-memory indexing system has been enhanced by Schooner Tehnologies and Membase, to include features analogous to the other key-value stores: persistence, replication, high availability, dynamic growth, backup, and so on. Without persistence or replication, memcached does not really qualify as a "data store". However, Membrain and Membase certainly do, and these systems are also compatible with existing memcached applications. This compatibility is an attractive feature, given that memcached is widely used; memcached users that require more advanced features can easily upgrade to Membase and Membrain.

The Membase system is open source, and is supported by the company Membase. Its most attractive feature is probably its ability to elastically add or remove servers in a running system, moving data and dynamically redirecting requests in the meantime. The elasticity in most of the other systems is not as convenient.

Membrain is licensed per server, and is supported by Schooner Technologies. Its most attractive feature is probably its excellent tuning for flash memory. The performance gains of flash memory will not be gained in other systems by treating flash as a faster hard disk; it is important that the system treat flash as a true "third tier", different from RAM and disk. For example, many systems have substantial overhead in buffering and caching hard disk pages; this is unnecessary overhead with flash. The benchmark results on Schooner's web site show many *times* better performance than a number of competitors, particularly when data overflows RAM.

2.7 Summary

All the key-value stores support insert, delete, and lookup operations. All of these systems provide scalability through key distribution over nodes.

Voldemort, Riak, Tokyo Cabinet, and enhanced memcached systems can store data in RAM or on disk, with storage add-ons. The others store data in RAM, and provide disk as backup, or rely on replication and recovery so that a backup is not needed.

Scalaris and enhanced memcached systems use synchronous replication, the rest use asynchronous.

Scalaris and Tokyo Cabinet implement transactions, while the others do not.

Voldemort and Riak use multi-version concurrency control (MVCC), the others use locks.

Membrain and Membase are built on the popular memcached system, adding persistence, replication, and other features. Backward compatibility with memcached give these products an advantage.

3. DOCUMENT STORES

As discussed in the first section, document stores support more complex data than the key-value stores. The term "document store" may be confusing: while these systems could store "documents" in the traditional sense (articles, Microsoft Word files, etc.), a document in these systems can be any kind of "pointerless object", consistent with our definition in Section 1. Unlike the key-value stores, these systems generally support secondary indexes and multiple types of documents (objects) per database, and nested documents or lists. Like other NoSQL systems, the

document stores do not provide ACID transactional properties.

3.1 SimpleDB

SimpleDB is part of Amazon's proprietary cloud computing offering, along with their Elastic Compute Cloud (EC2) and their Simple Storage Service (S3) on which SimpleDB is based. SimpleDB has been around since 2007. As the name suggests, its model is simple: SimpleDB has Select, Delete, GetAttributes, and PutAttributes operations on documents. SimpleDB is simpler than other document stores, as it does not allow nested documents.

Like most of the systems we discuss, SimpleDB supports eventual consistency, not transactional consistency. Like most of the other systems, it does asynchronous replication.

Unlike key-value datastores, and like the other document stores, SimpleDB supports more than one grouping in one database: documents are put into domains, which support multiple indexes. You can enumerate domains and their metadata. Select operations are on one domain, and specify a conjunction of constraints on attributes, basically in the form:

select <attributes> from <domain> where
<list of attribute value constraints>

Different domains may be stored on different Amazon nodes.

Domain indexes are automatically updated when any document's attributes are modified. It is unclear from the documentation whether SimpleDB automatically selects which attributes to index, or if it indexes everything. In either case, the user has no choice, and the use of the indexes is automatic in SimpleDB query processing.

SimpleDB does not automatically partition data over servers. Some horizontal scaling can be achieve by reading any of the replicas, if you don't care about having the latest version. Writes do not scale, however, because they must go asynchronously to all copies of a domain. If customers want better scaling, they must do so manually by sharding themselves.

SimpleDB is a "pay as you go" proprietary solution from Amazon. There are currently built-in constraints, some of which are quite limiting: a 10 GB maximum domain size, a limit of 100 active domains, a 5 second limit on queries, and so on. Amazon doesn't license SimpleDB source or binary code to run on your own servers. SimpleDB does have the advantage of Amazon support and documentation.

3.2 CouchDB

CouchDB has been an Apache project since early 2008. It is written in Erlang.

A CouchDB "collection" of documents is similar to a SimpleDB domain, but the CouchDB data model is richer. Collections comprise the only schema in CouchDB, and secondary indexes must be explicitly created on fields in collections. A document has field values that can be scalar (text, numeric, or boolean) or compound (a document or list).

Queries are done with what CouchDB calls "views", which are defined with Javascript to specify field constraints. The indexes are B-trees, so the results of queries can be ordered or value ranges. Queries can be distributed in parallel over multiple nodes using a map-reduce mechanism. However, CouchDB's view mechanism puts more burden on programmers than a declarative query language.

Like SimpleDB, CouchDB achieves scalability through asynchronous replication, not through sharding. Reads can go to any server, if you don't care about having the latest values, and updates must be propagated to all the servers. However, a new project called CouchDB Lounge has been built to provide sharding on top of CouchDB, see:

http://code.google.com/p/couchdb-lounge/

Like SimpleDB, CouchDB does not guarantee consistency. Unlike SimpleDB, each client does see a self-consistent view of the database, with repeatable reads: CouchDB implements multi-version concurrency control on individual documents, with a Sequence ID that is automatically created for each version of a document. CouchDB will notify an application if someone else has updated the document since it was fetched. The application can then try to combine the updates, or can just retry its update and overwrite.

CouchDB also provides durability on system crash. All updates (documents and indexes) are flushed to disk on commit, by writing to the end of a file. (This means that periodic compaction is needed.) By default, it flushes to disk after every document update. Together with the MVCC mechanism, CouchDB's durability thus provides ACID semantics at the document level.

Clients call CouchDB through a RESTful interface. There are libraries for various languages (Java, C, PHP, Python, LISP, etc) that convert native API calls into the RESTful calls for you. CouchDB has some basic database adminstration functionality as well.

3.3 MongoDB

MongoDB is a GPL open source document store written in C++ and supported by 10gen. It has some similarities to CouchDB: it provides indexes on collections, it is lockless, and it provides a document query mechanism. However, there are important differences:

- MongoDB supports automatic sharding, distributing documents over servers.
- Replication in MongoDB is mostly used for failover, not for (dirty read) scalability as in CouchDB. MongoDB does not provide the global consistency of a traditional DBMS, but you can get local consistency on the up-to-date primary copy of a document.
- MongoDB supports dynamic queries with automatic use of indices, like RDBMSs. In CouchDB, data is indexed and searched by writing map-reduce views.
- CouchDB provides MVCC on documents, while MongoDB provides atomic operations on fields.

Atomic operations on fields are provided as follows:

- The update command supports "modifiers" that facilitate atomic changes to individual values: $set sets a value, $inc increments a value, $push appends a value to an array, $pushAll appends several values to an array, $pull removes a value from an array, and $pullAll removes several values from an array. Since these updates normally occur "in place", they avoid the overhead of a return trip to the server.
- There is an "update if current" convention for changing a document only if field values match a given previous value.
- MongoDB supports a findAndModify command to perform an atomic update and immediately return the updated document. This is useful for implementing queues and other data structures requiring atomicity.

MongoDB indices are explicitly defined using an ensureIndex call, and any existing indices are automatically used for query processing. To find all products released last year costing under $100 you could write:

```
db.products.find(
  {released: {$gte: new Date(2009, 1, 1,)},
  price {'$lte': 100},})
```

If indexes are defined on the queried fields, MongoDB will automatically use them. MongoDB also supports map-reduce, which allows for complex aggregations across documents.

MongoDB stores data in a binary JSON-like format called BSON. BSON supports boolean, integer, float, date, string and binary types. Client drivers encode the local language's document data structure (usually a dictionary or associative array) into BSON and send it over a socket connection to the MongoDB server (in contrast to CouchDB, which sends JSON as text over an HTTP REST interface). MongoDB also supports a GridFS specification for large binary objects, eg. images and videos. These are stored in chunks that can be streamed back to the client for efficient delivery.

MongoDB supports master-slave replication with automatic failover and recovery. Replication (and recovery) is done at the level of shards. Collections are automatically sharded via a user-defined shard key. Replication is asynchronous for higher performance, so some updates may be lost on a crash.

3.4 Terrastore

Another recent document store is Terrastore, which is built on the Terracotta distributed Java VM clustering product. Like many of the other NoSQL systems, client access to Terrastore is built on HTTP operations to fetch and store data. Java and Python client APIs have also been implemented.

Terrastore automatically partitions data over server nodes, and can automatically redistribute data when servers are added or removed. Like MongoDB, it can perform queries based on a predicate, including range queries, and like CouchDB, it includes a map/reduce mechanism for more advanced selection and aggregation of data.

Like the other document databases, Terrastore is schema-less, and does not provide ACID transactions. Like MongoDB, it provides consistency on a per-document basis: a read will always fetch the latest version of a document.

Terrastore supports replication and failover to a hot standby.

3.5 Summary

The document stores are schema-less, except for attributes (which are simply a name, and are not pre-specified), collections (which are simply a grouping of documents), and the indexes defined on collections (explicitly defined, except with SimpleDB). There are some differences in their data models, e.g. SimpleDB does not allow nested documents.

The document stores are very similar but use different terminology. For example, a SimpleDB Domain = CouchDB Database = MongoDB Collection = Terrastore Bucket. SimpleDB calls documents "items", and an attribute is a field in CouchDB, or a key in MongoDB or Terrastore.

Unlike the key-value stores, the document stores provide a mechanism to query collections based on multiple attribute value constraints. However, CouchDB does not support a non-procedural query language: it puts more work on the programmer and requires explicit utilization of indices.

The document stores generally do not provide explicit locks, and have weaker concurrency and atomicity properties than traditional ACID-compliant databases.

They differ in how much concurrency control they do provide.

Documents can be distributed over nodes in all of the systems, but scalability differs. All of the systems can achieve scalability by reading (potentially) out-of-date replicas. MongoDB and Terrastore can obtain scalability without that compromise, and can scale writes as well, through automatic sharding and atomic operations on documents. CouchDB might be able to achieve this write-scalability with the help of the new CouchDB Lounge code.

A last-minute addendum as this paper goes to press: the CouchDB and and Membase companies have now merged, to form Couchbase. They plan to provide a "best of both" merge of their products, e.g. with CouchDB's richer data model as well as the speed and elastic scalability of Membase. See Couchbase.com for more information.

4. EXTENSIBLE RECORD STORES

The extensible record stores seem to have been motivated by Google's success with BigTable. Their basic data model is rows and columns, and their basic scalability model is splitting both rows and columns over multiple nodes:

- Rows are split across nodes through sharding on the primary key. They typically split by range rather than a hash function. This means that queries on ranges of values do not have to go to every node.
- Columns of a table are distributed over multiple nodes by using "column groups". These may seem like a new complexity, but column groups are simply a way for the customer to indicate which columns are best stored together.

As noted earlier, these two partitionings (horizontal and vertical) can be used simultaneously on the same table. For example, if a customer table is partitioned into three column groups (say, separating the customer name/address from financial and login information), then each of the three column groups is treated as a separate table for the purposes of sharding the rows by customer ID: the column groups for one customer may or may not be on the same server.

The column groups must be pre-defined with the extensible record stores. However, that is not a big constraint, as new attributes can be defined at any time. Rows are analogous to documents: they can have a variable number of attributes (fields), the attribute names must be unique, rows are grouped into collections (tables), and an individual row's attributes can be of any type. (However, note that CouchDB and MongoDB support nested objects, while the extensible record stores generally support only scalar types.)

Although most extensible record stores were patterned after BigTable, it appears that none of the extensible records stores come anywhere near to BigTable's scalability at present. BigTable is used for many purposes (think of the many services Google provides, not just web search). It is worthwhile reading the BigTable paper [1] for background on the challenges with scaling.

4.1 HBase

HBase is an Apache project written in Java. It is patterned directly after BigTable:

- HBase uses the Hadoop distributed file system in place of the Google file system. It puts updates into memory and periodically writes them out to files on the disk.
- The updates go to the end of a data file, to avoid seeks. The files are periodically compacted. Updates also go to the end of a write ahead log, to perform recovery if a server crashes.
- Row operations are atomic, with row-level locking and transactions. There is optional support for transactions with wider scope. These use optimistic concurrency control, aborting if there is a conflict with other updates.
- Partitioning and distribution are transparent; there is no client-side hashing or fixed keyspace as in some NoSQL systems. There is multiple master support, to avoid a single point of failure. MapReduce support allows operations to be distributed efficiently.
- HBase's log-structured merge file indexes allow fast range queries and sorting.
- There is a Java API, a Thrift API, and REST API. JDBC/ODBC support has recently been added.

The initial prototype of HBase released in February 2007. The support for transactions is attractive, and unusual for a NoSQL system.

4.2 HyperTable

HyperTable is written in C++. Its was open-sourced by Zvents. It doesn't seem to have taken off in popularity yet, but Baidu became a project sponsor, that should help.

Hypertable is very similar to HBase and BigTable. It uses column families that can have any number of column "qualifiers". It uses timestamps on data with MVCC. It requires an underyling distributed file system such as Hadoop, and a distributed lock manager. Tables are replicated and partitioned over servers by key ranges. Updates are done in memory and later flushed to disk.

Hypertable supports a number of programming language client interfaces. It uses a query language named HQL.

4.3 Cassandra

Cassandra is similar to the other extensible record stores in its data model and basic functionality. It has column groups, updates are cached in memory and then flushed to disk, and the disk representation is periodically compacted. It does partitioning and replication. Failure detection and recovery are fully automatic. However, Cassandra has a weaker concurrency model than some other systems: there is no locking mechanism, and replicas are updated asynchronously.

Like HBase, Cassandra is written in Java, and used under Apache licensing. It is supported by DataStax, and was originally open sourced by Facebook in 2008. It was designed by a Facebook engineer and a Dynamo engineer, and is described as a marriage of Dynamo and BigTable. Cassandra is used by Facebook as well as other companies, so the code is reasonably mature.

Client interfaces are created using Facebook's Thrift framework:

http://incubator.apache.org/thrift/

Cassandra automatically brings new available nodes into a cluster, uses the phi accrual algorithm to detect node failure, and determines cluster membership in a distributed fashion with a gossip-style algorithm.

Cassandra adds the concept of a "supercolumn" that provides another level of grouping within column groups. Databases (called keyspaces) contain column families. A column family contains either supercolumns or columns (not a mix of both). Supercolunns contain columns. As with the other systems, any row can have any combination of column values (i.e., rows are variable length and are not constrained by a table schema).

Cassandra uses an ordered hash index, which should give most of the benefit of both hash and B-tree indexes: you know which nodes could have a particular range of values instead of searching all nodes. However, sorting would still be slower than with B-trees.

Cassandra has reportedly scaled to about 150 machines in production at Facebook, perhaps more by now. Cassandra seems to be gaining a lot of momentum as an open source project, as well.

For applications where Cassandra's eventual-consistency model is not adequate, "quorum reads" of a majority of replicas provide a way to get the latest data. Cassandra writes are atomic within a column family. There is also some support for versioning and conflict resolution.

4.4 Other Systems

Yahoo's PNUTs system also belongs in the "extensible record store" category. However, it is not reviewed in this paper, as it is currently only used internally to Yahoo. We also have not reviewed BigTable, although its functionality is available indirectly through Google Apps. Both PNUTs and BigTable are included in the comparison table at the end of this paper.

4.5 Summary

The extensible record stores are mostly patterned after BigTable. They are all similar, but differ in concurrency mechanisms and other features.

Cassandra focuses on "weak" concurrency (via MVCC) and HBase and HyperTable on "strong" consistency (via locks and logging).

5. SCALABLE RELATIONAL SYSTEMS

Unlike the other data stores, relational DBMSs have a complete pre-defined schema, a SQL interface, and ACID transactions. Traditionally, RDBMSs have not achieved the scalability of the some of the previously-described data stores. As of 5 years ago, MySQL Cluster appeared the most scalable, although not highly performant per node, compared to standard MySQL.

Recent developments are changing things. Further performance improvements have been made to MySQL Cluster, and several new products have come out, in particular VoltDB and Clustrix, that promise to have good per-node performance as well as scalability. It appears likely that some relational DBMSs will provide scalability comparable with NoSQL data stores, with two provisos:

- *Use small-scope operations:* As we've noted, operations that span many nodes, e.g. joins over many tables, will not scale well with sharding.
- *Use small-scope transactions:* Likewise, transactions that span many nodes are going to be very inefficient, with the communication and two-phase commit overhead.

Note that NoSQL systems avoid these two problems by making it difficult or impossible to perform larger-scope operations and transactions. In contrast, a scalable RDBMS does not need to *preclude* larger-scope operations and transactions: they simply penalize a customer for these operations *if they use them.* Scalable RDBMSs thus have an advantage over the NoSQL data stores, because you have the convenience of the higher-level SQL language and ACID properties, but you only pay a price for those

when they span nodes. Scalable RDBMSs are therefore included as a viable alternative in this paper.

5.1 MySQL Cluster

MySQL Cluster has been part of the MySQL release since 2004, and the code evolved from an even earlier project from Ericsson. MySQL Cluster works by replacing the InnoDB engine with a distributed layer called NDB. It is available from MySQL (now Oracle); it is open source. A proprietary MySQL Cluster Carrier Grade upgrade provides administrative and automated management functionality.

MySQL Cluster shards data over multiple database servers (a "shared nothing" architecture). Every shard is replicated, to support recovery. Bi-directional geographic replication is also supported.

MySQL Cluster supports in-memory as well as disk-based data. In-memory storage allows real-time responses.

Although MySQL Cluster seems to scale to more nodes than other RDBMSs to date, it reportedly runs into bottlenecks after a few dozen nodes. Work continues on MySQL Cluster, so this is likely to improve.

5.2 VoltDB

VoltDB is a new open-source RDBMS designed for high performance (per node) as well as scalability.

The scalability and availability features are competitive with MySQL Cluster and the NoSQL systems in this paper:

- Tables are partitioned over multiple servers, and clients can call any server. The distribution is transparent to SQL users, but the customer can choose the sharding attribute.
- Alternatively, selected tables can be *replicated* over servers, e.g. for fast access to read-mostly data.
- In any case, shards are replicated, so that data can be recovered in the event of a node crash. Database snapshots are also supported, continuous or scheduled.

Some features are still missing, e.g. online schema changes are currently limited, and asynchronous WAN replication and recovery are not yet implemented. However, VoltDB has some promising features that collectively may yield an order of magnitude advantage in single-node performance. VoltDB eliminates nearly all "waits" in SQL execution, allowing a very efficient implementation:

- The system is designed for a database that fits in (distributed) RAM on the servers, so that the system need never wait for the disk. Indexes and record structures are designed for RAM rather than disk, and the overhead of a disk cache/buffer is eliminated as well. Performance will be very poor if virtual memory overflows RAM, but the gain with good RAM capacity planning is substantial.
- SQL execution is single-threaded for each shard, using a shared-nothing architecture, so there is no overhead for multi-thread latching.
- All SQL calls are made through stored procedures, with each stored procedure being one transaction. This means, if data is sharded to allow transactions to be executed on a single node, then no locks are required, and therefore no waits on locks. Transaction coordination is likewise avoided.
- Stored procedures are compiled to produce code comparable to the access level calls of NoSQL systems. They can be executed in the same order on a node and on replica node(s).

VoltDB argues that these optimizations greatly reduce the number of nodes needed to support a given application load, with modest constraints on the database design. They have already reported some impressive benchmark results on their web site. Of course, the highest performance requires that the database working set fits in distributed RAM, perhaps extended by SSDs. See [5] for some debate of the architectural issues on VoltDB and similar systems.

5.3 Clustrix

Clustrix offers a product with similarities to VoltDB and MySQL Cluster, but Clustrix nodes are sold as rack-mounted appliances. They claim scalability to hundreds of nodes, with automatic sharding and replication (with a 4:1 read/write ratio, they report 350K TPS on 20 nodes and 160M rows). Failover is automatic, and failed node recover is automatic. They also use solid state disks for additional performance (like the Schooner MySQL and NoSQL appliances).

As with the other relational products, Clustrix supports SQL with fully-ACID transactions. Data distribution and load balancing is transparent to the application programmer. Interestingly, they also designed their system to be seamlessly compatible with MySQL, supporting existing MySQL applications and front-end connectors. This could give them a big advantage in gaining adoption of proprietary hardware.

5.4 ScaleDB

ScaleDB is a new derivative of MySQL underway. Like MySQL Cluster, it replaces the InnoDB engine, and uses clustering of multiple servers to achieve scalability. ScaleDB differs in that it requires disks shared across nodes. Every server must have access to

every disk. This architecture has not scaled very well for Oracle RAC, however.

ScaleDB's sharding is automatic: more servers can be added at any time. Server failure handling is also automatic. ScaleDB redistributes the load over existing servers.

ScaleDB supports ACID transactions and row-level locking. It has multi-table indexing (which is possible due to the shared disk).

5.5 ScaleBase

ScaleBase takes a novel approach, seeking to achieve the horizontal scaling with a layer entirely on top of MySQL, instead of modifying MySQL. ScaleBase includes a partial SQL parser and optimizer that shards tables over multiple single-node MySQL databases. Limited information is available about this new system at the time of this writing, however. It is currently a beta release of a commercial product, not open source.

Implementing sharding as a layer on top of MySQL introduces a problem, as transactions do not span MySQL databases. ScaleBase provides an option for distributed transaction coordination, but the higher-performance option provides ACID transactions only within a single shard/server.

5.6 NimbusDB

NimbusDB is another new relational system. It uses MVCC and distributed object based storage. SQL is the access language, with a row-oriented query optimizer and AVL tree indexes.

MVCC provides transaction isolation without the need for locks, allowing large scale parallel processing. Data is horizontally segmented row-by-row into distributed objects, allowing multi-site, dynamic distribution.

5.7 Other Systems

Google has recently created a layer on BigTable called Megastore. Megastore adds functionality that brings BigTable closer to a (scalable) relational DBMS in many ways: transactions that span nodes, a database schema defined in a SQL-like language, and hierarchical paths that allow some limited join capability. Google has also implemented a SQL processor that works on BigTable. There are still a lot of differences between Megastore / BigTable "NoSQL" and scalable relational systems, but the gap seems to be narrowing.

Microsoft's Azure Tables product provides horizontal scaling for both reads and writes, using a partition key, row key, and timestamps. Tables are stored "in the cloud" and can sync multiple databases. There is no fixed schema: rows consist of a list of property-value pairs. Due to the timing of the original version of this paper, Azure is not covered here.

The major RDBMSs (DB2, Oracle, SQL Server) also include some horizontal scaling features, either shared-nothing, or shared-disk.

5.8 Summary

MySQL Cluster uses a "shared nothing" architecture for scalability, as with most of the other solutions in this section, and it is the most mature solution here.

VoltDB looks promising because of its horizontal scaling as well as a bottom-up redesign to provide very high per-node performance. Clustrix looks promising as well, and supports solid state disks, but it is based on proprietary software and hardware.

Limited information is available about ScaleDB, NimbusDB, and ScaleBase at this point; they are at an early stage.

In theory, RDBMSs should be able to deliver scalability as long as applications avoid cross-node operations. If this proves true in practice, the simplicity of SQL and ACID transactions would give them an advantage over NoSQL for most applications.

6. USE CASES

No one of these data stores is best for all uses. A user's prioritization of features will be different depending on the application, as will the type of scalability required. A complete guide to choosing a data store is beyond the scope of this paper, but in this section we look at some examples of applications that fit well with the different data store categories.

6.1 Key-value Store Example

Key-value stores are generally good solutions if you have a simple application with only one kind of object, and you only need to look up objects up based on one attribute. The simple functionality of key-value stores may make them the simplest to use, especially if you're already familiar with memcached.

As an example, suppose you have a web application that does many RDBMS queries to create a tailored page when a user logs in. Suppose it takes several seconds to execute those queries, and the user's data is rarely changed, or you know when it changes because updates go through the same interface. Then you might want to store the user's tailored page as a single object in a key-value store, represented in a manner that's efficient to send in response to browser requests, and index these objects by user ID. If you store these objects persistently, then you may be able to avoid many RDBMS queries, reconstructing the objects only when a user's data is updated.

Even in the case of an application like Facebook, where a user's home page changes based on updates made by the user as well as updates made by others, it may be possible to execute RDBMS queries just once

when the user logs in, and for the rest of that session show only the changes made by that user (not by other users). Then, a simple key-value store could still be used as a relational database cache.

You could use key-value stores to do lookups based on multiple attributes, by creating additional key-value indexes that you maintain yourself. However, at that point you probably want to move to a document store.

6.2 Document Store Example

A good example application for a document store would be one with multiple different kinds of objects (say, in a Department of Motor Vehicles application, with vehicles and drivers), where you need to look up objects based on multiple fields (say, a driver's name, license number, owned vehicle, or birth date).

An important factor to consider is what level of concurrency guarantees you need. If you can tolerate an "eventually consistent" model with limited atomicity and isolation, the document stores should work well for you. That might be the case in the DMV application, e.g. you don't need to know if the driver has new traffic violations in the past minute, and it would be quite unlikely for two DMV offices to be updating the same driver's record at the same time. But if you require that data be up-to-date and atomically consistent, e.g. if you want to lock out logins after three incorrect attempts, then you need to consider other alternatives, or use a mechanism such as quorum-read to get the latest data.

6.3 Extensible Record Store Example

The use cases for extensible record stores are similar to those for document stores: multiple kinds of objects, with lookups based on any field. However, the extensible record store projects are generally aimed at higher throughput, and may provide stronger concurrency guarantees, at the cost of slightly more complexity than the document stores.

Suppose you are storing customer information for an eBay-style application, and you want to partition your data both horizontally and vertically:

- You might want to cluster customers by country, so that you can efficiently search all of the customers in one country.
- You might want to separate the rarely-changed "core" customer information such as customer addresses and email addresses in one place, and put certain frequently-updated customer information (such as current bids in progress) in a different place, to improve performance.

Although you could do this kind of horizontal/vertical partitioning yourself on top of a document store by creating multiple collections for multiple dimensions, the partitioning is most easily achieved with an extensible record store like HBase or HyperTable.

6.4 Scalable RDBMS Example

The advantages of relational DBMSs are well-known:

- If your application requires many tables with different types of data, a relational schema definition centralizes and simplifies your data definition, and SQL greatly simplifies the expression of operations that span tables.
- Many programmers are already familiar with SQL, and many would argue that the use of SQL is simpler than the lower-level commands provided by NoSQL systems.
- Transactions greatly simplify coding concurrent access. ACID semantics free the developer from dealing with locks, out-of-date data, update collisions, and consistency.
- Many more tools are currently available for relational DBMSs, for report generation, forms, and so on.

As a good example for relational, imagine a more complex DMV application, perhaps with a query interface for law enforcement that can interactively search on vehicle color, make, model, year, partial license plate numbers, and/or constraints on the owner such as the county of residence, hair color, and sex. ACID transactions could also prove valuable for a database being updated from many locations, and the aforementioned tools would be valuable as well. The definition of a common relational schema and administration tools can also be invaluable on a project with many programmers.

These advantages are dependent, of course, on a relational DBMS scaling to meet your application needs. Recently-reported benchmarks on VoltDB, Clustrix, and the latest version of MySQL Cluster suggest that scalability of relational DBMSs is greatly improving. Again, this assumes that your application does not demand updates or joins that span many nodes; the transaction coordination and data movement for that would be prohibitive. However, the NoSQL systems generally do not offer the possibility of transactions or query joins across nodes, so you are no worse off there.

7. CONCLUSIONS

We have covered over twenty scalable data stores in this paper. Almost all of them are moving targets, with limited documentation that is sometimes conflicting, so this paper is likely out-of-date if not already inaccurate at the time of this writing. However, we will attempt a snapshot summary, comparison, and predictions in this section. Consider this a starting point for further study.

7.1 Some Predictions

Here are some predictions of what will happen with the systems we've discussed, over the next few years:

- Many developers will be willing to abandon globally-ACID transactions in order to gain scalability, availability, and other advantages. The popularity of NoSQL systems has already demonstrated this. Customers tolerate airline over-booking, and orders that are rejected when items in an online shopping cart are sold out before the order is finalized. The world is not globally consistent.
- NoSQL data stores will not be a "passing fad". The simplicity, flexibility, and scalability of these systems fills a market niche, e.g. for web sites with millions of read/write users and relatively simple data schemas. Even with improved relational scalability, NoSQL systems maintain advantages for some applications.
- New relational DBMSs will also take a significant share of the scalable data storage market. If transactions and queries are generally limited to single nodes, these systems should be able to scale [5]. Where the desire for SQL or ACID transactions are important, these systems will be the preferred choice.
- Many of the scalable data stores will not prove "enterprise ready" for a while. Even though they fulfill a need, these systems are new and have not yet achieved the robustness, functionality, and maturity of database products that have been around for a decade or more. Early adopters have already seen web site outages with scalable data store failures, and many large sites continue to "roll their own" solution by sharding with existing RDBMS products. However, some of these new systems will mature quickly, given the great deal of energy directed at them.
- There will be major consolidation among the systems we've described. One or two systems will likely become the leaders in each of the categories. It seems unlikely that the market and open source community will be able to support the sheer number of products and projects we've studied here. Venture capital and support from key players will likely be a factor in this consolidation. For example, among the document stores, MongoDB has received substantial investment this year.

7.2 SQL vs NoSQL

SQL (relational) versus NoSQL scalability is a controversial topic. This paper argues against both extremes. Here is some more background to support this position.

The argument for relational over NoSQL goes something like this:

- If new relational systems can do everything that a NoSQL system can, with analogous performance and scalability, and with the convenience of transactions and SQL, why would you choose a NoSQL system?
- Relational DBMSs have taken and retained majority market share over other competitors in the past 30 years: network, object, and XML DBMSs.
- Successful relational DBMSs have been built to handle other specific application loads in the past: read-only or read-mostly data warehousing, OLTP on multi-core multi-disk CPUs, in-memory databases, distributed databases, and now horizontally scaled databases.
- While we don't see "one size fits all" in the SQL products themselves, we do see a common interface with SQL, transactions, and relational schema that give advantages in training, continuity, and data interchange.

The counter-argument for NoSQL goes something like this:

- We haven't yet seen good benchmarks showing that RDBMSs can achieve scaling comparable with NoSQL systems like Google's BigTable.
- If you only require a lookup of objects based on a single key, then a key-value store is adequate and probably easier to understand than a relational DBMS. Likewise for a document store on a simple application: you only pay the learning curve for the level of complexity you require.
- Some applications require a flexible schema, allowing each object in a collection to have different attributes. While some RDBMSs allow efficient "packing" of tuples with missing attributes, and some allow adding new attributes at runtime, this is uncommon.
- A relational DBMS makes "expensive" (multi-node multi-table) operations "too easy". NoSQL systems make them impossible or obviously expensive for programmers.
- While RDBMSs have maintained majority market share over the years, other products have established smaller but non-trivial markets in areas where there is a need for particular capabilities, e.g. indexed objects with products like BerkeleyDB, or graph-following operations with object-oriented DBMSs.

Both sides of this argument have merit.

7.3 Benchmarking

Given that scalability is the focus of this paper and of the systems we discuss, there is a "gaping hole" in our analysis: there is a scarcity of benchmarks to substantiate the many claims made for scalability. As we have noted, there are benchmark results reported on some of the systems, but almost none of the benchmarks are run on more than one system, and the results are generally reported by proponents of that one system, so there is always some question about their objectivity.

In this paper, we've tried to make the best comparisons possible based on architectural arguments alone. However, it would be highly desirable to get some useful objective data comparing the architectures:

- The trade-offs between the architectures are unclear. Are the bottlenecks in disk access, network communication, index operations, locking, or other components?
- Many people would like to see support or refutation of the argument that new relational systems can scale as well as NoSQL systems.
- A number of the systems are new, and may not live up to scalability claims without years of tuning. They also may be buggy. Which are truly mature?
- Which systems perform best on which loads? Are open source projects able to produce highly performant systems?

Perhaps the best benchmark to date is from Yahoo! Research [2], comparing PNUTS, HBASE, Cassandra, and sharded MySQL. Their benchmark, YCSB, is designed to be representative of web applications, and the code is available to others. Tier 1 of the benchmark measures raw performance, showing latency characteristics as the server load increases. Tier 2 measures scaling, showing how the benchmarked system scales as additional servers are added, and how quickly the system adapts to additional servers.

In this paper, I'd like to make a "call for scalability benchmarks," suggesting YCSB as a good basis for the comparison. Even if the YCSB benchmark is run by different groups who may not duplicate the same hardware Yahoo specified, the results will be informative.

7.4 Some Comparisons

Given the quickly-changing landscape, this paper will not attempt to argue the merits of particular systems, beyond the comments already made. However, a comparison of the salient features may prove useful, so we finish with some comparisons.

Table 1 below compares the concurrency control, data storage medium, replication, and transaction mechanisms of the systems. These are difficult to summarize in a short table entry without over-simplifying, but we compare as follows.

For concurrency:

- Locks: some systems provide a mechanism to allow only one user at a time to read or modify an entity (an object, document, or row). In the case of MongoDB, a locking mechanism is provided at a field level.
- MVCC: some systems provide multi-version concurrency control, guaranteeing a read-consistent view of the database, but resulting in multiple conflicting versions of an entity if multiple users modify it at the same time.
- None: some systems do not provide atomicity, allowing different users to modify different parts of the same object in parallel, and giving no guarantee as to which version of data you will get when you read.
- ACID: the relational systems provide ACID transactions. Some of the more recent systems do this with no deadlocks and no waits on locks, by pre-analyzing transactions to avoid conflicts.

For data storage, some systems are designed for storage in RAM, perhaps with snapshots or replication to disk, while others are designed for disk storage, perhaps caching in RAM. RAM-based systems typically allow use of the operating system's virtual memory, but performance appears to be very poor when they overflow physical RAM. A few systems have a pluggable back end allowing different data storage media, or they require a standardized underlying file system.

Replication can insure that mirror copies are always in sync (that is, they are updated lock-step and an operation is not completed until both replicas are modified). Alternatively, the mirror copy may be updated asynchronously in the background. Asynchronous replication allows faster operation, particular for remote replicas, but some updates may be lost on a crash. Some systems update local copies synchronously and geographically remote copies asynchronously (this is probably the only practical solution for remote data).

Transactions are supported in some systems, and not in others. Some NoSQL systems provide something in between, where "Local" transactions are supported only within a single object or shard.

Table 1 compares the systems on these four dimensions.

Table 1. System Comparison (grouped by category)

System	Conc Contol	Data Storage	Repli-cation	Tx
Redis	Locks	RAM	Async	N
Scalaris	Locks	RAM	Sync	L
Tokyo	Locks	RAM or disk	Async	L
Voldemort	MVCC	RAM or BDB	Async	N
Riak	MVCC	Plug-in	Async	N
Membrain	Locks	Flash + Disk	Sync	L
Membase	Locks	Disk	Sync	L
Dynamo	MVCC	Plug-in	Async	N
SimpleDB	None	S3	Async	N
MongoDB	Locks	Disk	Async	N
Couch DB	MVCC	Disk	Async	N
Terrastore	Locks	RAM+	Sync	L
HBase	Locks	Hadoop	Async	L
HyperTable	Locks	Files	Sync	L
Cassandra	MVCC	Disk	Async	L
BigTable	Locks+s tamps	GFS	Sync+ Async	L
PNUTs	MVCC	Disk	Async	L
MySQL Cluster	ACID	Disk	Sync	Y
VoltDB	ACID, no lock	RAM	Sync	Y
Clustrix	ACID, no lock	Disk	Sync	Y
ScaleDB	ACID	Disk	Sync	Y
ScaleBase	ACID	Disk	Async	Y
NimbusDB	ACID, no lock	Disk	Sync	Y

Another factor to consider, but impossible to quantify objectively in a table, is code maturity. As noted earlier, many of the systems we discussed are only a couple of years old, and are likely to be unreliable. For this reason, existing database products are often a better choice if they can scale for your application's needs.

Probably the most important factor to consider is actual performance and scalability, as noted in the discussion of benchmarking. Benchmark references will be added to the author's website cattell.net/datastores as they become available. Updates and corrections to this paper will be posted there as well. The landscape for scalable data stores is likely to change significantly over the next two years!

8. ACKNOWLEDGMENTS

I'd like to thank Len Shapiro, Jonathan Ellis, Dan DeMaggio, Kyle Banker, John Busch, Darpan Dinker, David Van Couvering, Peter Zaitsev, Steve Yen, and Scott Jarr for their input on earlier drafts of this paper. Any errors are my own, however! I'd also like to thank Schooner Technologies for their support on this paper.

9. REFERENCES

[1] F. Chang et al, "BigTable: A Distributed Storage System for Structured Data", *Seventh Symposium on Operating System Design and Implementation,* November 2006.

[2] B. Cooper et al, "Benchmarking Cloud Serving Systems with YCSB", *ACM Symposium on Cloud Computing* (SoCC), Indianapolis, Indiana, June 2010.

[3] B. DeCandia et al, "Dynamo: Amazon's Highly Available Key-Value Store", *Proceedings 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, and partition-tolerant web services", *ACM SIGACT News 33*, 2, pp 51-59, March 2002.

[5] M. Stonebraker and R. Cattell, "Ten Rules for Scalable Performance in Simple Operation Datastores", *Communications of the ACM,* June 2011.

10. SYSTEM REFERENCES

The following table provides web information sources for all of the DBMSs and data stores covered in the paper, even those peripherally mentioned, alphabetized by system name. The table also lists the licensing model (proprietary, Apache, BSD, GPL), which may be important depending on your application.

System	License	Web site for more information
Azure	Prop	blogs.msdn.com/b/windowsazure storage/
Berkeley DB	BSD	oss.oracle.com/berkeley-db.html
BigTable	Prop	labs.google.com/papers/bigtable.html
Cassandra	Apache	incubator.apache.org/cassandra
Clustrix	Prop	clustrix.com
CouchDB	Apache	couchdb.apache.org

Dynamo	Internal	portal.acm.org/citation.cfm?id=1294281
GemFire	Prop	gemstone.com/products/gemfire
HBase	Apache	hbase.apache.org
HyperTable	GPL	hypertable.org
Membase	Apache	membase.com
Membrain	Prop	schoonerinfotech.com/products/
Memcached	BSD	memcached.org
MongoDB	GPL	mongodb.org
MySQL Cluster	GPL	mysql.com/cluster
NimbusDB	Prop	nimbusdb.com
Neo4j	AGPL	neo4j.org

OrientDB	Apache	orienttechnologies.com
PNUTs	Internal	research.yahoo.com/node/2304
Redis	BSD	code.google.com/p/redis
Riak	Apache	riak.basho.com
Scalaris	Apache	code.google.com/p/scalaris
ScaleBase	Prop	scalebase.com
ScaleDB	GPL	scaledb.com
SimpleDB	Prop	amazon.com/simpledb
Terrastore	Apache	code.google.com/terrastore
Tokyo	GPL	tokyocabinet.sourceforge.net
Versant	Prop	versant.com
Voldemort	None	project-voldemort.com
VoltDB	GPL	voltdb.com