Big Data Paper Summaries

[burgerm, jkleine, tgeorg]@ethz.ch

HS 2020

Contents

1 Windows Azure Storage (WAS) - burgerm

Source: https://sigops.org/s/conferences/sosp/2011/current/2011-Cascais/printable/11-calder.pdf

To store seemingly limitless amounts of data for any duration of time and pay only what is being used. WAS provides cloud storage in the form of *Blobs* (user files), *Tables* (structured storage) and *Queues* (message delivery). WAS claims to satisfy *strong consistency*, *high availability* and *partition tolerance* all at the same time; ie. all of *CAP*.

1.1 Global Partitioned Namespace

WAS leverages DNS to provide a single global and scalable namespace. The storage namespace is broken down into three parts:

- *account name* is the customer selected account to access the data; used to locate primary storage cluster for data, all requests to access this account's data go to said cluster
- *partition name* locates data within cluster, used to scale out data access across nodes if required by traffic needs
- *object name* used to access indiv. objects of a partition if present. A partition name can also directly point to data itself depending on the type of data. Atomic operations are supported across objects.

Access data by `http(s)://[AccountName].<service>.core.windows.net/[PartitionName]/[ObjectName]`

- Blob: access by Partition Name
- Table: each row has primary key (partitionName, objectName), can group rows into same partition to make Table
- Queues have Partition Name with messages having an Object Name

1.2 High Level Architecture

WAS is part of the Windows Azure Cloud platform. The Windows Azure Fabric Controller provides node management, network configuration, health monitoring, starting/stopping of service instances and service deployment. WAS is responsible for replication, placement, load balancing.

Storage Stamp is a cluster of N racks of storage nodes, each rack being an independent fault domain with redundant networking and power. To optimize economically a stamp should be kept at about 70% utilization in terms of capacity, transactions and bandwidth, but avoid going over 80% to maintain high performance and lifetime. Inter-stamp replication manages migration to maintain good utilization levels.

Location Service (LS) manages all the storage stamps and the account namespace and assignment of accounts across the stamps. Itself distributed and redundant, performs disaster recovery and load balancing by updating DNS entries to the respective exposed VIPs (Virtual IP) of the assigned stamp.

Three Stamp Layers bottom-up

- Stream Layer: stores data on disks and responsible for durability i.e. replication across the stamp (sort of a distr. file system within a stamp. *Files* called *Streams*, which are ordered lists of *Extents*.
- Partition Layer: manage, address, caching, consistency of the data and provide higher level abstractions (Blob, etc.). Scalability and load bal.
- Front-End (FE) Layer: Set of stateless servers responsible for request handling, authentication and authorization, also cache freq. accessed data directly

Two replication engines

- Intra-Stamp Replication *synchronous* in the Stream Layer; replicate blocks of disk storage, fault-tolerance
- Inter-Stamp Replication *asynchronous* in the Partition Layer; replicate objects and related transactions, geo-redundancy

1.3 Stream Layer

Provides internal interface used by Partition Layer only, see Fig. 1 for the layout. Write operations are append-only and appends are atomic (entire block or nothing), supports atomic multi-block append. Only last *Extent* can be appended to, others are *sealed* (sealed extents are immutable). Blocks are the minimum unit of storage and checksum validation is performed on a block-level for data integrity. Extents are the unit of replication in the stream layer. Depending on object size several extents are used or several objects put into even the same block. Stream looks like a big file to the Partition Layer. Partition Layer handles timeouts, retries and duplicates.

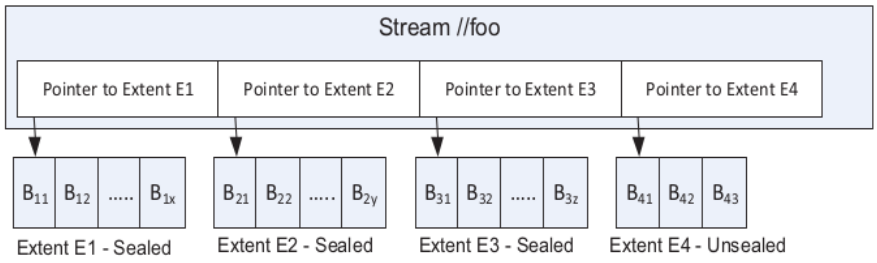


Figure 1: Stream Layout with Extents and Blocks

The *Stream Manager (SM)* manages a set of *Extent Nodes (EN)* and forms a Paxos-based cluster. The SM is only aware of Streams and Extents, but not Blocks. Each EN maintains storage for a set of extent replicas assigned to by the SM. An overview is in Fig. 2. The primary EN targeted by client is in charge of coordinating the writes to the secondary ENs.

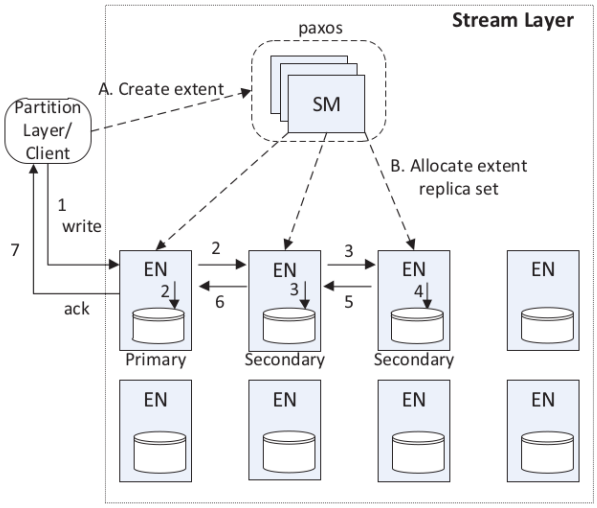


Figure 2: Stream Manager Overview

The SM coordinates the sealing operation among the ENs. Unavailable ENs during sealing will later be forced to sync to sealed commit length. The partition layer can read at specific locations (extend + location, length) and upon partition loading the metadata and commit log (both in streams) are being read sequentially until the end.

To save space sealed extents are erasure coded and fragments are distributed, which replaces the three-fold replication for sealed extents. If an extent cannot respond fast enough to a request, the request cancelled and can be redirected to a different EN by the client.

Spindle disks are optimized for long sequential operations, to still ensure fairness to e.g. short random access custom IO scheduling is used at the cost of slight increase in latency.

For durability a write is only acknowledged to the client after three copies have been made to durable storage. A *journal drive* (e.g. SSD) is being fed sequentially with incoming writes to the EN to saturate bandwidth and the write is ack'ed as soon as the journal has been written or the actual destination.

1.4 Partition Layer

The partition layer (Overview in Fig. 3 stores the different types of objects and understands what a transaction for a specific object means (Blob, Table, Queue).

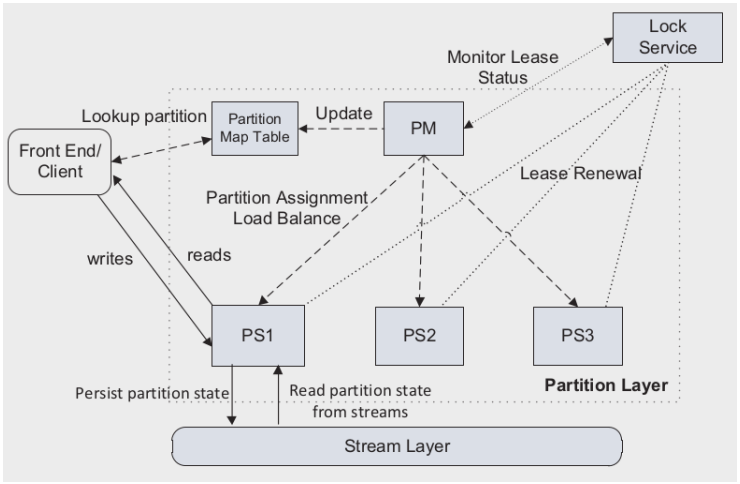


Figure 3: Partition Layer Overview, the *Lock Service* (Paxos based) determines the current leader PM and ensures the PS have non-overlapping partition ranges

The partition layer provides the *Object Table* (OT). OTs (up to Petabytes large) are broken into *RangePartitions*. A *RangePartition* is a contiguous range of rows from a given low-key to a high-key. OTs are Account Table, Blob Table, Entity Table, Message Table, Schema Table, Partition Map Table. They support standart insert, update, delete ops. on rows as well as query/get ops.

The *Partition Manager* (PM) splits the OTs into RangePartitions and assigns them to Partition Servers (PS), which serve the requests for assigned partitions. Load balancing takes place by reassigning RangePartitions to PS and splitting/merging RangePartitions.

Each *RangePartition* holds streams in the Stream Layer for Metadata, Commit-Log, Row-Data and Blob-Data. Various data-structures are kept in-memory for faster access and caching. The PS uses a commit log, checkpointing and in-memory buffering to achieve both durability and high throughput.

The Partition Layer in the primary stamp will asnychronously geo-replicate changes to the secondary stamp using inter-stamp replication.

1.5 Further sections

Application Throughput lays out some performance analysis: scaling number of worker VMs, table throughput, blob throughput etc.

Workload Profiles shows various large-scale example workloads for using WAS.

Design choices

- separate computation from storage for cloud offerings; allows to scale the two independently.

- range-based partitioning/indexing instead of hash-based (recall e.g. the assignment of the RangePartitions to the PS), hashes would auto-load balance, but ranges keep related data close (e.g. same customer data etc.)
- Others: Throttling/Isolation (of e.g. accounts), Auto. Load-Balance (of partitions), Append-only system, Upgrades (Fault and upgrade domains)
- CAP, claimed to hold within a single storage stamp under conditions observed in practice i.e. within their fault model.

2 Dynamo: Amazon Key-Value store - jkleine

Source: `https://dl.acm.org/doi/10.1145/1323293.1294281`

Dynamo is a highly available key-value store used internally by Amazon and backs many of their services. It follows a simple query model, allowing only simple *put* and *get* instructions to store *blobs* (binary objects) usually less than 1 MB in size. No operations span multiple data items and there is no need for relational schemes. Dynamo sacrifices consistency to some extend to provide high availability. Note Dynamo is designed for Amazon internal use only and assumes the operating environment to be non-hostile, thus authentication and authorization are not part of the design considerations.

The goal is to have a system that conforms to latency constrains in the 99.9^{th} percentile of of the distribution. Additionally Dynamo targets the design space of an "always writable" data store, meaning write requests are not rejected even during failure of nodes and network partitions. Additionally the system should embrace the following principles:

- *Incremental scalability*: Dynamo should be able to scale out one storage 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.
- *Decentralization*: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control.
- *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.

2.1 System Architecture

Partitioning Dynamo partitioning relies on consistent hashing (in the form of MD5 hash of the key) to extract a 128-bit identifier that maps the object to circular space. This circular space is divided among the storage nodes. To account for heterogeneous hardware, nodes in the system are implemented as virtual nodes with one physical node running multiple virtual nodes depending on its hardware resources. Each virtual node is assigned a position on the ring (called *token*). This virtual node is then the coordinator of all the keys between its token and the token of the predecessor (next virtual node counter clockwise on the ring).

The specific strategy by which is the circle is devided between the nodes is visualized in Figure 4 Strategy 3. The space is divided into Q equal parts (tokens), with ever node being assigned Q/S tokens (where S is the number of nodes in the system and $Q >> S$). Additionally each token is also replicated by the first N

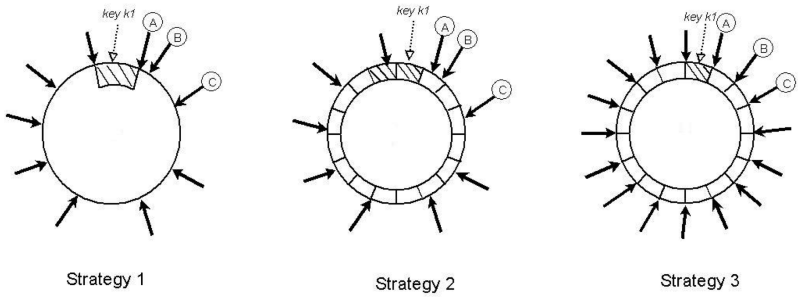


Figure 4: Overview of strategies explored to divide the hash space. Strategy 1 nodes are assigned a random token and store all the data between it and the Nth node in counter-clockwise direction. Strategy 2 was deemed inefficient, not further discussed in the summary (mix of 1 & 3). Strategy 3 divides the space into Q equally sized sections, assigning each node Q/S sections. each section is replicated by the first N distinct nodes in clockwise direction.

distinct nodes in clockwise direction on the circle. When a node leaves the network, its tokens are divided among the remaining nodes, such that every node again has Q/S tokens. Analogously, when a node enters the network it steals an equal amount of tokens from the other nodes.

The nodes replicating a particular key are stored in a *preference list*.

Execution of `get()` and `put()` The node handling the read or write operation is called the *coordinator*. Typically its the first node in the preference list, but load balancing might decide to give the task to a different node on the preference list.

To maintain consistency the nodes follow a protocol that mimics quorums, called *sloppy quorums*. Hereby every read request needs to be answered by R nodes, and every write request needs to be answered by W nodes. More specifically the coordinator will generate a vector clock for the new item and writes it locally. It then sends the new item along with the vector clock to all the N highest ranked reachable nodes. If W nodes respond the write operation is considered successful. Choosing $R + W > N$ guarantees that the set of nodes that answered the write requests and the nodes answering the read request overlap. Alternatively choosing $W = 1$ guarantees that write requests will be answered as long as there is one active node left in the system. This allows to configure different Dynamo instances to the specific use case.

Handling Failures To handle temporary node failures, Dynamo implements *hinted handoffs*, where messages send to a node that is down will be forwarded to the next available node. This node will detect that the message was intended for another node and will save it in a separate local database. Upon reconnecting with the failed node it will send all the data from the local database to the failed node to bring it "up to date".

For permanent node failure, where this hinted handoff is not enough to recover the data (the node temporarily storing the data might fail as well), Dynamo synchronizes replica nodes regularly. To minimize the transferred data Dynamo uses Merkel trees, where each leaf is the hash of some data and the inner nodes are the hash of the child nodes. This way it is possible to find where exactly the data differs without sending unnecessary copies.

3 The Hadoop Distributed File System - tgeorg

Source: `https://doi.org/10.1109/MSST.2010.5496972`

3.1 INTRODUCTION AND RELATED WORK

Two types of nodes:

- **NameNode** stores metadata
- **DataNode** stores application data

All nodes are fully connected and communicate with each other using TCP-based protocols. There are no data protection mechanisms in within DataNodes, instead data replication between nodes is used for redundancy.

3.2 ARCHITECTURE

NameNode "The HDFS namespace is a hierarchy of files and directories."
Files and directories are represented by *inodes* on the NameNode.
"An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client."
Blocks are typically 128MB but are user selectable file-by-file.
For writing data, the client first connects to the NameNode to nominate which DataNodes to write the replicas to.

- **Image:** inode data and list of blocks belonging to a file. Stored in RAM.
- **Checkpoint:** persistent record of an image.
- **Journal:** modification log of the image stored on the NameNode on disk.

DataNodes "Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp."

- namespace ID
  - Identifies the cluster. Nodes with different namespace ID cannot join the cluster.
  - New nodes do not have one and receive it when joininig the cluster.
- storage ID
  - Identifies the DataNode
  - Persists through reboots

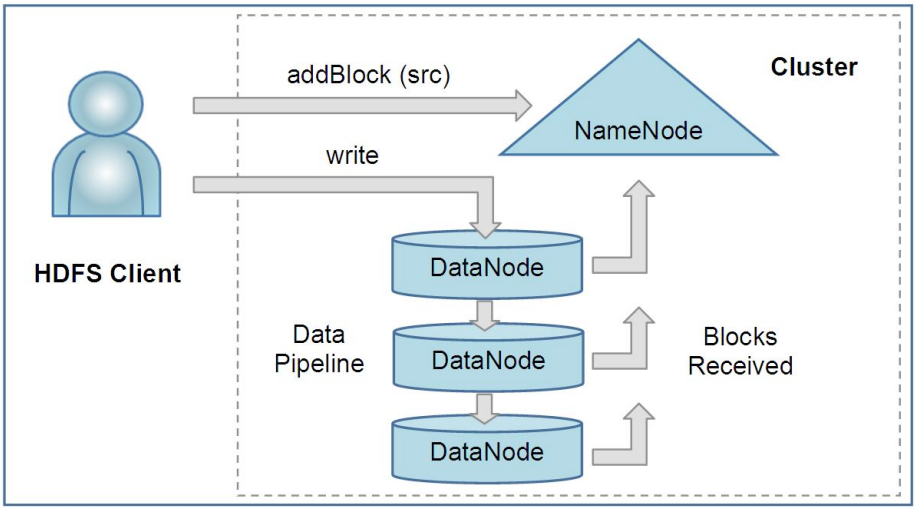


Figure 5: HDFS client creating a new file

Heartbeats

- every 3 seconds
- timeout if none received within 10 minutes → NameNode considers DataNode out of service / replicas as unavailable
- content of heartbeat (total storage capacity, fraction of storage in use, and the number of data transfers currently in progress.)

HDFS Client Abstract interface, for user it seems as they are using standard filesystem (directories, etc.); user does not see where data is stored

- **Read:** Ask NameNode which datanote contains which (replicas of) files, then contact DataNode directly
- **Write:** Ask NameNode, which DataNodes should host the (replicas of the) files

HDFS provides API that exposes location of file blocks (unlike conventional FS) Default replication factor of block is 3

Image and Journal

- **namespace image:** file system metadata
- **checkpoint file:** persistent record of namespace image
- **journal:** write-ahead commit log for changes to the file system. Missing/corrupt checkpoint or journal results in (partly) lost namespace information.

Transactions are batched to prevent/reduce bottle neck caused by synchronous flush-and-sync

CheckpointNode Alternative roles for NameNodes: *CheckpointNode* and *BackupNode* The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. It downloads current check-point and journal files from NameNode, merges them locally, and returns new checkpoint back to NameNode.

BackupNode Next to periodic checkpoints, maintains up-to-date image of file system namespace. Quasi read-only NameNode. It can perform all operations of NameNode as long as they are read-only.

"Use of a BackupNode provides the option of running the NameNode without persistent storage, delegating responsibility for the namespace state per-sisting to the BackupNode"

Upgrades, File System Snapshots *Snapshot*: current state of file system saved persistently Local Snapshot on DataNode is created by creating copy of storage directory (not files themselves) and hard linking existing blocks. As deletion are done via removing original hard links and writes are COW[1], old blocks can still be recovered via the snapshot. Snapshot is coordinated over all nodes.

3.3 FILE I/O-OPERATIONS AND REPLICA MANGEMENT

File Read and Write Data is added by creating new file and writing to it. After closing file cannot be changed except for appends.
For writing the writer receives a lease (a lock essetially), lease is periodically renewed via heartbeat.
Lease has soft and hard limit if not renewed

- **soft limit:** Another client can kick writer
- **hard limit:** DataNode revokes lease of writer

Reading is never blocked by lease

Integrity of blocks is verified via checksums. when block is requested, checksum is sent together with it. Client verifies checksum and notifies NameNode if they do not match.
Block replicas are read based on distance, i.e. closest replica of a block first, if that fails / is corrupt then second closest is selected, ... etc.
HDFS I/O is optimised for batch processing

[1] Copy-On-Write

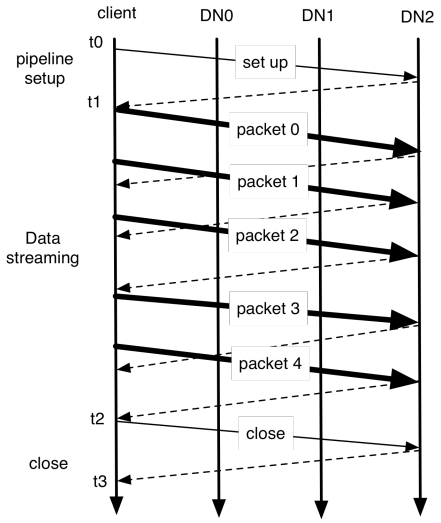


Figure 6: Data pipeline during block construction

Block Placement Default HDFS replica placement:
General distribution: no more than one replica per node and no more than two replicas per rack (provided sufficient racks)

First replica is placed close to the writer. To determine distance network bandwidth and rack location are used. Block placement policy is adjustable.
After selecting targets for replicas, they are organised in a pipeline.
Reads are performed from the closest node, that hosts the wanted replica, first.
Together this should reduce inter-node and inter-rack traffic.

Replication management

- **Over-replicated:** If a block becomes over-replicated the NameNode decides which copy to remove. NameNode tries to maintain number of racks over which block has been replicated and prefers removing it from DataNode with low free disk space.
- **Under-replicated:** Block gets put in replication priority queue. Less replicas $\rightarrow$ higher priority Replication follows previous rule ($\leq$ 1 per node, $\leq$ 2 per rack)

Too many replications in one rack: Treat block as under-replicated $\rightarrow$ replicate on different rack $\rightarrow$ now block is over-replicated $\rightarrow$ rule for over-replication is applied, removing the block from the rack in which it violates the rule $\leq$ 2 per rack

Balancer Block placement does not take disk space utilisation into account. DataNode disk usage is balanced if it is close to the mean total disk usage of

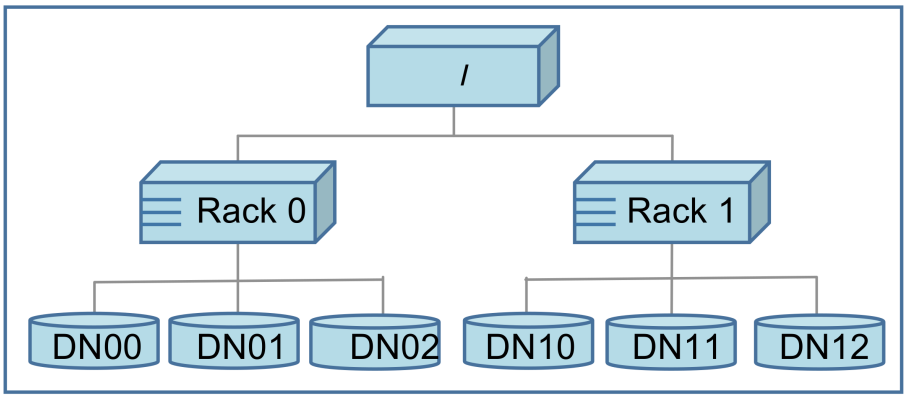


Figure 7: Cluster topology example

the cluster minus some threshold.
"*balancer*" tool is responsible for ensuring balanced disk usage on all DataNodes and moves blocks around accordingly. Has to garantee that number total number of replicas and number of racks hosting replicas does not get reduced.
Optimised to reduce inter-rack copying as well as possibility of bandwidth limits.

Block Scanner "Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data." "If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica."
Verification time for each block is stored in log.
If block scanner or client detects corrupt block, NameNode is informed which marks block as corrupt. Corrupt blocks are not immediately deleted, instead a good copy is replicated first. Therefore data (even if corrupt) is preserved as long as possible.

Decommissioning Cluster administrator specifies which nodes can and cannot register with the cluster.
Cluster admin can command re-evaluation of include/exclude list. If DataNode is newly excluded, it gets marked for decommissioning. DataNode continues to serve read requests but no new data is added and existing blocks are replicated on other DataNodes. Once all blocks are replicated, the DataNode is markes as decommissioned.

Inter-Cluster Data Copy Done via a tool called DistCp. Uses MapReduce.

3.4 PRACTICE AT "YAHOO!"

(*skipped, not relevant*)

Durability of Data Three time replication is robust guard against data loss caused by uncorrelated node failures.
No guaranteed revocery against correlated node failures, e.g. whole rack falling out due to losing a rack switch, power loss, etc.
Block scanner regularly scans for corrupted data.

Caring for the Commons (*skipped, not relevant*)

Benchmarks (*skipped, not relevant*)

3.5 FUTURE WORK

Improve BackupNode ability to take over if NameNode is down.
Improve scalability of NameNode $\rightarrow$ introduce multiple NameNode with independent namespaces.

4 Hadoop: The Definitive Guide (Chapter 3) - burgerm

Source: `http://proquest.safaribooksonline.com/book/databases/hadoop/9781491901687` chaptre I.3

Introducing the **Hadoop Distributed Filesystem** (HDFS); designed for very large files and built around the processing pattern of writing once and reading many times. The time to read the whole dataset is more important than the latency of reading the first word. It is designed to run on commodity hardware.

Not suited for:

- Low-latency data access
- Lots of small files
- Multiple writers, arbitrary file modifications

4.1 HDFS concepts

Blocks Files are broken up into blocks of 128MB (default). A file smaller than the block size does not occupy the full block length in storage. Having blocks allows larger files, simplifies the storage subsystem (blocks are fixed size) and well fit replication for fault tolerance and availability.

Nodes There are *Namenodes and Datanodes* operating in a master-worker pattern. The Namenode manages the filesystem namespace and the block locations for a file (reconstructed at startup from datanode information). Datanodes hold the actual blocks and receive and send them to the clients; they also periodically update the Namenode with list of their blocks. The Namenode persistent state can be backed-up on network storage, *secondary Namenodes* can be hot standby or backup replacement.

Block Caching Datanodes can cache frequently used blocks directly in memory (a block is usually only cached in a single Datanode (default)).

HDFS Federation As the Namenode keeps a reference to each file and block, memory on the Namenode becomes the limiting factor to scaling. HDFS Federation allows multiple Namenodes for are each responsible for a portion of the filesystem namespace.

HDFS High Availability The system is resistant to node failures, however restarting a new Namenode after failure can take 30min or more (long recovery time). To improve active-standby Namenodes are introduced, which take over immediately in case the primary fails. This requires a highly available shared storage between the Namenodes for the edit log, which allows the standby to sync up quickly if required. For this an NFS filer or a *quorum journal manager* (QJM) (HDFS dedicated implementation) is used.

4.2 Various

HDFS offers a command-line interface. The permission model resembles the standart POSIX model with rwx bits with owner, group and mode. The Namenode process is the superuser.

Hadoop supports interaction with various filesystems e.g. local filesystems, HDFS, Amazon S3 and more. Hadoop is written in Java, so most Hadoop filesystem interactions are mediated through the Java API.

HDFS nodes can also directly serve HTTP requests (instead of via the Java API) or HDFS Proxy's serve as an itermediary for HTTP requests to the filesystem.

Hadoop provides a C library that mirrors the Java FileSystem interface. It is possible to mount HDFs on a local filesystem using the NFS gateway or FUSE (Filesystem in Userspace); which then allows to use POSIX interaction methods.

4.3 Data Flow

Read The client accesses the filesystem via the API and emits RPCs (remote procedure calls) to determine the location of the first few blocks in a file. The Namenode returns the adresses of the corresponding Datanodes (which are sorted according to their proximity in the cluster, proximity proportional to available bandwidth between nodes approximated by heuristic). Data is then streamed from the Datanode back to the client. When a block is fully transmitted and find the best Datanode for the next block; if an error or corruption is encountered the next best Datanode is choosen. The client always contacts Datanodes directly after learning the adresses.

Write New files are created on the Namenodes filesystem namespace using RPCs with no blocks associated with it. The clients requests a a list of suitable Datanodes for storing and replication from the Namenode. The client then sends data in a pipelined fashion to the first Datanode, which in turn then continue the stream to the Datanodes for replication building a single pipeline for storing all replicas in one go. When all packets have arrived and have been acknowledged the client informs the Namenode of the succesful transfer. The Namenode knows the blocks for the file as the client asked for block allocations, so it only waits for minimal replication from the Datanodes to confirm the transfer.

Replica placement is driven by a reliability (spread) and read/write-bandwidth (keep close) tradeoff.

5 XML in a Nutshell (Chapters 2, 4.1, and 4.2) - jkleine

Source: https://proquest.safaribooksonline.com/0596007647

5.1 Introduction

An element is comprised of a start tag, an end tag and content in between, like so: <name>CONTENT<\name>. Tags can be nested to form a tree structure, however there can be only one root node. Names in XML can be comprised of any alphanumeric character combined with hyphens (-), underscores (\), period (.). Note, numbers, -, and . are not allowed to start the name. The content of tags can also contain other special characters, but notably never <, it must always be written as <. The other four escape sequences available in XML are:

- &: The ampersand (&)
- >: The greater-than sign, a.k.a. the closing angle bracket (>)
- ": The straight, double quotation marks (")
- ': The apostrophe, a.k.a. the straight single quote (')

Attributes An element can have attributes. These are defined in the start tag of element and are of the form attr_name="value". A attribute can be defined at most once per element. For example <mytag foo="abc" bar="def">...<\mytag> would be allowed, <mytag foo="abc" foo="def">...<\mytag> would not be allowed.

CDATA In case you want to include longer sections that include characters that need to be escaped, you may use a CDATA section. Anything written in between <![CDATA[and]]> will be interpreted as plain text (except for]]> itself for obvious reasons).

Comments Comments look like follows, they specifically cannot contain any double hyphens ("--"). <!-- My super cool comment -->

Processing Instructions XML provides a way of passing information to particular applications that may read the document in the form of processing instructions. These instructions may appear anywhere in the XML document. Processing instructions are enclused between <? and ?>. A good example of this is PHP, as seen in Figure 8.

A special case of this is the XML declaration which contains version and encoding information. <?xml version="1.0" encoding="ASCII" standalone="yes"?>

```
<?php
  mysql_connect("database.unc.edu", "clerk", "password");
  $result = mysql("HR", "SELECT LastName, FirstName FROM Employees
    ORDER BY LastName, FirstName");
  $i = 0;
  while ($i < mysql_numrows ($result)) {
     $fields = mysql_fetch_row($result);
     echo "<person>$fields[1] $fields[0] </person>\r\n";
     $i++;
  }
  mysql_close( );
?>
```

Figure 8: Processing Instructions Example

```
<rdf:RDF xmlns:rdf="http://www.w3.org/TR/REC-rdf-syntax#">
 <rdf:Description
      about="http://www.cafeconleche.org/examples/impressionists.xml">
    <title> Impressionist Paintings </title>
    <creator> Elliotte Rusty Harold </creator>
    <description>
      A list of famous impressionist paintings organized
      by painter and date
    </description>
    <date>2000-08-22</date>
  </rdf:Description>
</rdf:RDF>
```

Figure 9: Example of a namespace. Here the `rdf` namespace is defined only within the outer `rdf:RDF` tag

5.2 Name Spaces

Qualified Names, Prefixes, and Local Parts Since URIs contain special characters such as /, %, and ~, they are not allowed as XML names. Instead a prefix is associated with each URI. Prefixed elements look like this: `xsl:template`. They are separated by exactly one colon, everything before the colon is the *prefix*, everything after is the *local part*. The complete thing is calles the *qualified name*, *QName*, or *raw name*.

Prefixes are bound to namespace URIs by attaching an `xmlns:prefix` attribute to the prefixed element or one of its ancestors. Bindings have scope within the element where they're declared and within its contents. See Figure 9 for an example.

Namespace URIs The URIs do not have to point to anything, they are simply used as identifier to allow the user to change the prefix if needed. It doesn't even have to follow the `http` scheme, it might aswell follow the `mailto:` scheme.

However, it is a good idea to place some sort of documentation at the URI. It is also possible to place a Resource Directory Description Language (RDDL) document at the namespace URI, which might allow automated resolution in some cases.

Default Namespace It is possible to set a deafult namespace by using the `xmlns` attribute without a prefix. For example the tag `<svg xmlns="http://www.w3.org/2000/svg" width="12cm" height="10cm">` would set `http://www.w3.org/2000/svg` to be the default namespace within that tag. It behaves as if every XML name without a prefix had a prefix referencing the above URI. You can override the default namespace in a nested tag by using the `xmlns` attribute again.

Tags with prefix will ignore the name space and behave as usual.

6 The JSON Data Interchange Syntax - tgeorg

Source: `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

TL;DR: *Just defines JSON formally. If you know JSON syntax, you can happily skip this.*

Introduction JSON is language agnostic and allows for interchange of data between all programming languages. Syntax is defined by Unicode code points (characters). Numbers are represented in text format which makes them human readable and independent of constraints of binary representation.

Objects and arrays can nest which allows for representing trees in JSON.

6.1 Scope

Lightweight, text-based, and language-independent syntax based on ECMAScript (JavaScript).

6.2 Conformance

Conforming JSON text is a sequence of Unicode points that conform to the JSON grammar.

6.3 Normative References

Reference to Unicode and RFC8259, not relevant - skipped

6.4 JSON Text

Structural tokens:

- `[` left square bracket
- { left curly bracket
- `]` right square bracket
- } right curly bracket
- `:` colon
- `,` comma

Litaral tokens

- `true`
- `false`
- `null`

Whitespace is allowed before/after tokens.

6.5 JSON Values

A JSON value can be an object, array, number, string, `true`, `false`, or `null`.

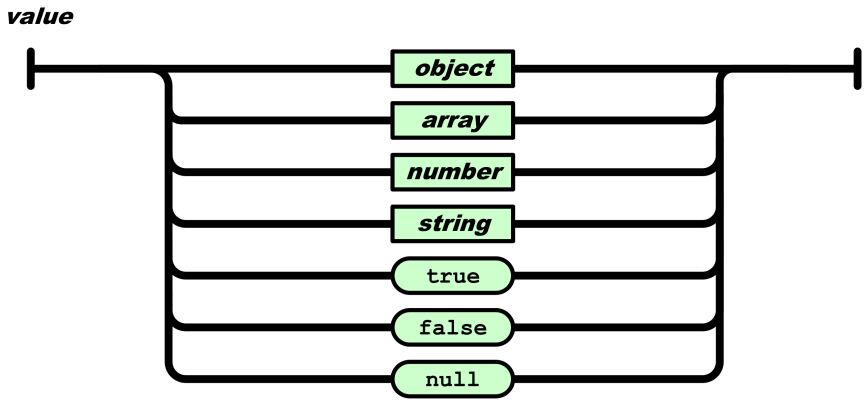


Figure 10: JSON value

6.6 Objects

Represented as a pair of curly bracket tokens surrounding zero or more name/value pairs.
Name is string, followed by token, followed by value. Additionally entries seperated by comma. Strings don't need to be unique.

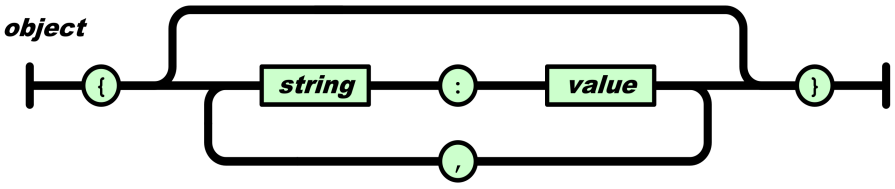


Figure 11: JSON object

6.7 Arrays

Sequence of values, separated by commas, surrounded by square brackets.

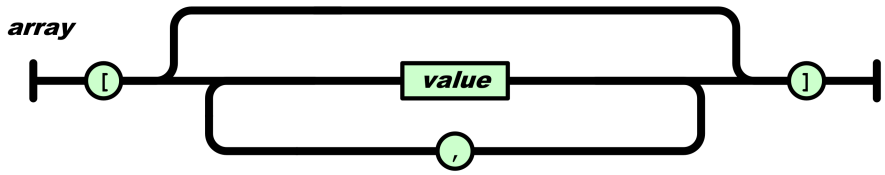


Figure 12: JSON array

6.8 Numbers

Decimal digits with no superfluous leading zero. Can have leading minus sign. Fractional part prefixed by decimal point. Exponent through prefix upper/lower case e and optional sign of exponent.

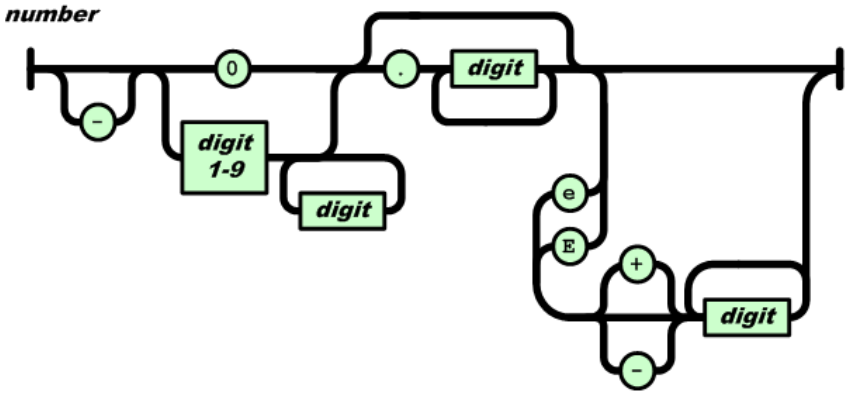


Figure 13: JSON number

6.9 String

Strings are surrounded by quotes. Backslash used as escape character.

- \" for quotes
- \\ for backslash
- \/ for forward slash
- \b for backspace character
- \f for form feed character

- \n for new line
- \r for carriage return
- \t for tab

For enter arbitary Unicode characters \u is used followed by the characters Unicode number in hexadecimal, e.g. \u002F. (Both upper and lower case characters can be used.)

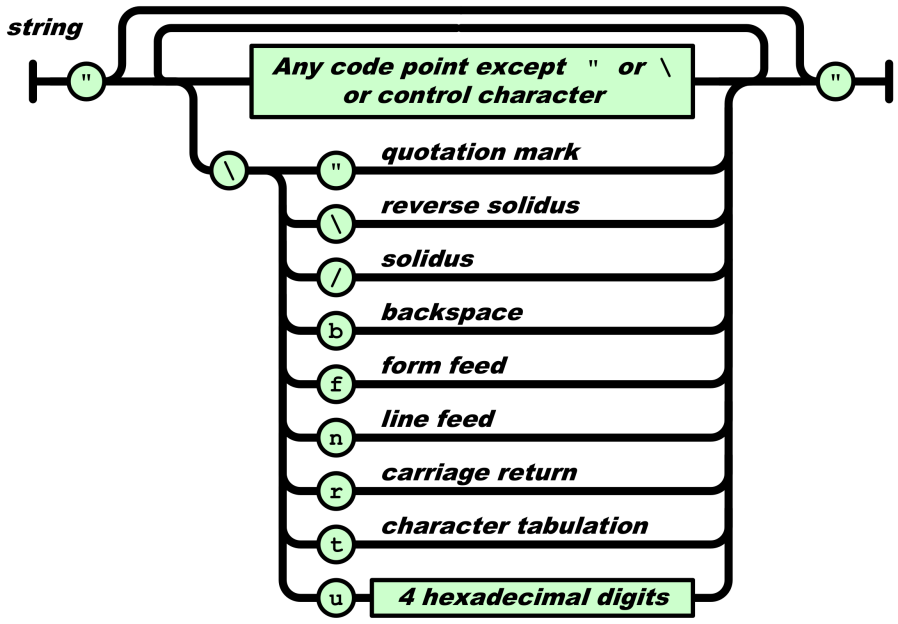


Figure 14: JSON string

7 Bigtable: A Distributed Storage System for Structured Data - burgerm

Source: `https://research.google.com/archive/bigtable-osdi06.pdf`

Bigtable (in the following referred to as *BT*) is a distributed storage system for managing *structured data* and is designed to scale to very large sizes (PB of data).

It does not support a full relational data model, instead providing clients with a simple data model that supports dynamic control over data layout and format. *BT* schema parameters let clients dynamically control whether to serve data out of memory or from disk, which allows to optimize latency-sensitive end user services or throughput-oriented batch processing.

7.1 Data Model

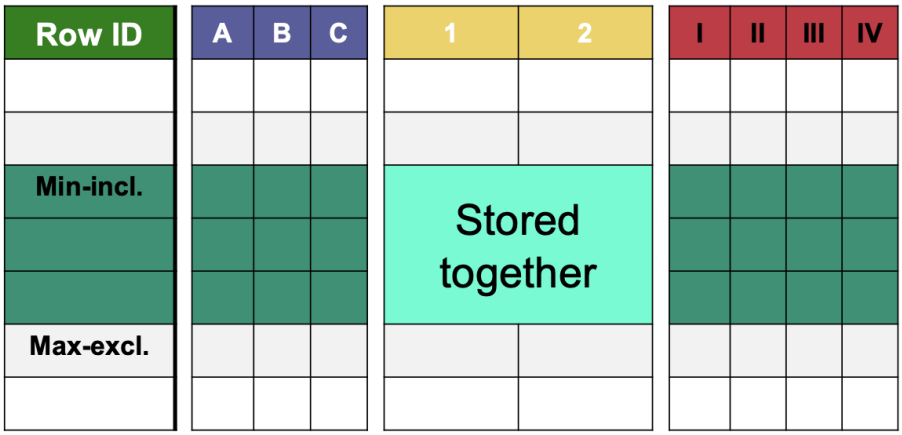


Figure 15: Bigtable Data Model - Table Layout

A *BT* is a sparse, distributed, persistent multi-dimensional map. The map is indexed (and sorted) by a row key, column key and a timestamp; each value in the map is an uninterpreted array of bytes. See Fig. 15 for the table layout.

Rows The row keys are arbitrary strings and each operation under a given row key is atomic. A *tablet* i.e. a range of row keys is the unit of distribution and load balancing. By choosing row keys which, in lexicographic order, group simultaneously accessed data together, performance can be optimized.

Column Families Column keys are grouped into sets called *column families* which form the basic unit of access control, as well as disk and memory accounting. All data stored in a column family is usually of the same type (data in same column

family is compressed together. Number of distinct column families in a table should be kept small and rarely change. But a table may have unlimited columns within these families and increase the column family size dynamically.

Timestamps Each cell can contain multiple version indexed by timestamps and sorted in decreasing order for efficient access to the most recent element. Garbage collection keeps the number of versions in check based on preferences (keep n most recent, keep 10 days back etc.).

7.2 API

Client app. can write or delete values, lookup indiv. rows or iterate over subset (range) of rows. Also supports single-transactions i.e. atomic read-modify write ops. under same row key. Client-supplied scripts can be run in the server address space to do inline data processing e.g. filter (but not write back). Tight integration with *MapReduce* is supported.

7.3 Structure

BT uses (distributed) GFS (Google File System) to store data. The Google *SSTable* file format is used to persist data on disk. *BT* relies on *Chubby* a Paxos-based lock service, which handles failures, replication, access control etc.

One master server assigns *tablets* to tablet servers, detects the addition and expiration of tablet servers, load balances the tablet servers and garbage-collects files from the underlying distributed file system (e.g. GFS); also handles schema changes. Client data does not go through the master; clients directly communicate with the tablet servers.

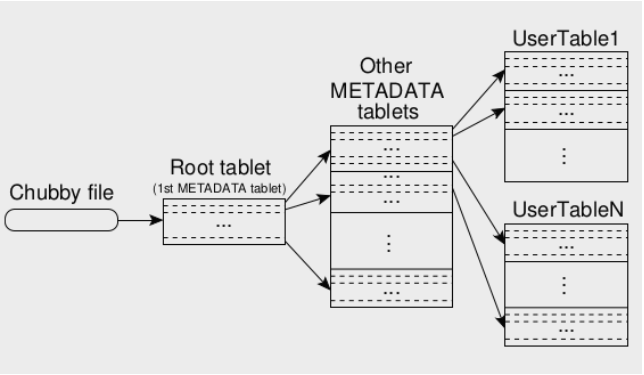


Figure 16: Tablet location store hierarchy on the master server

The master server stores the tablet locations in a hierarchy similar to a B+ tree as can be seen in Fig. 16.

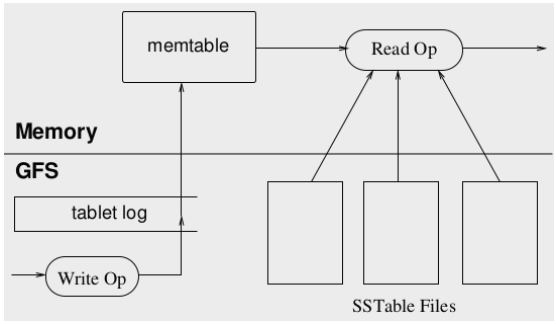


Figure 17: Tablet representation - persistent storage in GFS and fast memory storage in *memtable*

The tablet layout can be seen in Fig. 17. New data goes through a persistent commit log into a fast *memtable*, which is then periodically flushed out to persistent storage in *SSTable* files according to caching policies. Read requests are performed on a union of the *memtable* and the *SSTable* files. This process creates a new *SSTabe* file upon each memtable flush. *SSTabes* are periodically merged to keep the growth of the number of total *SSTable* files logarithmically.

7.4 Refinements

Locality groups allow clients to group often jointly accessed columns together and separate those that are usually not accessed together. This increases efficiency as unnecessary reads are prevented and writes have better locality.

Compression can be configured for the SSTable file's blocks; the block level compression trades off some loss in space efficiency for faster access to partial pieces of a SSTable file as the blocks can be decompressed separately.

Caching is performed in a *Scan Cache* which caches key-value pairs of SSTables (good for temporal locality) and a *Block Cache* caches SSTables read from GFS (good for spatial locality)

Bloom Filters can improve access times as they store the potential candidate SSTable files and memtables, where a value could be stored.

Commit-log implementation is tuned for performance and thus uses only one file for all tablets on a server; this however complicates recovery. Optimizations such as parallel sorting of the log file in case of recovery, two log files written by different threads to be able to switch to a different log file if performance is poor due to inconsistent GFS performance (log entries contain seq. numbers to order the two logs).

8 HBase: The Definitive Guide (Chapters 1 and 3)

- jkleine

Source: `http://proquest.safaribooksonline.com/9781449314682`

HBase stores data on disk in a column oriented format. This column oriented format is basically the only similarity to RDBMS. While RDBMS allow real-time analytical access to data, HBase excels at providing key-based access to a specific cell of data, or a sequential range of cells.

HBase takes a lot of concepts from Bigtable. Rows are organized in columns which are grouped in column families. While there can only exist a few tens of column families, each family can contain millions of columns. The columns with a column family are stored together and can be expanded dynamically. They also reside in the same low level storage, i.e. one column family can be stored in faster storage than another.

While Null values must explicitly be written in a RDBMS, in HBase the value can simply be omitted.

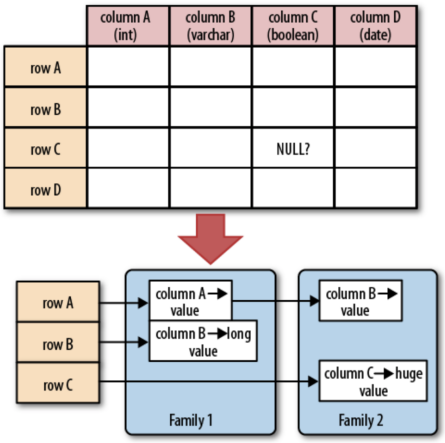


Figure 18: Rows and columns in HBase

Access to a single row is atomic, that is, any number of columns in that row can be read as written to atomically. However there are no atomic transactions spanning multiple rows.

Sharding Contiguous rows are stored in *regions*. These regions are stored on region servers. Each region is saved on exactly one *region server*, with one region server is responsible for ideally 10-1000 regions, each region being 1GB-2GB in size. If a region becomes to large/small it is automatically split/fused to better balance the load between servers.

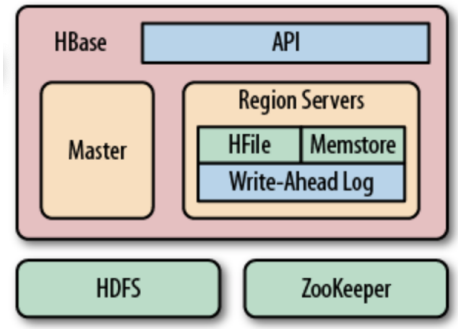


Figure 19: HBase implementation build upon existing systems

API There exists a *scan* API that allows to iterate over ranges of rows and be able to limit which columns are returned or the number of versions of each cell. Clients can also run client provided code in the address space of the server. This framework is called *coprocessor*. The code has access to the server local data and can be used to implement lightweight batch jobs, or use expressions to analyze or summarize data based on a variety of operators. Finally, the system is integrated with the *MapReduce* framework by supplying wrappers that convert tables into input source and output targets for MapReduce jobs.

Implementation The data is stored in store files, called HFiles, which are persistent and ordered immutable maps from keys to values. The store files are typically saved in the Hadoop Distributed File System (HDFS), which provides a scalable, persistent, replicated storage layer for HBase. When data is updated it is first written to a commit log, called a write-ahead log (WAL) in HBase, and then stored in the in-memory memstore. Once the data in memory has exceeded a given maximum value, it is flushed as an HFile to disk.

To assign regions to specific region servers the head server uses *Apache ZooKeeper*

8.1 API Basics

Put The `put` method expects one or a list of `Put` objects. A put object is is created from a row identifier (a byte array `byte[] row`, though methods are provided to convert other data types into byte arrays, essentially allowing arbitrary key values), but can additionally take a row lock and/or a time stamp. Once created, data can be added to the put object through the use of add methods. The add method call specifies the family and qualifier of the column, together with the actual data (again given as byte array). Optionally one can pass another timestamp, if this is not passed the timestamp used in the constructor will be used instead.

Alternative to specifying family, qualifier, and timestamp one may use the `KeyValue` object which identifies a cell in the three dimensional structure. `KeyValues` offer more functionality than this, which is omitted here.

Each put operation is effectively a remote procedure call, which is not feasible if a client is storing thousands of values per second. For this purpose HBase provides a client-side write buffer. It automatically collects multiple stores by region and sends them collectively to the correct region.

HBase also provides a special put operation calles `checkAndPut` which allows for atomic, server-side mutations (comparable to CAS).

Get Getting works very similar to setting. One specifies the row to create a `Get` object. This can then be narrowed down further by specifying a any one or more of the following: family, column, timestamp, and time range.

The call returns a `Result` object containing all the matching cells. Further details are omitted here.

Delete The delete functionality again follows the same principle, by first having to create a `Delete` object by specifying the row, which can then be further narrowed down by column family, column, timestamp, etc..

Analogous to the `checkAndPut` a `checkAndDelete` also exists.

Row Locks While normal operations like `put`, `delete`, and so on, are atomic, one might need more explicit control. If needed we can create a lock on a row in the first call and pass this lock to subsequent calls later on. Once not used anymore, the lock needs to be released again.

Scans Scan is similar to get, but working on a range of rows. Like previously shown, we can narrow down the selection by specifying a column (family) and so on. Additionally a filter can be supplied to filter out unwanted rows in the range of rows.

9 XML in a Nutshell (Chapter 17 except 17.3) - tgeorg

Source: https://proquest.safaribooksonline.com/0596007647

9.1 Overview

An XML Schema is an XML document containing a formal description of what comprises a valid XML document.

- An xsi:schemaLocation attribute on an element contains a list of namespaces used within that element and the URLs of the schemas with which to validate elements and attributes in those namespaces.
- An xsi:noNamespaceSchemaLocation attribute contains a URL for the schema used to validate elements that are not in any namespace.
- A validating parser is used to validate an XML document against a schema.

9.2 Schema Basics

Example: The schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="fullName" type="xs:string"/>
</xs:schema>
```

validates

```
<?xml version="1.0"?>
<fullName>Scott Means</fullName>
```

Note how the XML doc contains an element `fullName` which is referenced in the schema by `<xs:element name="fullName" type="xs:string"/>` with the name attribute in the schema matching the element in the XML.

`xsi:noNamespaceSchemaLocation` is used to tell the location of a schema without namespace, i.e.:

```
<?xml version="1.0"?>
<fullName xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="address-schema.xsd">
    Scott Means
</fullName>
```

Document Organization "Every schema document consists of a single root `xs:schema` element. This element contains declarations for all elements and attributes that may appear in a valid instance document."
"Instance elements declared using top-level xs:element elements in the schema (immediate child elements of the `xs:schema` element) are considered global elements.

The simple schema in the example above globally declares one element: `fullName`. According to the rules of schema construction, any element that is declared globally may appear as the root element of an instance document."

Annotations XML comments not used for annotations as they might be lost during parsing. Instead "To accommodate this extra information, most schema elements may contain an optional xs:annotation element as their first child element. The annotation element may then, in turn, contain any combination of `xs:documentation` and `xs:appinfo` elements, which are provided to contain extra human-readable and machine-readable information, respectively."

Example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:annotation>
        <xs:documentation xml:lang="en-US">
            Simple schema example from O'Reilly's
            <a href="http://www.oreilly.com/catalog xmlnut">XML in a Nutshell.</
            Copyright 2004 O'Reilly Media, Inc.
        </xs:documentation>
    </xs:annotation>
    <xs:element name="fullName" type="xs:string" />
</xs:schema>
```

"The `xs:documentation` element permits an xml:lang attribute to identify the language of the brief message. This attribute can also be applied to the `xs:schema` element to set the default language for the entire document."

`xs:documentation` used for human-readable content and `xs:appinfo` for application-specific extension information, e.g. to generate tool-tips in a GUI.

Element Declarations See figure 20

Attribute Declarations Attributes are referenced in the schema using the element `xs:attribute`, so `<xs:attribute name="fruit" type="xs:string"/>` defines an attribute `fruit` of type string.

9.3 Working with Namespaces

Target Namespaces Use attribute `targetNamespace` to associate schema with namespace.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://namespaces.oreilly.com/xmlnut/address">
```

Namespace of schema and content of XML doc must match. E.g. if schema requires some element `A` and that element in some XML doc is not in the same namespace as the schema, then the schemal will detect it as missing.

Type	Description
anyURI	A Uniform Resource Identifier
base64Binary	Base64-encoded binary data
boolean	May contain either true or false, 0 or 1
byte	A signed byte quantity >= -128 and <= 127
dateTime	An absolute date and time
duration	A length of time, expressed in units of years, months, days, hours, etc.
ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS	Same values as defined in the attribute declaration section of the XML 1.0 Recommendation
integer	Any positive or negative integer
language	May contain same values as xml:lang attribute from the XML 1.0 Recommendation
Name	An XML name
string	Unicode string

Figure 20: Built-in XML schema types

9.4 Complex Types

While simple types are basic building blocks (integers, strings, etc.) complex types are a combination of different types, kinda like objects in object-oriented programming. "Only elements can contain complex types. Attributes always have simple types."

XML:

```
<?xml version="1.0"?>
<addr:address xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://namespaces.oreilly.com/xmlnut/address address-schema
 xmlns:addr="http://namespaces.oreilly.com/xmlnut/address" addr:language="en"
>
  <addr:fullName>
      <addr:first>Scott</addr:first>
      <addr:last>Means</addr:last>
  </addr:fullName>
</addr:address>
```

Resulting schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://namespaces.oreilly.com/xmlnut/address"
 xmlns:addr="http://namespaces.oreilly.com/xmlnut/address"
 elementFormDefault="qualified"
>
    <xs:element name="address">
      <xs:complexType>
```

```
        <xs:sequence>
          <xs:element name="fullName">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="first" type="addr:nameComponent" />
                <xs:element name="last" type="addr:nameComponent" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:complexType name="nameComponent">
      <xs:simpleContent>
        <xs:extension base="xs:string" />
      </xs:simpleContent>
    </xs:complexType>
  </xs:schema>
```

"The nested element declaration is for the fullName element, which then repeats the xs:complexType and xs:sequence definition process. Within this nested sequence, two element declarations appear for the first and last elements."

Occurence Constraints Use attributes `minOccurs` and `maxOccurs` to set min/-max number of occurences.

```
<xs:element name="fullName">
  <xs:complexType>
    <xs:sequence>
    <xs:element name="first" type="addr:nameComponent" />
    <xs:element name="middle" type="addr:nameComponent" minOccurs="0" />
    <xs:element name="last" type="addr:nameComponent" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

9.5 Empty Elements

element has no content. All data is stored in attributes. Use complex type that only defines the attribute but no further elements → element must be empty to conform schema.

9.6 Simple Content

"The `xs:simpleType` element can define new simple data types, which can be referenced by element and attribute declarations within the schema."

Defining New Simple Types Example:

```
<xs:complexType name="contactsType">
  <xs:sequence>
    <xs:element name="phone" minOccurs="0">
      <xs:complexType>
        <xs:attribute name="number" type="xs:string" />
        <xs:attribute name="location" type="addr:locationType" />
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="locationType">
  <xs:restriction base="xs:string" />
</xs:simpleType>
```

Facets "A facet is an aspect of a possible value for a simple data type." Facet types that are supported by a schema processor:

- `length` (or minLength and maxLength) (restricts length of string)
- `pattern` (to enforce a specific format, e.g. `"\d\d-\d\d\d-\d\d\d"` for Legi number)
- `enumeration` (only allows predefined valid options)
- `whiteSpace` (preserve, replace, collapse)
- `maxInclusive`, `maxExclusive`, `minInclusive`, and `minExclusive` (to restrict range of numeric values)
- `totalDigits` (total number of digits allowed in number)
- `fractionDigits` (`totalDigits` after the decimal point)

Facets are applied to simple types using the `xs:restriction` element. Use lists to allow multiple of same type (e.g. multiple middle names) and unions to allow multiple types (e.g. accept both strings and integers)

9.7 Mixed Content

Allowing Mixed Content By setting the attribute `mixed="true"` of a complex type we say that the elements contained in this complexType can be of different types.

Controlling Element Placement

- `xs:sequence` for a sequence of elements listed in the same order as the schema
- `xs:all` same as `xs:sequence` but without the order restriction
- `xs:choice` for enumeration, i.e. only values allowed are those listed within

Using Groups `xs:group` can be used to define a structure of elements and reference it to reuse it. Kinda like functions are used in programs to reuse code.

9.8 Allowing Any Content

Use `xs:any` if you don't care about the content type.

Using Multiple Documents Use `xs:include` with attribute `schemaLocation` to "import" other schema files and re-use their content.

If you want to adjust the imported schema, use the element `xs:redefine` instead of `xs:include`. To import the schema under a different namespace, use `xs:import` with attributes `schemaLocation` and `namespace`.

Derived Complex Types The element `xs:extension` can be used to extend parts of the imported schema similar to object-oriented programming where you can create a subclass of and object and extend it.

9.9 Controlling Type Derivation

Similar to object-oriented programming "the schema language allows schema authors to place restrictions on type extension and restriction."

Abstract Elements and Types "The abstract attribute applies to type and element declarations. When it is set to true, that element or type cannot appear directly in an instance document."

The Final Attribute The attribute `final` allows to prevent extensions of an element in the schema.

Uniqueness and Keys "The `xs:unique` element enforces element and attribute value uniqueness for a specified set of elements in a schema document." I.e. no same value for element and attributes where uniqueness is enforced.
`xs:key` also enforces uniqueness but unlike `xs:unique` gives an error if no value exists.

10 Understanding JSON Schema - burgerm

Source: `https://json-schema.org/understanding-json-schema/`

So this is schema for JSON in JSON. Basically for each field instead of the value you would put there, you put the type and then there are some special cases and syntactic sugar features.

{} **or** `true` accepts any valid JSON string. Similarly will `false` reject any input.

Types can be declared as follows: { `"type": "string"` }. To enforce the type on a value, the value is replaced with the previous object i.e. a key-value pair with keyword `type` and the desired type as a value.

Declaration To declare a JSON file/object as a schema the `$schema` can be set e.g. { `"$schema": "http://json-schema.org/schema"` }

Unique identifier It is best practice to also include a unique identifier for each schema by setting `$id` e.g. { `"$id":"http://yourdomain.com/schemas/myschema.json"` }.

10.1 Reference

Types available are:

- string
- Numeric types (arbitrarily long numbers, fractionals etc. must all be handled by the user upon loading the data)
- object
- array
- boolean
- null

Multiple different types for a single field can be accepted by passing an array to the `type` e.g. { `"type": ["number", "string"]` }.

Strings Length can be constrained as follows

```
{
  "type": "string",
  "minLength": 2,
  "maxLength": 3
}
```

There is regex support by setting the `pattern` key on the string object. By setting the `format` key one can select amongst a range of predefined patterns/formats such as dates, times, emails, network addresses etc.

Numbers i.e. `integer` or `number`. The following keywords allow to enforce multiples and ranges:

- multipleOf
- minimum
- exclusiveMinimum
- maximum
- exclusiveMaximum

Objects are enforced by setting { `"type": "object"` }, which allows any valid object as a value. By setting the `property` key a complete substructure for the object can be enforced as in the following example Listing 1:

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    }
  }
}
```

Listing 1: More complex structure for objects can be enforced by setting `property`

The `additionalProperties` key is used to allow/disallow more properties on the object. `required` key expects an array of previously defined properties, which are required. `propertyNames` can be passed a regex to only allow specific names. Size and dependencies of some properties on others can also be configured. A regex in the property name can be used to enforce constraints for specifically named properties as seen in the following Listing.

```
{
  "type": "object",
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  },
  "additionalProperties": false
}
```

array are used for ordered elements. The `item` key is used to set the type of the contained elements as follows:

```
{
  "type": "array",
  "items": {
    "type": "number"
  }
}
```

Alternatively the `contains` key can be used to ensure the array is non-empty. An ordered tuple can be defined by passing multiple type objects to the array's `items` key as in the following Listing:

```
{
  "type": "array",
  "items": [
{
      "type": "number"
    },
{
      "type": "string"
    },
    {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
}, {
      "type": "string",
      "enum": ["NW", "NE", "SW", "SE"]
    }
] }
```

Can forbid additional items with **additonalitems**, fix length by using `minItems` and `maxItems` and ensure uniqueness with `uniqueItems`.

boolean by setting { `"type": "boolean"` } and it does not have any additional properties

null by using { `"type": "null"` }

Annotations to the schema without influencing the validation can be done by using any of these: `title, description, default, examples`. Newer version allow for comments by setting `$comment` key.

Enumerations by using `enum` as seen in Listing 1 can be used together with a type or also as self-standing type and even with a mixed type selection.

Various `concentMediaType` allows to set the MIME type for encoded data of a string. `contentEncoding` specifies the used encoding. The use of `allOf, anyOf, oneOf, not` and passing an array of schemas, allows to combine schemas. Conditionals can also be included as in the following Listing:

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string"},
    "country": {
      "enum": ["United States of America", "Canada"]
    }
  },
  "if": {
    "properties": { "country": { "const": "United States of America" } }
  },
  "then": {
    "properties": { "postal_code": { "pattern": "[0-9]{5}(-[0-9]{4})?" } }
  }, "else": {
    "properties": { "postal_code": { "pattern": "[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" } }
  }
}
```

By using the `$ref` key we can refer to schemas defined elsewhere and it allows to do recursive validation of schemas.

The `$id` property is a URI-reference that serves two purposes:

- It declares a unique identifier for the schema.
- It declares a base URI against which `$ref` URI-references are resolved.

11 Dremel: Interactive Analysis of Web-Scale Datasets.

- jkleine

Source: `https://dl.acm.org/doi/10.1145/1953122.1953148`

Dremel is a scalable, interactive ad hoc query system for analysis of read-only nested data. 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.

11.1 Introduction

Performing interactive data analysis at scale demands a high degree of parallelism. For example, reading a terabyte of compressed data from secondary storage in 1s would require more than 10,000 commodity disks. The same applies for compute intensive tasks.

Since Dremel is designed for a cluster environment a single worker might take much longer than others or never complete due to failure.

The data queries id often non relational. Hence a flexible data model is essential.

Dremel supports a high-level, SQL-like language to express ad hoc queries. Unlike other query systems (such as Pig and Hive) it executes the queries naively and does not translate them into a series of map reduce jobs.

11.2 Data Model

The datatype is based on strong typed nested records. the syntax is given by:

$$\tau = \textbf{dom} | \langle A_1 : \tau[*|?], \ldots, A_n : \tau[*|?] \rangle$$

τ is a type. It is either an atomic type (in **dom**), such as integer, floating-point numbers, strings, etc. or a record consisting of one or more fields. Field i in a record has name A_i and a value defined again by τ. The field has a associated carnality, either it must exist exactly once (no symbol), it may occure multiple times ($*$), or it is optional, i.e., may be missing ($?$). Apparently this notation is never used, but instead what is shown in Figure 22a.

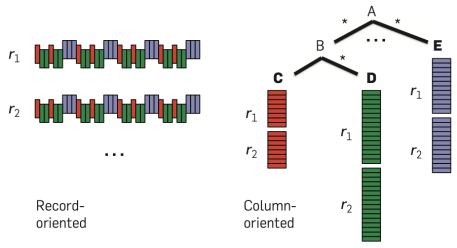
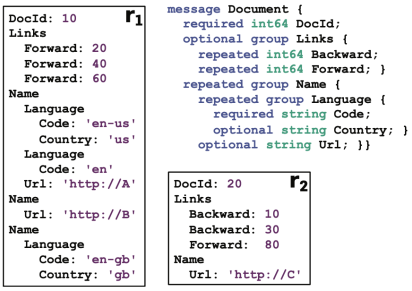
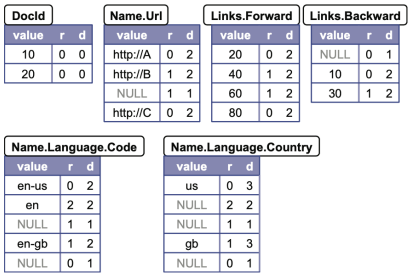


Figure 21: Record oriented storage (left, data of a record is stored together) vs. column oriented storage (right, all values of given filed are stored together).



(a) Top right: Schema for *Document*; Rest: Records r_1, and r_2



(b) Column-striped representation of the data in Figure 22a

Figure 22: Example

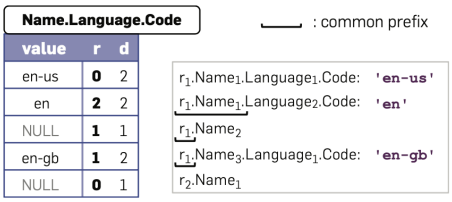


Figure 23: Encoding of the `Name.Language.Code` field of the records in Figure 22a

11.3 Nested Columnar Storage

The goal is to store all values of a given field consecutively to improve retrieval efficiency. Further it must be possible to assemble records efficiently for any given subset of columns.

Repetition and definition levels Figure 22b shows the records in Figure 22a in columnar format. For each value a *repetition level* (r) and a *definition level* (d) is stored.

The encoding works as follows, shown using the example of the `Name.Language.Code` filed of the records in Figure 22a. The encoding is shown in Figure 23. First, we only look at the fields `Name`, `Language`, and `Code`. Second, we represent the stripped records as a list of root-to-leaf paths. The subscripts denote positions of the respective fields within their enclosing records. The repetition level encodes the length of the common prefix of p_{i-1} and p_i, while definition level encodes the length of p_i's suffix (i.e. what is left after the common prefix).

The repetition level is defined as the number of repeated fields in the common prefix (including the first path element identifying the record). The definition level specifies the number of optional and repeated fields in the path. Required fields are not counted as they are always present. A definition level smaller than the maximal number of repeated and optional fields in a path denotes a NULL.

A table is stored as a set of *tablets*. A tablet is a self-contained horizontal

partition of the table. In addition to the actual data, the tablet contains the schema and extra metadata that includes specification of keys, sorting order, value ranges, etc. There is some additional compression by leaving out implicit values (not further summarized here).

Record Assembly To allow applications like Map-Reduce, a subsets of fields must be able to construct the original records as if they were the only records in the field. For this one can construct a finite state machine that can start at any point of the list and reconstruct all pairs after that point. (not further summarized)

11.4 Query Execution

Using an SQL like query language one can execute queries on the data. The query is translated into a query execution tree which is then split across multiple nodes to form a serving tree.

Basically the the query is split across multiple layers of execution and multiple partitions of the data. The leave node of the serving tree are the storage nodes, which perform the initial part of the query, relaying their results to the next higher level.

Query Dispatcher Since Dremel is a multi user system the queries are staged and dispatched based on priorities.

11.5 Experiements

Dremel go Brrrrr...

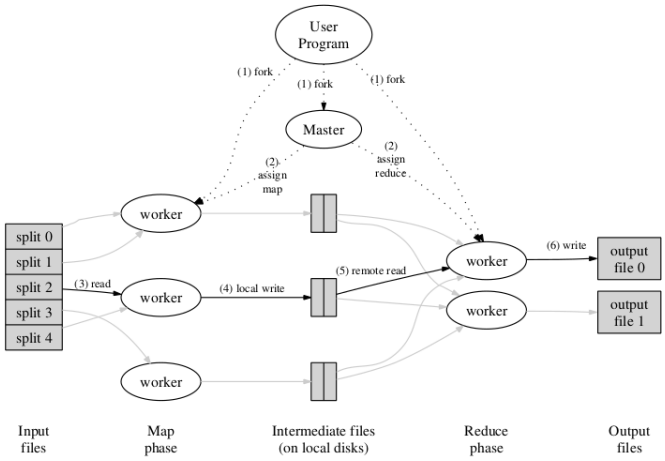


Figure 24: Map Reduce Overview

12 MapReduce: Simplified Data Processing on Large Clusters) - tgeorg

Source: `https://research.google.com/archive/mapreduce-osdi04.pdf`

MapReduce is a framework to run highly parallelisable computations on many machines made from commodity hardware, while simultaneously keeping the complexity of creating parallel jobs simple.
Most computations are conceptually straightforward but the logic around them (how to distribute data, failsafes, etc) is not. MapReduce as a framework tries to provide the logic to allow for easily distributing data and instructions without having to worry about the setup.

12.1 Programming Model

MapReduce consists of two functions:

1. In the mapping phase input key/value pairs are taken and a set of intermediate key/value pairs are created.
2. In the mapping phase a key I and key/value pair associated with that key are used to produce the output value(s).

Example: Wordcount

- **Map:** output 1 for each word
- **Reduce:** sum up all integers

12.2 Implementation

Input data partitioned into M splits which are processed in parallel. The intermediate data produced by the map job is then partitioned into R splits using a hash function.
When a MapReduce job is started the program holding the instructions for the map and reduce functions is copied to all machines that are part of the job. One of the copies of the program is special and is called the "master".

The Master keeps track of the progress of individual machines. It also propagates the location of data of completed jobs to the workers.

12.2.1 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 workers periodically. If a worker connection times out the progress of the concerning partition is reset and the partition is assigned to a new worker. Other workers are notified to read the data from the new machine.
As jobs from offline workers get re-executed by available ones, large-scale worker failures causes no data-loss but simply increases the time until the result of the MapReduce job becomes available.

Master Failure Master can write checkpoints to an off-machine storage medium which a new master can use to resume work if the old one dies. Otherwise a MapReduce job can also be aborted and restarted.

Semantics in the Presence of Failures As long as the map and reduce operators are deterministic, the parallel execution of a MapReduce job is equal to its sequential execution which simplifies reasoning about correctness.
If the operators are non-deterministic there exists a weaker guarantee that the result of a particular reduce task R_1 is equal to the sequential execution of R_1 but a different reduce task R_2 corresponds to the output for R_2 produced by a different sequential execution of the non-deterministic program.

12.2.2 Other implementation details

Locality When assigning jobs the master takes into account the location of data and nodes to reduce overall network usage.

Task Granularity By making the number map and reduce partitions (R and M) greater than the number of machines, multiple partitions can dynamically assigned to the same machine to allow for load balancing (i.e. weaker machines get less

partitions and faster machines get more). The upper bound of the number of partitions is limited by the masters ability to perform scheduling decisions.

Backup Tasks Slowdown of a MapReduce job is frequently caused by a single machine taking significantly longer than the rest. As the job is only completed when all partitions have been processed, a MapReduce job close to completion will run the same remaining partitions on multiple machines and take the result of the first machine that completes. This prevents a single machine from slowing down the job.

12.3 Refinements

Within a given partition the key value pairs are processed in increasing key order.

Combiner Function A *combiner function* is reduce task run on the machine running the ,mapping function to reduce the amount of data sent over to network to the machine executing the reduce task

Skipping Bad Records Some records may cause crashes of the map/reduce function. If a crash happens the worker sends a signal to the master that indicates the record causing the crash. If multiple signals with the same record a received, the master indicates to skip this record.

Status Information Master runs a HTTP server that gives human readable information on the progress of the MapReduce job.

(Chapters *Performance*, *Experience*, *Related Work*, and *Conclusions* skipped, due to not being relevant for knowing how MapReduce works)

13 HBase: The Definitive Guide (Chapter 7) - burgerm

Source: `http://proquest.safaribooksonline.com/9781449314682`

One of the great features of HBase is its tight integration with Hadoop's MapReduce framework.

13.1 MapReduce

Solves the problem of processing in excess of terabytes of data in a scalable way. MapReduce follows a divide-and-conquer approach by splitting the data located on a distributed filesystem across the available machines and cores; at the end of the processing data needs to be aggregated as per the requirements of the computation.

Splits The framework first creates *splits* of the input, which can be distributed (need to be choosen in a smart way and in accordance with the input format). The `InputFormat` class is responsible for this splitting and creation of the *key-value* pairs; the unit of processing for the further steps. HBase provides tight integration with MapReduce and we can efficiently split an entire table stored in HBase.

Mapper The `Mapper` classes are responsible for processing each *key-value* pair and emitting other (potentially of different types) pairs. This step essentially transforms the input into a different format more suited for further processing. Again there are a couple of pre-implemented Mappers for integration with HBase e.g. the `IdentityTableMapper` simply passing through the key-value pairs of the table.

Reducer The `Reducer` stage receives the output of the `Mapper` stage after the data has been *shuffled* and/or *sorted* according to the configured partitioner and setup. Each reducer is assigned a set of shuffled (i.e. random) keys; this subset is received by the `Reducer` in sorted order with all the values corresponding to any of the assigned keys as per output of the mapping stage.

OutputFormat in a final stage the job persists the data in various locations. HBase exposes e.g. the `TableOutputFormat` class to write a specific HBase output table. Altough many `Mappers` and `Reducers` can process the data only a single `OutputFormater` takes each output records from its Reducer subsequently and writes the to a file or table.

TableMapReduceUtil class helps in setting up MapReduce jobs over HBase by providing a set of static methods.

13.1.1 Locality

Hadoop handles block replication automatically on the underlying distributed filesystem (e.g. HDFS). As the data in HDFS is split up in blocks Hadoop can run MapReduce tasks close to the data they process e.g. on the same node where the data

is stored in HDFS. Each block is assigned to a map task to process the contained data; larger blocks means fewer tasks. Due to block replication the framework can choose multiple locations for the same block to be processed.

By running regular compactions (or trigger global compactions them manually) and integration with HDFS, HBase keeps data of the tables on the server responsible for the data or a collocated server. Thus MapReduce tasks running over HBase together with HDFS have similar data locality guarantees.

One has to be careful with Region movements due to load balancing as they temporarily violate data locality, but locality will be again restored after running compactions.

13.2 Integration of MapReduce and HBase

Lots of *blabla* on how to actually set up MapReduce tasks to run over HBase and various code examples to interact with HBase data with MapReduce.

14 Apache Hadoop YARN: Yet Another Resource Negotiator - burgerm

Source: `http://doi.org/10.1145/2523616.2523633`

The goal of YARN is the seperation of resource management functions and the programming model. The goal of YARN was to satisfy the following properties:

1. Scalability
2. Multi-tenancy (Multiple clients share same application and hardware resources)
3. Serviceability
4. Locality awareness (for performance)
5. High cluster utilization
6. Reliability/Availability
7. Secure and auditable operation
8. Support for programming model diversity (i.e. not only MapReduce style communication and programming patterns)
9. Flexible resource model (to be able to change allocated amount of resources to a job while running)
10. Backwards compatibility

14.1 Architecture

YARN uses a per cluster *ResourceManager* (RM), which tracks resource usage and node liveness, enforces allocation invariants, and arbitrates contention among tenants. An *ApplicationManager* then requests resources from the RM, generates a physical plan from the received resources, and coordinates the execution around faults.

Overview The RM runs on a dedicated machine and can enforce fairness, capacity and locality across tenants. It schedules and dynamically allocates *containers* to applications to run on particular nodes; a container being a logical bundle of resources e.g. $2GBRAM, 1CPU$ bound to a node. On each node runs a *NodeManager* which exchanges heartbeats with the RM and is responsible for monitoring resource availability, reporting faults, and container life cycle management (start, kill, ...). Jobs are submitted to the RM and go through an admission control phase. Accepted jobs are passed to the scheduler and once enough resources are available the job is moved from accepted to running state; meaning allocating a container for the AM and spawning it on a node. The AM is the head of the job and manages all lifecycle aspects. It can be written in any language and run arbitrary user code.

It communicates with the RM and NM via an encoded extensible communication protocol. The AM requests the required nodes for a job from the RM, which generates leases for containers on different nodes to the AM. The token lease is then presented to the NM by the AM.

Resource Manager (RM) Handles clients submitting applications, AMs requesting resources, and communication with the NMs for monitoring. One or more `ResourceRequests` by the AMs consist of:

- #containers
- resources per container (memory, CPU etc.)
- locality preferences
- priority of requests within the application

The RM also generates containers together with access tokens to make resources accessible. The RM can also symmetrically request resources back from an AM, if cluster resources become scarce. The AMs can then decide how to handle the preemption request e.g. checkpoint and then yield the container.

Application Master (AM) An application can be a static set of processes, a logical description of work, or even a long-running service. It is run itself in a container of the cluster and coordinates the execution of the application across the cluster. The AM communicates with the RM via heartbeat messages. It encodes its preferences and constraint for resources in those msg. and the RM responds with container *leases*. The actual process spawned is note bound to the request, but to the *lease*; meaning the conditions of the request may not remain true, but the ones granted in the lease. The AM can also update requests dynamically adjusted to the already received grants, which affected present and future requirements.

As the AM itself is running in a container of the cluster it should be resilient to failure.

Node Manager (NM) authenticates container leases, manages containers' dependencies, monitors their execution, and provides a set of services to containers.

It registers with the RM and reports status/metrics in heartbeats. All containers are described by a *container launch context* (CIC) (env. variables, dependencies, startup sequence, etc.). The NM configures the environment accordingly for the container. NM garbage collects dependencies no longer in use and kills containers as per command of the AM or RM. The NM periodically monitors the health of the node e.g. disk issues. It also provides services such as *log aggregation*, which stores application output in HDFS after the application terminated. NMs can expose *auxiliary services* such as persisting data across container launches within the same application.

YARN framework/application writer The Application Master can be written in any language and run any code as long as it adheres to the standardized communication interface and communicates with the RM. It should also be resilient against failures as it is run itself inside an unreliable container unit of the cluster.

Fault tolerance and availability The RM is a single point of failure in YARN and recovers its state from persistent store on initialization (in paper release version). It then kills all running containers (including live AMs) and launches new instances of each AM, who need to deal with the fault within their application themselves. AMs surviving an RM crash is a work in progress.

If an NM fails, the RM marks all the containers on the node as killed when the heartbeat response times out and reports to the AMs. For transient faults the NM can re-synchronize wit the RM.

Since the AM itself is running within the cluster it does not affect the availability of the cluster, but is itself affected by instabilities of the cluster (e.g. failing RM). The AM is responsible for handling application fault tolerance due to failing containers, requesting new containers etc.

14.2 Practical

YARN log aggregation can (especially for large jobs) in large clusters put a lot of pressure on the NameNode of the underlying HDFS file system and thus this NameNode can become a bottleneck for scaling up cluster size.

Lots of stuff on how great YARN is doing at Yahoo and how all important applications have been ported with amazing performance.

15 Dominant Resource Fairness: Fair Allocation of Multiple Resource Types - jkleine

Source: `https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Ghodsi.pdf`

The intuition behind Dominant Recourse Fairness (DRF) is that in a multi-resource environment, the allocation of a user should be determined by the user's dominant share, which is the maximum share that the user has been allocated of any resource. In a nutshell, DRF seeks to maximize the minimum dominant share across all users.

Principal Manage a job by its most dominant resource. For example if a job requires 10% of the available CPU and 30% of the available Memory, treat it like the job requested 30% of the cluster. Then try to maximize the overall allocation of the system, while trying to match the dominant shares the the theoretical shares of the system, and making sure the total doesn't exceed any one recourse (i.e. if 100 GB are available, don't allocate 120 GB...). One can formulate this as LP but I couldn't be bothered.

Example Assume the resources (9 cores, 18 GB memory) are supposed to be split 50/50, user A runs jobs that need 1 CPU and 4GB memory each, while user B runs jobs that require 3 CPUs and 1 GB memory each. The dominant recourse utilization are $4/18 = 2/9$ for user A and $3/9 = 1/3$ for user B. We now try to allocate as many tasks as possible, while balancing the dominant recourse as given by the 50/50 split. The result is that user A can run 3 jobs and user B can run 2. This way the dominant resources are distributed as follows: A: $3 \cdot 2/9 = 2/3$, and B: $2 \cdot 1/3 = 2/3$.

Note that even though both users seem to be allocated two thirds of the system, this is not the case. The goal is just to keep these number in relation to the reservation of the system (i.e. 50/50, so equal in this case).

The actual utilization of the system is thus $3 \cdot 1 + 2 \cdot 3 = 9$ CPUs, and $3 \cdot 4 + 2 \cdot 1 = 14$ GB memory.

Properties The following properties should be fulfilled by a resource manager:

- *Sharing incentive:* Each user should be better off sharing the cluster, than exclusively using her own partition of the cluster. Consider a cluster with identical nodes and n users. Then a user should not be able to allocate more tasks in a cluster partition consisting of n1 of all resources.
- *Strategy-proofness:* Users should not be able to benefit by lying about their resource demands. This provides incentive compatibility, as a user cannot improve her allocation by lying.
- *Envy-freeness:* A user should not prefer the allocation of another user. This property embodies the notion of fairness [13, 30].

- *Pareto efficiency:* It should not be possible to increase the allocation of a user without decreasing the allocation of at least another user. This property is important as it leads to maximizing system utilization subject to satisfying the other properties.
- *Single resource fairness:* For a single resource, the solution should reduce to max-min fairness.
- *Bottleneck fairness:* If there is one resource that is percent-wise demanded most of by every user, then the solution should reduce to max-min fairness for that resource.
- *Population monotonicity:* When a user leaves the system and relinquishes her resources, none of the allocations of the remaining users should decrease.
- *Resource monotonicity:* If more resources are added to the system, none of the allocations of the existing users should decrease.

Dominant Recourse fairness achieves all but the last property, which is also not satisfied by the systems they use for comparison.

16 Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing - tgeorg

Source: `https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf`

Resilient Distributed Datasets (RDDs) are "a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner"

16.1 Introduction

Some cluster computing frameworks like MapReduce lack abstractions for leveraging distributed memory. There "the only way to reuse data between computations is to write it to an external stable storage system" which causes heavy I/O usage that can dominate application execution times.
As data reuse is common in some workloads such as iterative machine learning and graph algorithms a new framework is needed to provide efficient data reuse without causing heavy I/O while maintaining fault-tolerance. "The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently."
Existing solutions provide consistency for fine-grained changes (e.g. updating cell in table). "In contrast to these systems, RDDs provide an interface based on coarse-grained transformations. [...] This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition."
"RDDs are a good fit for many parallel applications, because these applications naturally apply the same operation to multiple data items."

16.2 Resilient Distributed Datasets (RDDs)

RDD Abstraction "An RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either data in stable storage or other RDDs." These operations are called *transformations*.
"RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage."

Advantages of the RDD Model "The main difference between RDDs and distributed shared memory (DSM) is that RDDs can only be created (written) through coarse-grained transformations, while DSM allows reads and writes to each memory location. This restricts RDDs to applications that perform bulk writes, but allows for more efficient fault tolerance. [...] Furthermore, only the lost partitions of an RDD need to be recomputed upon failure, and they can be recomputed in parallel

on different nodes, without having to roll back the whole program."
"A second benefit of RDDs is that their immutable nature lets a system mitigate slow nodes (stragglers) by running backup copies of slow tasks as in MapReduce."

Further, "in bulk operations on RDDs, a runtime can schedule tasks based on data locality to improve performance" and "RDDs degrade gracefully when there is not enough memory to store them, as long as they are only being used in scan-based operations. Partitions that do not fit in RAM can be stored on disk and will provide similar performance to current data-parallel systems."

Applications Not Suitable for RDDs Because RDDs perform updates on a coarse-grained level, it is not well suited for applications that make asynchronous fine- grained updates to shared state

16.3 Spark Programming Interface

"To use Spark, developers write a driver program that connects to a cluster of workers. The driver defines one or more RDDs and invokes actions on them." Spark code on the driver tracks RDDs' lineage. "Workers are long-lived processes that can store RDD partitions in RAM across operations."

16.4 Representing RDDs

The interface for RDDs exposes five pieces of information:

1. `partitions()`: Return a list of Partition objects
2. `preferredLocations(p)`: List nodes where partition p can be accessed faster due to data locality
3. `dependencies()`: Return a list of dependencies
4. `iterator(p, parentIters)`: Compute the elements of partition p given iterators for its parent partitions
5. `partitioner()`: Return metadata specifying whether the RDD is hash/range partitioned

Dependencies are split into *narrow* and *wide* dependencies.

- **narrow:** "each partition of the parent RDD is used by at most one partition of the child RDD"
- **wide:** "where multiple child partitions may depend on it."

"This distinction is useful for two reasons. First, narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis. In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReducelike operation.

Second, recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution."

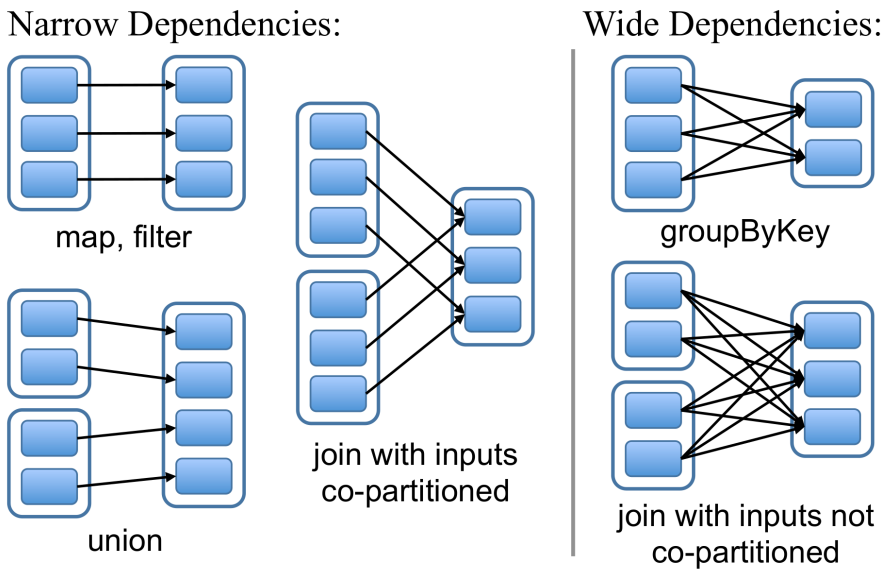


Figure 25: RDD Dependencies

16.5 Implementation

Job Scheduling "Whenever a user runs an action (e.g., count or save) on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of stages to execute."
"If a task needs to process a partition that is available in memory on a node, we send it to that node. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations (e.g., an HDFS file), we send it to those.[...] If a task fails, we re-run it on another node as long as its stage's parents are still available. If some stages have become unavailable (e.g., because an output from the "map side" of a shuffle was lost), we resubmit tasks to compute the missing partitions in parallel."

Memory Management "Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage."
"The first option provides the fastest performance, because the Java VM can access

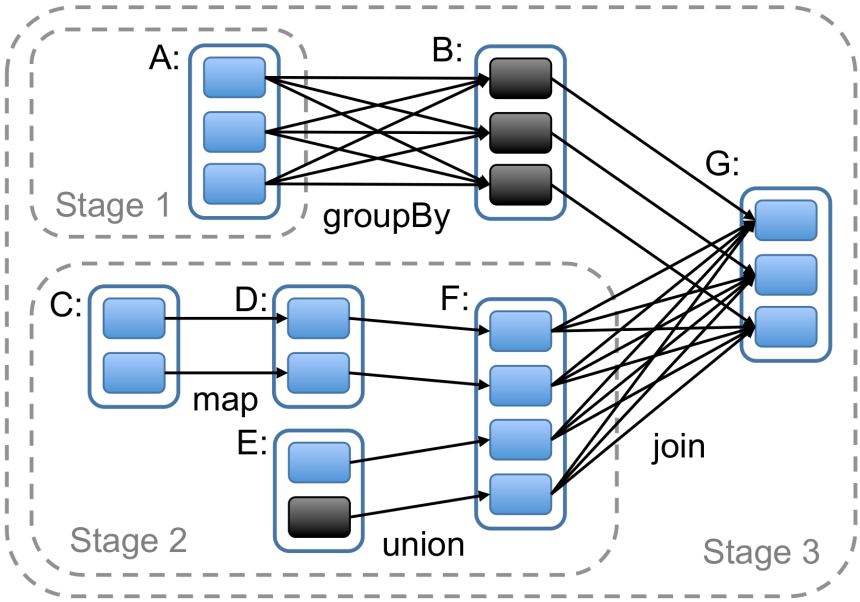


Figure 26: Black boxes are RDD in RAM. B has already been computed and is in RAM hence Stage 2 is executed and then using the result (F) and B, G is produced in Stage 3

each RDD element natively. The second option lets users choose a more memory-efficient representation than Java object graphs when space is limited, at the cost of lower performance. 8 The third option is useful for RDDs that are too large to keep in RAM but costly to recompute on each use." To manage limited memory, an LRU eviction policy at the level of RDDs is used.

Support for Checkpointing "Although lineage can always be used to recover RDDs after a failure, such recovery may be time-consuming for RDDs with long lineage chains. Thus, it can be helpful to checkpoint some RDDs to stable storage."

16.6 Discussion

Expressing Existing Programming Models RDDs can efficiently express a various cluster programming models like MapReduce and performing distributed SQL queries.

Leveraging RDDs for Debugging For fault tolerance, RDDs are designed to be deterministically recomputable which allows for easy replaying of computations. This property also facilitates debugging.

16.7 Conclusion

Resilient distributed datasets (RDDs) "can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation, and new applications that these models do not capture. Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage."

17 Spark SQL: Relational Data Processing in Spark

- **burgerm**

Source: `https://doi.org/10.1145/2723372.2742797`

Spark SQL is a module in Apache Spark that integrates relational processing with Spark's functional programming API. The goal is to provide users with an interface that seamlessly mixes relational (declarative database style) and procedural programming models to access data and perform computations. This is realized through the *Spark DataFrame API* and a new optimizer *Catalyst*. The base Spark API is a general-purpose cluster copmuting engine with APIs in Scala, Java, and Python and libraries for streaming, graph processing and machine learning.

It is based on a functional programming API, where users manipulate RDDs. Please refer to paper summary section 16 for more information on RDDs and basic Spark operation.

Goals for Spark SQL

- Support relational processing within Spark on native RDDs and on external data sources
- Provide high performance using established DBMS techniques/optimizations
- Easily support new data sources; including semi-structured and external databases
- Enable extensions for graph processing and machine learning

17.1 Programming Interface

DataFrame API is the main abstraction in Spark SQL; a distributed collection of rows with the same schema. It is equivalent to a table in relational databases. They keep track of their schema and support relational operations, thus leading to more optimized execution.

DataFrames can be created from external sources or RDDs and then be manipulated by relational operations or also viewed as an RDD of *Row objects*, which can be manipulated by the procedural Spark API functions.

All operations are *lazy* and are only executed upon a specific request to compute the results; this allows for rich set of optimizations.

Data Model Spark SQL supports all the known base types (booleans, integers etc.), complex data types (structs, arrays, maps etc.), as well as custom complex data types.

DataFrame Operations are expressed in a DSL (domain specific language), which includes all common relational operations such as projection (`select`), filter (`where`), `join`, and aggregations (`groupBy`). The operators take *expressions* as arguments in a limited DSL, which allows Spark to capture the structure of the

computation in an AST (abstract syntax tree) which is passed to the optimizer *Catalyst*. This is opposed to the native Spark API which executes arbitrary host framework code, which is opaque to the runtime engine.

The DataFrames can also be registered as temporary tables and queried using SQL. To simplify programming in DataFrames the API can *analyze* logical plans *eagerly* to identify errors, but still *compute lazily*.

Spark can also infer schemas from language native collections of objects. For example inferring a schema from an RDD of class objects by directly using class attribute names and create the appropriate DataFrame.

In-Memory Caching is more efficient as the compact columnar storage of DataFrames is more space-efficient than the object-based storage of RDDs.

UDF (User-Defined Functions can be registered in Spark SQL inline by passing native language (Scala, Python, Java) functions. This allows to extend the framework for application specific processing in e.g. machine learning.

17.2 Catalyst Optimizer

Catalyst is an extensible optimizer based on functional programming constructs in Scala. It is at its core based on a general library for representing *trees* and *rules* to manipulate them. The main datatype in Catalyst is thus a *tree* with *node* objects of various types. Trees can be manipulated by rules, which define a tree-to-tree mapping; rules are applied using *pattern-matching* (built into many functional programming languages; thus also Scala) on sub-trees. Rules can be applied until it reaches a *fixed point* i.e. the tree does not change anymore.

Catalyst in Spark SQL and its tree-based approach is used for:

- Analyzing a logical plan to resolve references and types.
- logical plan optimization (based on common DBMS techniques)
- physical planning using Spark physical operators (multiple results may be compared based on cost)
- code generation to compile parts of the query to Java bytecode

Extension points Users can extend Spark SQL through adding:

- Custom *Data Sources* by using several APIs implementing various levels of access to the data source from simply scanning it completely to use e.g. native filtering implementations of the data source.
- User-Defined Types (UDT) (e.g. vector type for ML, graph types for graph processing etc.); UDTs are mapped to structures of built-in types using Catalyst (the user provides the mapping upon implementing the UDT). This allows the framework to use all the optimizations also on the UDTs.

17.3 Advanced Analytics Features

Features added to improve *big data* type of workloads.

- Schema Inference for semi-structured data e.g. JSON input data
- Integration with Sparks ML library by exposing an API based on DataFrames; allowing to build ML *pipelines*
- Query Federation to External Databases; instead of loading data into local processing environment, execute the required query or parts of it on the external data source directly to reduce traffic and improve performance.

17.4 Evaluation, Research Applications, Related Work

DataFrames make everything faster, simpler, and even researchers like Spark SQL.

18 Scalability! But at what COST? - jkleine

Source: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

The paper presents the COST metric, or the **C**onfiguration that **O**utperforms a **S**ingle **T**hread. With this metric the authors try to to shed some light on misleading scaling behaviours of distributed algorithms.

Often, when a distributed algorithm or system is presented, the bulk of the attention rests on the scalability of the algorithm, while disregarding the actual real-world runtime. The authors show this by applying their COST metric to a set of distributed graph processing libraries on two specific problems, namely Page Rank and Connected Components.

For the comparison the authors implement a baseline single-threaded versions of the algorithms and compare these to the distributed counterparts. The baseline was run on a "high-end 2014 laptop", while the distributed version ran on cluster with up to 512 cores.

The authors found that many of the distributed libraries need hundreds of cores to beat the runtime of the single threaded application. For example, the GraphX and GraphLab systems, when running on 128 cores, needed 252 seconds and 242 seconds respectively to compute the connected components on the twitter social graph, while the basic single-threaded application that used the same algorithm needed 153 seconds on the laptop. Even further, the authors show that the algorithmic approach taken by the distributed systems introduces massive inefficiencies, just to be scalable. When using an algorithm (Union-Find) that is more efficient, but inherently more single threaded, the performance of the baseline got an order of magnitude better at 15 seconds.

The authors also compare two distributed applications with one another. Application A shows a mush more superior scaling behaviour when compared to application B. However when comparing the COST of both applications the authors found that application B needed 10 core to archive the speed of the single-threaded application, while application A needed around 100. This shows that while application A scales a lot better, it needs many times more cores than application B to even reach the same real world speed. *On a personal note: the authors prove this by showing a runtime plot in addition to the speedup plot. This information suffices in my mind to show the above, the COST metric is thus not strictly necessary.*

While the authors conclude, that many systems show the above behaviour, they also state that the distributed systems, especially in the Big Data domain, often make specific trade-offs in order to achieve additional objectives apart from scalability and speed, such as integration into an existing ecosystem, high availability, or security, which a simpler system, cannot provide.

19 MongoDB: The Definitive Guide (2nd ed.) (Chapters 3, 4, and 5) - tgeorg

Source: http://proquest.safaribooksonline.com/book/databases/mongodb/9781449344795

19.1 Chapter 3: Creating, Updating, and Deleting Documents

19.1.1 Inserting and Saving Documents

`db.foo.insert({"bar" : "baz"})` to insert a document
`_id` key is automatically added if not exists.

Batch Insert

```
1> {db.foo.batchInsert([{"\_id" : 0}, {"\_id" : 1}, {"\_id" : 2}])
2> db.foo.find()
3{"_id" : 0 }
4{"_id" : 1 }
5{"_id" : 2 }
```

"Batch inserts are only useful if you are inserting multiple documents into a single collection: you cannot use batch inserts to insert into multiple collections with a single request."
(Not really used for importing data)

Insert Validation Basic checks for invalid data are performed when inserting.

19.1.2 Removing Documents

`db.foo.remove()` will remove all documents in the collection `foo`. The collection itself will stay.
Can be used to with a query document to only remove elements that match the filter
`db.mailing.list.remove("opt-out" : true)`

19.1.3 Updating Documents

"Updating a document is atomic [...] conflicting updates can safely be sent in rapid-fire succession without any documents being corrupted: the last update will "win.""

Document Replacement `db.people.update("name" : "joe", new_joe)` will search for a matching document and will replace the first match with the one stored in `new_joe`.

Using Modifiers Given:

```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 52
}
```

using

```
db.analytics.update({"url" : "www.example.com"}, {"$inc" : {"pageviews" : 1}})
```

gives

```
> db.analytics.find()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 53
}
```

Note that `$inc` is the increment operator. I.e. we search for the document where `url` matches `www.example.com` and apply the increment operation on `pageviews`.

The value `1` defines how much the value is incremented, i.e. replacing it with `50` results in adding $+50$. If the field does not exist it will be created and the initial value `0` is assumed.

Other operators:

- `$set` sets the value of a field. If the the field didn't exist, it will be created.
- `$push` is used to add another element to an array
- `$each` to `$push` multiple elements at once
- `$sort` does what it says
- `$slice` is like `LIMIT` in SQL
- `$addToSet` adds to an array if the element does not already exist.
- `$pop` to remove an element from an array (front or back, can be specified)

Elements in arrays can be accessed by index via `my_array.0.parameter_of_some_object` where `0` is the desired index.

Upserts "An upsert is a special type of update. If no document is found that matches the update criteria, a new document will be created by combining the criteria and updated documents. If a matching document is found, it will be updated normally. Upserts can be handy because they can eliminate the need to "seed" your collection: you can often have the same code create and update documents."

The code

```
// check if we have an entry for this page
blog = db.analytics.findOne({url : "/blog"})
// if we do, add one to the number of views and save
if (blog) {
    blog.pageviews++;
    db.analytics.save(blog);
}
// otherwise, create a new document for this page
else {
    db.analytics.save({url : "/blog", pageviews : 1})
}
```

can be replaced with:

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"pageviews" : 1}}, true)
```

which does the same thing but is atomic. The `true` as the third argument tells the database the update is an *upsert*.

Updating Multiple Documents Updates are only applied to the first matched document. To update all documents, perform a multiupdate by passing a `true` as the fourth parameter, i.e.

```
db.users.update({"birthday" : "10/13/1978"}, {"$set" : {"gift" : "Happy Birthday!"}}, false, true)
```

Note, the third parameter (`false`) refers to the upsert, the fourth (`true`) is for the multiupdate.

Returning Updated Documents The command `findAndModify` can modify documents and will return all modified documents in the state they were in before performing the update.

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}})

{
    "ok" : 1,
    "value" : {
        "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
        "priority" : 1,
        "status" : "READY"
    }
}
```

19.1.4 Setting a Write Concern

"Write concern is a client setting used to describe how safely a write should be stored before the application continues. By default, inserts, removes, and updates wait for a database response"

19.2 Chapter 4: Querying

19.2.1 Introduction to find

`find` is used to query for documents. `find` returns every document that matchs the query.

```
db.users.find({"age" : 27})
```

returns all documents with where `age == 27`. If multiple fields are passed all have to match for the document to be returned, i.e.

```
db.users.find({"username" : "joe", "age" : 27})
```

requires `username == jow AND age == 27` for the document to be returned.

An empty query ({}) matches everything.

Specifying Which Keys to Return The second parameter in `find` is used to to select the fields to display. Example:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

`_id` is always displayed by default. Passing a `0` instead of a `1` for some field, means this field should not be displayed.

Limitations The value of a query has to be constant, i.e. cannot be a variable assigned during the run of the query.

19.2.2 Query Criteria

Query Conditionals "`$lt`" , "`$lte`" , "`$gt`" , and "`$gte`" are all comparison operators, corresponding to $<$, $<=$, $>$, and $>=$, respectively.

OR Queries The operator `$in` can be used with `find` to search for elements fitting one out of a list of values:

```
> db.users.find({"user_id" : {"$in" : [12345, "joe"]})
```

As in the example, the values searched are not required to be the same datatype. The operator `$nin` is the opposite of `$in` and only returns result that do not contain the passed values.
The operator `$or` can be used to search elements fitting one or more criteria:

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

$not Is self explanatory. Returns result not fitting a criteria.

Conditional Semantics One can use conditional operator like `$lt` for queries

```
> db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})
```

There are also boolean operators: "`$and`", "`$or`", and "`$nor`".

19.2.3 Type-Specific Queries

null One can query for values of type `null` but should be carefull that if querying for a field that does not exist, `null` matches the missing field. Use the operator `$exists` to ensure the field actually exists.

Regular Expressions RegEx word like in other languages, i.e.

```
> db.users.find({"name" : /regexStuff/})
```

Note that if a document contains the same RegEx expression it will also be matched.

Querying Arrays When querying arrays a result is found if the element from the query is inside the array. If one once to make sure that the array contains certain elements (plural), then use operator `$all`:

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})

> db.food.find({fruit : {$all : ["apple", "banana"]}})

{"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
{"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

When querying for an array without using `$all` only documents will be matched where the arrays match each other in elements and their order.
The operator `$size` can be used to look for arrays of certain size.
The operator `$slice` can be used to take a slice out of the array, i.e. first x elements, elements between index y and z, etc.

Querying on Embedded Documents Querying for an embedded document by using a document in the query means it will only match if query and embedded document contain the same values in the same order:

```
db.people.insert({
    "name" : {
    "first" : "Joe",
    "middle" : "Moe",
    "last" : "Schmoe"
    },
```

```
    "age" : 45
})
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})

/* No match */
```

Instead one should query for the keys, so:

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

will match.

19.2.4 Cursors

Cursors allow you to step through the results of the query a document at the time.

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
...     print(x.name);
... };
```

Important to note that the variable `cursor` does not hold the results of the query, rather with every loop iteration the next document is fetched.

Limits, Skips, and Sorts Limit result to first 3:

```
db.c.find().limit(3)
```

Skip first 3:

```
db.c.find().skip(3)
```

Sort results by "username" ascending and "age" descending:

```
db.c.find().sort({username : 1, age : -1})
```

Avoiding Large Skips Don't use large skips as data still has to be fetched, rather make use of the composition of the data.

19.2.5 Database Commands

`runCommand` can be used to run commands on the database.

```
> db.runCommand({"drop" : "test"}); // Drop collection 'test'
> db.runCommand({shutdown:1}) // Shutdown database
```

19.3 Chapter 5: Indexing

19.3.1 Introduction to Indexing

Database can use index to find desired document quicker. Otherwise full table scan is performed.
One can make some field an index by using `ensureIndex`:

```
db.users.ensureIndex({"username" : 1})
```

Creating too many indexes can slow down the database when updating/inserting documents. Hence there is a max limit of 64 indexes per collection. Generally, indexes should only be used for fields that are queried the most often.

Compound Indexes An index can also be created based on multiple fields:

```
db.users.ensureIndex({"age" : 1, "username" : 1})
```

Replacing the 1 with a −1 reverses the sort order for that element. An example where compound indexes are useful would be when searching for documents matching on the first index and sorting them by the second. Both actions are sped-up with compound indexes as querying for an indexed element is almost instant and the indexed elements are also sorted, hence the sort operation on the second index is trivial.

Also, the prefixes of compound indexes are again, (compound) indexes, i.e. {"age" : 1, "username" : 1, "level": 1} is implicitly indexed by {"age" : 1, "username" : 1}

How $-Operators Use Indexes Inefficient operators

- `$exists` has to go through entire table anyway
- `$ne` has to go through all elements except the matching one, so index usually only reduce search space by a small amount

Querying **ranges** is faster on the first index then later ones.

When querying for multiple fields only one will be searched by index even if multiple are indexed. `$or` is the exception, as it use one index per `$or` clause.

Indexing Objects and Arrays Indexes can be created on fields of embedded documents using their key:

```
{
    "username" : "sid",
    "loc" : {
        "ip" : "1.2.3.4",
        "city" : "Springfield",
        "state" : "NY"
    }
}
> db.users.ensureIndex({"loc.city" : 1})
```

Entire embedded documents can also be used as an index but this is only useful when looking for the exact same document as a query, otherwise the index won't be of use.
Indexing arrays is also possible, which allows using the index to quickly search for specific array elements. However, an index entry is creates for each element of the array, which can become expensive with large arrays. Index keys with array fields are flagged as multikey indexes.

Index Cardinality *Cardinality* refers to number of distinct values a field can take, e.g. by intuition, "age" has higher cardinality than "gender". The higher the cardinality the more useful an index as more values can be filtered out while performing the query.

19.3.2 Using explain() and hint()

`explain()` can be used to get diagnostic information about a query. Example:

```
> db.users.find({"age" : 42}).explain()
{
    "cursor": "BtreeCursor age_1_username_1",
    "isMultiKey": false,
    "n": 8332,
    "nscannedObjects": 8332,
    "nscanned": 8332,
    "nscannedObjectsAllPlans": 8332,
    "nscannedAllPlans": 8332,
    "scanAndOrder": false,
    "indexOnly": false,
    "nYields": 0,
    "nChunkSkips": 0,
    "millis": 91,
    "indexBounds": {
        "age": [
            [
                42,
                42
            ]
        ],
        "username": [
            [
                {
                    "$minElement": 1
                },
                {
                    "$maxElement": 1
                }
            ]
        ]
    },
    "server": "ubuntu:27017"
}
```

`hint()` on the other side is used to force the database to use a certain index for querying. Example:

```
db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1, "age" : 1})
```

19.3.3 When Not to Index

Index scans are more effective the smaller the subset of data retrived compared to the size of the collection. This is because using an index requires two lookups, one for the index entry and one for following the index' pointer, while a table scan only performs a single lookup per entry when scanning through the table.
"Rule of thumb: if a query is returning 30% or more of the collection, start looking at whether indexes or table scans are faster."

19.3.4 Types of Indexes

Unique Indexes Index where each value only appears once in the table is considered *unique*. For compound indexes, the combination of values most only appear once for the index to be considered unique. Building an index via unique values is achieved by:

```
db.users.ensureIndex({"age" : 1}, {"unique" : true})
```

The action will fail if the values are not unique. As a drastic measure one can use the following command to drop all documents with duplicate values while building the index

```
db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups" : true})
```

Sparse Indexes Missing values are considered `null` when interacted with. Therefore by default building a unique index with sparse values is not possible. If an indexe is desired where all existing values must be unique, use the parameter `sparse`:

```
db.ensureIndex({"email" : 1}, {"unique" : true, "sparse" : true})
```

The result of some queries might change depending on whether or not a sparse index is used.

19.3.5 Index Administration

Use `getIndexes()` to see all index information about a given collection.

```
> db.foo.getIndexes()
[
    {
        "v": 1,
        "key": {
            "_id": 1
        },
        "ns": "test.foo",
        "name": "_id_"
    },
    {
        "v": 1,
        "key": {
```

```
            "y": 1
        },
        "ns": "test.foo",
        "name": "y_1"
    }
]
```

Identifying Indexes Index names are, by default, `keyname1_dir1_keyname2_dir2_..._keynameN_dirN`, where `keynameX` is the index's key and `dirX` is the index's direction (1 or -1).

Custom names can be given via:

```
> db.foo.ensureIndex({"a" : 1, "b" : 1, "c" : 1}, {"name" : "alphabet"})
```

Changing Indexes Use `dropIndex()` to drop a index from a collection given its name (which we got from `getIndexes()`)

```
db.people.dropIndex("x_1_y_1")
```

20 Rumble: Data Independence for Large Messy Data Sets - burgerm

Source: `https://arxiv.org/pdf/1910.11582.pdf`

Rumble is q query execution engine for large, heterogeneous, and nested collections of JSON objects built on top of Apache Spark. It uses *JSONiq* as interface langauge; a standardized language for querying JSON documents. To fit the recursive structure of JSON objects onto the Spark's execution primitives (e.g. DataFrames) the *JSONiq* expression is translated into a tree of iterators. Being built on top of Spark allows to process large scale data sets and inherits Sparks support for various data formats.

Introduction Large JSON datasets are widely used because of the ease of use of dumping information as JSON objects. This can yields large and heterogenous datasets over the years. Most commonly available tools (usually table based) drop values which cannot fit into a regular structure, keep nested objects as complete strings or map to non-existing values etc. Conversion to a data frame thus seems to be inadequate for JSON data sets. Further `SQL` is not build for querying nested data structures and reaches its limits when working with such data sets.

JSONiq is a declarative, functional and Turing-complete query language. It manipulates ordered and potentially heterogenous *sequences* of *items*. Items can be:

- atomic values (JSON types, dates, binaries, etc.)
- structured items (objects, arrays)
- function items

Sequences are always flat, unnested automatically, and may be empty. Operators query nested objects and can return flattened sequences even if they found matches at different depth levels and they also deal with absent data e.g. by returning empty sequences.

For more complex structures the *FLWOR* expressions based on {`for, let, where, order by, group by, return`} clauses allow to process each item of a sequence individually. Other expressions include literals, function calls, sequence predicates and concatenation, range construction, `if-then-else`, `switch`, and `try-catch` expressions. See Fig. 27 for an example. For more details about JSONiq and related query languages check the lecture materials (e.g. lecture *12 Big Data - Querying trees*).

20.1 Design and Implementation

On a high level Rumble follows a traditional database architecture: submit queries, Rumble parses the queries into an AST, the AST is translated into a physical execution plan. The execution plan consists of *runtime iterators*, which roughly

```
for $e in $events         (: iterate over input :)
let $top-committer := (
  for $c in $e.commits[]  (: iterate over array :)
  group by $c.author
  stable order by count($c) descending
  return $c.author)[1]
return [$e.commits[][$$.author eq $top-committer]]
```

Figure 27: JSONiq `FLWOR` expression

correspond to JSONiq expressin or FLWOR clauses. The result can be displayed, stored to file or written to a distributed file by Spark. The key challenge is the mapping of the recursive structure of JSONiq onto Sparks mostly flat execution primitives.

In the following a quick overview of the primitives of JSONiq and how they are mapped to Spark operators and local operators. For a more detailed description of how they are implemented check the paper (highly unnecessary in my opinion).

Runtime Iterators The iterators can be executed in three different ways, *local execution* i.e. direct single-threaded Java implementations, *RDD-based Spark* or *DataFrame-based Spark*. Based on the type of operations and available information (statically known structure, heterogenous sequence etc.), vailable operators for an execution mode, user preferences and the chosen operator for a nested operation, a suitable mode is chosen. Rumble can switch seamlessly between local and Spark-based execution modes through appropriate interfaces.

Sequence-transforming Expressions transform sequences of items onf any length i.e. operate on *items**. This is e.g. implemented using Sparks {`map()`, `flatMap()`, `filter()`} transformations. The use of `flatMap` allows to return multiple items per transformed input item. The output is then chained with a `filter` or `map` transformation to apply the desired function.

Sequence-producing Expressions are operators that take a local input and trigger distributed execution by creating DataFrames or RDDs.

Sequence-combining Expressions are performed using Spark RDD transformations. Control flow operations from JSONiq allow for optimization by eager evaluation of conditionals and the the launch of a Spark job only on required branches and only if an actually executed child branch requires results.

FLWOR Expressions correspond roughly to `SQL SELECT` statements. The different clauses, functions and operators are mapped to suitable Spark primitives.

20.2 Experiments, Results, Related Work

Rumble scales, Rumble is as fast or faster than competitors, Rumble can handle more cases, and where Rumble is slower it is still better because we have a better programming interface, which is more accessible.

Some related work on query languages for JSON, document stores and in-situ data-analysis.

Conclusion Demonstrate data independence for JSON processing is achievable with reasonable performance on top of large clusters.

21 Robinson, I. et al. (2015). Graph Databases (2nd ed.) - jkleine

Source: `http://proquest.safaribooksonline.com/book/databases/9781491930885`

21.1 Data Modeling with Graphs

21.1.1 Labeled Property Graph

A labeled property graph is made up of *nodes*, *relationships*, *properties*, and *labels*. Nodes contain properties and represent documents that store properties in the key-value format. In Node4j valid values are (Java) Strings, primitive data types and arrays of the previous types. Nodes are tagged with labels to group nodes together (for example each node represents a Person, then Person would be the tag). A relationships connect two nodes to from a graph. The connections are directed and never dangling.

21.1.2 Querying Graphs with Cypher

The philosophy of Cypher is to "find things like this" in the Graph. "Things like this" are graph patterns, intuitively defined by ASCII art. For example, the ASCII art

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

defines the graph in Firgure 28.

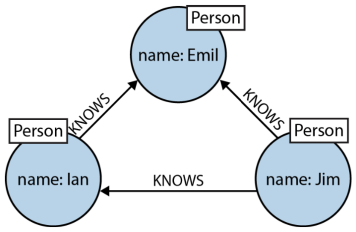


Figure 28: Sample graph pattern

To further specify what we are looking fore, we can define a few more properties, like node type, properties, etc.. The new ASCII art looks like follows:

```
(emil:Person {name:'Emil'})
  <-[:KNOWS]-(jim:Person {name:'Jim'})
  -[:KNOWS]->(ian:Person {name:'Ian'})
  -[:KNOWS]->(emil)
```

MATCH The match clause is the specification by example part of the query language. The query selects all matches of the patter. Combining this with the `RETURN` clause we can specify what part of the matched patter to return. We can

also define further properties using the `WHERE` clause, very similar to how SQL works. Take a look at the following query:

```
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c
```

This query would match all triangle patterns (similar to Figure 28), where a node *a* "knows" a node *b*, which in turn "knows" *c*, *a* also knows *b* directly. Further *a* is labeled as Person. The `WHERE` clause further specifies that node *a* must have a property *name* with value *"Jim"*. The query returns the nodes *b* and *c* of all matching occurrences.

A query can return node, relationships, and properties from the matched data.

The nodes in a query that map to specific real nodes are called *anchors* as they anchor the query to real nodes.

Further syntax examples:

```
1.  (a:Label {prop:'value'})-[r:RELATION]->(b:Label)
2.  (a)-[]->()
3.  (a)-[:REL_A|REL_B*1..3]->(b)
```

The first is pretty basic, it matches a node with label "Label" and property "prop" equal to `'value'`, which is connected via a relation "RELATION" to another node with label "Label". The nodes are identified by *a* and *b* respectively, the relationship by *r*. These identifiers can be used in the rest of the query, for example in `RETURN` and `WHERE` clauses. If the node is not needed specifically one can use the *anonymous* node

List of other Cypher clauses

`WHERE` Provides criteria for filtering pattern matching results.

`CREATE` (and `CREATE UNIQUE`) Create nodes and relationships.

`MERGE` Ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates, or by creating new nodes and relationships.

`DELETE` Removes nodes, relationships, and properties.

`SET` Sets property values.

`FOREACH` Performs an updating action for each element in a list.

`UNION` Merges results from two or more queries.

`WITH` Chains subsequent query parts and forwards results from one to the next. Similar to piping commands in Unix.

21.1.3 Relational vs Graph Modeling

tldr; Turn entities into node, and relations into edges. No magic... Its a graph...

The book explains this with the example of a system management domain (i.e. a set of server, racks, VMs, apps, clients, etc. that communicate/are related in certain ways). In a relational database each individual type would have an individual table that contains all instances of that type. Further, depending on the relation, there are additional tables connecting multiple rows of one table with multiple tables of another table (e.g. which app uses which services). Alternatively, in a 1-to-N relationship, this would be handled in the N side of the relation.

In a graph database each individual element in this construct would be a node, i.e., one node for each app, one node for each service, etc.. The nodes are then connected to represent the relations, all metadata is stored in labels and properties.

Query The advantage of graph databases becomes apparent when you frequently query certain patterns in the graph. For example to find faulty equipment after a user complaint the following query would return all distinct faulty elements that affect `User 3`.

```
MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User 3' AND asset.status = 'down'
RETURN DISTINCT asset
```

Common Modeling Pitfalls It is easy to miss important information when going from a relational model to a graph model. In the book this is shown in the form of an email relationship that stores who emailed whom with CC and BCC relationships to detect fraudulent behaviour, but the actual content of the email was lost in the process. Basically, you need to think about what you are doing... Shocker!

Avoid Anti-Patterns In general, do not encode entities into relationships. Use relationships to encode the *how* two entities are related, and the quality of the relationship. It is easy to confuse nouns (which map to entities) with verbs (which map to relations). For example, while we might say "Alice *emailed* Bob", it actually means "Alice *send* this email *to* Bob", which makes it more apparent that the email is an entity itself and *send* and *to* are the relationships. I email is not a entity, we would loose the content of the email as this doesn't really fit into a relationship property (as an email might be send to multiple people).

Graphs are naturally additive structures. Adding many entities might feel wrong, as one wants to preserve query time, but this is not a consideration one should make while designing the database. To quote the book: "Graph databases maintain fast query times even when storing vast amounts of data. Learning to trust our graph database is important when learning to structure our graphs without denormalizing them."

Evolving the Domain Adding additional data to a graph database is sometimes a lot easier than in relational databases, as we can simple add new nodes and edges, without affecting the preexisting graph.

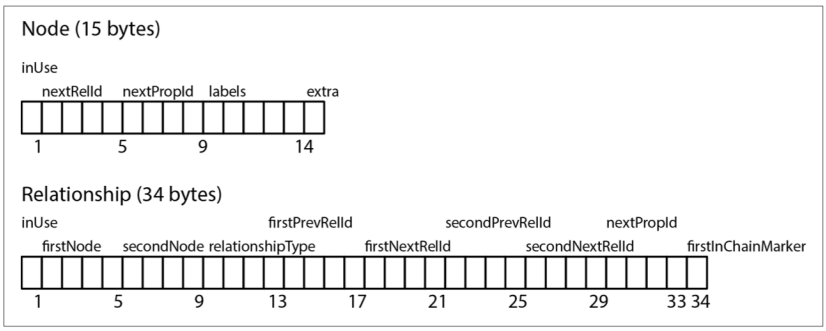


Figure 29: Neo4j node and relationship store file record structure

21.1.4 Identifying Nodes and Relationships

Loose guideline to translate a concept into a graph:

- Common nouns become Labels: "user" and "email" become the labels `User` and `Email`.
- Verbs that take an object become relationship names: "sent" and "wrote," for example, become `SENT` and `WROTE`.
- A proper noun—a person or company's name, for example—refers to an instance of a thing, which we model as a node, using one or more properties to capture that thing's attributes.

21.2 Graph Database Internals

This subsection takes a brief look at the underlying architecture of a graph database.

Native Graph Processing A native graph database like *Neo4j* stores the relationships index free. That means there is not one table that lists all relationships of a specific type, but rather, each node store its outgoing connections locally, which greatly speeds up graph traversal.

Native Graph Storage Neo4j stores the graph in seperate stores which are represented by files on disks. There exists a store for nodes, relationships, properties, and labels. Storing the data for example for nodes and properties separately makes the underlying model slightly different from the view exposed to the user, however this greatly increases traversal performance.

The node store, like many other stores in Neo4j, is a fixed size store, that is every node in the graph will be exactly 9 bytes in size (I assume there is a mistake in the book as Figure 29 shows 15 bytes). This greatly aids lookup speeds. In the case of node data the first byte is a usage flag to indicate whether Neo4j can reclaim those bytes for new nodes. The next four bytes indicate the ID of the first

relationship connected to the node, followed by four bytes for the first property for the node. The next five bytes point to the label store for this node. The final byte is reserved for flags and future use.

The relationship store is similar, with fixed sized entries. The record contains the IDs for both nodes of the connection, a pointer to the relationship type, two pointers for each node in the the relationship to the next/previous relationship (the form a doubly linked list for each node), and some flag bytes.

Traversing the data is very efficient. Due to the fixed sized entries one can compute the location of the next element in constant time, instead of doing a global lookup in a table (in the case of non-native graph databases).

To further improve traversal speeds, Neo4j implements an *LRU-K page cache*, where each store is divided into distinct regions which are then cached in an LRU fashion. This allows the caching of graphs, which exceed the size of main memory.

Kernel API This is the lowest level API available in Neo4j. It allows the user to listen to all transaction flowing through the system and react to them. One use case would be to intercept the deletion of nodes and only mark them as logically deleted in case the user wants to roll back the database at a later time.

Core API The core API is a imperative Java API. It is lazily evaluated, as the nodes are traversed only when the code demands the next node. It also allows for transaction management capability to ensure ACID.

Traversal Framework In this framework the user defines some restrictions for a traversal in a declarative Java API. The traversal is then performed by Neo4j as specified.

Cipher Cipher is the SQL like query language seen above with the definition of the graph patterns in ASCII-art.

21.2.1 Nonfunctional Characteristics

Transactions Transactions in Neo4j are semantically identical to traditional database transactions. Writes occur within a transaction context, with write locks being taken on any nodes and relationships involved in the transaction. On successful completion of the transaction, changes are flushed to disk and the locks are released. If the transaction fails, the writes are discarded. In case multiple transactions try to modify the same part of the graph, the transactions will be serialized.

Recoverability When recovering from an unclean shutdown Neo4j checks in the most recently active transaction log and replays any transactions it finds against the store. In a cluster setting this is called *local recovery*. After the local recovery is done the node needs to "catch up" with the other nodes. For this purpose the node request any transactions that occurred during the fail from another node (typically the master).

Availability Typically, multiple database instances are clustered for high availability, with one master instance relaying all transactions to the other instances. At any moment the master and some replicas will be up to date, with some replicas catching up (in the order of milliseconds behind). Reads are generally distributed across the replicas to scale read performance.

Scale Instead of only focusing on transactions per second, Neo4j focuses on a combination of capacity (graph size), latency, and read & write throughput.

Neo4j can handle graphs with tens of billions of nodes, relationships, and properties. Apparently it could handle the Facebook graph.

Latency is not such a big issue in graph databases as in traditional relational databases. Since data is queried via pointer chasing and pattern matching, the latency depends on the size of data being queried, not the size of the data set.

For throughput one can note that complex read/write operations typically access the same region, i.e. sub-graph, of the graph multiple times. Due to the way the graph is stored it is very efficient to traverse local sections of a graph via pointer chasing.

Clustering To spread a graph across multiple machines there are a few options. One way would be to split the graph into logical partitions from the client point of view, i.e. maintain a set of smaller graphs that are relatively independent.

If this is not case it is necessary to split a single graph across multiple machines. For this synthetic keys are used that point to data on other nodes. These keys are then resolved by the database when traversed.

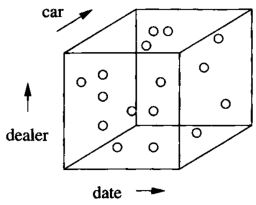


Figure 30: Datacube

22 Garcia-Molina, Ullman, Widom: Database Systems: The Complete Book. Pearson, 2. Edition, 2008. (Chapters 10.6 and 10.7) - tgeorg

Source: No link found :(

22.0.1 On-Line Analytic Processing

OLAP - Online Analytic Processing: examination of data for patterns using complex queries possibly using multiple aggregations.

OLTP - On-Line Transaction processing: common database operations that only touch a tiny portion of the database.

22.0.2 OLAP and Data Warehouses

"It is common for OLAP applications to take place in a separate copy of the master database, called a *data warehouse*. Data from many separate databases may be integrated into the warehouse. In a common scenario, the warehouse is only updated overnight, while the analysts work on a frozen copy during the day."

Data Warehouses are used as mean to centrally organise scattered data and because OLAP queries might be too resource intensive for databases handling live transactions.

22.0.3 A Multidimensional View of OLAP Data

"In typical OLAP applications there is a central relation or collection of data, called the fact table. A fact table represents events or objects of interest [...]. Often, it helps to think of the objects in the fact table as arranged in a multidimensional space, or "cube.""

ROLAP - Relational OLAP: Data stored in a star schema; one of the relations is the fact table; "Other relations give information about the values along each dimension. The query language, index structures, and other capabilities of the system may be tailored to the assumption that data is organized this way."

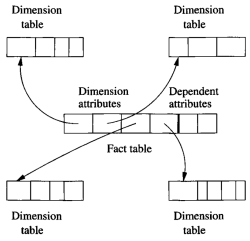


Figure 31: Star schema

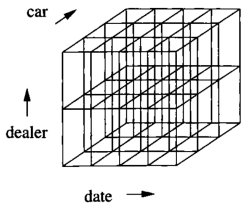


Figure 32: Dicing the cube by partitioning along each dimension

MOLAP - Multidimensional OLAP: Data cube is used to hold the data

22.0.4 Star Schemas

"A star schema consists of the schema for the fact table, which links to several other relations, called "dimension tables." The fact table is at the center of the "star," whose points are the dimension tables. A fact table normally has several attributes that represent dimensions, and one or more dependent attributes that represent properties of interest for the point as a whole."

22.0.5 Slicing and Dicing

Partitioning the data along certain attributes is called "dicing". We can then decide to work on a particular partition (slice).

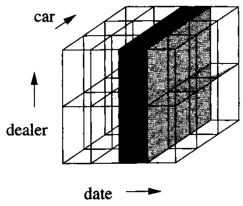


Figure 33: Selecting a slice of a diced cube

22.1 Data Cubes

"the formal data cube precomputes all possible aggregates in a systematic way."
"In the data cube, it is normal for there to be some aggregation of the raw data of the fact table before it is entered into the data-cube and its further aggregates computed."

22.1.1 The Cube Operator

Given a fact table F , we can define an augmented table `CUBE(F)` that adds an additional value, denoted *, to each dimension. The * has the intuitive meaning "any," and it represents aggregation along the dimension in which it appears.

A tuple of the table `CUBE(F)` that has * in one or more dimensions will have for each dependent attribute the sum (or another aggregate function) of the values of that attribute in all the tuples th at we can obtain by replacing the *'s by real values. In effect, we build into the data the result of aggregating along any set of dimensions. Notice, however, that the `CUBE` operator does not support aggregation at intermediate levels of granularity based on values in the dimension tables.