

# Big Data

{wnicole}@ethz.ch

ETH Zürich, HS 2020

This is a summary for the course *Big Data* at ETH Zürich.

I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. Feel free to point out any erratas. It's probably smart to reread your old DMDB summary first!

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                      | <b>4</b>  |
| 1.1      | Concepts . . . . .                                       | 4         |
| 1.2      | SQL Brush Up . . . . .                                   | 6         |
| 1.3      | Exercises . . . . .                                      | 6         |
| 1.3.1    | SQL Brush Up . . . . .                                   | 6         |
| <b>2</b> | <b>Storage</b>   | <b>7</b>  |
| 2.1      | Object Storage . . . . .                                 | 7         |
| 2.1.1    | Amazon S3 . . . . .                                      | 8         |
| 2.1.2    | Microsoft Azure Blob Storage . . . . .                   | 9         |
| 2.2      | Key-Value Store . . . . .                                | 11        |
| 2.3      | Distributed File Systems . . . . .                       | 15        |
| 2.3.1    | Hadoop Distributed File System (HDFS) . . . . .          | 16        |
| 2.4      | Syntax . . . . .   | 19        |
| 2.4.1    | JSON . . . . .   | 20        |
| 2.4.2    | XML . . . . .  | 21        |
| 2.5      | Exercises . . . . .                                      | 24        |
| 2.5.1    | Object and Key-Value Stores . . . . .                    | 24        |
| 2.5.2    | HDFS . . . . .   | 24        |
| 2.5.3    | XML . . . . .  | 24        |
| <b>3</b> | <b>Models</b>  | <b>25</b> |
| 3.1      | Wide Column Stores . . . . .                             | 25        |
| 3.1.1    | Bigtable . . . . .                                       | 25        |
| 3.1.2    | HBase . . . . .  | 29        |
| 3.2      | Data Models and Schemas . . . . .                        | 30        |
| 3.2.1    | JSON Data Model . . . . .                                | 31        |
| 3.2.2    | JSON Schema . . . . .                                    | 31        |
| 3.2.3    | XML Information Set and Schema . . . . .                 | 34        |
| 3.2.4    | In General . . . . .                                     | 34        |
| 3.2.5    | Protocol Buffers . . . . .                               | 34        |
| 3.2.6    | Validation . . . . .                                     | 34        |
| 3.3      | Graph Databases . . . . .                                | 35        |
| 3.3.1    | Neo4j . . . . .  | 35        |
| 3.3.2    | GDB Reading Assignment . . . . .                         | 36        |
| 3.4      | Cubes . . . . .  | 36        |
| 3.5      | Dremel . . . . .   | 37        |
| <b>4</b> | <b>Processing</b>  | <b>39</b> |
| 4.1      | Two Step Distributed Query Processing . . . . .          | 39        |
| 4.1.1    | MapReduce . . . . .                                      | 39        |
| 4.2      | Resource Management . . . . .                            | 41        |
| 4.2.1    | YARN: Yet Another Resource Negotiator . . . . .          | 42        |
| 4.2.2    | Scheduling Strategies . . . . .                          | 43        |
| 4.2.3    | Dominant Resource Fairness . . . . .                     | 45        |
| 4.3      | DAG-Based Distributed Query Processing (Spark) . . . . . | 46        |
| 4.3.1    | Resilient Distributed Dataset . . . . .                  | 46        |
| 4.3.2    | Physical Layer . . . . .                                 | 48        |
| 4.3.3    | DataFrames . . . . .                                     | 49        |
| 4.3.4    | SparkSQL . . . . .                                       | 49        |

|          |  |           |
|----------|--|-----------|
| 4.4      | Performance at Large Scales . . . . .      | 50        |
| 4.4.1    | Scalability! But at what COST? . . . . .   | 51        |
| <b>5</b> | <b>Management</b>                          | <b>52</b> |
| 5.1      | Document Store . . . . .                   | 52        |
| 5.2      | MongoDB . . . . .                          | 52        |
| 5.3      | Query Languages . . . . .                  | 53        |
| 5.3.1    | Rumble . . . . .                           | 53        |
| <b>6</b> | <b>Exercises and Quizzes</b>               | <b>55</b> |
| 6.1      | SQL Brush Up . . . . .                     | 55        |
| 6.2      | Object and Key-Value Stores . . . . .      | 55        |
| 6.2.1    | Exploring Azure . . . . .                  | 55        |
| 6.2.2    | Vector Clocks . . . . .                    | 56        |
| 6.2.3    | Merkle Trees . . . . .                     | 58        |
| 6.2.4    | Virtual Nodes . . . . .                    | 58        |
| 6.3      | Storage Models . . . . .                   | 58        |
| 6.3.1    | Hadoop and HDFS . . . . .                  | 58        |
| 6.3.2    | Different Storage Models . . . . .         | 61        |
| 6.4      | XML and JSON . . . . .                     | 61        |
| 6.4.1    | XML . . . . .                              | 61        |
| 6.4.2    | JSON . . . . .                             | 62        |
| 6.5      | Wide Column Stores . . . . .               | 62        |
| 6.5.1    | HBase Architecture . . . . .               | 62        |
| 6.6      | Validation . . . . .                       | 64        |
| 6.6.1    | XML Information Set . . . . .              | 64        |
| 6.6.2    | XML Schema . . . . .                       | 65        |
| 6.6.3    | JSON Schema . . . . .                      | 66        |
| 6.6.4    | Document Validation . . . . .              | 66        |
| 6.6.5    | Dremel . . . . .                           | 66        |
| 6.7      | MapReduce . . . . .                        | 67        |
| 6.7.1    | Reverse Engineering . . . . .              | 67        |
| 6.7.2    | True/False, Facts . . . . .                | 67        |
| 6.8      | YARN and Spark . . . . .                   | 68        |
| 6.8.1    | YARN . . . . .                             | 68        |
| 6.8.2    | Schedulers . . . . .                       | 68        |
| 6.8.3    | Spark Architecture . . . . .               | 69        |
| 6.9      | Spark Dataframes and SparkSQL . . . . .    | 69        |
| 6.10     | Document Stores (MongoDB) . . . . .        | 69        |
| 6.11     | Rumble . . . . .                           | 70        |
| 6.12     | Graph Databases and Neo4j . . . . .        | 70        |
| 6.13     | OLAP and Cubes, Data Warehousing . . . . . | 70        |

# 1 Introduction

Big Data is a portfolio of technologies that were designed to store, manage and analyze data that is too large to fit on a single machine while accommodating for the issue of growing discrepancy between capacity, throughput and latency.

|            | Concepts                               | Technologies           |
|------------|--|------------------------|
| Storage    | Object storage                         | S3, Azure Blob Storage |
|            | Distributed file systems               | HDFS                   |
|            | Syntax                                 | XML, JSON              |
| Models     | Wide column stores                     | HBase                  |
|            | Data models and schemas                | XML/JSON Schema        |
|            | Cubes                                  | OLAP                   |
|            | Graphs                                 | neo4j, Cypher          |
| Processing | 2-step distributed query processing    | Hadoop MapReduce       |
|            | Resource management                    | YARN                   |
|            | DAG-based distributed query processing | Spark                  |
| Management | Document storage                       | MongoDB                |
|            | Query languages                        | JSONiq                 |




Figure 1: Lecture overview.

|                      |                             |
|----------------------|-----------------------------|
| Lots of rows         | Object Storage              |
| Lots of rows         | Distributed File Systems    |
| Lots of nesting      | Syntax                      |
| Lots of rows/columns | Column storage              |
| Lots of nesting      | Data Models                 |
| Lots of rows         | Massive Parallel Processing |
| Lots of nesting      | Document Stores             |
| Lots of nesting      | Querying                    |

Figure 2: Scaling up.

|           |  |
|-----------|--|
| kilo (k)  | 1,000 (3 zeros)                              |
| Mega (M)  | 1,000,000 (6 zeros)                          |
| Giga (G)  | 1,000,000,000 (9 zeros)                      |
| Tera (T)  | 1,000,000,000,000 (12 zeros)                 |
| Peta (P)  | 1,000,000,000,000,000 (15 zeros)             |
| Exa (E)   | 1,000,000,000,000,000,000 (18 zeros)         |
| Zetta (Z) | 1,000,000,000,000,000,000,000 (21 zeros)     |
| Yotta (Y) | 1,000,000,000,000,000,000,000,000 (24 zeros) |

You must know this by ♥!

Figure 3: Prefixes (International System of Units).

## 1.1 Concepts

**Table** A table in a relational model is also called a collection.

**Attribute** An attribute in a relational model is also called a column, field or property.

|           |  |
|-----------|--|
| kibi (ki) | 1,024 (2 <sup>10</sup> )                             |
| Mebi (Mi) | 1,048,576 (2 <sup>20</sup> )                         |
| Gibi (Gi) | 1,073,741,824 (2 <sup>30</sup> )                     |
| Tebi (Ti) | 1,099,511,627,776 (2 <sup>40</sup> )                 |
| Pebi (Pi) | 1,125,899,906,842,624 (2 <sup>50</sup> )             |
| Exbi (Ei) | 1,152,921,504,606,846,976 (2 <sup>60</sup> )         |
| Zebi (Zi) | 1,180,591,620,717,411,303,424 (2 <sup>70</sup> )     |
| Yobi (Yi) | 1,208,925,819,614,629,174,706,176 (2 <sup>80</sup> ) |

You must NOT know this by ♥ !

Figure 4: Prefixes for base 2.

**Primary Key** The identifying attribute is also called row ID or name of a row.

**Row** A row in a relational model is also called a business object, item, entity, document or record.

**Relational Algebra** A table in a relational model is a relation  $R$ . It is made of a set of attributes and an extension (set of tuples). (String to Value mappings)

$$\text{Attributes}_R \subseteq \mathbb{S}$$

$$\text{Extension}_R \subseteq \mathbb{S} \mapsto \mathbb{V}$$

**Tabular Integrity** Each row of a table has the same schema and no empty values.

**Atomic Integrity (1st Normal Form)** The domain of each attribute contains only atomic (indivisible) values and the value of each attribute contains only a single value from that domain. E.g. a column cannot include entire table as values.

**Domain Integrity** Each attribute has a specified data type (domain) and its values are from that domain. E.g. a Boolean column contains only true and false (no zeros and ones).

**SQL vs. NoSQL** SQL has all the integrity properties mentioned above while NoSQL (not relational) can break any or even all of those.

**Relational Queries** We can have:

- **Set:** Union, intersection, subtraction.
- **Filter:** Selection (subset of rows) and projection (subset of columns).
- **Renaming:** Relation and attribute renaming.
- **Binary:** Cartesian product, natural join, theta join.
- **Grouping:** Make groups of the same values in a specific column and summarize the groups items (sum, avg, etc.).

**Normal Forms / Database Normalization** A relational database can be structured to be in accordance to one or a set of normal forms. A normalized database reduces data redundancy and improves data integrity. For data to be consistent, we need to avoid:

- **Update anomalies:** A row with the same identifier has two different values for the same attribute. Happens if there are too many places to update (and they're not linked).
- **Deletion anomalies:** The deletion of unwanted information causes desired information to be deleted as well. Happens if too much information is stored in a single record.
- **Insertion anomalies:** New data cannot be inserted since some information is not available yet. Happens if a new record is missing a value for one of the necessary attributes (new prof, no assigned course yet).

**0NF / UNF** Any table that is not normalized.

**1NF** Each field contains a single value and each row is unique (and can be identified by some primary key).

**2NF** Relation is in 1NF and no non-key attribute is dependent on only a subset of any candidate key.

**3NF** Relation is in 2NF and ...

**Transactions and ACID** A transaction is a set of operations. It is correct if it follows the ACID principles:

- **Atomicity:** Each transaction is atomic, i.e. it is either applied as a whole or undone.
- **Consistency:** A Transaction starts and ends in a consistent state (if it commits)."
- **Isolation:** Intermediate states of a transaction are not seen by any other transactions. Transactions feel like they're alone.
- **Durability:** Committed updates do not disappear again (in case of failures).

**Index**

**OLTP vs. OLAP**

**RDBMS vs. Big Data**

**1.2 SQL Brush Up**

**1.3 Exercises**

**1.3.1 SQL Brush Up**

## 2 Storage

Storage models for processing large amounts of data when using multiple machines with commodity hardware. In Big Data scenarios, we mostly read data (very little modification besides uploading data at the beginning).

Lowest level of the stack - we have: local filesystems, NFS, GFS, HDFS, S3, Azure Blob Storage, etc.

**Scaling Up vs. Out** Scaling up refers to enhancing a single machine (more cores, more processing power, etc.) and scaling out refers to using more machines. The first is more expensive in terms of hardware cost while the latter has just linear cost. A data center uses the second approach to process large amounts of data.

**Data Center** There are typically 1'000 to 100'000 machines (electricity and cooling are logistic limits for the amount of machines) in a data center with each server having 1 to 100 cores. Per server, we have a local storage of 1 to 20TB and RAM of 16GB to 6TB. The network bandwidth of a server typically lies between 1 and 100 GBps. Data centers are made out of racks, which are modular units consisting of rack units = nodes (servers, storage, routers, etc.).

**CAP Theorem** The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

- **Consistency:** Every read receives the most recent write or an error (all nodes see the same data). This is different from ACID consistency.
- **Availability:** Every request receives a (non-error) response, without the guarantee that it contains the most recent write. It is possible to query the DB at all times.
- **Partition Tolerance:** The system continues to operate despite an arbitrary number of messages being dropped / delayed by the network between nodes.

If a network partition failure happens we can either cancel the operation (decreasing availability, ensuring consistency) or proceed (providing availability, risking consistency). The CAP theorem stands in contrast to the ACID principles.

### 2.1 Object Storage

Object storage systems allow retention of massive amounts of unstructured data with unlimited scalability (no system to keep up).

**Object** Each object typically includes the data itself, a variable amount of metadata (descriptive info associated with object) and a globally unique identifier.

**Logical Model** A global key-value model maps keys (global IDs) to objects and their associated metadata.

**Logical vs. Physical Model** Careful: a logical model is not the same as the physical model. With OS, the physical model is storing objects and the logical model is the KV model.

**OS vs. FS** File storage is a hierarchy of folders. In OS, the hierarchy is flattened and each object has its own name that includes the folders it would be in in a FS.

**OS vs. BS** Block storage divides files into singular blocks of data that are stored independently, each with a different address (in disk sectors and tracks). In OS, data/metadata/ID is bundled.

**RESTful API** Objects are typically accessed through the HTTP-based RESTful API. To locate an object, the API queries its metadata via the Internet from anywhere - we then know where to route our requests to for a specific object.

### 2.1.1 Amazon S3

A service offered by Amazon Web Services (AWS) that provides object storage through a web service interface.

**S3 Model** Storage is divided into buckets, each containing multiple objects. To access an object, we need to know the bucket ID and the object ID. The size of an object in a bucket can be at most 5TB and an S3 accounts can have at most 100 buckets. Objects are a blackbox (details not known - physical model is proprietary, we only know the logical model).

S3 can be used to host a static website<sup>1</sup> or for dataset hosting.

#### S3 SLAs

- **Durability:** Loss of 1 in  $10^{11}$  objects in a year (99.999999999%).
- **Availability:** Down 1h per year (99.99%).
- **Response Time:** In 99.9% of the cases, the response time is sub 10ms.

There are different classes one can choose when using S3. The standard storage class promises high availability, middle is less availability and cheaper storage and Amazon Glacier is low cost but a GET might take hours.

**S3 API** S3 can be accessed through many different APIs and drivers (PHP, Java, Python, etc.). The most common API is the REST API.

#### REST API

- **URI:** A resource = object in S3 has its own URI (uniform resource identifier).
- **HTTP:** Buckets / objects are queried using HTTP methods (GET, PUT, DELETE and POST)<sup>2</sup>. Each request is answered with a status code.

---

<sup>1</sup>Example URI: `http://<bucket-name>.s3-website-us-east-1.amazonaws.com/`

<sup>2</sup>POST vs. PUT: the first is to modify and update a resource and the latter is used to create a resource (or overwrite it). PUT is idempotent - sending a PUT request multiple times results in the same effect as sending it once. PUT:  $x = 5$  and POST:  $x++$ .



**URI** First we have the scheme (http), then the authority (www.name.com), the path and after the ? is the query with the fragment (starting with #). Example: `http://www.example.com/api/collection/foo/object/bar?id=foobar#head`

## HTTP Status Codes

**Replication** Data replication is used to be fault tolerant (in case of failures, local vs. regional) and to improve latency (geographical).

### 2.1.2 Microsoft Azure Blob Storage

See reading assignment "Windows Azure Storage: A Highly Available Cloud Storage Service With Strong Consistency". The following is a brief summary of the paper. There are many more details on the SL and PL (and design choices) that have not been included in this brief summary.

#### Introduction

- Scalable cloud storage system, only pay for what you use and store.
- Local and geographical replication of data for durability.
- Blob storage (files), Table storage (structured) and Queue storage (message delivery between application components - middleware service).
- Claim: all three properties in the CAP theorem are provided.
- Global namespace to access data from anywhere.
- **WAS is responsible for managing the replication and data placement across disks and load balancing data and application traffic within a storage cluster.** It is given network topo. info, physical layout of the clusters and hardware configs of the storage nodes (from the Fabric Controller - not important detail).
- Multiple locations, each its own datacenter with multiple storage stamps. All components are extensible (new regions, locations, new SS, etc.).

#### Global Partitioned Namespace

- All data is accessible via URIs: `http(s)://AccountName.<service>.core.windows.net/PartitionName/ObjectName` - routed to virtual IP of primary storage stamp.
- AccountName: customer selected name (globally unique) for storage account.
- DNS is used to map AccountName to primary storage cluster and data center where our data is stored (Virtual IP). Primary location is where all requests go to reach data for that account. Applications might use multiple AccountNames to store data across different locations.
- PartitionName: locate data once storage cluster is reached.
- ObjectName: if PartitionName holds multiple objects, this name identifies each one. Atomic transactions across objects with same PartitionName are supported.
- Blob - blob name = PartitionName; Table - row = primary key = PartitionName and ObjectName; Queue - queue name = PartitionName and message = ObjectName.

**High Level Architecture** See Figure 5 for an overview.

- **Storage Stamps (SS):** Cluster of  $N$  racks (typically 10-20) with each rack in a different fault domain (own networking and power). Each rack typically has 18 disk-heavy storage nodes. Only filled until 70-80%. Primary SS is the one closest to account holder - requests are routed to it.
- Partition and stream servers (daemons in respective layer) co-located on each node in SS.
- Front-end layer: stateless servers taking incoming requests (authentication and authorization of AccName and request, routing to partition server in PL based on PartitionName). Keep track of which PartitionName is served by which PL server in a cache.
- **Location Service (LS):** Manages all SSs and the account namespace across SSs. Accounts are allocated to SSs and managed across SSs for disaster recovery and load balancing (migration). Resources of each SS are tracked.
- **Replication:** Intra-stamp (hardware failure) managed by SL and synchronous (only ACK request once we replicated) - replicates blocks of disk storage. Inter-stamp (geo-redundancy) managed by PL and asynchronous (background) - replicates objects and transactions.

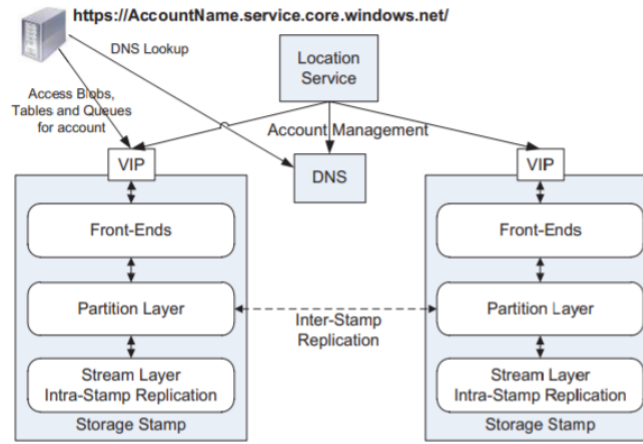


Figure 5: High-level architecture of WAS.

### Stream Layer (SL)

- Layer storing bits on disk, i.e. DFS within a SS.
- In charge of storing, distributing and replicating (intra) data across many servers within SS.
- **Stream:** = file = ordered list of extent pointers - managed by stream manager. Can be opened/closed/deleted/renamed/read/appended to/concatenated by PL.
- Extent: sequence of append blocks (up to 1GB in total). Unit of replication - typically 3 times.
- Block: up to  $N$  bytes (typically 4MB). Minimum data unit to read and append (writes are append-only). Accessed by giving offsets (random read of stream possible).
- PL keeps track of object to stream / extent / block mappings.

## Partition Layer (PL)

- Data on SL is accessed through PL.
- Manages and processes higher level data abstractions (blob, table, queue) and ObjectNames. Stores and caches object data on top of SL.
- Provides transaction ordering and strong consistency for objects.
- Each partition (PartitionName) is served by a different partition server.

## Design Choices / Lessons Learned

- **Separate Storage and Compute:** Easier to scale both components independently. Layer of isolation in between in multi-user scenario. Each component does its own load balancing. Needs high bandwidth between the two.
- **Range Partition (Index) instead of Hashing:** Better locality and isolation for each object.
- **Append-Only System:** Simplifies replication protocol and handling of failure scenarios. Allows snapshots. Issue diagnosis easier. Needs good garbage collection (extra I/O).

**Blob** Binary large object. Collection of binary data stored as a single entity in a DBMS - usually very large ones (multimedia objects).

**S3 vs. Azure** See Figure 6.

|            | S3              | Azure   |
|------------|-----------------|---|
| Object ID  | Bucket + Object | Account + Container + Blob                        |
| Object API | Blackbox        | Block/Append/Page                                 |
| Limit      | 5 TB            | 4.78 TB (block)<br>195 GB (append)<br>8 TB (page) |

Figure 6: S3 vs. Azure.

## 2.2 Key-Value Store

### Object Store vs. Key-Value Store

- In an object store, the latency is much larger than in a typical DB (100-300ms vs. 1-9ms).
- Similar to object storage: store unstructured data (value) that is addressed and located with a key.
- In KVS, the object is much smaller than in OS (5TB vs. 400KB in Dynamo).
- KVS do not have metadata but keys can have variable length.
- Simple API: get(key) and put(key, value) (delete is just a put of nothing).
- KVS are very simple with little features and eventual consistency (needs conflict resolution). This allows for better performance (high availability) and scalability compared to a normal RDBMS.

- The most efficient data structure to query a KVS is an associative array aka. a map (single machine: tree / hash map, dis. machine: Dynamo).

See reading assignment "Dynamo: Amazon's Highly Available Key-Value Store". The following is a brief summary of the paper. Many details are left out.

## Introduction

- Always-on experience - sacrifice consistency under certain failure scenarios (CAP theorem).
- Eventual consistency: object versioning and application-assisted conflict resolution (quorum-like technique). All updates reach all updates eventually.
- Partition and replicate data using consistent hashing.
- Gossip-based distributed failure detection and membership protocol.
- Decentralized.
- Simple key/value interface.
- See summary of techniques in Figure 7.

| Problem                            | Technique   | Advantage   |
|------------------------------------|---|---|
| Partitioning                       | Consistent Hashing                                      | Incremental Scalability   |
| High Availability for writes       | Vector clocks with reconciliation during reads          | Version size is decoupled from update rates.  |
| Handling temporary failures        | Sloppy Quorum and hinted handoff                        | Provides high availability and durability guarantee when some of the replicas are not available.                  |
| Recovering from permanent failures | Anti-entropy using Merkle trees                         | Synchronizes divergent replicas in the background.  |
| Membership and failure detection   | Gossip-based membership protocol and failure detection. | Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information. |

Figure 7: Techniques used in Dynamo.

## System Assumptions and Requirements

- Data stored as blobs, identified by unique keys. Data is small (sub 1MB).
- Simple read and write operations that don't span multiple data items.
- No need for relational schema.
- No isolation guarantee, only single key updates.
- Commodity hardware.
- Non-hostile environment (Dynamo used by Amazon's internal services).

## Service Level Agreement (SLA)

## Design Considerations

- Conflict resolution - when? We can always write but if we read, we need to do CR first.
- Conflict resolution - who? By application since data store CR is limited (last write wins). Application can choose CR method based on data schema and goals.
- Incremental scalability (add/remove nodes easily - needs dynamic partitioning).
- Symmetry (each node has the same responsibilities).
- Decentralization (P2P techniques, all nodes connected with all other nodes).
- Heterogeneity (proportional work distribution if nodes have different amount of resources).
- Each node knows enough routing information for zero hop latency.

## Background

- **P2P Systems:** Unstructured (arbitrary links, "search" query flooded to find all peers holding data). Structured (any node can efficiently route requests to other node holding requested data - reduce hops by maintaining local routing info).
- **DFS:**

## System Architecture

- **Interface:** `get(key)` that gives all object replicas and their associated contexts. `put(key, context, object)` that writes object with key and context. Context is used for conflict resolution (see later). Keys are hashed into 128-bit IDs to locate the storage nodes responsible for serving.
- **Partitioning:** Consistent hashing to spread load over all storage nodes (see below).
- **Replication:** Per-key coordinator node replicates its data across  $N - 1$  nodes (clockwise successors). Per-key preference lists keep track of which nodes are allowed to store the associated objects (more than  $N$ ). On a put-call, data is replicated asynchronously.
- **Data Versioning:** Modifications are treated like new items (in a put, always define which version is being modified - context). Vector clocks are used to resolve conflicts between modifications of same item (see below).
- See more architecture components below.

**Consistent Hashing** Output range of hash function treated as ring ( $\text{mod } 2^n$  for wrap around). Each storage node assigned to random position on ring. Hash key of object, store in node located clockwise of its hash value. Adding and removing nodes just re-assigns (transfer) all object accordingly.

**Uniform Distribution and Heterogeneity:** Randomly distribute virtual nodes (tokens) across ring. Actual nodes are assigned one or more tokens according to their capacity. Allows for better load balancing (failure, adding/removing nodes).

**Vector Clocks** Assume the same object is updated multiple times. See Figure 8 for an example.

- Create new object - written by node A.
- Modify this object - write by node A.
- Both node B and node C modify the object last modified by A at the same time.
- Client wants to read object and receives both versions. Client chooses the correct version based on application-specific heuristics and orders node A to write the correct version (or a new value) of the object.

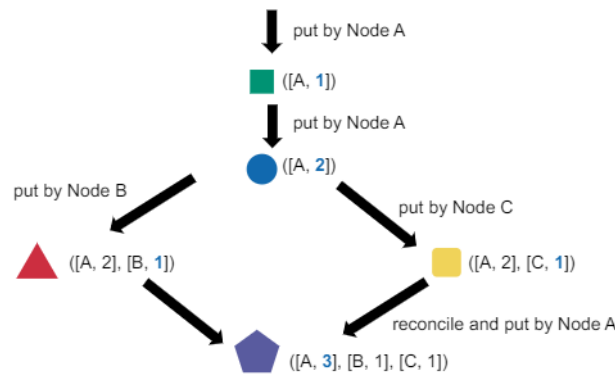


Figure 8: Vector clock example.

### Get and Put Execution

- **Select Node:** Either through a generic load balancer that chooses node based on some LB heuristics, that node then routes requests (to coordinator) or with a partition-aware client that routes requests directly to appropriate coordinator node.
- **Coordinator Node:** Handles get/put requests, typically first of the  $N$  key-specific nodes. Top  $N$  healthy nodes of preference list are involved after that.
- **Consistency:**  $R$  = min. number of nodes ACKing reads.  $W$  = min. number nodes ACKing writes. Quorum-like system if  $R + W > N$ . Latency of operation determined by slowest node of the  $R / W$  nodes. Clients can tune those values!  $((N, R, W) = (3, 2, 2)$  typically).
- **Put:** Coordinator generates vector clock, writes new version locally, sends it to  $N$  highest reachable nodes and succeeds if  $W - 1$  respond.
- **Get:** Coordinator requests all versions from  $N$  highest reachable nodes, waits for  $R$  responses and returns all of them to client. Client reconciles and writes back newest version.

**Failure Handling (Hinted Handoff)** If an intended recipient node is unreachable during a write, its replica is send to next reachable node on preference list with note that it was originally intended for another node. If original node recovers, temporary node sends intended replica to it.

**Permanent Failures** To keep replicas synchronized (in cases where hinted replicas will not go back to intended), Merkle trees are used. Reduces amount of data that needs to be transferred when checking for inconsistencies. Each node has a Merkle tree for each key range it hosts. Two nodes exchange roots to check.

**Merkle Tree:** Leaves = hashes of values of individual keys. Parent nodes = combined hashes of children.

### Membership Detection

- First time: node gets tokens, mapping kept in disk of node.
- Reconciliation s.t. each node has its own tokens.

**Failure Detection** If node B does not respond to node A's messages, node A considers B failed (even if it might not actually be). A periodically retries during traffic.

**Ensuring Uniform Load Distribution** Divide hash ring into  $Q$  equally sized partitions. Each node is assigned  $Q/S$  tokens where  $S$  is total number of active nodes in system.

## 2.3 Distributed File Systems

Use OS with the KV model when dealing with a huge amount of large files and use BS with the FS model when dealing with a large amount of huge files. Now, throughput is top priority (latency is secondary).

Instead of doing random access of small parts of a file, we want to scan entire files (good for batch processing). With files, writes are append-only.

**Parallelism vs. Batch Processing** Capacity has grown a lot, throughput and latency hasn't. To achieve higher throughput with lots of capacity, we can parallelize (work on different data at the same time). To achieve low latency with lots of capacity we can do batch processing (move and process large amounts of data as one unit).

**FS Logical Model** Files are stored in a hierarchically instead of flat as in the KV model.

**BS Physical Model** A file is divided into a sequence of blocks that are exposed to the user. Files need to be broken down since they can be bigger than a single disk. In contrast, OS objects are treated as a blackbox package / collection of bits - the user does not see a further division of an object.

**Block Size** In a simple FS, blocks are usually 4KB. In a RDBMS, blocks are between 4KB and 32KB. In a DFS, blocks are between 64MB and 128MB.

If the size is too small, we run into latency issues (too much individual fetching). If it's too big, we run into storage issues and strain the network bandwidth. Choose the size s.t. system is dominated by throughput and not latency.

**Hadoop** Framework for distributed (parallel) processing of large data sets across clusters of computers using simple programming models. Each machine can do local computations and has local storage - compute close to data. Hadoop includes multiple components: HDFS, HBase, MapReduce, YARN, etc.

See reading assignment "Hadoop: The Definitive Guide". The following is a brief summary of chapter 1.

- Disk storage capacities massively increased - throughput has not kept up (data reading rate).
- Read and write time can be reduced if we read / write from multiple disks at once = parallelism.
- Batch processing utilized by MapReduce - for each query, the entire dataset can be processed (see later section).
- MapReduce is slow and should be used for offline use.
- For online access: use HBase (KVS over HDFS). Provides online read/write access of individual rows and batch operations for reading / writing data in bulk.
- Better processing: use YARN (cluster resource management system) - allows distributed programs to run over Hadoop cluster.
- Compared to RDBMS: lots of data, batch access instead of interactive and batch, write once and read many times, no ACID, schema-on-read, low integrity, linear scaling.
- Semi and unstructured, denormalized data.

### 2.3.1 Hadoop Distributed File System (HDFS)

See reading assignment "The Hadoop Distributed File System". The following is a brief summary of the paper. Some details and the entire "Practice at Yahoo!" section are left out. Also, see this video for an easy explanation of HDFS: [click](#).

#### Introduction

- Store large data sets reliably and stream them at high bandwidth to user applications.
- Storage and computation is distributed across many servers - easy scalability.
- Commodity hardware.
- Metadata is stored in NameNodes and application data is stored in DataNodes.
- All servers are fully connected (master-slave architecture), communication protocols based on TCP.
- Reliability through replication of file content across multiple DataNodes.
- Single-writer, multiple-reader model (closed files can only be appended to).

**Architecture** Master-slave architecture with NameNode as master and multiple DataNodes as slaves. Single NameNode per cluster. See Figure 9 and 10.



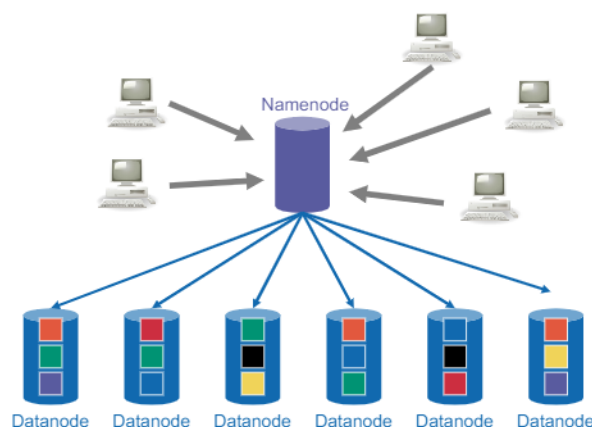


Figure 9: HDFS architecture.

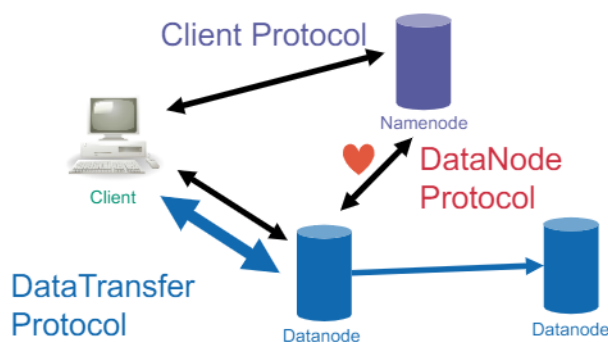


Figure 10: All communication protocols in HDFS.

**File** File content is split into large blocks (typically 128MB, user selectable per-file) - HDFS blocks are not the same as disk blocks. Each block has a 64-bit block ID. Blocks of a file are independently replicated at multiple DataNodes (typically 3, user selectable per-file). Replica locations can change over time (not part of persistent checkpoint).

**NameNode** Keeps track of system-wide activity. It maintains the following in RAM: namespace tree, file to block mapping and mapping of file blocks to DataNodes (i.e. physical location of file data).

On restart, a namenode can recover the previous state by reading the namespace file (path to block ID) mapping and replaying the journal stored on its local disk (see below).

**Instructions to DN:** replicate blocks to other DNs, remove local replicas, re-register/shutdown, send block report.

**Namespace** Hierarchy of files and directories - represented as inodes on the NameNode (in RAM). Namespace is a tree with files as leaves, thus each file is a specific path in the tree.

A namespace gets an ID which is assigned to a HDFS instance when it is formatted. ID is persistently stored on all nodes of the cluster. Cluster belongs to one HDFS instance only!

**inode** Records attributes such as permissions (POSIX style), modification / access times, namespace / disk space quotas, etc.

**Metadata** Inode data plus list of blocks belonging to a file. Collection of all metadata = image. A persistent image is kept on the local disk of the NameNode (= checkpoint). A modification log of the image is also stored on the local disk = journal. Journal is write-ahead commit log.

**DataNode** Stores its HDFS blocks on local disk. Blocks can be cached in DN RAM. Each HDFS block is stored as two disk files (actual data and associated metadata). Disk files can be further divided - depends on disk, this allows access of values in the middle of a HDFS block. This also means that a 1MB file put into a 128MB HDFS block uses only 1MB of local disk space and does not fill up entire 128MB on local disk since local disk block sizes are used to store).

**Startup:** Connect to NameNode and perform handshake to verify (or assign) DataNode's namespace ID (and HDFS version). After the handshake, the DataNode registers with the NameNode with their persistently stored and never changing storage ID (assigned by a NameNode if first time).

**Block Reports:** Every hour, a DN sends a block report to NN - consists of block ID, generation stamp and block length of each block it hosts.

**Heartbeat:** Every 3 seconds, DN tells NN that it is alive and its blocks are available (and storage capacity, used capacity, nr. of current transactions). NN uses heartbeats for space allocation and load balancing mechanisms. No heartbeat: NN considers DN dead and schedules replication of its blocks to other DNs. NN never directly contacts DN, it sends instructions as replies to heartbeats.

**Client** User application connects to HDFS with HDFS client - read / write / delete files and create / delete directories (control). Files and directories are referenced by namespace paths. API exposes block locations to client for efficient processing.

**Read:** Ask NN for list of DN hosting replicas of blocks of file and the block IDs. Contact DN directly to retrieve blocks (closest DN first).

**Write:** Ask NN to choose list of DNs that should host replicas of first block of file. Send first block to first DN in pipeline, DN then sends it to the other DNs in prepared pipeline. After ACK, repeat for all other blocks. The replicas of the blocks of the file as individually distributed across DNs.

## Replica Management

- NN detects if a block is under- or over-replicated with DN block reports. It then balances this with instructions.
- Replica 1: same node as client (or random) in rack A.
- Replica 2: node in different rack B.
- Replica 3: different node in same rack B.
- Replica 4 and beyond: random but if possible at most one replica per node and at most two per rack.
- Balancer balances disk space usage in an HDFS cluster - input threshold value (fraction in range from 0 to 1). Cluster is balanced if for each DN, its utilization (ratio of used space to total capacity) differs from utilization of whole cluster by no more than threshold value. It iteratively moves replicas from DN with high util. to DN with low util. Process is optimized by minimizing inter-rack data copying.
- Removing a replica: system tries not to reduce number of racks hosting replica and tries to remove it from DN with least amount of space.

**Distance Calculation** See Figures 11 and 12.

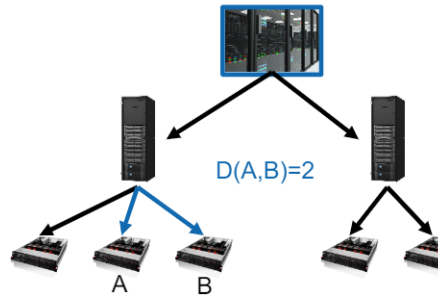


Figure 11: Calculate distance in HDFS.

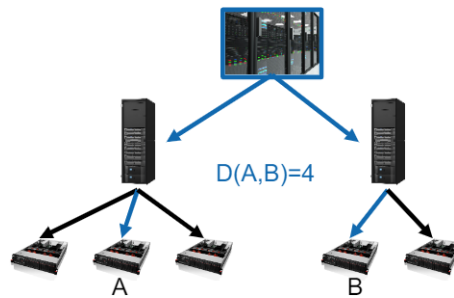


Figure 12: Calculate distance in HDFS.

**Backup** Secondary Namenode keeps namespace image and edit log of primary NameNode, periodically merges namespace image and edit log. (Alternative: federated DFS with multiple NN responsible for partitions of namespace.)

**Hadoop: The Definitive Guide** See reading assignment "Hadoop: The Definitive Guide". The following is a brief summary of chapter 3 (of points not yet mentioned above).

- Huge files: hundreds of MB, GB or TB in size. Hadoop clusters store PB of data. We don't want lots of small files.
- Write-once, read-many-times data processing pattern. Analysis involves large proportion of / entire dataset. Time to read it (throughput) is more important than latency of reading first record.
- Not good for low-latency applications. We have high throughput.
- HDFS block is not the same as disk block!
- etc.

## 2.4 Syntax

Physical representation of data on a syntax level (files). Examples include: text, CSV, XML, JSON, etc. The files we feed into a Big Data storage system are written in any of those formats.

These formats allow for data to be represented in a denormalized (non-atomic values, nestedness) and heterogeneous (rows don't all have values for all attributes) way. Such data can be queried with NoSQL systems.

For read-intensive workloads, denormalized data works well - if we avoid joins (they can be impossible). In contrast, for write-intensive workloads, we want high levels of normalization to avoid update anomalies (and else).

**Comma Separated Values (CSV)** A CSV file can be easily represented as a table. The first row are the attribute names and all following rows are the attribute values. Words are separated by comma (or else in other standards).

**Structured** A normalized table.

**Semi-Structured** Document can be expressed in a tree data model (XML, JSON, etc. - see later).

**Unstructured** A text file.

**Well-Formedness** A document is well-formed if it actually belongs to the language it is written in (abides to the syntax / grammar rules). Else, it is not well-formed.

### 2.4.1 JSON

See reading assignment "JSON ECMA Standard". The following is a brief summary.

#### Introduction

- JSON = text syntax, facilitates structured data interchange between programming languages.
- Language-independent - used to define data interchange formats.
- Meaningful data interchange requires agreement between producer and consumer on semantics attached to particular use of JSON syntax (see later section).
- Agnostic to semantics of numbers. Numbers are simply a sequence of digits.
- Simple notation to express collections of name/value pairs.
- Support for ordered lists of values (that can also be nested).
- No support for cyclic graphs.
- Not for applications requiring binary data.

**Text** Sequence of tokens. Six structural tokens (square brackets, curly brackets, colon and comma), strings, numbers and three literal name tokens (true, false, null). Whitespace is allowed everywhere besides inside tokens (except inside strings).

## Values

- **Object:** Curly bracket tokens surrounding zero or more name/value pairs. Names are always string and followed by single colon. Commas separate value from next name. Names don't have to be unique (but really should be...) and the name/value ordering doesn't matter.
- **Array:** Square bracket tokens surrounding zero or more values separated by commas. Ordering doesn't matter.
- **Number:** Any sequence of decimal digits, leading minus okay, decimal point okay, exponent okay. Inf and NaN not permitted.
- **String:** Sequence of unicode code points wrapped by quotation marks. Some code points must be escaped (see below). Any code point can also be written in hex representation.
- **true / false / null**

**String Escapes** Reverse solidus followed by: quotation marks, reverse solidus, solidus, b for backspace, f for form feed, n for line feed, r for carriage return, t for tabulation character.

**Some Examples** See JSON examples below (taken from the slides).

```
1 {  
2   "target": "Italian",  
3   "sample": "af45858ejdgi02i3n4j",  
4   "choices": ["Chinese", true, {"foo": false}, null, 3.12],  
5   "bar": true,  
6   "object": {"School": "ETH"},  
7   "Q": null  
8   "Exponent": -1.23E+5,  
9   "unicode": "foo\nbar\u005f"  
10 }
```

## 2.4.2 XML

See reading assignment "XML in a Nutshell". The following is a brief summary of chapters 1, 2, 4.1, 4.2 and 16.

### Introduction

- Generic syntax to mark up data with simple, human-readable tags.
- Provides standard format for documents that is flexible enough to be customized for various domains.
- Basically the same as above.
- XML application: tag set defined by individuals / companies if their application only wants to use a subset of XML tags.
- Documents can be associated with an XML schema to check for validity (does the document fit a given mold). See later section.
- XML document contains text, never binary data. XML documents are simply text files with a special structure (just like with JSON) read by XML parsers.

## Fundamentals

- **Element:** Start-tag, content, end-tag. The tags name the element. Elements can contain more elements (surrounded by content). E.g. `<person> Alan Turing </person>` - this element is called "person".
- **Empty Element:** `<person></person>` is the same as `<person/>`.
- XML is case-sensitive.
- XML document is a tree: nested tags with content. Only one root = document element. Each child has only one parent and a parent can have several children. XML doc must start with an element. See example below.
- **Attributes:** Elements can have attributes = name-value pairs, attached to the start-tag. E.g. `<person born="1912-06-23" died="1954-06-07"> Alan Turing </person>` (single instead of double quotes is okay).
- Convention: data is content and metadata / additional info for data is attribute.
- **Names (element / attribute):** May only contain characters defined in Unicode 3.0 and later, no length limit, can contain A-Z, a-z, 0-9, non-English letters, dot, minus, underscore. Don't start with "xml" (reserved). Don't start with a number.
- **References:** escape `<` character with `"&lt;"` tag (entity reference), `"&#60;"` (numeric character reference) or `"&#x3C;"` (hex. num. char. reference). Escape `&` with `"&amp;"` or `"&#38;"`. There is also `"&gt;"` for `>`, `"&quot;"` for `"` and `"&apos;"` for `'`. Sequence `"]>"` must be written `"]&gt;"`.
- **CDATA Sections:** If a text includes many escape characters, put text in a CDATA section: `<![CDATA[ &<>" hello, <<<]]>` - everything in `[]` is treated as raw character data.
- **Comment:** `<!-- example-->`, double hyphen never inside comment text. Comments can appear anywhere. Don't use additional hyphen to close (`-->`).
- **Processing Instructions:** Alternative way to pass info (in any syntax, can be code) to particular application reading the XML document. Begins with `<?` and ends with `?>`. Can appear anywhere. Don't put XML and variations of that into it.
- **XML Declaration:** Looks the same as processing instruction. Not necessary but if included must be very first thing. Always use version 1.0. Default encoding is UTF-8. Encoding specified by metadata wins if in conflict with declaration. Standalone: if not standalone, application might need to read external DTD to determine proper values for parts of doc - default is no.

```
1 <?xml version="1.0" encoding="ASCII" standalone="yes"?>
2 <!-- example XML decl. above -->
3
4 <person>
5   <name>
6     <first_name>Alan</first_name>
7     <last_name>Turing</last_name>
8   </name>
9   <profession>computer scientist</profession>
10  <profession>mathematician</profession>
11  <profession>cryptographer</profession>
12 </person>
13
14 <!-- same as above (with attributes): -->
```

```

15
16 <person>
17   <name first="Alan" last="Turing"/>
18   <profession value="computer scientist"/>
19   <profession value="mathematician"/>
20   <profession value="cryptographer"/>
21 </person>

```

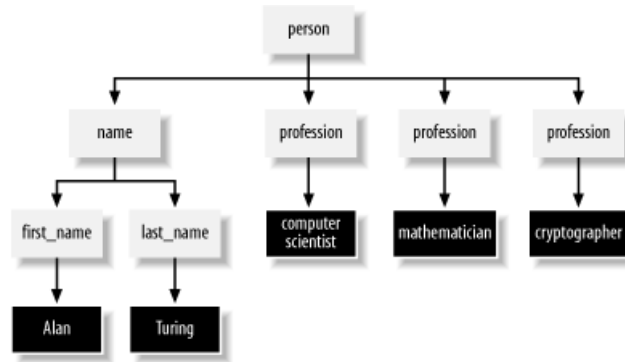


Figure 13: The person XML as a tree.

### Checking for Well-Formedness

- Every start-tag must have a matching end-tag.
- Elements may nest but may not overlap.
- There must be exactly one root element.
- Attribute values must be quoted.
- An element may not have two attributes with the same name.
- Comments and processing instructions may not appear inside tags.
- No unescaped < or & signs may occur in the character data of an element or attribute.
- No whitespace in element tag.
- No "lower case" quotation marks.

**Namespaces** Distinguish between elements / attributes from different vocabularies with different meaning that happen to have the same name and group all related elements / attributes from a single XML application together (easy to recognize for software). E.g. a join of two datasets with different meaning (networks vs. CRM) - two elements might have the name "client".

- With a namespace, a local name turns into an expanded name.
- Namespace implementation: attach a prefix to each element and attribute. Each prefix is mapped to a URI. Same namespace if element / attribute has same prefix.
- URI / IRI doesn't actually have to map to a resolved webpage.
- Qualified name (Qname / Raw Name): prefix:localname and namespace URI.
- Localname identifies element within the namespace.

- If the prefix is not specified, we are simply in the default namespace - use rarely. Prefixed elements are in their namespace but non-prefixed children in default namespace. Unprefixed attributes are never in any namespace even if they're in scope.
- If the local name and namespace is the same but the prefixes differ, we still have the same QName. Same attribute name and namespace but different prefix is not well-formed.
- To make it tidier, namespace bindings can be put into root element.
- Backward compatibility: parser that doesn't know any namespaces will have no problem reading document with.

```

1 <prefix:localname xmlns:prefix="http://example.com/prefix">
2
3 <!-- scope of prefix namespace binding -->
4
5 </prefix:localname>

```

```

1 <!-- example of default namespace: -->
2
3 <math xmlns="http://www.w3.org/1998/Math/MathML">
4
5 </math xmlns="http://www.w3.org/1998/Math/MathML">

```

```

1 <!-- tidy example: -->
2
3 <?xml version "1.0"?>
4
5 <prefix1:foo
6     xmlns:prefix1="http://example.com/prefix1"
7     xmlns:prefix2="http://example.com/prefix2">
8
9     <prefix1:some_name>
10         "Something"
11     </prefix1:some_name>
12
13     <prefix2:other_name prefix2:attr="value">
14         "Hello!"
15     </prefix2:other_name>
16
17 </prefix1:foo>

```

## 2.5 Exercises

### 2.5.1 Object and Key-Value Stores

### 2.5.2 HDFS

### 2.5.3 XML



### 3 Models

RDBMS are for small scale applications running on a single machine. We can either scale up (costly) or scale out (complex, high maintenance cost).

#### 3.1 Wide Column Stores

Wide column stores are a type of NoSQL DB. Also uses tables, rows and columns but the names and the format of the columns can vary from row to row in the same table. Interpret this as a two-dimensional key-value store. In WCS, we store together what is accessed together - good for distributed systems since it reduces the amount of data transferred.

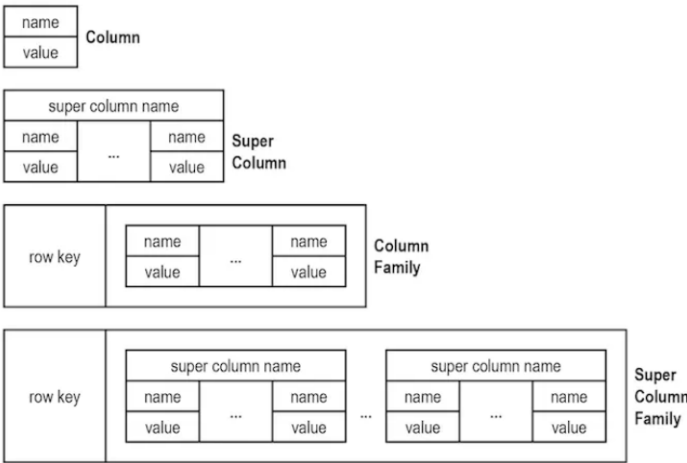


Figure 14: WCS DB objects.

Column families have to be specified in advance, but columns can be added on the fly (see Figure 15). The row ID is always its own column.

| Row ID | A | B | C | 1 | 2 | I | II | III | IV |
|--------|---|---|---|---|---|---|----|-----|----|
| 000    |   |   |   |   |   |   |    |     |    |
| 002    |   |   |   |   |   |   |    |     |    |
| 0A1    |   |   |   |   |   |   |    |     |    |
| 1E0    |   |   |   |   |   |   |    |     |    |
| 22A    |   |   |   |   |   |   |    |     |    |
| 4A2    |   |   |   |   |   |   |    |     |    |

Figure 15: WCS example - individual columns can be added on the fly.

**Usual Column Store** CS columnar data layout is adopted s.t. each column is stored separately on disk. In WCS, column families containing multiple columns are stored together in a row-by-row fashion by column family.

##### 3.1.1 Bigtable

See reading assignment "Bigtable: A Distributed Storage System for Structured Data". The following is a brief summary of this paper.

## Introduction

- **Bigtable** is a distributed storage system for managing structured data that is designed to scale to a very large size (PB across thousands of servers).
- Build on top of GFS (similar to HDFS).
- Serves applications with varying demands (data size and latency). Throughput-oriented batch processing vs. latency-sensitive jobs serving data to real-time users.
- Clients have dynamic control over data layout and format (locality).
- Wide applicability, scalability, high performance, high availability.
- No support for a full relational model.
- Data is indexed using row and column names (arbitrary strings). Data is treated as uninterpreted strings (clients serialize structured / semi-structured data into these strings).
- Data can be served out of memory or disk (client decision).

**Data Model** Sparse, distributed, persistent multi-dimensional sorted map. Example in Figure 16.

(row:string, column:string, time:int64)  $\rightarrow$  string

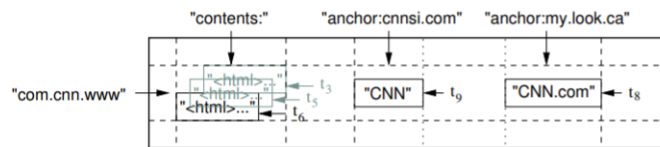


Figure 16: Bigtable Web pages example.

**Rows:** Row keys are arbitrary strings (up to 64KB, usually sub 100B). Read/write under single row key is atomic. Sorted in lexicographical order. Table row range is dynamically partitioned (horizontally) into **tablets** (unit of distribution and load balancing).

**Column Families:** Column keys are grouped into column families = basic unit of access control and disk/memory accounting. Data in CF usually of same type. Families created in advance - small number, individual cols unbounded number. Column key: *family:qualifier* (qualifier can be empty if family contains only one col).

**Timestamps:** Cells can contain multiple versions of same data indexed by a 64b timestamp. For applications wanting to avoid collisions, timestamps have to be unique. Most recent version is read first. Client specifies: either keep only last  $n$  values or only new-enough values.

## API

- Create and delete tables and col. families.
- Changing cluster, table and col. family metadata (e.g. access control rights).
- Write, delete values. Look up values from individual rows or iterate over subset of data.
- Iterate over multiple col. families, possible to limit rows, cols, timestamps produced by a scan.
- Single-row transactions to perform atomic read-modify-write sequences (single row key).
- No transactions over several row keys but writes across row keys can be batched.

- Cells can be used as integer counters.
- Client-supplied scripts can be executed in address space of servers (Sawzall language).
- MapReduce can be used.

## Building Blocks

- On top of GFS - stores log and data files.
- Bigtable depends on cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures and monitoring machine status.
- Store any Bigtable data: **SSTable** file format used internally = persistent and ordered immutable map from keys to values (arbitrary byte strings).
- Point queries and range queries are possible (key-value operations).
- SSTable divided into sequence of GFS blocks. At end of SSTable - block index keeping track of GFS block locations.
- Opening SSTable - block index is loaded into memory.
- Lookup: single disk seek (binary search on in-memory index and read appropriate block from disk).
- Optionally: completely map SSTable into memory (no disk seeks).
- Bigtable relies on highly available and persistent distributed lock service **Chubby**. Why: store bootstrap location of Bigtable data, discover tablet servers, finalize tablet server deaths, store schema info (col. family info for each table), store access control lists.

**Chubby Service:** Five active replicas (Paxos for consistency), one is master service and serves requests. Namespace with directories and small files, both of which can be used as locks. Atomic reads and writes to files.

**Chubby Client:** Consistent caching of Chubby files. Clients maintain session with service (needs to be renewed continuously, else client loses locks).

**Implementation** Library linked into every client, one master server, many tablet servers (dynamic add / remove possible).

- **Master Server:** Assigning tablets to tablet servers, detect addition / expiration of tablet servers, balancing load, garbage collection of GFS files, handle schema changes (table / col. family deletion / creation).
- **Tablet Server:** Manages set of tablets, handle read / write requests, splits too large tablets.
- **Client Communication:** Not through master! Communication directly to tablet servers (read / write). Client does not have to ask master for tablet location information.

**Tablet Location** Three level hierarchy to store tablet location information in memory (see Figure 17). Chubby file contains location of root tablet. Root tablet (never split) contains location of all tablets in special METADATA table. Each METADATA tablet contains location of a set of user tablets (row key = encoding of tablet's table ID and end row).

Client can cache tablet locations (GFS accesses not required). Clients also prefetch table locations.

METADATA also stored log of all events pertaining to each tablet for debugging and performance analysis.

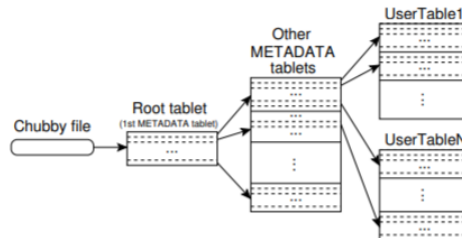


Figure 17: Tablet location hierarchy.

**Tablet Assignment** Each tablet assigned to one tablet server at a time. Master keeps track of live TS and current assignments (and unassigned tablets). Unassigned tablets are assigned to TS with enough room by master.

Chubby keeps track of tablet servers. On startup, TS receive x-lock on unique file in Chubby directory (monitored by master to discover TS).

Master reassigns tablets from dead TS into unassigned pile. Monitor TS with periodic check of lock status. Current assignments don't change if master fails.

(More details are omitted here).

**Tablet Serving** Persistent state of a tablet stored in GFS. Updates are committed to a commit log storing redo records. (Recovery details omitted).

Write: TS checks well-formedness and user authorization. If valid, write to commit log stored in GFS (disk). Write contents stored in memtable (RAM).

Read: TS checks well-formedness and user authorization. If valid, execute read on merged view of sequence of SSTables and memtable (disk and RAM). Merge easy since both are lexicographically sorted.

## Compactions

- Minor compaction: if memtable to big, write to GFS as SSTable and create new memtable.
- Merging compaction: periodically executed to merge sequence of SSTables and memtable (in background) into a new SSTable.
- Major compaction: rewrite all SSTables into one - contains no deleted values.

## Refinements

- **Locality Group:** Client group multiple col. families together into separate SSTable for efficient reads. Locality groups can be declared to be in RAM (lazy loading). Useful for small pieces of data accessed frequently. E.g. used for METADATA table.
- **Compression:** Client can choose if and how LG on SSTable is compressed - applied to each SSTable block (don't need to decompress whole table for small read).
- **Caching:** Speed up reads with two layer caching at TS. One for frequent and one for locality.
- **Bloom Filters:** Reduce number of disk accessed on a read (SSTables) by using BF in SSTables for specific LG (chosen by client). If SSTable might contain data we want, it is fetched.
- **Commit Log:** ommitted.
- **Tablet Recovery:** ommitted.
- **Exploit Immutability:** ommitted.

**Performance Evaluation** Ommitted.

**Lessons Learned** Ommitted.

**Related Work** Ommitted.

### 3.1.2 HBase

See reading assignment "HBase: The Definitive Guide". The following is a brief summary of chapters 1, 3 and 8. Some things are also from chapter 20 of "Hadoop: The Definitive Guide".

#### Basics

- **HBase is a distributed column-oriented database built on top of HDFS. Used when real-time read/write random access on very large datasets is required.**
- HBase scales linearly just by adding nodes.
- Not relational, no support for SQL.
- Hosts very large, sparsely populated tables on clusters made from commodity hardware.
- **Data Model:** Applications store data in labeled tables. Tables are made of rows and columns. Table rows are sorted by row key (primary key, byte arrays) - byte-ordered. Table cells — the intersection of row and column coordinates — are versioned. By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion. A cell's content is an uninterpreted array of bytes (same as Bigtable). See example in Figure 18.
- Physically, column families are stored together on HDFS. Members should have same characteristics and access pattern.
- **Regions:** Tables automatically partitioned horizontally into regions (same as tablets in Bigtable). First row inclusive, last row exclusive.
- **Locking:** Row updates are atomic.

- Same physical layout as HDFS with different words (see Figure 19).
- HMaster responsible for bootstrapping, assigning regions to registered regionservers and for recovering regionserver failures. Regionservers serve client read/write requests and manage region splits.

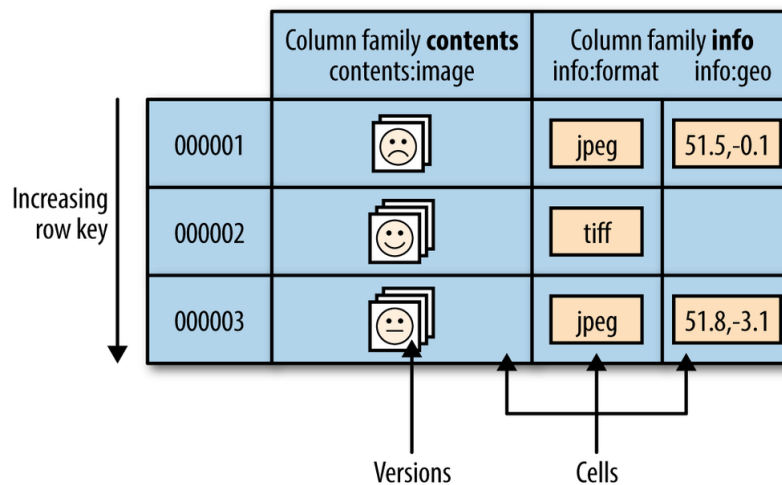


Figure 18: HBase data model example.

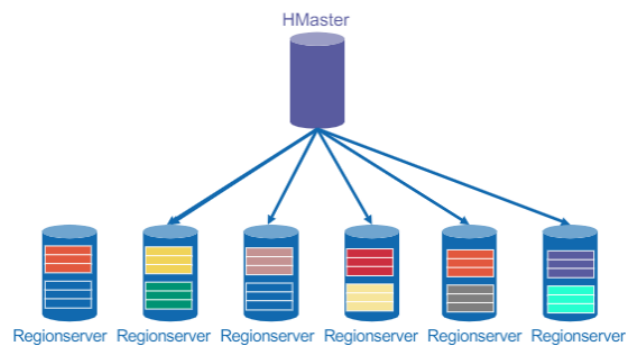


Figure 19: HBase implementation model.

<http://www.benstopford.com/2015/02/14/log-structured-merge-trees/>

## 3.2 Data Models and Schemas

Physical view: actual syntax (CSV, JSON, XML, etc.). Logical view: data model (e.g. relational table). A physical view can be interpreted with different logical views. E.g. relational table, Excel spreadsheet, etc.

**Structured Data** Data that is easy to search and organize, usually contained in rows and columns with its elements mapped into fixed pre-defined fields. Often managed by SQL.

**Unstructured Data** Data that cannot be contained in a row-column database and that doesn't have an associated data model. Difficult to search, manage and analyse. Usually accessed and processed with ML / AI technologies. E.g. text.

**Semi-Structured Data** Mix of the two above. Data with defining / consistent characteristics but not conforming to rigid structures - usually just data where heterogeneity and nestedness is allowed. Usually comes with metadata to make it easier to organize. E.g. Email messages (text unstructured but everything else structured).

### 3.2.1 JSON Data Model

JSON documents can be interpreted as trees (see Figure 20). In JSON, labels are on the edges (in contrast to XML where labels are in nodes).

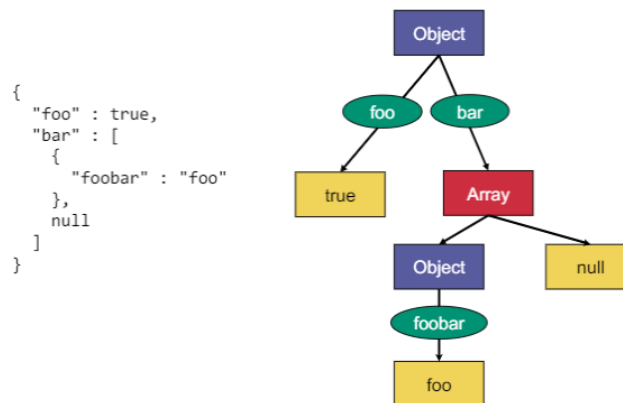


Figure 20: Tree-based visual model of JSON.

With JSON, structured and semi-structured data can be represented. Design of a document depends on its use. With a JSON schema, we can define the type of JSON data an application wants.

### 3.2.2 JSON Schema

See reading assignment "Understanding JSON Schema". The following is a brief summary of the document.

**Introduction** JSON Schema is a tool for describing and validating the structure of JSON data. A JSON schema is itself written in JSON. JSON Schema is used to validate structure, but not semantics of a JSON document.

**Declaration** To declare that a JSON document is a schema, use the `$schema` keyword (not required, just good practice).

```
1 { "$schema": "http://json-schema.org/schema#" }
```

**Unique Identifier** Good practice to include `$id` property as a unique identifier for each schema.

```
1 { "$id": "http://yourdomain.com/schemas/myschema.json" }
```

**Accept Any Valid JSON** The schema is simply the empty object `{}` or `true` (`false` for a schema rejecting everything).

**Restrict Type** The data type of JSON document to accept can be specified like these few examples (invalid since many top-level types):

```
1 { "type": "string" }
2 { "type": ["number", "string"] }
3 { "type": "object" }
4 { "type": "integer" }
5 { "type": "array" }
6 { "type": "boolean" }
```

Type restrictions can have additional keywords (i.e. attributes):

```
1 {
2   "type": "string",
3   "minLength": 2,
4   "maxLength": 3
5 }
6
7 {
8   "type": "number",
9   "multipleOf": 1.0,
10  "minimum": 0,
11  "exclusiveMaximum": 100
12 }
13
14 {
15   "type": "array",
16   "items": { "type": "number" }
17 }
18
19 {
20   "type": "array",
21   "contains": { "type": "number" }
22 }
23
24 {
25   "type": "array",
26   "minItems": 2,
27   "maxItems": 3
28 }
29
30 {
31   "type": "array",
32   "uniqueItems": true
33 }
```

**Regular Expressions** To restrict a string to a particular regex, use the pattern keyword:

```
1 {
2   "type": "string",
3   "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
4 }
```



**Format** With the format keyword we can do basic semantic validation on certain kinds of commonly used string values. E.g. date/time, email, hostnames, ipaddr, uri/iri (template), JSON pointer, regex.

**Properties** Leaving out or additional properties is valid (and therefore also the empty object). See example:

```
1 {
2   "type": "object",
3   "properties": {
4     "number": { "type": "number" },
5     "street_name": { "type": "string" },
6     "street_type": { "type": "string",
7                     "enum": ["Street", "Avenue", "Boulevard"]
8                       }
9   }
10 }
```

The above schema accepts for example:

```
1 { "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

To disallow additional properties, use this after property object:

```
1 "additionalProperties": false
```

To allow additional properties with a specific type, use this after property object:

```
1 "additionalProperties": { "type": "string" }
```

To specify which properties are required, use this after property object:

```
1 "required": ["name", "email"]
```

**Property Names** To not define the value but only the key name, use (for example):

```
1 {
2   "type": "object",
3   "propertyNames": {
4     "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
5   }
6 }
```

**Size** To limit the amount of properties of an object, use (for example):

```
1 {
2   "type": "object",
3   "minProperties": 2,
4   "maxProperties": 3
5 }
```

## Dependencies

## Pattern Properties

**Array Tuple Validation** Each item in an array has a different schema and the ordinal index of each item is meaningful. JSON document does not need to contain all items or can contain more - attention: order is important! Missing or additional items only at the end (set `additionalItems` to `false` if you don't want any, same with object examples above). Example:

```
1 {
2   "type": "array",
3   "items": [
4     {
5       "type": "number"
6     },
7     {
8       "type": "string"
9     },
10    {
11      "type": "string",
12      "enum": ["Street", "Avenue", "Boulevard"]
13    },
14    {
15      "type": "string",
16      "enum": ["NW", "NE", "SW", "SE"]
17    }
18  ]
19 }
```

### 3.2.3 XML Information Set and Schema

### 3.2.4 In General

types, cardinality

### 3.2.5 Protocol Buffers

### 3.2.6 Validation

json / xml = tree (physical vs logical model)

logical model needed for writing code / doing operations (abstraction level)

general model for logical data independence (atomic types, structured data, etc) - what all models have in common

protocol buffers (vs. textual serialization models like json and xml)

and rest

### 3.3 Graph Databases

**Why** Easier joins with more efficient relationships between primary and foreign keys. Relationships allow data in the store to be linked together directly and often retrieved in a single operation. Good for heavily inter-connected data.

#### Components

- Nodes and (un)directed edges.
- Properties for edges and nodes
- Labels (=names) for edges and nodes

#### Graph Representations

- Adjacency list: node and edges table, for each node, keep a list of nodes connected to it.
- Adjacency matrix: rows and cols are nodes, 1 if connected, 0 if not.
- Incidence matrix: rows nodes, cols edges, 1 if edge is incoming, -1 if it is outgoing, 0 if not connected to node.

**Graph as Table / RDF** A graph can be stored as a triple-store table (RDF), i.e. attributes are: source (node), target (node) and name (edge) = subject, object, property. E.g. Alice, Bob, knows. Nodes can be blank.

|            | Subject | Property | Object |
|------------|---------|----------|--------|
| IRI        | YES     | YES      | YES    |
| Literal    | NO      | NO       | YES    |
| Blank node | YES     | NO       | YES    |

Figure 21: RDF restrictions (in generalized graphs, all is allowed).

#### Syntax

**Querying** With Cypher (for general) or with SPARQL (for RDF). Cypher does pattern matching to find all subgraphs matching shape of query (also a graph).

##### 3.3.1 Neo4j

- Neo4j is an architecture for graph databases.
- No sharding! Graphs are stored as a whole for fast traversal.
- Master-slave architecture. Can have multiple master servers = core servers (vs. read replicas = non-core).
- Asynchronous (non-blocking) synchronization between master and read replica servers. Allows for reads from all nodes.

- Write: either directly to core or do a read replica node sending write also to core. Write is blocked until graph is on absolute majority of every core server  $(n/2 + 1)$ .
- Graphs are stored as graphs in memory (index-free adjacency) but are dissected into their components (nodes, edges, etc.) when stored on disk (fixed-sized records).

### Storing Stuff

- Label storage: labels of a node are stored as linked list to node.
- Property storage: properties of a node are stored as linked list to node where elements are key/value pairs.
- Relationship storage: linked list of outgoing edges

### 3.3.2 GDB Reading Assignment

See reading assignment "Graph Databases".

## 3.4 Cubes

**Data Warehouse** Subject-oriented (customers, sales, products, events), integrated (data comes from many applications and ETLed into the data warehouse), time-variant, nonvolatile (load and access, no updates) collection of data in support of management's decision-making process (analyze, report, mine).

**OLTP vs. OLAP** See Figure 22.

|              | OLTP                        | OLAP                            |
|--------------|-----------------------------|---------------------------------|
| Source       | Original (operational)      | Derived (consolidated)          |
| Purpose      | Business tasks              | Decision support                |
| Interface    | Snapshot                    | Multidimensional views          |
| Writing      | short and fast, by end user | period refreshes, by batch jobs |
| Queries      | Simple, small results       | Complex and aggregating         |
| Design       | Many normalized tables      | Few denormalized cubes          |
| Precision    | ACID                        | Sampling, confidence intervals  |
| Freshness    | Serializability             | Reproducibility                 |
| Speed        | Very fast                   | Often slow                      |
| Optimization | Inter-query                 | Intra-query                     |
| Space        | Small, archiving old data   | Large, less space efficient     |
| Backup       | Very important              | Re-ETL                          |

Figure 22: OLTP vs. OLAP.

**Cube Data Model** Data (= facts) is stored in multidimensional hypercubes. E.g. year - country - product = key with some value (price or something) - can be empty.

Transform into a table by mapping dimensions into their own fields with cube content as additional field in corresponding row.

**Aggregation** Remove one dimension of the cube.

**Slicing** Extract one stage of cube (for a specific year, give me all countries and products).

**Dicing** Extract sub-cube.

**Pivoting** Turn cube around.

**Drill Down** Zoom in into one dimension while still keeping cube shape.

**Drill Up / Roll Up** In contrast to drill down (e.g. month to year).

**Implementation** ROLAP - simply as table either with one value per multi-dimensional key or multiple measures (sub-values) - see star / snowflake schemas) - just use SQL here. Or MOLAP with MDX .....

### 3.5 Dremel

See reading assignment "Dremel: Interactive Analysis of Web-Scale Datasets". The following is a brief summary of that paper.

#### Introduction

- **Dremel is a scalable, interactive ad hoc query system for analysis of read-only nested data.**
- Combine multilevel execution trees and columnar data layout - aggregation queries over huge amounts of rows in seconds.
- Shared clusters with commodity hardware.
- Dremel can execute many queries usually requiring a sequence of MapReduce jobs but at a fraction of execution time.
- Not a replacement for MR, often used in conjunction.
- Serving tree and high level language to execute queries natively (no translation to MR jobs).
- Column-striped storage representation for nested data (see Figure 23).
- Used on top of GFS (distributed storage layer, data is replicated, etc.). (Dremel is like Hive in Hadoop).

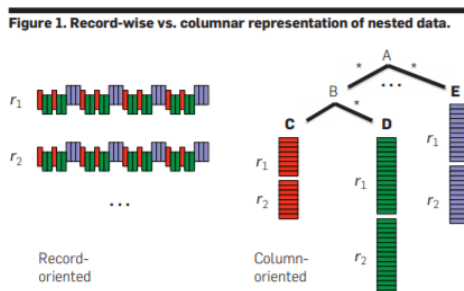


Figure 23: Access A.B.C without anything else.

**Data Model**

**Nested Columnar Store**

**Query Language**

**Query Execution**

**Observations**

## 4 Processing

### 4.1 Two Step Distributed Query Processing

MapReduce

types of files that can be processed with mapreduce

physical vs logical architecture

shuffle done with HTTP calls (moving data) - optimize shuffle with combining

block vs split vs shard

mapreduce examples-MR to sql relational algebra mapping

java API (job vs slot vs task etc - careful) good slot example in video 9.1

inputformats and outputformats

again block vs split

#### 4.1.1 MapReduce

See reading assignment "MapReduce: Simplified Data Processing on Large Clusters". The following is a brief summary of that paper (enhanced with extra notes from the lecture).

##### Introduction

- **Programming model and implementation to process and generate large data sets.**
- **Map:** User-specified map function that processes a key/value pair and generates a set of intermediate key/value pairs. Intermediate value with the same key are grouped together.
- **Reduce:** User-specified reduce function that merges all intermediate values associated with the same intermediate key.
- Programs are automatically parallelized.
- Run-time system partitions input data and schedules execution across set of machines.
- Example: count number of occurrences of each word in a large collection of documents. Map: create (word, 1) pairs and group them together into (word, 1, 1, ..., 1) objects. Reduce: go through each object and count the ones together to create (word, n) pairs.

##### Programming Model

- **Map Type:**  $(k1, v1) \longrightarrow \text{list}(k2, v2)$
- **Reduce Type:**  $(k2, \text{list}(v2)) \longrightarrow \text{list}(v2)$
- Input keys and values normally from different domain than output keys and values. Intermediate keys and values same domain as output keys and values.

## Implementation: Execution Overview

- Partition input data into  $M$  splits.
- Map invocations are distributed across multiple machines, processing input data in parallel.
- Partition intermediate key space into  $R$  pieces with a partitioning function (e.g.  $\text{hash}(\text{key}) \bmod R$ ).  $R$  and function is user-specified.
- Reduce invocations are distributed across multiple machines, processing input data in parallel.
- Ideally,  $M$  and  $R$  are much larger than number of worker machines. In practice:  $M$  s.t. input roughly 16 - 64MB (effective locality optimization) and  $R$  a small multiple of expected number of worker machines.
- Avoiding stragglers: when MR operation close to end, master schedules backup executions of remaining tasks.

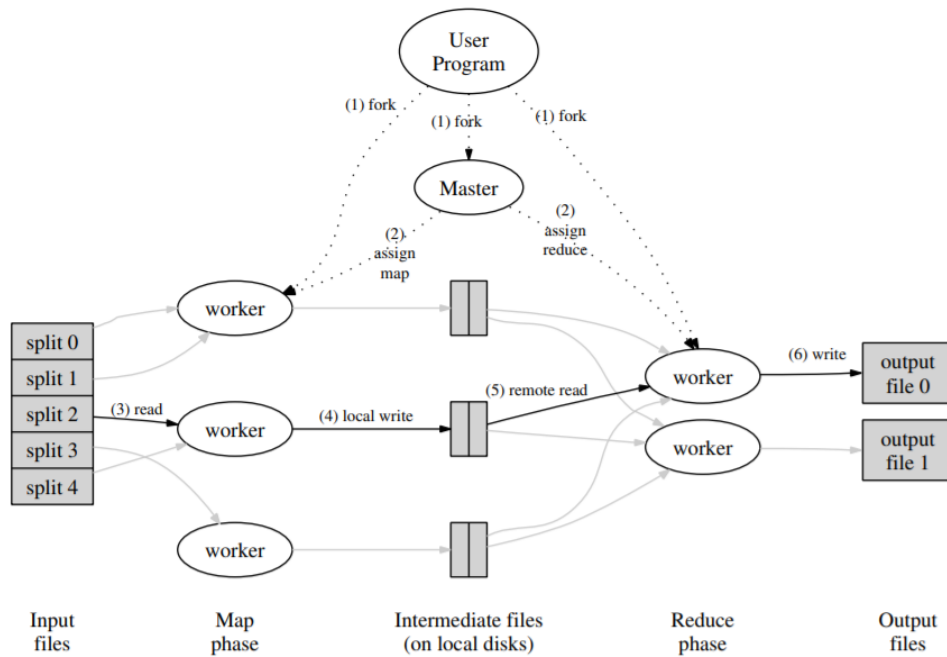


Figure 24: MR execution overview.

1. Split input files into  $M$  splits (16 - 64MB a piece). Start up copies of user program on cluster of machines (one copy = master, rest are worker programs).
2. Master has  $M$  map and  $R$  reduce tasks to assign - pick idle workers and assign each one a map and a reduce task.
3. Worker with map task: read contents of corresponding input split, parse key/value pairs, pass each pair to user-defined map function. Buffer intermediate key/value results in memory.
4. Buffered pairs are periodically written to local disk that is partitioned to  $R$  regions by partitioning function.
5. Master notifies reduce worker about locations of intermediate key/value pairs. Worker uses remote procedure calls to read buffered data from local disks of map workers. All read - sort by intermediate keys (grouping). Use external sort if this doesn't fit in memory.



6. Reduce worker iterates over sorted intermediate data, for each unique key, pass key and all its values to user-specified reduce function. Append output to final output file for this reduce partition.
7. All completed - master wakes up user program and MapReduce call returns. Output is available in the  $R$  output files (can be passed to another MR call s.t. user doesn't have to combine themselves).

**Implementation: Fault Tolerance** Master keeps track of workers and re-schedules map tasks on worker failures (even completed map tasks since they're stored on local disk - inaccessible). Completed reduce tasks don't need re-schedule, output is stored in global file system.

Master failure is rare - MR task is simply aborted if it fails, clients retry.

**Implementation: Locality** Locality to conserve network bandwidth. Underlying architecture is GFS - files divided into blocks and replicated on different machines. Master tries to schedule a map task on machine with or near a replica of that task.

**Refinements: Combiner Function** The intermediate data is shuffled around s.t. the reduce workers can process it (fetch from map workers local disks). To optimize amount of I/O, we can use a combiner function. A map worker can combine the data when flushing it to disk (or when compacting in HBase).

The combiner function is identical to the reduce function if: key/value types are always the same for input/output of combiner and reduce function and reduce function is commutative and associative.

## 4.2 Resource Management

### MR Master / JobTracker Responsibilities

- Resource management
- Scheduling jobs
- Monitoring workers
- Job life cycle monitoring
- Fault tolerance

### Issues

- Not scalable
- Bottleneck
- Jack of all trades
- TaskTrackers / Workers are divided into slots, each possibly taking on many map tasks (or reduce tasks) exclusively. Slots with reduce tasks are idle until slots with map tasks finish.

Issues can be fixed with using YARN. The YARN resource manager does the scheduling and application management while the many application masters do the monitoring.

### 4.2.1 YARN: Yet Another Resource Negotiator

See reading assignment "Apache Hadoop YARN: Yet Another Resource Negotiator".

**Goal** Separate resource management functions from the programming model (e.g. MapReduce). YARN supports fair scheduling, FIFO, capacity scheduling but not preemptive priority scheduling.

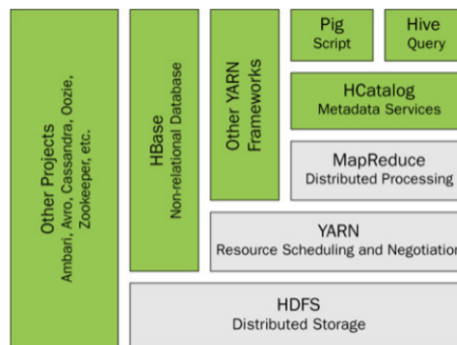


Figure 25: Hadoop components overview.

### Architecture

- **ResourceManager:** one per cluster, tracks resource usage and node liveness, enforces allocation invariants (fairness, capacity, locality across tenants, SLAs, not job type tho) and arbitrates contention among tenants. RM can take resources away from a running job (preemption).
- **ApplicationManager:** requests resources from RM, generates physical plan from received resources and coordinates execution around faults. Heartbeats exchanged with RM. Many per cluster, usually not per every node - can have multiple per node if for different applications and they are not aware of each other. AM are not privileged services in the YARN architecture.
- **Container:** RM schedules and dynamically allocates containers to applications. Containers are many per node and represent a logical bundle of resources (e.g. 10GB RAM and 4 cores).
- **NodeManager:** Running on each node (one per node) offering a container, heartbeats are exchanged with RM (resource availability, health/faults, container life cycle, liveness, etc.). Authenticates presented tokens (container leases), manages containers dependencies, provides set of services to container, does garbage collection, etc.
- **Job Submission:** A job is a program (arbitrary user code / language). AM submits job and RM goes through admission control procedure. If accepted, job is scheduled and once enough resources available, job can run on spawned containers. For each container, AM gets a token that it has to present to each NM s.t. the job can start. AM can manage multiple processes (= jobs). AM itself runs in a container.
- **Resource Request:** AM requests to RM consist of: nr. of containers, resources per container, locality preferences and priority requests within application.

## 4.2.2 Scheduling Strategies

**FIFO Scheduler** Processes are queued and scheduled in the order that they arrive (time-based). In a shared cluster, small applications have a disadvantage.

If application 1 requests containers before application 2, subsequent container requests of a1 get served first even if a2 requested some before a1's second request.

**Capacity Scheduler** Different applications are guaranteed a percentage of the cluster capacity. E.g. IT department gets 60% and Math department gets 40% of overall capacity. Both departments then have their own scheduling algorithms, i.e. we have two separate FIFO queues. Division can be hierarchical and requests have to be renormalized to 100% - see example in Figure 26.

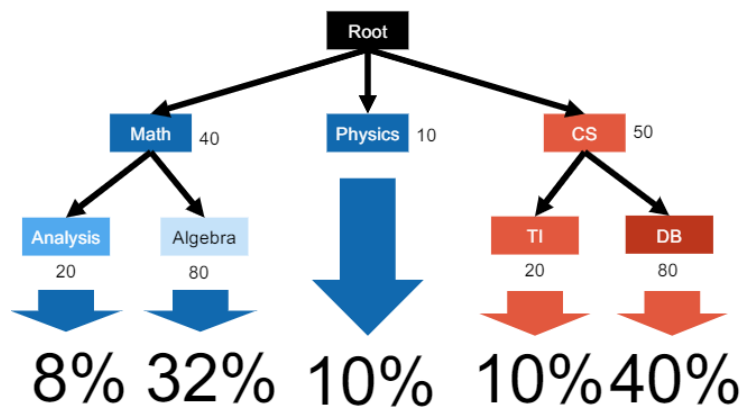


Figure 26: Hierarchical queues example.

**Cluxel** A cluster divided into smaller chunks. For example, a cluster with 1'000GB capacity in total can be divided into 100 10GB cluxels.

**Queue Elasticity** If one queue is currently unused/empty, another queue can take some of its resources. We usually define capacity shares and maximum capacity allowed per class (e.g. Math can normally use 40% of the entire cluster's capacity but if other departments are idle, it can use up to 80% of the entire cluster's capacity). Between queues, the ones with lowest usage ratio have priority (one class = one queue).

Careful: each application of one class can still only take original share (see Figure 27).

**Fair Scheduling** See next paragraphs. Different from capacity or simple FIFO queues.

**Steady Fair Share (Soll)** Pre-configured amount of resources reserved for each queue.

E.g.: 1'000 GB of memory, IT wants 40%, Physics wants 10% and CS wants 50%. Memory is divided into: 400GB, 100GB and 500GB.

**Current Share (Ist)** What is actually happening in the cluster = current usage of the resources. This might deviate from the steady fair share.

E.g. IT is not using the cluster (0%), Math is using 30% and Physics 20%.

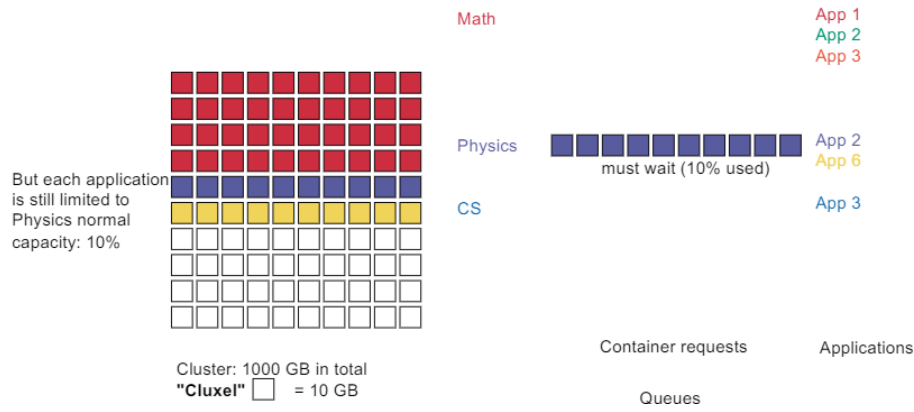


Figure 27: Elastic queues example.

**Example: Soll vs. Ist** Keep track of queue / class, steady fair share, current share and delta (ist - soll). We start with: Math wants 40, Physics wants 10 and CS wants 50.

1. Math, Physics and CS all want to use several cluxels.
2. Look at class with lowest delta (CS with -50) and assign 10 cluxels s.t. delta is now the same as second lowest (math -40). Update CS current to 10.
3. Assign 40 to CS and Math each, s.t. they both match the -10 delta from Physics. Currents are now: Math 30, Physics 0 and CS 40. Cluxels in use: 70 out of 100.
4. Physics is the only one asking, give remaining 30 to it. Deltas are now: -10 for Math, 20 for Physics and -10 for CS.

**Instantaneous Fair Share** If one class is idle, re-normalize other shares to 100%. E.g. 10% and 50% to 17% and 83%.

Replace Soll column with instantaneous fair share numbers and update deltas accordingly. Now we can see who gets more if one class stops after all was full.

If a class is still under-using while another is over-using, it can preempt the other class (harsh, usually just wait nicely) s.t. delta is 0 for all.

**Multiple Resources** Computing the steady fair share is easy but how to compute current share? See DRF below.

Resources can be: cores, memory, disk space, network bandwidth, etc.

**Dominant Resource Fairness (DRF) Example** One class uses 300GB out of 1TB memory (30%) and 4 out of 100 cores (4%), the other uses 10GB out of 1TB memory (1%) and 50 out of 100 cores (50%). For each class, take the dominant resource (mem for first, cores for second) and calculate current share by re-normalizing those, e.g. 37.5% vs. 62.5% usage of cluster. With this, we are back to the original algorithm.

Find smallest puzzle to meet steady demand and then repeat.

With this, there will be some leftover idle resources - allocate more cluxels than we actually have (since they're 2D).

### 4.2.3 Dominant Resource Fairness

See reading assignment "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types".

**Goal** In a multi-resource environment, the allocation of a user should be determined by the user's dominant share = maximum share that the user has been allocated of any resource. DRF seeks to maximize the minimum dominant share across all users.

#### Example

- Resources: 9 cores, 18GB RAM. Split them 50/50 across two users.
- Demands of jobs run by user A: 1 core, 4GB RAM each.
- Dominant resource utilization of A:  $4/18 = 2/9$ .
- Demands of jobs run by user B: 3 cores, 1GB RAM each.
- Dominant resource utilization of B:  $3/9 = 1/3$ .
- To allocate 50/50, we have A:  $3 * 2/9 = 2/3$  and B:  $2 * 1/3 = 2/3$ .
- Actual allocation:  $3 * 1$  (A) +  $2 * 3$  (B) = 9 cores and  $3 * 4$  (A) +  $2 * 1$  (B) = 14GB RAM.

**Properties** Properties that a resource manager should fulfill. DRF achieves all but last one in practice.

- **Sharing Incentive:** Each user should be better off sharing the cluster than exclusively using her own partition of the cluster.
- **Strategy-Proofness:** Users should not be able to benefit from lying about their resource demands.
- **Envy-Freeness:** A user should not prefer the allocation of another user.
- **Pareto Efficiency:** It should be impossible to increase the allocation of a user without decreasing it for another (= maximize utilization).
- **Single Resource Fairness:** For a single resource, solution is simple max-min fairness.
- **Bottleneck Fairness:** If every user has the same dominant resource demands, solution for that resource is simple max-min fairness.
- **Population Monotonicity:** User freeing resources should not decrease any allocations in users still running.
- **Resource Monotonicity:** Adding more resources should not decrease any allocations.

### 4.3 DAG-Based Distributed Query Processing (Spark)

**Spark** MapReduce is under-using YARN since YARN supports any DAG, i.e. dividing into one map and one reduce stage with a shuffle in the middle is limiting. With Spark, the data processing can follow any DAG. The dataflow does not have to be linear as in MR - we can now efficiently do iterative processing or interactive analysis of data.

Spark is best known for its ability to keep large working datasets **in memory between jobs** (instead of writing to and loading from disk in between MR stages).

Spark can run over any distributed storage system (HDFS, S3, etc.) and it needs some form of cluster management (e.g. YARN).

**Problem of MapReduce** It lacks the abstraction that allows it to use distributed memory. The only way to reuse data in between computations is to write and read it from disk - heavy I/O usage = bottleneck.

**Why not simply shared RAM?** Distributed shared memory allows reads and writes to each memory location. RDDs can only be created through coarse-grained transformations - this restricts RDDs to bulk write applications but allows for better fault tolerance. Upon failure, only lost partitions need to be recomputed (in parallel) - no need to roll back whole program.

#### 4.3.1 Resilient Distributed Dataset

See reading assignment "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing".

**Introduction** Distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. For Spark, RDDs are one possible API (next to DataFrames/DataSets, etc.), i.e. ways to interact with Spark. DataFrames /DataSets are an abstraction of the underlying RDD abstraction.

An RDD is a collection of read-only<sup>3</sup> structured or unstructured data items/records partitioned across the nodes of the cluster that can be operated in parallel. RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory.

RDDs are the nodes of the processing DAG. This is in contrast to the key/value pair files in MapReduce.

RDDs are created by starting a file in the Hadoop file system and transforming it deterministically. Upon executing an action, the transformations are actually materialized (lazy evaluation) and presented to the user (not as an RDD). Lazy evaluation allows Spark to change the DAG for better efficiency. See Figure 28.

One can provide a partition function to efficiently partition the RDD data across nodes (exploit locality when executing Spark).

---

<sup>3</sup>RDDs are resilient and immutable. Applying a transformation to an RDD simply creates a new one - this is how we get a DAG structure. We can always read past RDDs.

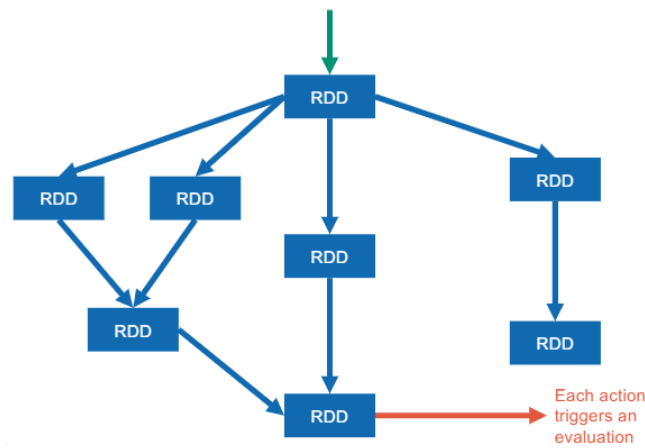


Figure 28: Example Spark DAG (one creation, multiple transformations and one action).

**RDD vs. DataFrame/DataSet Abstraction** Since RDD is such a low-level API, a programmer actually has to think about **how** to do things, not just **what** to do (as in SQL). RDDs are best to use when we want control and we're dealing with unstructured data. But, using RDDs can make code more inefficient since Spark has less room for optimizations (Spark doesn't know what data actually is since it is unstructured and what we actually want to do with it - opaque).

## Example Spark Execution

- **Creation:** `val rdd1 = sc.parallelize(List("Hello, World!", "Hello, there!"))`
- **Transform:** `val rdd2 = rdd1.flatMap(value => value.split(" "))`
- **Action:** `rdd2.countByValue()`

## Transformations on one RDD

- **filter:** Accept one type of value and reject another.
- **map:** Map each value to a new value.
- **flatMap:** One value can be mapped into multiple values.
- **distinct:** Filter out duplicate values.
- **sample:** Given a fraction and a random seed, sample a random amount of values. Dice at every value.
- etc.

## Transformations on two RDDs

- **union:** Combine values of two RDDs into one RDD.
- **intersection:** Combine all values that are the same in both RDDs into one.
- **subtract:** Only put values in first RDD that are not in the other in a single RDD.
- **cartesian product:** Clear.

## Transformations Key/Value Pairs

- **keys:** Put all keys of input RDD into output RDD.
- **values:** Put all values of input RDD into output RDD.
- **reduceByKey:** Combine values with same key with some reduce function - makes key/value pairs with single values.
- **groupByKey:** Group values with same key together, values stay separated.
- **sortByKey:**  $\leq$  sort.
- **mapValues:** Provide function that takes values and maps into other values, same key.
- **join:** For each key/value pair on the left, join with fitting key on the right into key-value-value for example.
- **subtractByKey:** Only keep key/value pairs of left that don't have a key in the right.

**Actions** Output of an action is **not** an RDD!

- **collect:** Output all values of result RDD.
- **count:** Output number of final values.
- **countByValue:** Each distinct value is counted, output key/value pairs.
- **take:** Output n first values.
- **top:** Output n last values.
- **takeSample:** Randomly output n values.
- **reduce:** Reduce all values into one (e.g. add up numbers).
- **countByKey:** For each key, count same key/value pairs.
- **lookup:** Given key, output its value.

**Persisting RDDs** If the transformation sets for some or all actions at the bottom of the DAG overlap, don't recalculate intermediate stages for better performance. Use persisting RDDs instead = pre-calculation of subset of DAG (higher-levels). Typically one RDD.

### 4.3.2 Physical Layer

#### RDD Partition Dependencies

- **Narrow:** Each partition of the parent RDD is used by at most one partition of the child RDD. Better parallelization and locality, easier recovery.
- **Wide:** Multiple child partitions might depend on a partition of the parent RDD. Shuffling required.



Parallel execution: divide into tasks. Default: one task per HDFS block. Spread tasks over executors and their cores.

Sequence of parallelizable transformations: we would like it to be streamlined. Easy for narrow dependency. Divide wide dependency into stages. Job = sequence of stages.

Stage 1: all narrow, transformations stay on same machine, can run in parallel.

Shuffle: only start shuffling around until stage 1 ended.

Stage 2: again narrow, etc.

Avoid wide dependency: pre-partition, e.g. key values with same key already on same machine.

### 4.3.3 DataFrames

**Introduction** In short: it is a distributed collection of rows with the same schema.

DataFrames (and DataSets, SQL) are a more structured and higher-level Spark API (on top of RDD abstraction<sup>4</sup>). DFs offer compile time syntax error detection and runtime analysis error detection (reported before a distributed job starts).

A DataFrame (i.e. table, fixed-length column store) is made up of several data items = `DataSet<Row>` = attribute names, types and attribute values of one table row. Names and types are statically known (= schema).

DFs can be manipulated with relational operations or with simple RDD functions by treating a DF as a RDD of row objects. No matter what, all operations are again lazy.

Can use SparkSQL with DataFrames (vs. PySpark with RDDs). Types: Data formats: Catalyst optimizations EXPLODE SQL function pyspark - with JSON vs. dataframes = table (higher level than RDD - easier to optimize) (everything in SQL can be implemented on top of spark / mapreduce) - fixed length col store - dataframe data types and type mapping - catalyst: optimize query plan (todo?) rdbms vs. mapreduce / spark 10.4 dealing with structured types - explode function (SQL extensions) SQL dots for objects - this needs structure - heterogeneity a limit of DFs can switch between RDD and DF with sufficient structure (see code) <https://www.youtube.com/watch?v=0fk7G3GD9jk> example also here

### 4.3.4 SparkSQL

See reading assignment "Spark SQL: Relational Data Processing in Spark".

**Introduction** Integrate relational processing with Spark's functional programming API - despite using semi-structured data. Mix declarative vs. procedural programming models seamlessly to access and process data. SparkSQL uses the Spark DataFrame API and Catalyst (optimizer). Write it with either Java, Scala or Python.

---

<sup>4</sup>DF can also be created externally.

## Data Model

- Base types: number types, string, boolean, binary, timestamp, date
- Complex types: structs, arrays, maps
- Custom complex types

**Operations** Domain specific language, includes all common relational operations. Operators take expressions as arguments. Spark then captures structure of computation in an ASL and passed to optimizer Catalyst (not possible with RDDs!). To just use SQL, DFs can be registered as temporary tables.

**Storage: RDD vs. DF** DF is stored in compact columnar storage and RDD in object-based storage. Former is more compact and therefore in-memory caching is more efficient.

## Catalyst Optimizer

- Core: library to represent trees and rules to manipulate these trees.
- Main data type: tree object, node object (of various types).
- Rules: tree-to-tree mapping, apply them using pattern matching on sub-trees, can be applied until a fixed-point is reached (no more changes possible).
- Used in SparkSQL to: analyze AST to resolve references and types, optimize AST, physical planning using Spark physical operators (cost-based heuristic stuff), code generation to compile parts of the query to Java bytecode.

## Additional Features for Big Data Workloads

- Schema inference of semi-structured data, e.g. JSON document
- Machine Learning stuff
- Query federation to external DBs (instead of loading data into local processing env., execute query on external data source - reduce traffic, improve performance)

## 4.4 Performance at Large Scales

**Bottleneck Sources** In most systems, one of the below sources is the main source of bottleneck.

- Memory
- CPU
- Disk I/O (MR and Spark are used if disk is a bottleneck)
- Network I/O

## Measurements

- Latency (arrival of first piece of data), typical latencies are between ns and ms.
- Throughput (how fast can we transmit data), bits resp. bytes per second.
- Total response time = latency + transfer time.

You must know this by ❤️!

|                  |   |
|------------------|---|
| <b>milli (m)</b> | 0.001 (3 places)                              |
| <b>micro (μ)</b> | 0.000 001 (6 places)                          |
| <b>nano (n)</b>  | 0.000 000 001 (9 places)                      |
| <b>pico (p)</b>  | 0.000 000 000 001 (12 places)                 |
| <b>femto (f)</b> | 0.000 000 000 000 001 (15 places)             |
| <b>atto (a)</b>  | 0.000 000 000 000 000 001 (18 places)         |
| <b>zepto (z)</b> | 0.000 000 000 000 000 000 001 (21 places)     |
| <b>yocto (y)</b> | 0.000 000 000 000 000 000 000 001 (24 places) |

Figure 29: Prefixes.

## Speedup

- Normal: speedup = old latency / new latency.
- Amdahl:  $1 / (1-p + p/s)$  where  $p$  = percent paralellizable (e.g. 0.3 for 30%) and  $s$  = speedup on that part.
- Gustafson:  $1 - p + s*p$
- Amdahl assumes constant problem size (faster, same size) and Gustafson assumes constant computing power (same time, increasing size).

**Scaling out vs. up** Scaling up should be last resort.

### 4.4.1 Scalability! But at what COST?

See reading assignment "Scalability! But at what COST?".

**COST Metric** Configuration that Outperforms a Single Thread.

**Issue** New distributed algorithms are praised for their scalability but we forget their real-world runtime performance which is sometimes worse than running a single-threaded version of the algorithm. Inefficiencies are introduced just to be scalable.

## 5 Management

### 5.1 Document Store

**NoSQL** Nestedness and heterogeneity. Joins are hard. Document-oriented DBs are one of the main categories of NoSQL

**Document Store** Storing many small documents (e.g. JSON/XML documents = "record") a.k.a a collection of trees (tree per document) a.k.a semi-structured data. Validate after data was populated. MongoDB is one implementation of a document store.

#### Encodings (Char to 0/1)

- ASCII
- ISO Latin 1
- UTF-8
- UTF-16

### 5.2 MongoDB

See reading assignment "MongoDB: The Definitive Guide".

**Introduction** MongoDB is a document-oriented DB program - classified as NoSQL. It uses JSON-like documents with optional schemas. Main features include:

- Ad-hoc queries such as point queries, range queries and regex searches. Queries can return specific fields of documents.
- Fields in a document can be indexed (primary on `_id` and secondary on other fields).
- Data is replicated to provide high availability. Writes and reads are done on primary replica, secondaries maintain a copy (eventual consistency). If primary fails, a new primary is selected.
- Horizontal scaling by using sharding. Shard key chosen by user determines how data in collection will be distributed. Data is split into ranges based on key and distributed across multiple shards (=master with one or more replicas). Even distribution with hash partitioning possible.
- MongoDB provides the capability to validate documents during updates and insertions.
- There are no joins in MongoDB!

**MongoDB Stack** See Figure 30.

**Querying: CRUD** Create, read, update, delete. Writing atomicity is at document-level. No need for schema until projection.

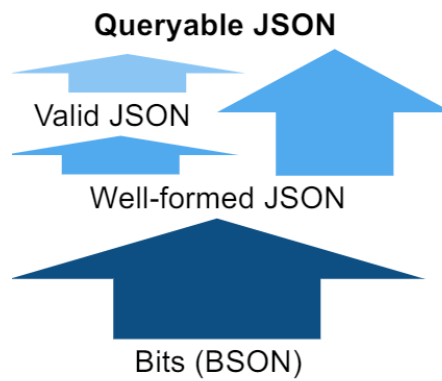


Figure 30: MongoDB Stack.

**Architecture** mongodb can compare anything with anything

architecture (no hdfs below mongodbl!) - primary nodes, write 3 replicas ACK and then asynch - 1 doc = 1 tree - 1 shard = multiple docs

indices - there have been none until now, we always scan everything (need to be built by the programmer) - doc stores have indices for programmer - hash, good for point - b+ trees, how to create it, good for point and range

mongodb data fits on single machine (usually)

## 5.3 Query Languages

querying trees

jsoniq (data model, operations, data flow, flwor, types) json navigation (jsoniq, rumble)

query processing, iterative/streamed vs. materialization

### 5.3.1 Rumble

See reading assignment "Rumble: Data Independence for Large Messy Data Sets".

**Introduction** Rumble is a query execution engine for large, heterogeneous and nested collections of JSON objects built on top of Apache Spark (usually running over HDFS). It uses JSONiq as the interface language, which is a standardized language for querying JSON documents.

Rumble shows that data independence for JSON processing is achievable with reasonable performance on top of large clusters.

Other formats are possible (txt, parquet, csv, etc.) and other file systems are possible (local, S3, Azure, etc.).

**Issues Rumble Solves** Values which cannot fit a regular structure are dropped (in table-based tools), nested objects are kept as complete strings or mapped to non-existing values. Simply converting JSON documents to DataFrames is therefore uncool.

## JSONiq

- Fit recursive JSON structure on Spark's DataFrames: JSONiq expression is translated into a tree of iterators.
- Declarative language.
- Manipulating ordered and potentially heterogeneous sequences of items (atomic values, structured items such as objects and arrays, function items).
- Sequence: always flat and unnested, separated by comma, can be empty. A sequence of one item is the same as the item itself.
- Process each item of a sequence individually with FLWOR expressions (for, let, where, order by, group by, return) - similar to Haskell programming.

**Query Result** Results can be displayed, stored to file or written to a distributed file by Spark.

## Basic Operations

**Precedence** See Figure 31. Use parentheses to override precedence ordering.

| Precedence (low first)                                      |
|---|
| Comma   |
| Data Flow (FLWOR, if-then-else, switch...)                  |
| Logic   |
| Comparison  |
| String concatenation  |
| Range   |
| Arithmetic  |
| Path expressions  |
| Filter predicates, dynamic function calls                   |
| Literals, constructors and variables                        |
| Function calls, named function references, inline functions |

Figure 31: Higher precedence first.

## 6 Exercises and Quizzes

### 6.1 SQL Brush Up

writing SQL queries plotting results

### 6.2 Object and Key-Value Stores

#### 6.2.1 Exploring Azure

##### WAS Questions

- There are multiple types of blob that can be created in WAS (block, append and page blobs).
- All resources in Azure Storage are accessed through a REST API. The existing libraries wrap this REST API in specific programming language calls.
- A storage account can be configured by default for (un)frequent access. We can choose the default configuration for all blobs inside a storage account at creation time of the storage account.
- Block blobs can be at most 4.75TB and Page Blobs up to 8TB.
- The partition layer provides, among other services, transaction ordering and strong consistency for access to objects.
- Intra-stamp replication provides durability against hardware failures, whereas inter-stamp replication provides geo-redundancy against geo-disasters.
- The checksum validation is done at block level, not at extent-level.
- The intra-stamp replication is synchronous to guarantee durability and no data loss on client writing.

**REST** REST stands for **r**epresentational **s**tate **t**ransfer.

##### HTTP Response Code Questions

- Authenticated user tries to access resource that he has no permission for. The returned HTTP response code is: 403.
- 400: Request could not be understood by the server due to malformed syntax.
- 401: User is not authenticated or needs to authenticate again
- 404: Server has not found anything matching the request URI.
- 302: Requested resources temporary resides under different URI
- 202: Request accepted for processing, processing has not yet completed.
- 204: Server has fulfilled request, does not need to return an entity body and might want to return updated meta-info.

## HTTP Response Codes Summary

- **1xx:** Informational
- **2xx:** Success
  - **200:** OK, resource has been fetched / put / updated, can contain message
  - **201:** Created, new resource created, usually response after POST
  - **202:** Accepted, request received, not yet acted upon
  - **204:** No Content, request no content in answer but headers might be useful
- **3xx:** Redirection
  - **300:** Multiple Choice, request has more than one possible answer, please choose
  - **301:** Moved Permanently, URL changed
  - **302:** Found, URL changed temporarily
  - **304:** Not Modified, caching purposes, response has not been modified
- **4xx:** Client Error
  - **400:** Bad Request, invalid syntax
  - **401:** Unauthorized = unauthenticated
  - **403:** Forbidden, no access rights, client identity known
  - **404:** Not Found
  - **409:** Conflict, current state of server
- **5xx:** Server Error
  - **500:** Internal Server Error
  - **503:** Service Unavailable

### 6.2.2 Vector Clocks

**Recap** VC is an algorithm used to generate a partial ordering of events and to detect causality violations in a distributed system. For all  $n$  processes / nodes, there is an entry in an array (= vector clock) with length  $n$  (initialized to 0). Each entry is the logical clock of the corresponding process / node. Each object has its own DAG of vector clocks and value  $i$  gets incremented by 1 if node  $i$  writes that object (see example below).

The purpose of VS is **not** to keep different versions of objects or to produce atomic updates on distributed objects.

**Partial Ordering** If  $VC(x)$  is the vector clock of event  $x$  (= a write to an object) and  $VC(x)_z$  is the component of that clock for node  $z$ , we can define the partial ordering property as:

$$VC(x) < VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z] \wedge \exists z'[VC(x)_{z'} < VC(y)_{z'}]$$



**Antisymmetry** If an event happened before another as defined above then it is not possible that the second event has also happened before the first, i.e.:

$$VC(x) < VC(y) \implies \neg(VC(y) < VC(x))$$

**Antisymmetry Proof** Proof the antisymmetry property using proof by contradiction:

1. Assume  $VC(x) < VC(y)$  is true.
2. Using the partial ordering property defined above, there must be  $z'$  where  $VC(x)_{z'} < VC(y)_{z'}$  holds.
3. Assume  $VC(y) < VC(x)$  is also true.
4. Using the partial ordering property, it is implied that  $\forall z[VC(y)_z \leq VC(x)_z] \wedge \exists k[VC(y)_k < VC(x)_k]$ .
5. This stands in contradiction with 2). QED.

**Transitivity** If  $VC(a) < VC(b)$  and  $VC(b) < VC(c)$  then also  $VC(a) < VC(c)$ .

**Filling Version Evolution DAGs** Shorthand notation: with two nodes in the system, node 0 writing the value  $aa$  is denoted as  $S_0 : aa([1,0])$ . Each DAG is for one value. For missing written values, anything is valid. When merging two (or more) clocks (current node can reach both/all previous nodes), for each element choose largest parent value. See examples in Figure 32.

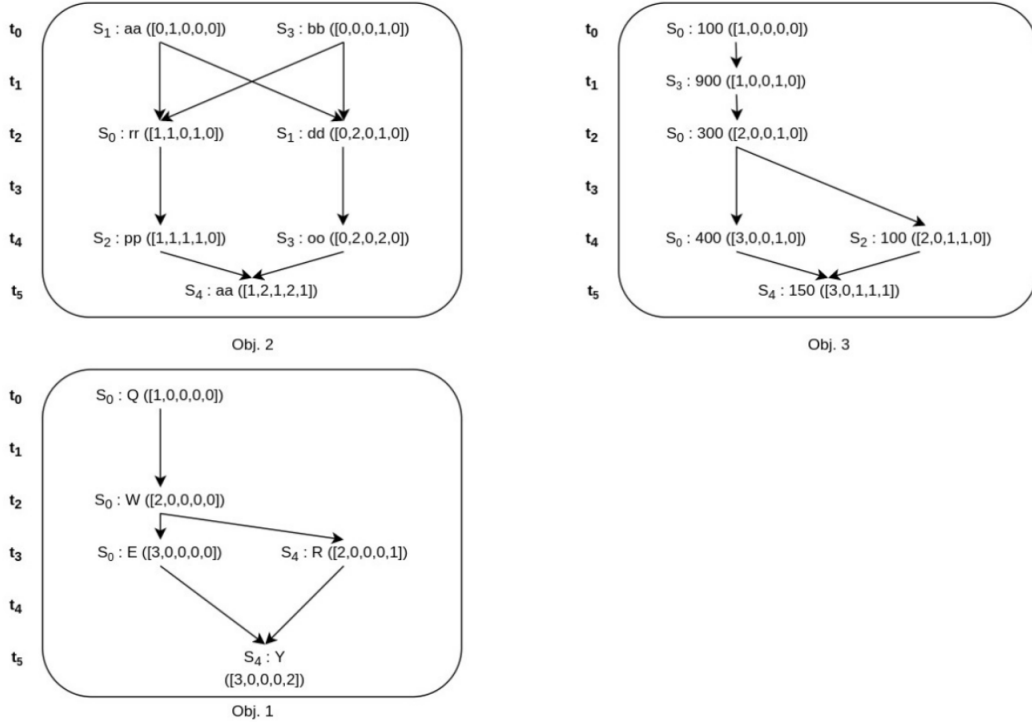


Figure 32: Version Evolution DAGs examples.

**Resolving Get Requests** When a Dynamo coordinator node receives a get request for a specific key, it collects the latest vector clocks for that value from itself and the top  $N - 1$  healthy nodes in its preference list for that key. Given a list of value and vector clock pairs, draw the version DAG reconstructed by the coordinator node.

If a get request arrives and we have a DAG with multiple leaves, all vector clocks are returned and the client is responsible for reconciliation (merging the different VCs).

### 6.2.3 Merkle Trees

**Recap** Leaves are data blocks and every non-leaf node is labelled with the hash value of its children. Some KeyValue stores use MT to efficiently detect inconsistencies in data between replicas. To compare MT, we first compare the root node - if the values match, the replicas are consistent. Else, compare the next level and follow the tree down until the inconsistent leaf/leaves are identified.

**Comparing Merkle Trees** Pay special attention to:

- If the root hash differs between two MTs but the hash values of the root children are the same in both MTs, it would imply that our hash function produced two different values for the same input values - this is impossible.
- It is theoretically possible that two different sets of input values produce the same hash value (parent) because hash functions can have a non-zero probability for hash collisions. This is not an issue with hash functions that are strong enough but it can happen in practice.

### 6.2.4 Virtual Nodes

Given a set of nodes, each node with different main memory capacities, and given a number of tokens, find the amount of virtual tokens to give to each node for a fair partition.

#### Tips

- $1\text{TB} = 1024\text{ GB}$
- Capacity per token: total capacity / total number of tokens
- It is good practice to give the node with the smallest capacity more than one token (it might be unlucky and get a lot of responsibility).

## 6.3 Storage Models

### 6.3.1 Hadoop and HDFS

**Hadoop Recap** Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm. Some components of Hadoop are: HDFS (distributed file system), MapReduce (distributed computation framework), HBase (column-oriented table service), etc.

## HDFS Statements

- The HDFS namespace is a hierarchy of files and directories. In contrast with the Object Storage logic model, HDFS is designed to handle a relatively small amount of huge files. A hierarchical file system can therefore be handled efficiently by a single NameNode.
- The default size for blocks of a file is either 64 or 128 megabytes - this can be easily changed in the configuration.
- No data goes through the NameNode. The client writes data directly to the DataNodes.
- A DataNode may execute multiple application tasks for different clients concurrently.
- The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster.
- NameNodes keep the namespace in RAM and an image of such namespace is also persisted in the NameNode file system.
- The locations of block replicas are not part of the persistent checkpoint that the NameNode stores in its native file system since they can change over time (re-distribution upon DataNode failure / addition).
- If the HDFS block size is 64MB, a 80MB file will be stored on disk as a block of 64MB and 16MB. We don't require extra space since the HDFS blocks are not rounded up to their nominal block size when they're stored (as in disk blocks of traditional file systems). The underlying system stored the two HDFS blocks with their own disk block system (rounded up). We don't require 128MB of physical storage for the 80MB file.
- Hardware cost grows linearly as a function of amount of data stored.
- NameNode is single point of failure (DataNode failure is handled easily). Run two redundant NameNodes for security (not the same as secondary NameNode which makes startups faster by efficient amount of keeping log and image info). Secondary NameNode does not store data blocks.

**HDFS Block Size** The typical HDFS block size is either 64MB or 128MB (compared to 4KB / 4096B in a typical file system - a lot smaller!). Advantages of a large block size are:

- Minimize seek cost (much smaller than transfer time). Transfer time is usually at disk transfer rate.
- Less client-master interactions - reads/writes on same chunk only require one block location information request from to master. Usual pattern: sequential read/write of large files.
- Reduce network overhead since read/write on large chunk (as in typical access pattern) allows for longer persistent TCP connection.
- Less metadata stored in master and can be therefore stored in memory.

## HDFS Properties

- **Scalability:** Partition files into blocks and distribute them to many servers operating in parallel. Arbitrarily increase storage capacity by simply adding more DataNodes.
- **Durability:** HDFS creates multiple copies of each block (by default 3, on different racks) to minimize the probability of data loss.
- **High sequential read/write performance:** By splitting huge files into blocks and spreading these into multiple machines.

**Replication Policy** For each block individually, default is three replicas. One randomly in a node and the other two in a different rack on two separate nodes. Probability of rack failure is much lower of node failure, thus no problem of having 2/3 replicas in same rack.

**Write New File** 2-5 are repeated for each block.

1. C asks NN to create new file.
2. C asks NN or DN to host block.
3. NN replies with a list of DN and locations for block.
4. C writes to first DN, DN replicates to next and so on.
5. DNs send ACKs to previous DNs. If all replied, first DN sends ACK to C.
6. C asks NN to close file and release lock.
7. DNs check with NN for minimal replication.
8. NN sends ACK to C.

**Read File**

1. C requests file from NN.
2. NN replies with list of blocks and locations of each replica.
3. C reads each block from closest DN.

**Distance Rules** Network bandwidth is estimated by distance - the shorter the more bandwidth is available. Distance between node and parent = 1, distance between nodes = sum up their distances to closest common ancestor.

To calculate distances, draw a tree where: root = cluster, root kids = datacenters, datacenter kids = racks, rack kids = nodes. From node to node: sum up edges of shortest path. Distance in same node = 0.

**HTTP Return Codes**

- Successful append operation to a file: 307 and 200
- Successful HTTP GET operation for reading and opening a file: 307 and 200
- Successful HTTP GET request for listing files: 200
- Successful HTTP PUT operation for setting the replication factor: 200
- Failed HTTP AUTHENTICATE request: 401
- Malformed HTTP GET request for listing files: 400

**Latency vs. Throughput Block Size**

- High latency and high throughput: choose largest possible block size to maximize throughput.
- Low latency and normal throughput: choose small(est) block size

## 6.3.2 Different Storage Models

### Object Storage vs. Block Storage

- Block Storage implements file storage API, whereas Object Storage provides only key-value interface.
- Pure Object Storage has a limit on object size, since the object cannot be partitioned across machines. Block Storage does not have this limitation and can split objects into blocks. Therefore, Block Storage can store PB files, whereas Object Storage is limited by the storage capacity of a single node. On the other hand, object storage can store more files than Block Storage.
- BS for huge amount of large files, OS for large amount of huge files.
- BS allows for block level access, OS can only retrieve entire files.
- OS is better for: Netflix movies with many concurrent accesses (movies small enough, simple KV is enough), auto backups of smartphones (written once, rare reads where partial access is not essential)
- BS is better for: experimental and simulation data from CERN (large files, store lots of data).

## 6.4 XML and JSON

### 6.4.1 XML

#### Well-Formedness Additional Tips

- Element names must start with a letter or underscore.
- Some escape entities need to be defined like example below for copyright.
- Not allowed to have two attributes with same name. Attribute values have to be quoted.
- XML is case-sensitive.
- Element names can contain letters, digits, hyphens, underscores, and periods.
- Element names cannot contain spaces.
- Same tag name is okay

```
1 <!DOCTYPE catalog [  
2 <!ENTITY cright "&#169;">  
3 ]>
```

#### Predefined Entities

- &lt; <
- &gt; >
- &amp; &
- &quot; " (okay to not escape)
- &apos; ' (okay to not escape)

## 6.4.2 JSON

### Well-Formedness Additional Tips

- Double quotes
- Commas in objects
- "type": ["home"] is okay
- "@number": "646 555-4567" is okay
- "1phone": 212-3242 needs quotes since there is a dash in number
- null with small n
- No duplicate keys
- Keys are always strings
- Using whitespaces and non-ascii characters for key names is allowed although not recommended.
- Mixing proper boolean values and strings used as boolean values (ie. "true") is considered a bad practice.

## 6.5 Wide Column Stores

### 6.5.1 HBase Architecture

**Returning Key Value Pairs** Given a query for a specific key, the most recent versions of each component are returned (memstore and HFiles). If they have the same timestamp, return both. If one is newer than the other and in the same component and same col., return newest. If they are in different columns, return both (no matter the timestamp)! Multiple values can be returned!

**Bloom Filter** One BF per HFile, used to avoid checking HFiles for a specific key. If no match in BF, don't read file. Key input in various hash functions and then map to array (1 for output value, else 0), for new key, check if sets of 1 is present.

**Index** To avoid scanning HFiles entirely upon get(key) - index helps to skip to HBase block that may hold the key. HBase blocks are not the same as HDFS or FS blocks. They come in 4 varieties: DATA, META, INDEX, and BLOOM. Default size: 64KB and contains whole key/value pairs (block grows to not split a key/value pair).

Build an index exercise: start with first key/value pair in sorted HFile (first pointer). Fill first HBase block by counting bytes of entries (with spill). Second pointer points to first value not fitting in the first HBase block. Index contains: rowID of first key/value pair, key without value, pointer to first key/value pair of block index points to. See Figure 18 for an example.

### Schema: Choice of Row Key

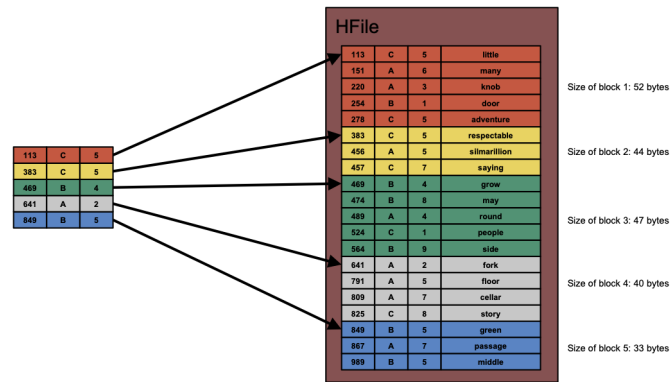


Figure 33: Example HBase index.

**Log Structured Merge Trees** A LSM tree is highly efficient in applications using wide column storage where insertions in memory happen quite often. As opposed to B+-tree which has a time complexity of  $O(\log n)$  when inserting new elements,  $n$  being the total number of elements in the tree, LSM tree has  $O(1)$  for inserting, which is a constant cost.

All of the stores are always sorted by key, so no reordering is required to fit new keys in between existing ones. LSM tree promises high write throughput. LSM tree allows random insertion of key-value pairs. Flush and compaction do not happen at the same time.

**Insert:** Writes are inserted in sorted MemStore, which is flushed to disk as a HFile segment if MemStore is too big. Old HFiles are periodically compacted together to save disk space and reduce fragmentation of data.

**Read:** Lookup key in MemStore. With hash index, search in one or more HFiles (depends on status of compaction).

**Delete:** Special case of update - store delete marker, use during lookup to skip “deleted” keys. When the pages are rewritten asynchronously, the delete markers and the key they mask are eventually dropped.

**Exercise:** Writing, reading and deleting key/value pairs. When inserting key/value pairs, sort them in MemStore and note down timestamp (increase also for non-matching keys). Flush MemStore if threshold is reached = sorted HFile on disk. Upon read, check both MemStore and Disk and get latest value. Compact HFiles on disk if threshold reached = sorted HFile, remove duplicate keys with earlier timestamp. Upon delete, keep key/value pair but also store delete marker.

**HBase Atomicity Guarantees** Row-level atomicity.

**HBase Replication Functionality** Setting up a backup HBase cluster in a separate datacenter to be synced with the current cluster. NOT the management of row replicas on many RegionServers.

## HBase vs. RDBMS

- HBase for: Data schema is difficult to determine in advance, or changes frequently and most of the data is sparse.
- RDBMS for: Dense and highly-structured data and Database consistency must always be maintained.

**Write-Ahead Log (WAL)** Without the WAL, in order to keep guaranteeing durability every write operation would require sorting an HFile.

## 6.6 Validation

### 6.6.1 XML Information Set

**Recap** XML "Information Set" provides an abstract representation of an XML document — it can be thought of as a set of rules on how one would draw an XML document on a whiteboard. An XML document has an information set if it is well-formed and satisfies the namespace constraints. There is no requirement for an XML document to be valid in order to have an information set. An information set can contain up to eleven different types of information items, e.g., the document information item (always present), element information items, attribute information item, etc. Information sets can be drawn as trees.

**Example 1:** Tree can contain: document information item, elements, character information items, and attributes. See tree in 34.

```
1 <Burger>
2   <Bun>
3     <Pickles/>
4     <Cheese origin="Switzerland" />
5     <Patty/>
6   </Bun>
7 </Burger>
```

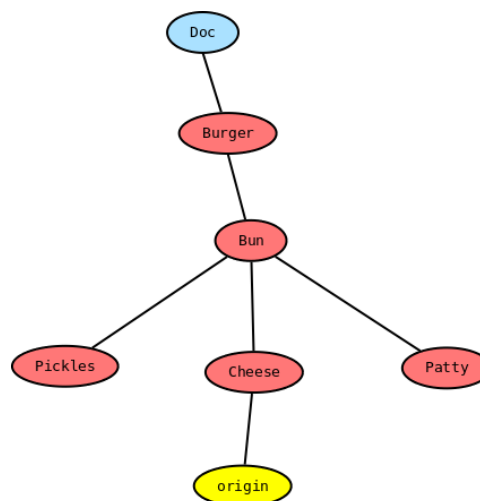


Figure 34: Example 1 XML tree.

**Example 2:** Tree can contain: document information item, elements, character information items, and attributes. See tree in 35.

```
1 <catalog>
2   <!-- A list of books -->
3   <book id='bk101'>
4     <author>Gambardella, Matthew</author>
5     <title>XML Developer's Guide</title>
6     <genre>Computer</genre>
```



```

7      <price>44.95</price>
8      <publish_date version='hard' version2='soft'>2000-10-01</
        publish_date>
9  </book>
10 </catalog>

```

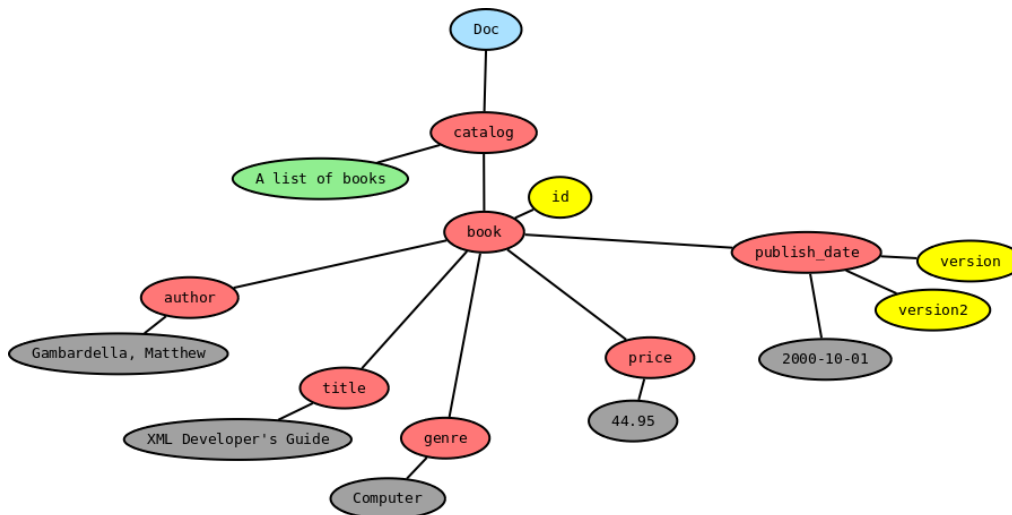


Figure 35: Example 2 XML tree.

**Example 3:** Tree can contain: document information item, elements, character information items, namespace items and attributes. See tree in 36.

```

1 <?xml version="1.0"?>
2 <!DOCTYPE eth>
3 <eth xmlns="http://www.ethz.ch" xmlns:ethdb="http://www.dbis.ethz.ch"
  date="11.11.2006" ethdb:date="12.11.2006">
4   <date>16.11.2017</date>
5   <president since="2015">Prof. Dr. Lino Guzzella</president>
6   <ethdb:Rektor>Prof. Dr. Sarah M. Springman</ethdb:Rektor>
7 </eth>

```

### 6.6.2 XML Schema

**Recap** An XML Schema describes the structure of an XML document. The purpose of an XML Schema is to define the legal building blocks of an XML document: the elements and attributes that can appear in a document, the number of (and order of) child elements, data types for elements and attributes, default and fixed values for elements and attributes.

#### Validation: Document Matches Schema

#### Provide Schema

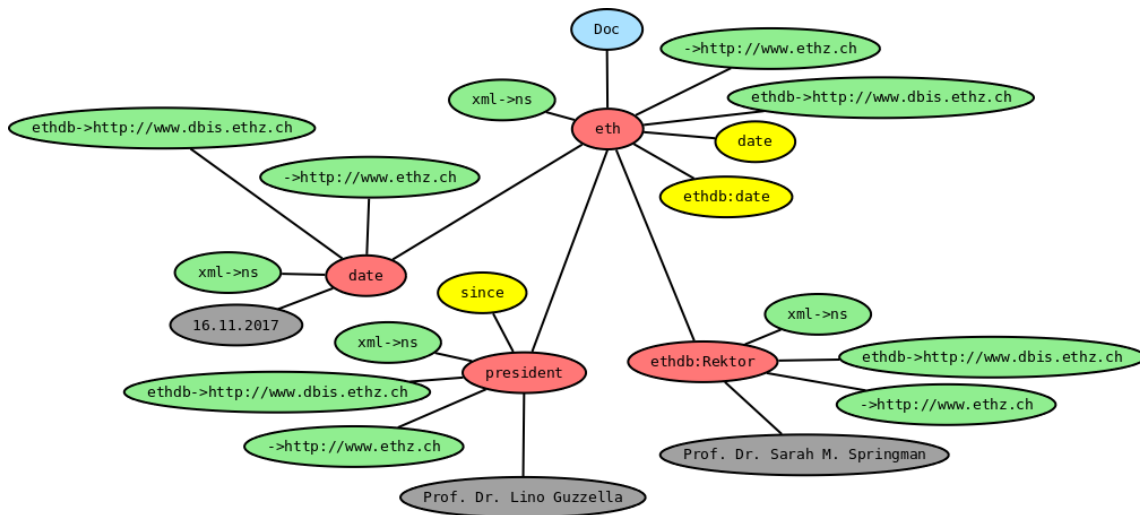


Figure 36: Example 3 XML tree.

### 6.6.3 JSON Schema

**Recap** JSON Schema is a vocabulary that allows you to annotate and validate JSON documents. It is used to: describe your existing data format(s), provide clear human- and machine- readable documentation, validate data, i.e., automated testing, ensuring quality of client submitted data.

### 6.6.4 Document Validation

#### Document Type and Validation

- **XML:** XML Schema, DTD, Schematron, RelaxNG, or not validated at all
- **JSON:** JSON Schema, JSound, Kwalify, or not validated at all
- **Protocol Buffers:** Protocol buffer schema language, it always has to be validated
- **XHTML:** XHTML XML Schema

### 6.6.5 Dremel

**Recap** Dremel is a query system developed at Google for deriving data stored in a nested data format such as XML, JSON, or Google Protocol Buffers into column storage, where it can be analyzed faster.

**Conversion Algorithm** Convert nested data format with an associated schema (e.g. Google Protocol Buffer document and schema) into column storage.

#### Statements

- False: Column storage takes up more disk space than nested data to allow for faster read-only operations.
- Dremel does not use standard SQL as its query language.
- False : After converting to column storage using the Dremel method, we can use this representation to speed up writes to the original data source.

- Column storage is faster than row storage at performing projections.

## 6.7 MapReduce

### 6.7.1 Reverse Engineering

### 6.7.2 True/False, Facts

#### True

- MapReduce splits might not correspond to HDFS blocks. Since splits respects logical record boundaries, they might contain data from multiple HDFS blocks.
- One single Reducer is applied to all values associated with the same key. This is the principle behind partitioning: one Reducer is responsible for all values associated with a particular key.
- Multiple Reducers can be assigned pairs with the same value. Values are not relevant in partitioning.

#### False

- Each mapper must generate the same number of key/value pairs as its input had. Why: For each input pair, the mapper can emit zero, one, or several key/value pairs.
- The TaskTracker is responsible for scheduling mappers and reducers and make sure all nodes are correctly running. Why: The JobTracker is responsible for this.
- The input key/value pairs of mappers are sorted by the key. Why: mapper input is not sorted.
- In Hadoop MapReduce, the key-value pairs a Reducer outputs must be of the same type as its input pairs. Why: Reducer's input and output pairs might have different types.

#### Benefits of Combine Function

- Decrease memory requirements on Reducers (but not in mappers)
- Decrease the overall communication volume
- Does not decrease mapper computation time

**Load Balancing** Requires prior knowledge about the distribution of keys - good partitioning function. Not automatic.

**Inefficiency** MR is always inefficient if: reducers get a lot of data with a small amount of possible keys.

## 6.8 YARN and Spark

### 6.8.1 YARN

#### Issues Addressed

- **Scalability:** MapReduce has limited scalability, while YARN can scale to 10,000 nodes.
- **Rigidity:** MapReduce v1 only supports MapReduce specific jobs. There is a need, however, for scheduling non-MapReduce workloads. For instance, we would like the ability to share cluster with MPI, graph processing, and any user code.
- **Resource Utilization:** in MapReduce v1, the reducers wait on the mappers to finish (and vice-versa), leaving large fractions of time when either the reducers or the mappers are idle. Ideally all resources should be used at any given time.
- **Flexibility:** mapper and reducer roles are decided at configuration time, and cannot be reconfigured.

### 6.8.2 Schedulers

**FIFO** Application in queue, run in order of arrival. No time guarantees.

**Fair Scheduler** All applications have same priority and resources are fairly distributed. Resources are dynamically balances between running jobs. New job - find new balance. If preemption is enabled, reclaiming of resources is instant (termination of running jobs possible) - else wait until resources freed up.

**Capacity Scheduler** Each user has certain minimum capacity guarantees. Resources are allocated over a set of predetermined queues. Each queue gets only a fraction of the cluster resources. As a result, each queue has a minimum guaranteed resource allocation. Capacity Schedulers feature two different metrics for assigning resources: SFS and IFS.

**Compute SFS** Given resources (memory, cores, amount of nodes) and hierarchical queue config, compute SFS.

1. Calculate total capacity (total amount of memory and cores).
2. SFS for each queue: first level simply demand / 100, second level parent result times (demand / 100).
3. For each node (!), result times resources.

#### CS vs. FS

- In CS and with no queue elasticity, applications submitted to a queue that does not have enough resources will be rejected.
- In FS, always count with the full resources.

#### DFS Exercise

### 6.8.3 Spark Architecture

## 6.9 Spark Dataframes and SparkSQL

### 6.10 Document Stores (MongoDB)

**Normalized Data in DS** References can be used for data normalization. In Figure 37, instead of storing contact and access as nested objects in the user document, new documents with a foreign key are created and referenced in the original.

Different relationships between data can be represented by references and embedded documents.

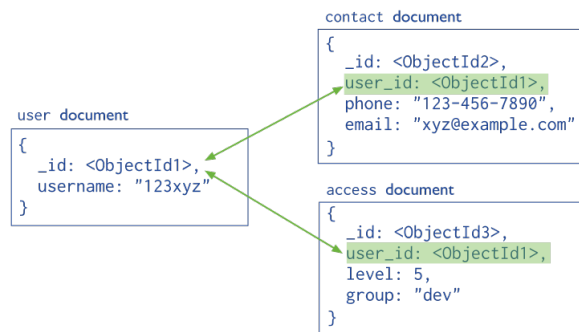


Figure 37: How to normalize data in MongoDB.

**DS vs. KVS** Document-oriented databases are inherently a subclass of the key-value store. The difference lies in the way the data is processed: in a key-value store, the data is considered to be inherently opaque to the database, whereas a document-oriented system relies on an internal structure of the documents in order to extract metadata that the database engine uses for further optimization. Although the difference is often mostly in tools of the systems, conceptually the document-store is designed to offer a richer experience with modern programming techniques.

### Data Model to Document Mapping

### Writing MongoDB

### MongoDB Indices

**Padding** The padding factor is the amount of extra space MongoDB leaves around new documents to give them room to grow. When MongoDB has to move a document, it bumps the collection's padding factor. You can see the padding factor by running `db.coll.stats()`.

**Cursor** Almost every method on a cursor object returns the cursor itself so that you can chain options in any order.

**Type Comparison Order** <https://docs.mongodb.com/manual/reference/bson-type-comparison-order/>

## 6.11 Rumble

**Idempotent Queries** Any JSON document is also a JSONiq query. Running a JSON document as a query just outputs itself. This also works the other way round: if your query outputs an object or an array, you can use it as a JSON document.

## 6.12 Graph Databases and Neo4j

- In Neo4j, relationships can be traversed in either direction with the same cost.
- The starting points of our graph queries are called bound nodes.
- In Neo4j an edge stores pointers to all the edges of both the source and the target nodes, using double-linked lists.
- With index free adjacency every record stores a pointer to the relationships connected to that node, making the lookup of relationship in constant time.
- B-Tree index lookups on a relational database with  $n$  elements :  $O(\log n)$
- Doing an  $m$  step/hop traversal on a graph with  $n$  elements and use index free adjacency:  $O(m)$
- Traverse a network of  $m$  steps/hops with  $n$  elements using index lookups :  $O(m \cdot \log n)$
- Looking up immediate relationships in a graph database with index free adjacency :  $O(1)$
- Neo4j transactions are ACID-compliant, committing data to disk in Neo4j uses a Write Ahead Log
- Neo4j (labelled property graph) and RDF (triple stores)
- Neo4j uses a query by example paradigm, while SPARQL follows a declarative query paradigm similar to SQL
- In RDF a property can also be the source or the target of another triple
- 

## 6.13 OLAP and Cubes, Data Warehousing

Given a fact table with  $m$  dimensions and  $n$  different possible values for each dimension and assuming that it has no duplicate rows or missing values, the amount of rows in the fact table is:  $n^m$ , this represents all combinations between all possible values for all dimensions.

Creating a view for the full data cube (adding rows with Nulls for slice aggregations) out of this fact table gives us  $(n + 1)^m$  rows (as we now have the possibility of aggregating over a dimension, and we represent this by using a Null value for that dimension. Therefore, it's as if we have a new "possible value" for our dimensions, this value being Null).