ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Big Data

Roman Böhringer

February 13, 2021

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Introduction

The three paramount factors when dealing with data are capacity (how much data can we store?), throughput (how fast can we transmit data?), and latency (when do I start receiving data?). While capacity has increased a lot in the last decades, throughput and especially latency couldn't catch up.

1.1 History

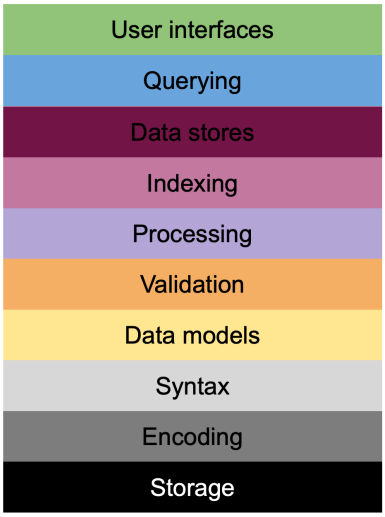
To find books in libraries, the dewey decimal system was invented. It consists of three numbers, the first indicates the category, the second the sub-category and so on.

Before relational databases, there were file systems in the 1960s. In the 1970s, relational databases were introduced. The 1980s are known as the "object era" whereas the 2000s are the NoSQL era (key-value stores, triple stores, column stores, document stores, ...). Big data consists of the three Vs: Volume, variety, and velocity. Data is worth more than the sum of its parts. It can be represented in different shapes:

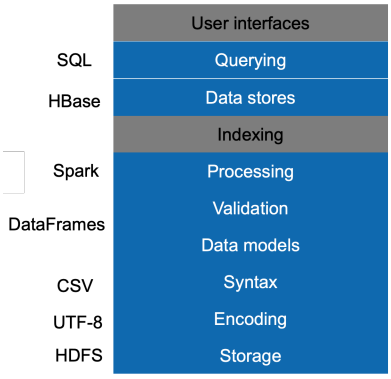
- Tables
- Trees
- Graphs
- Cubes
- Text (Information Retrieval lecture)

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

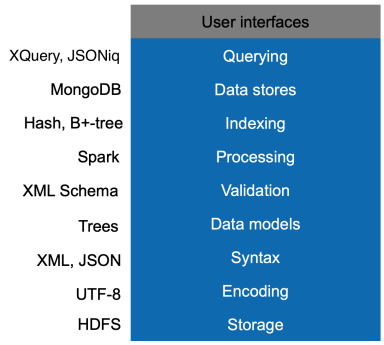
accommodating for the issue of growing discrepancy between capacity, throughput and latency. It consists of a modular technology stack:



With tables, the stack looks like this:



And with trees:



10 principles of big data are:

1. Learn from the past
2. Keep the design simple
3. Modularize the architecture
4. Homogeneity in the large
5. Heterogeneity in the small
6. Separate metadata from data
7. Abstract logical model from its physical implementation
8. Shard the data
9. Replicate the data
10. Buy lots of cheap hardware

1.2 Prefixes

The International System of Units defines these prefixes:

- **pico (p)**: 0.000 000 000 001 (12 places)
- **nano (n)**: 0.000 000 001 (9 places)
- **micro (μ)**: 0.000 001 (6 places)
- **milli (m)**: 0.001 (3 places)
- **kilo (k)**: 1,000 (3 zeros)
- **Mega (M)**: 1,000,000 (6 zeros)
- **Giga (G)**: 1,000,000,000 (9 zeros)

- **Tera (T)**: 10^{12}
- **Peta (P)**: 10^{15}
- **Exa (E)**: 10^{18}
- **Zetta (Z)**: 10^{21}
- **Yotta (Y)**: 10^{24}

1.3 Scaling

There are two main approaches, scaling up (using better/more expensive hardware) and scaling out (using more nodes). Scaling out has the advantage that the costs increase linearly, whereas the increase can be exponential when scaling up (as better hardware becomes more and more expensive). The third approach is to use better algorithms and optimize the software.

1.4 CAP Theorem

Any two of the following three properties can be achieved at the same time, but not all of them:

- (Atomic) **Consistency**: All nodes see the same data.
- **Availability**: It is possible to query the database at all times.
- **Partition Tolerance**: The database continues to function even if the network gets partitioned.

No distributed system is safe from network failures, thus network partitioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability. When choosing consistency over availability, the system will return an error or a time out if particular information cannot be guaranteed to be up to date due to network partitioning. When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

In the absence of network failure – that is, when the distributed system is running normally – both availability and consistency can be satisfied.

1.5 HTTP/REST

A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource. A Uniform Resource Locator (URL) is a URI that specifies the means of acting upon or obtaining the representation of a resource, i.e. specifying both its primary access mechanism and network location. A URL is therefore a type of URI that identifies a resource via a representation of its primary access mechanism (e.g., its network "location"), rather than by some other attributes it may have. A Uniform Resource Name (URN) is a URI that identifies a resource by name in a particular namespace. A URN may be used to talk about a resource without implying its location or how to access it, e.g. `urn:isbn:0123456789`. A URI consists of:

- Scheme (e.g. `http`)
- Authority (e.g. `//www.ethz.ch`)
- Path (e.g. `/api/collection/object`)
- Query (e.g. `?id=abc`)
- Fragment (e.g. `#head`)

The main HTTP methods that are used in REST APIs are:

- GET: Side-effect free
- PUT: Idempotent
- DELETE
- POST: Most generic and with side effects

1xx status codes are for informational responses, 2xx for success, 3xx for redirections, 4xx for client errors, and 5xx for server errors. The most important status codes are:

- 200: OK
- 201: Created
- 301: Permanent Redirect
- 302: Temporary Redirect
- 400: Bad Request
- 401: Unauthorized
- 403: Forbidden

- 404: Not Found
- 405: Method Not Allowed
- 410: Gone
- 500: Internal Server Error
- 501: Not Implemented
- 502: Bad Gateway
- 503: Service Unavailable
- 504: Gateway Timeout

Chapter 2

Relational Databases

A key concept that was introduced by Edgar Codd is data independence: The logical data model (e.g. a table) is separated from the physical storage (e.g. file system on a hard drive or multiple nodes). The data model defines what data looks like and what you can do with it.

Key concepts in relational databases are tables (collections), attributes (columns, fields or properties), primary keys (row IDs, names), and rows (business objects, items, entities, documents, records). We can view a table as a relation, i.e.:

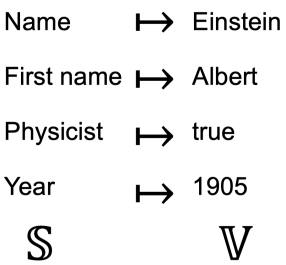
$$R \subseteq \mathcal{D}_1 \times \mathcal{D}_2 \times \mathcal{D}_3$$

On the other hand, we can view it also as a collection of partial functions from some properties $\mathbb{S}$ to some values $\mathbb{V}$, i.e.

$$f \in \mathbb{S} \nrightarrow \mathbb{V}$$

$\mathbb{S}$ is made of a set of attributes ($\text{Attributes}_R \subseteq \mathbb{S}$) and the table consists of an extension (set of tuples) $\text{Extension}_R \subseteq \mathbb{S} \nrightarrow \mathbb{V}$ such that

$$\forall t \in \text{Extension}_R, \text{support}(t) = \text{Attributes}_R$$



In SQL, we have three key concepts:

- **Tabular Integrity**: All tuples / partial functions have the same attributes (which is stated by the formula above)
- **Atomic Integrity**: The domain of each attribute contains only atomic (indivisible) values and the value of each attribute contains only a single value from that domain.
- **Domain Integrity**: All columns must be declared upon a defined domain.

Relational algebra allows different types of relational queries:

- **Set Queries**: Union, Intersection, Subtraction
- **Filter Queries**:
  - Selection: Select rows that fulfill criteria, e.g. $S = \sigma_{B \leq 2}(R)$
  - Projection: Select subset of columns, e.g. $S = \pi_{A,C}(R)$
- **Renaming Queries**:
  - Relation renaming
  - Attribute renaming
- **Binary Queries**:
  - Cartesian Product: Combine all tuples of two relations, $T = R \times S$
  - Natural Join: Tables are joined on matching column, $T = R \bowtie S$
  - Theta Join

Normal forms are introduced for consistency (preventing update, delete, and insert anomalies):

1. Normal Form: Atomic integrity
2. Normal Form: Every non-prime attribute (that is not a part of any candidate key) is dependent on the whole of every candidate key. This is for instance violated in a table "CD_Track" with key ("CD_ID", "Track number") and fields artist, album title, and year of founding, because the fields only depend on "CD_ID".
3. Normal Form: All Attributes are functionally dependent on solely the primary key. If we split the previous example into a relation with ("CD_ID", "artist", "album title", "founding year"), it would still violate the third normal form, as founding year depends on artist.

SEQUEL (Structured English Query Language) in the early 1970s was the first commercial relational query language and System R the first commercial relational database. It is a set-based declarative language and was renamed to SQL because of trademark issues. The logical model defines what, but not how, so there are multiple possible physical executions (query plans, parallelism) possible. The DML (data manipulation language) deals about queries, insertions, and deletions whereas the DDL (data definition language) is to create tables/schemas and drop them.

ACID refers to a standard set of properties that guarantee database transactions are processed reliably:

- **Atomicity**: Either the entire transaction is applied or none of it.
- **Consistency**: After a transaction, the database is in a consistent state again.
- **Integrity**: A transaction feels like nobody else is writing to the database.
- **Durability**: Updates made do not disappear again.

Chapter 3

Object Storage

File content is often stored in blocks because otherwise, latency would dominate. Explicit block storage gives the application control over the locality of blocks.

Object storage consists of:

- "Black-box" objects
- Flat and global key-value model
- Flexible metadata
- Commodity hardware

They are generally replicated to be more fault tolerant. For regional fault tolerance and lower latency, they are replicated among regions.

3.1 Amazon S3

Logically, S3 consists of buckets and objects. Every object is identified by a bucket + object ID. The maximum size of an object is 5TB and Amazon guarantees high durability and availability. The objects can be accessed over a REST API, e.g. PUT/DELETE/GET `http://bucket.s3.amazonaws.com/object-name`. Keys can have slashes to emulate a hierarchy, but they are just strings.

3.2 Azure Blob Storage

In Azure Blob Storage, an object is identified by the account + container + blob. Instead of only providing blackbox objects, block blobs, append

blobs (optimized for append operations, e.g. logging), and page blobs (optimized for random read/writes, e.g. for Azure virtual machine disks) are provided. The physical architecture consists of multiple storage stamps (10-20 racks with a total storage of 30PB). There's synchronous intra-stamp replication and asynchronous inter-stamp replication. The account name is mapped to the virtual IP of the primary stamp.

3.3 Key-Value Storage

Object stores like S3 or Azure Blob Storage make use of the key-value model, but aren't key-value stores in the classical sense. Key-Value stores have smaller objects (400KB for Dynamo) and no metadata. The API is very basic and usually consists of `get(key)` and `put(key, value)`. In the Dynamo API, a context (vector clock value) is provided in addition to the value when calling `get` that is used in subsequent `put`'s. Key-Value stores usually only provide eventual consistency (if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value) which allows higher performance and better scalability.

3.3.1 Distributed Hash Table

The design principles for distributed key-value stores are:

- Incremental Stability: New nodes can be added or removed
- Symmetry: Machines behave the same (e.g. no master/slave)
- Decentralization: No central node with special roles (different principle than symmetry as it is possible to elect central nodes, i.e. have symmetry but still centralization)
- Heterogeneity: Nodes can be equipped with different kind of hardware.

In distributed hash tables, keys are hashed to a n-bit ID that are organized in a logical ring (i.e. $\mod 2^n$). A node picks a n-bit hash randomly and is placed on the ring. All IDs that are stored until the next node are assigned to the node. When adding/removing nodes, only the neighboring ones have to be contacted. For fault tolerance, each node can be responsible for k consecutive ranges instead of only one. Finger tables index all nodes at distance 2^i and therefore allow logarithmic traversal to any other node. In dynamo, every node knows about the ranges of all other nodes (which allows constant traversal). Writes are confirmed by at least W nodes (quorum). Pros of distributed hash tables are:

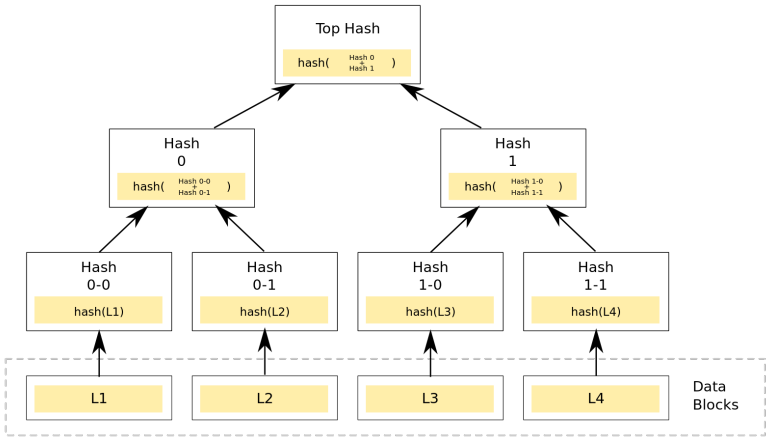
- Highly scalable
- Robust against failure
- Self organizing

And Cons are:

- Only lookup, no search
- Data integrity
- Security issues

If every node is only assigned one hash, it can happen that the range is split non-uniformly (especially for only a few nodes) by chance. Furthermore, one cannot incorporate heterogenous performance. Therefore, virtual nodes/tokens (additional hashes) are introduced. Every node can have multiple tokens and the number of tokens per node can differ (based on the performance).

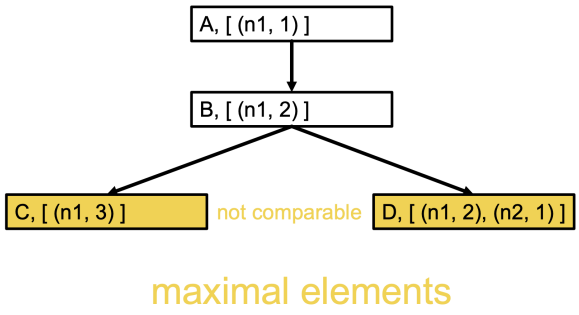
To efficiently detect data corruptions/differences among replicas, anti-entropy protocols are used. For instance, Merkle Trees can be used. The leaves are hashes of data blocks and nodes further up in the tree are the hashes of their respective children.



To detect differences, one starts comparing the top hashes. If they agree, there are no differences, if they do not, we continue with hash 0 and 1, until the differing data blocks are found.

Vector Clocks

A vector clock is an algorithm for generating a partial ordering of events in a distributed system. Each node has a timestamp that is incremented on an operation (put). The states of the values and vector clocks therefore form a directed acyclic graph, where the elements can become non comparable on a partition:



In such a situation, both values are returned to the client (including the merged vector clock/context, e.g. `C, D, [(n1, 3), (n2, 1)]` and it's the responsibility of the client to solve the conflict and put a new value including the old context, e.g. `put(key1, [(n1, 3), (n2,1)], E)`, resulting in an absolute maximum again. During synchronization, old values/contexts can be cleaned up or kept to allow versioning. Formally, we have:

$$VC(x) \leq VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z]$$

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

Chapter 4

Distributed File Systems

Object storages are suitable for a huge amount of large files (billions of TB files), whereas distributed file systems are used for a large amount of huge files (millions of PB files). In object storage, we use the key-value model with a flat hierarchy whereas in block storage/distributed file system, we have a file system that forms a tree. Fault tolerance and robustness are key concerns for distributed file systems because in large clusters, nodes will fail. This requires monitoring, error detection, and automatic recovery.

When reading and updating files, we can have mainly random access or mainly scans and upserts/appends. The latter is common in data analytics and distributed file systems are optimized for it. Because of this access pattern, the top priority is throughput and latency is secondary. The appends are usually done by 100s of clients in parallel.

In distributed file systems, the logical model is a hierarchy of files that are mapped to (possibly multiple) blocks (chunks in GFS parlance). Therefore, we have physically multiple blocks in contrast to object storage with large objects. Blocks allow files that are bigger than a disk and are a simpler level of abstraction. The blocks of distributed file systems are usually in the range of 64MB - 128MB (much larger than those of simple file systems as the latency of the network is much larger). Because a peer-to-peer architecture would result in very complex deployments, a Master-Slave architecture is usually used for distributed file systems.

4.1 Hadoop/HDFS

Apache Hadoop is a collection of open-source software utilities (including the Hadoop distributed file system, inspired by Google's GFS) that

facilitate using a network of many computers to solve problems involving massive amounts of data and computation.

The Hadoop Distributed File System (HDFS) is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. In HDFS, there is a NameNode and many DataNodes. A file is divided into 128MB blocks (taking up only the space that is needed when they are smaller) that are replicated among DataNodes for fault tolerance (but all replicas are equal, i.e. there is no primary replica). Clients first access the NameNode which stores (in RAM):

1. File namespace (and possibly access control)
2. The file to block mapping
3. The block locations

DataNodes store blocks on their local disks.

Communication is done using three protocols:

- **Client Protocol**: A RPC-based protocol that is used for metadata operations (querying location of files, metadata queries, file creation, etc...) between clients and the NameNode.
- **DataNode Protocol**: The DataNode protocol for communication between the NameNode and DataNodes is RPC-based as well and communication is always initiated by the DataNodes. It is used for registration, heartbeat messages containing additional information such as the storage capacity, fraction of storage in use, number of data transfers in progress (every 3s per default), and BlockReports (full report of stored blocks including hashes, every 6h per default). NameNodes can respond to heartbeat messages with block operations, e.g. the order to replicate blocks, which are acknowledged by the DataNodes with BlockReceived messages.
- **DataTransfer Protocol**: A streaming protocol for the actual transfer of data blocks between the client and DataNodes (the NameNode isn't involved). For writes (with at most one concurrent writer per file, controlled by leases), replication pipelining is used: A client only writes to a single DataNode which then replicates the write to another DataNode, which then replicates to another DataNode, ...

When reading a file, a client asks the NameNode for the file and gets the block locations (multiple DataNodes for each block, sorted by distance). The client then communicates with the DataNodes, gets the blocks, and

verifies their checksums. The Java library provides an input stream that hides these steps. DataNodes cache commonly accessed blocks and job schedulers can take advantage of cached blocks.

When writing a file, a client contacts the NameNode and gets the DataNodes for the first block. This node is contacted to organize the pipeline and data is sent (in a streaming fashion with smaller packets) to the node, which forwards it to the next pipeline node. After the acknowledgement of this node, the NameNode is contacted for the second block and the same procedure is done. This is all done simultaneously until the last block is written, after which the client contacts the NameNode to close/release the lock. The NameNode then checks for minimal replication (using the DataNode protocol) and acknowledges the write if the criteria is met. Possible further replicas are created asynchronously by the NameNode. The replication factor is a system-wide configuration but can also be set per file. Any content written a file is not guaranteed to be visible directly afterwards except a `hflush()` operation is performed. For parallel copying, the tool `distcp` (a MapReduce job) is provided.

For the replica placement, we should consider reliability, read/write bandwidth and block distribution. HDFS places the first replica on the same node as the client (or on a random node if the client is not part of the cluster). The second and third replicas are placed in the same rack (different to the one of the client/the first replica), but on different nodes. More replicas are placed randomly but if possible in such a way that there is at most one replica per node and at most two replicas per rack. With this placement policy, a good distribution of blocks is achieved (placing the first two replicas on the same rack as the client would mean that 2/3 of a files replicas are on the same rack).

The HDFS shell is used to access HDFS clusters, e.g. `hadoop fs -ls`, `hadoop fs -cat /dir/file`, or `hadoop fs -copyFromLocal localFile1 localFile2 /dir1/dir2`.

Apache Flume collects, aggregates, and moves log data into HDFS whereas Apache Sqoop imports from a relational database.

4.1.1 Availability

Because the NameNode is a single point of failure, the file namespace and the file-to-block mapping are written to a namespace file on persistent storage (checkpoint) and to prevent constant rewriting of this file, an edit log/write-ahead commit log (journal) is used for subsequent changes. Those files should be backed up and can be written to multiple locations

automatically. The block locations are not persisted as they can be rebuilt with the DataNodes. When a NameNode starts up, the namespace file is read and the edit log is played to get all the changes after the creation of the namespace file. Block locations are recreated using the DataNode protocol. This whole process takes around 30 minutes. Because of this, multiple approaches have been developed historically:

- Checkpoints with Secondary NameNodes: A secondary NameNode remerges the edit log periodically into a new namespace file such that it does not need to be merged after the startup. Checkpoint nodes are an advancement of secondary NameNodes but basically do the same (secondary NameNodes do not upload to the NameNode whereas checkpoint NameNodes do).
- Backup NameNodes: A different NameNode that maintains mappings and locations in memory (but does not provide automatic failover) and periodically saves to disk.
- Standby NameNodes: Provide automatic failover if the NameNode fails (High Availability).

The namespace can also be sharded/partitioned, leading to Federated DFS. If the mappings no longer fit into memory of a single node, different NameNodes are responsible for different parts of the namespace (e.g. one for `/foo` and one for `/bar`).

File system snapshots are supported to roll back the whole system in case of a failed update. Occasionally, block checksums are verified by DataNodes and the blocks are moved in case of imbalances.

Chapter 5

Syntax

JSON and XML are both syntaxes to represent trees. They allow highly denormalized data. For read-intensive workloads, we generally want to avoid joins (for performance reasons) and therefore have denormalized/pre-joined data. Collection of tuples can be easily represented in XML and JSON.

Relational databases are highly structured, whereas text is unstructured data. Tree-like documents like XML and JSON are semi-structured documents. XML (and HTML) is standardized by the W3C, JSON by ECMA. The goal of them is to be human-readable and processable by a machine at the same time.

Both are well-formed languages, meaning a document is either well-formed or not (in contrast to HTML, where browsers often try to fix mistakes).

Besides XML/JSON, there is YAML (very similar to JSON) and CSV for tables.

5.1 JSON

The basic data types are:

- String: Delimited with double-quotation marks, backslash escaping syntax is supported. Unicode characters are represented by `\u` and the unicode code point.
- Number: There are no size limits, numbers can be written as floating point numbers, integers and in scientific notation, e.g. `-1.23E+5`.
- Boolean: `true` and `false`
- Null: `null`

- Array: An ordered list of zero or more values that can be any type, denoted by square brackets with comma-separated elements. Trailing commas are not allowed.

```
[
        3.14159265368979,
        true,
        "This is a string",
        { "foo" : false },
        null
]
```

- Object: A collection of name-value pairs where the names (keys) are strings. Trailing commas are not allowed as well, duplicate keys should not be used.

```
{
        "foo": 3.14159265368979,
        "bar": true,
        "str": "This is a string",
        "obj": { "school" : "ETH"},
        "Q": null
}
```

5.2 XML

In XML, an element consists of an opening tag and a closing tag and contains potentially more XML: `<foo>[more XML]</foo>` For empty elements, the short form `<bar/>` can be used which is equivalent to `<bar></bar>`. Attributes are key value pairs, duplicate keys are not allowed. The quotes are always needed, but single and double quotes can be used: `<a attr="value"/>` Elements can contain text which is not enclosed by quotes, e.g. `<a>Text</a>`. Comments are denoted by `<!-- Comment -->`. They are transmitted to the application (in contrast to comments in programming languages) and `--` is not allowed in comments. XML also contains processing instructions of the form `<?myapp do whatever ?>`. An optional text declaration of the form `<?xml version="1.0" encoding="UTF-8"?>` can be used. Valid versions are 1.0 and 1.1, but the differences are minor (related to non-ASCII characters, line ending characters, and control characters). Nesting of the elements needs to be proper, i.e. `<a><b></a></b>` is invalid. Escaping is done by entity references:

- `&lt;` → `<`
- `&gt;` → `>` (often not needed, but still done for symmetry reasons)

- `&apos;` → ' (can use double quotes/single quotes to enclose attribute values, when using double quotes single quotes do not need to be escaped and vice versa)
- `&quot;` → "
- `&` → &

When we for instance integrate an HTML document into a XML document, we sometimes need to use double escaping, e.g. `2 &lt; 3` which the XML parser translates to `2 < 3` and then the HTML parser to `2 < 3`. An alternative to escaping are CDATA sections where everything is interpreted as textual data (not marked up content). They start with `<![CDATA[` and end with `]]>`. In the document type definition, custom entities can be declared that can then be referenced like the other ones:

```
<?xml version "1.0"?>
<!DOCTYPE document [
<!ENTITY myownentity "foobar">
]>
<document>
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
&myownentity;
eiusmod tempor incididunt ut labore et dolore magna aliqua.
</document>
```

It is also possible to include external parsed entities/XML fragments (with relaxed conditions, multiple elements and text at top level are allowed) or external unparsed entities like images:

```
<?xml version "1.0"?>
<!DOCTYPE document [
<!ENTITY myownentity SYSTEM "/path/to/file.xml">
]>
<document>
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
&myownentity;
eiusmod tempor incididunt ut labore et dolore magna aliqua.
</document>
```

Character references are used to encode arbitrary Unicode characters. `&#960;` or `&#x03C0;` (decimal/hexadecimal) are for instance used to encode Π.

XML names cannot start with a digit, -, or . (but those characters are allowed within) and all other special characters (including spaces) except _ (which is allowed anywhere) are disallowed for names. The name `xml` is reserved for the W3C and nobody else is allowed to use names starting with `xml`.

Every XML document needs to have exactly one root element, these documents are therefore not well-formed:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo/>
<bar/>
```

```
<?xml version="1.0" encoding="UTF-8"?>
text
<foo>
<bar/>
</foo>
text
```

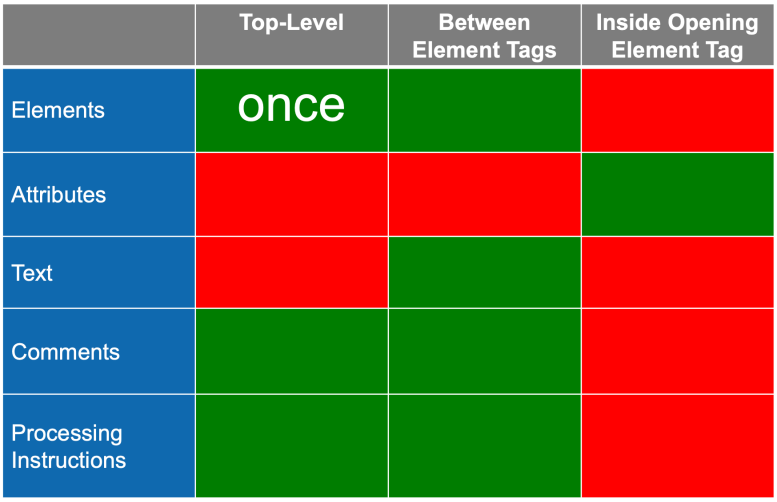
But text inside elements is allowed, e.g.:

```
<foo>
text <bar/> text
</foo>
```

Furthermore, there can be multiple comments/processing instructions:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is a comment -->
<!-- this is another comment -->
<?processing instruction?>
<foo>
<bar/>
</foo>
<?processing instruction?>
<!-- this is a comment -->
```

Inside attribute values, < is not allowed and needs to be escaped. A summary of what is allowed where is:



5.2.1 Namespaces

Namespaces are used to prevent name collisions. The namespace in addition with the local name forms the expanded name. Using the attribute `xmlns:m="http://www.w3.org/1998/Math/MathML"`, one binds the prefix `m` to the namespace `"http://www.w3.org/1998/Math/MathML"` (where URIs are used for name spaces because they identify resources, not because they relate to a web page). The combination of prefix, local name and namespace is called a QName. Semantically, only the local name and the namespace matters, these are therefore equivalent (with local name math and the MathML namespace):

```
1 <foo:math xmlns:foo="http://www.w3.org/1998/Math/MathML"/>
2 <m:math xmlns:m="http://www.w3.org/1998/Math/MathML"/>
3 <math xmlns="http://www.w3.org/1998/Math/MathML"/>
```

The last line introduces a default namespace which means that in this element, the names without a namespace (except for attribute names, unprefixed attributes are in no namespace even if there is a default namespace in scope!) are in this namespace. A best practice is to put all namespace bindings in the root element and to keep the same bindings across all documents. The default namespace should only be used if the document mainly uses one single namespace. Note that using the default namespace on the root element (which means that all elements are in the default namespace) and not using a namespace (which means that all elements are in no namespace) is not the same. The following snippet is not well-formed XML because the attribute name on the foobar element are semantically identical (although a different prefix is used, it refers to the same namespace).

```
<?xml version "1.0"?>
<bar
xmlns:foo="http://example.com/foo"
xmlns:bar="http://example.com/foo">
        <foo/>
        <foobar foo:attr="value" bar:attr="value"/>
</bar>
```

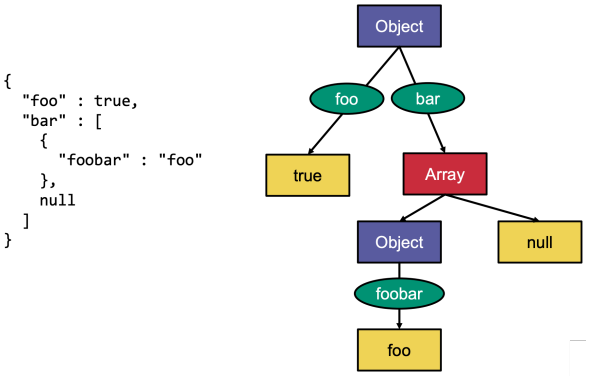
Chapter 6

Data Models

One needs to distinguish between the physical view which is the syntax and the logical view which is the data model. For instance, the data model of a table can be expressed in CSV syntax and the one of a tree in XML/JSON.

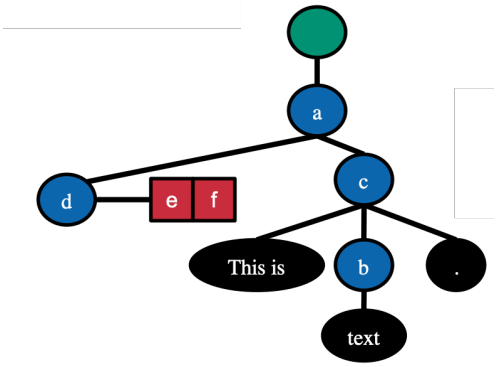
6.1 Trees

JSON consists of the atomic values strings, numbers, booleans, and null as well as the structured values objects and arrays. We can visualize a JSON document as a tree:



The same can be done with an XML document:

```
<a>
        <d e="f"/>
        <c>This is <b>text</b>.</c>
</a>
```



For JSON, the labels are on the edges whereas they are on the nodes for XML. Another difference (in memory) is that XML trees usually have backpointers and JSON's do not.

XML Information Set is a W3C specification describing an abstract data model of an XML document in terms of a set of information items. The main XML information items are:

- Document (virtual node at the top of the tree)
- Element
- Attribute
- Processing Instruction
- Character (Text)
- Comment
- Namespace

A document information item consists of the children and the version. Element information items consist of:

- Namespace Name
- Local Name
- Prefix
- Children
- Atrributes (attribute information items)
- In-Scope Namespaces (namespace information items)
- Parent

Attribute information items also have a namespace name, local name, and prefix and furthermore a normalized value and an owner element. Text information items have characters (their content) and an owner element and namespace information a prefix and a namespace name.

6.2 Types

There are many different type systems, but almost all have in common:

- Atomic and structured types
- Same categories of atomic types
- Lists and maps as structured types
- Sequence type cardinalities

The commonly used atomic types are:

- Strings
- Numbers (interval-based, arbitrary precision, and/or floats/doubles)
- Booleans
- Dates and Times
- Time Intervals (usually a separation between year/month and day/hour/minute/seconds, i.e. no combination of those allowed, as number of days per month vary)
- Binaries
- Null

We distinguish between the value space and the lexical space (how the values appear in a document, same value can have multiple entries in the lexical space, but there is often a canonical mapping). A subtype (usually) means that its value space is a subset of the supertype.

For the cardinalities, one often distinguishes:

- One (common adjective "required")
- Zero or more (common sign $*$, common adjective "repeated")
- Zero or one (common sign ?, common adjective "optional")
- One or more (common sign +)

6.3 XML Validation

Well-Formedness answers the question if the document can be parsed as a tree, whereas validation is done after well-formedness and checks if the document adheres to some schema. Annotation outputs a modified document with additional type information.

XML schemas are XML documents themselves (usually with a `.xsd` extension for XML schema definition), an example schema looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="foo" type="xs:string"/>
</xs:schema>
```

A valid document that references this schema looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="schema.xsd">
 This is text.
</foo>
```

There are simple and complex types in XML schema. There are built-in simple types for strings (string), numbers (decimal, float, double, ...), dates and times (dateTime, time, ...), time intervals (duration, ...), and binaries (hexBinary, base64Binary), but not for null. There are also three possibilities for a user to specify simple types:

1. **Restriction**: Further restrict a simple type with so-called facets, e.g. create a simple type for strings of length 3 with the length facet:

```
<xs:simpleType name="myFixedLengthString">
  <xs:restriction base="xs:string">
    <xs:length value="3"/>
  </xs:restriction>
</xs:simpleType>
```

2. **List**: A list of simple types (elements are always separated by spaces), e.g. a list of strings:

```
<xs:simpleType name="myList">
  <xs:list itemType="xs:string"/>
</xs:simpleType>
```

3. **Union**: Allows multiple simple types, e.g. integer or boolean:

```
<xs:simpleType name="myUnion">
  <xs:union memberTypes="xs:integer,xs:boolean"/>
</xs:simpleType>
```

For complex types, we distinguish the following four forms:

1. **Empty**, e.g. `<foo/>`:

```
<xs:complexType name="emptyType">
  <xs:sequence/>
</xs:complexType>
```

2. **Simple Content**, e.g. `<foo country="Switzerland">2014-12-02</foo>`: Without attributes, this corresponds to a simple type, with attributes (where the type of the attributes is a simple type) we have:

```
<xs:complexType name="dateCountry">
  <xs:simpleContent>
    <xs:extension base="xs:date">
      <xs:attribute name="country" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

3. **Complex Content**, e.g. `<foo><a/><b/></foo>`: Can contain detailed cardinality information:

```
<xs:complexType name="complexContent">
<xs:sequence>
  <xs:element name="twotofour" type="xs:string" minOccurs="2"
      maxOccurs="4"/>
  <xs:element name="zeroorone" type="xs:boolean" minOccurs="0"
      maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
```

   The order of the elements matter, a valid document looks like this:

```
<foo>
  <twotofour>foobar</twotofour>
  <twotofour>foobar</twotofour>
  <twotofour>foobar</twotofour>
  <zeroorone>true</zeroorone>
</foo>
```

4. **Mixed Content**, e.g. `<foo>Text<a/>Text<b/></foo>`: Activated by setting `mixed` to `true`:

```
<xs:complexType name="mixedContent" mixed="true">
  <xs:sequence>
    <xs:element name="b" type="xs:string" minOccurs="0" maxOccurs="
        unbounded"/>
  </xs:sequence>
</xs:complexType>
```

We can have named and anonymous types, the following two declarations are therefore equivalent:

```
<xs:complexType name="empty">
  <xs:sequence/>
</xs:complexType>
<xs:element name="c" type="empty">
</xs:element>
```

```
<xs:element name="c">
  <xs:complexType>
    <xs:sequence/>
  </xs:complexType>
</xs:element>
```

It is also possible to specify that an element must have unique keys:

```
<xs:key name="fooDid">
  <xs:selector xpath="foo"/>
  <xs:field xpath="@id"/>
</xs:key>
```

6.3.1 Namespaces

For elements without namespaces, the schema is referenced with `xsi:noNamespaceSchemaLocation`. In a schema, one introduces a namespace with `targetNamespace="NS"`. It is good practice to also bind the namespace in the schema (although it is not always needed). The schema is then referenced with `xsi:schemaLocation="NS schema.xsd"`. Therefore, an example schema/XML file with namespaces looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.example.com/bigdata"
 xmlns:big="http://www.example.com/bigdata">
  <xs:element name="foo" type="xs:string"/>
</xs:schema>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<big:foo
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.example.com/bigdata schema.xsd"
 xmlns:big="http://www.example.com/bigdata">
 This is text.
</big:foo>
```

For the name attribute of elements, we do not need to specify the namespace (a schema can be in exactly one namespace, therefore the namespace is always clear). But for the type, the namespace has to be specified.

We can import schemas in other namespaces with `xs:import` and reference elements in those schemas with `ref`, which is illustrated in the following example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<big:document
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
 http://www.example.com/bigdata schema-big.xsd
 http://www.example.com/mediumdata schema-medium.xsd
 http://www.example.com/smalldata schema-small.xsd
 "
 xmlns:big="http://www.example.com/bigdata"
 xmlns:medium="http://www.example.com/mediumdata"
 xmlns:small="http://www.example.com/smalldata">
  <medium:page>
    <small:paragraph/>
    <small:paragraph/>
    <small:paragraph/>
  </medium:page>
  <medium:page>
    <small:paragraph/>
    <small:paragraph/>
    <small:paragraph/>
  </medium:page>
</big:document>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:small="http://www.example.com/smalldata"
 targetNamespace="http://www.example.com/smalldata">
  <xs:element name="paragraph" type="small:empty-string"/>
  <xs:simpleType name="empty-string>
    <xs:restriction base="xs:string">
      <xs:length value="0"/>
    </xs:restriction>
```

```
  </xs:simpleType>
</xs:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:small="http://www.example.com/smalldata"
 xmlns:medium="http://www.example.com/mediumdata"
 targetNamespace="http://www.example.com/mediumdata">
  <xs:import namespace="http://www.example.com/smalldata"
  schemaLocation="schema-small.xsd"/>
  <xs:element name="page" type="medium:paragraphs"/>
  <xs:complexType name="paragraphs">
    <xs:sequence>
      <xs:element ref="small:paragraph" maxOccurs="unbounded">
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:big="http://www.example.com/bigdata"
 xmlns:medium="http://www.example.com/mediumdata"
 targetNamespace="http://www.example.com/bigdata">
  <xs:import namespace="http://www.example.com/mediumdata"
  schemaLocation="schema-medium.xsd"/>
  <xs:element name="document" type="big:pages"/>
  <xs:complexType name="pages">
    <xs:sequence>
      <xs:element ref="medium:page" maxoccurs="unbounded">
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

6.4 JSON Validation

JSON Schema is similar to XML schema, schemas are JSON documents that optionally contain the `$schema` keyword to declare that the document is JSON schema. An example looks like this:

```
{
  "$id": "https://example.com/person.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string",
```

```
      "description": "The person's first name."
    },
    "lastName": {
      "type": "string",
      "description": "The person's last name."
    },
    "age": {
      "type": "integer",
      "minimum": 0
    }
  }
}
```

There are type-specific keywords like `minLength`, `maxLength`, `pattern`, `format` (string), `multipleOf`, `minimum`, `exclusiveMinimum` (number/integer), `properties` (where each key is the name of a property and value the schema to validate the property), `additionalProperties`, `requiredProperties`, `dependencies` (schema changes based on properties), or `patternProperties`. For arrays, list validation (all elements must match the schema) or tuple validation (each item may have a different schema) can be performed. Furthermore, there are enums, schema annotations (like `description`), constructs like `allOf` / `anyOf` to combine schemas, and `if/then/else` constructs. Reuse is supported by separate definitions that are referenced.

6.5 Data Formats

6.5.1 Protocol Buffers

Protocol Buffers (Protobuf) are a method of serializing structured data. It consists of messages (maps), scalar types, structured types, and cardinalities. Source code for programming languages is automatically generated based on the schema, the data always adheres to the schema (i.e. homogeneous data).

6.5.2 Avro

Avro is a binary format for storing tree data. It is language neutral and interoperable. The schema (in JSON format) is part of the file and not known at compile time. It has atomic and structured types (arrays, maps, and records; where the values must be the same time for maps but not for records). The splittable data files consist of multiple blocks and the metadata (including the schema).

6.5.3 Dremel

Dremel is a large scale query system for analysis of read-only nested data that is capable of operating on in situ (e.g. on a DFS) data. It introduces a novel columnar storage format for nested data that enables it to efficiently assemble records from any given subset of columns. The basic idea is to store all possible fields (which are known in advance because it uses a fixed schema) individually, e.g. $a.b.c.d$. Besides the values, a repetition level and definition level is stored. The repetition level encodes the length of the common prefix of two consecutive paths (where only the fields that are tagged repeated and the initial path element identifying the record are counted) and the definition level the overall length of the path. These two additional information allow reconstruction of the record structure.

Tables itself are stored as horizontal partitions (tablets). Dremel also provides an SQL-like query language and a parallel query executor based on a serving tree architecture. It serves as the foundation of BigQuery and can be significantly faster than an equivalent MapReduce job.

6.5.4 Parquet

Parquet is a Column Storage. It uses an approach similar to Dremel for the mapping of trees to tables.

Chapter 7

Query Languages

Data independence allows to change the physical layer but keep the logical layer (the query language). Data independence for heterogeneous, denormalized data is achieved with languages such as JSONiq, SQL++, GraphQL, UnQL, ... Generally, the languages should be declarative (what instead of how) and functional (only consisting of expressions that take instances of the model and return instances of the model, for set-based languages with collections as instances). The language ecosystem for XML/JSON looks like this:

	XML	**JSON**
Navigation	XPath	JSONPath JSONSelect
Transform	XSLT	JSONT
Query	XQuery 1.0/3.0	XQuery 3.1 JSON Query JSONiq
Update, Scripting	XQuery Update Facility & Scripting	JSONiq

7.1 JSONiq

In the JSONiq Data Model (JDM), there are only sequences of items (an item is the same as the sequence with only that item). Sequences are flat and the items are atomic values, objects, arrays (which itself can contain items, nesting is therefore performed in this way), or functions.

An expression takes sequences of items and returns sequences of items. `json-doc("file-name.json")` returns the content of the JSON file. Navi-

gation is done with a dot, `[]` unboxes arrays (and `[[i]]` accesses the *i*-th element with 1-based indexing). If we have the following document:

```
{
   "countries":[
      {
         "name":"Switzerland",
         "code":"CH"
      },
      {
         "name":"France",
         "code":"F"
      },
      {
         "name":"Germany",
         "code":"D"
      },
      {
         "name":"Italy",
         "code":"I"
      },
      {
         "name":"Austria",
         "code":"A"
      }
   ]
}
```

`json-doc("myfile.json")` returns an object (a sequence of 1 item), `json-doc("myfile.json").countries` returns an array (sequence of 1 item), `json-doc("myfile.json").countries[]` returns a sequence of 5 objects, and `json-doc("myfile.json").countries[].name` returns the names of the 5 countries. `$$` allows referencing the context item, `json-doc("myfile.json").countries[][$$.code = "CH"]` e.g. returns the Switzerland object. JSONiq can also read JSON Lines files (with one object per line) using `json-file("filename.json")`.

Strings, numbers, booleans, objects, and arrays are constructed in the same way as in JSON (for numbers, the type is integer if no dot is provided, decimal with a dot and double if the number is written in scientific notation). Non-native JSON types from XML schema are written as `long("12345")`, `date("2013-05-01Z")`, `hexBinary("0CD7")`, etc... Sequences are constructed with a comma, e.g. `[2,3], true, "foo"` or with special syntax like `1 to 100` (sequence with numbers from 1 to 100).

Basic arithmetic operations are addition (`1 + 1`), subtraction (`1 - 1`), multiplication (`1 * 1`), division (`42.3 div 7`), integer division(`42 idiv 7`), and modulo (`42 mod 9`). Addition with the empty sequence produces the empty

sequence, addition with other types such as string or other cardinalities produces an error. Strings can be concatenated with `"foo" || "bar"` or `concat("foo", "bar")`. There are also operations like `string-join(seq, char)`, `substr(str, start, len)`, or `string-length(str)`. Value Comparison for equality/inequality is done with `eq`/`ne`, greater than/less than with `gt`&`ge`/`lt`&`le`. Comparison with the empty sequence always produces the empty sequence and there is also an error on type mismatch. Besides value comparison, there is also general comparison with `=`, `<`, `<=`, `>`, and `>=` which does existential quantification: At least one element in the left sequence must be in the relation with at least one element in the right sequence. A query like `json-file("countries.json").name = "Switzerland"` therefore returns true if the name field of any object is Switzerland. If one knows that there is only one item, value comparison should be used (throws an error if there are multiple items). The logical operations `and`, `or`, `not` are supported by JSONiq. When a non-boolean is passed to an `if()` expression, it is casted. The empty string, empty sequence, and 0 evaluate to false, any other string, any object, any other number to true. Comma has the lowest precedence and literals, constructors, variables, function calls the highest. Parentheses can always be used to override the precedence. The following shows an example construction in JSONiq:

```
{
  "attr" : string-length("foobar")
  "values" : [
    for $i in 1 to 10
    return string($i)
  ]
}
```

```
{
  "attr" : 6,
  "values" : [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
}
```

Multiple objects are merged using `{|` and `|}`, we have:

```
{| { "foo" : "bar" }, { "bar" : "foo" } |}
```

```
{ "foo" : "bar", "bar" : "foo" }
```

7.1.1 Data Flow

```
if(count(json-doc("file.xml").country) gt 1000)
then "Large file!"
else "Small file."
```

is a conditional expression (that returns a value unlike the if/else in imperative languages). Switch expressions are also supported:

```
switch($country.code)
case "CH" return
  ("gsw", "de", "fr", "it", "rm")
case "F" return "fr"
case "D" return "de"
case "I" return "it"
default return "en"
```

And try/catch expressions:

```
try {
  integer($country.code)
} catch * {
  A country code is not an integer!"
}
```

7.1.2 FLWOR Expressions

"for let where order by return" (FLWOR) expressions are the analogue to SELECT queries. `let` binds a sequence of items to a variable (but it is not an assignment), e.g.:

```
let $x := 2
return $x * $x
```

`for` clauses take items one after another and bind them to a variable, e.g.:

```
for $x in (1, 2, 4)
return $x * $x
```

`where` clauses filter items for which the predicate is not satisfied, e.g.:

```
for $x in 1 to 10
where $x - 2 gt 5
return $x * $x
```

It is also possible to return other types such as objects:

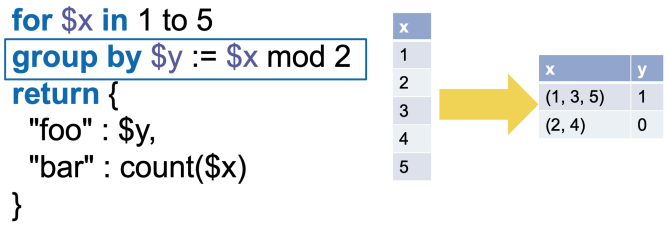
```
for $x in 1 to 10
return
  {
    "number": $x,
    "square": $x * $x
  }
```

`order by` allows sorting of the items:

```
for $x in json-file("countries.json")
order by $x.population
return $x.name
```

With `group by` clauses, it is possible to group by an expression. Note that unlike in SQL, no aggregation is needed when grouping, i.e. all items in the group can be accessed (using the variable that was used in the right hand side of the group by statement):

```
{
  "continents" : [
    for $x in json-file("countries.json")
    group by $continent := $x.continent
    return
    {
      "code": $continent,
      "countries" : [
        for $country in $x
        return $country.name
      ]
    }
  ]
}
```



A count clause adds a new variable that binds the position in the sequence, e.g.:

```
for $i in json-file("/players.json")
group by $nat := $i.nationality
order by $nat
count $c
where $c <= 5
return count($i)
```

7.1.3 Types

Simple types (e.g. `integer`) and complex types (e.g. `object`) are supported. Cardinality can be specified with `?`, `+`, and `*` (and no sign means

exactly one), e.g. `boolean?` or `item*`, the most generic type. Types do not have to be specified, but can be optionally:

```
let $x as integer := 2
return $x + $x
```

`treat as type` throws an error if the type is different, `cast as type` allows casting (also possible with the `type()` syntax), with `instance of type` it is possible to check the type of an expression, and with `expression castable as type` one can check if a value is castable to the type:

```
for $x as array in (
  [ 1 ],
  [ 2 ]
)
return $x + $x treat as integer
```

```
for $x as array in (
  [ 1 ],
  [ 2 ]
)
return $x + $x cast as double
```

```
for $x as array in (
  [ 1 ],
  [ 2 ]
)
return double($x + $x)
```

```
(3.14,"foo") instance of integer*
```

```
3.14 castable as double
```

Types can be useful in function declarations, but are also optional there:

```
declare function is-big-data(
  $threshold as integer,
  $objects as object*
) as boolean
{
  count($objects) gt $threshold
};
```

7.1.4 Rumble

Rumble is a JSONiq query processing engine built on top of Spark. In general, a query is parsed to an abstract syntax tree. This is then converted

to an expression tree (e.g. with `ForClause`, `ReturnClause`, ...) which is optimized and converted to an iterator tree (e.g. with `ForIterator`, `ReturnIterator`, ...) where each iterator returns a stream of tuples/items. For the execution of the iterator tree, we can use materialized execution (whole stream of one iterator is returned and stored in memory before continuing with the next iterator), streamed execution (one-at-a-time is returned and passed to the next iterator), and parallel execution (input is sharded to multiple hosts). Materialized execution requires a lot of memory, whereas streamed execution can be slow and is not possible depending on the iterator (e.g. for distinct). Rumble constantly switches between the three execution models.

When the internal structure is known statically, DataFrame-based execution is used. On the other hand, if no structure is known (e.g. for generic sequence of items), the engine uses RDD-based execution. Tuple streams in FLWOR expressions are represented as DataFrames with the variable names as columns. To allow grouping by heterogeneous types (not supported in spark), items in the grouping variables are shredded into three new columns (each homogeneous).

Chapter 8

Wide Column Stores

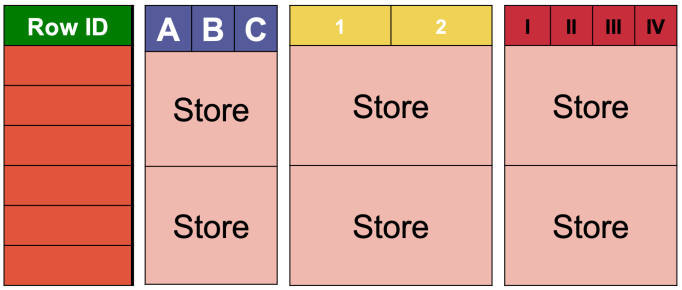
The problem with relational databases is that they are optimized for small scale/single machines and scaling out is very hard to set up and results in high maintenance costs. Wide column stores adhere to the relational/tabular model but run by design on a scalable cluster of commodity hardware. To avoid expensive joins, data is denormalized.

Column-oriented storage is different to wide column stores. In column-oriented storage, data is simply stored column-by-column (instead of row-by-row).

8.1 HBase

HBase is a wide column store that runs on top of HDFS. It is modeled after Google's Bigtable. Every row has a row ID and columns are grouped into column families. The families must be known in advance, but columns can be added on the fly. The primary operations are `get`, `put` (which may add new columns), `scan` and `delete`.

Physically, HBase shards the rows into regions (a range of IDs). A column family in combination with a region is stored together and called a store.



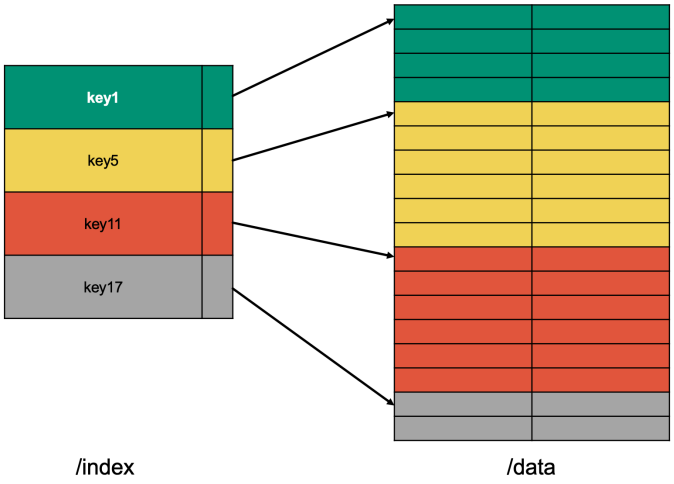
HBase also follows the Master-Slave architecture with a HMaster (can be the same machine as the NameNode) and Regionservers (usually the same machines as DataNodes).

The HMaster is responsible for DDL operations (creation/deletion of tables, ...), assigns regions to Regionservers (where assigning means assigning responsibility and not data, as the data is stored in HDFS), splits regions if necessary and handles failovers of Regionservers. There's no duplication of regions because the data is stored on HDFS (where it is duplicated).

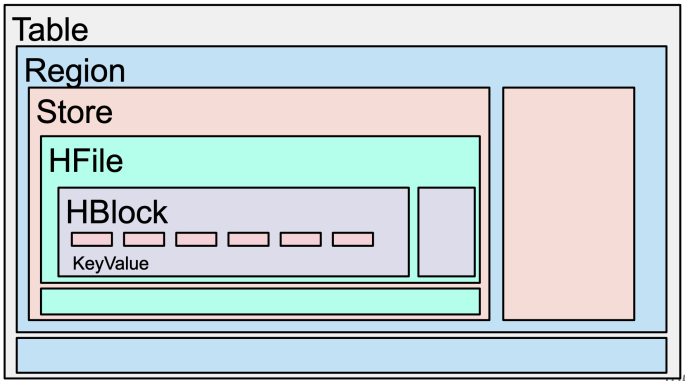
Regionservers are responsible for the serving of a region. One store is stored on HDFS in (one or more) HFiles. An HFile is a sorted list of key-value pairs where each cell (combination of row ID and column) + the version corresponds to one key (old versions are kept for versioning). One KeyValue pair inside an HFile contains the key and value length and the actual key and value. The key consists of:

- Row length
- Row (key)
- Column family length
- Column family
- Column qualifier (name of column; length is not needed because it can be calculated with all other lengths)
- Timestamp (for versioning)
- Key type (for marking as deleted)

HBase guarantees ACID on the row level and the versioning is therefore a total order (unlike Dynamo). A HFile is further divided into "HBlocks", 64kB blocks of KeyValues that are read at a time. Inside an HFile there is an index file (loaded in memory) which denotes the key that starts an HBlock and therefore allows faster lookup.



Overall, the levels of physical storage look like this:



Log-structured merge trees (LSM trees) allow efficient addition of new cells (as only appending is possible in HDFS, but key-value pairs need to be inserted in sorted order). There is an additional MemStore in memory which stores the recently inserted pairs. When memory is full, this MemStore is flushed to an HFile on disk (and the pairs are sorted before flushing/already kept in sorted order). When there are many HFiles, reading can take long (as the MemStore and all HFiles have to be accessed). Therefore, the HFiles are compacted regularly to one larger HFile. Generally, we can have multiple levels in LSM trees (memory, SSD, HDD, ...). As soon as there are two HFiles on one level, the file is merged and put onto the next level (if there was already one there, it is merged again, etc...)

For reliability, the system keeps a write-ahead log (HLog) with all changes and changes are first persisted to disk before they land in the MemStore.

Clients communicate with the HMaster for DDL operations and with Regionservers when they want to get the location of regions (for that, they communicate with the Regionserver hosting the meta table) and for queries (after they know the responsible Regionserver).

HBase has a LRU block cache and an additional bucket cache to accelerate reads to often accessed blocks. This is not helpful for batch processing and random access patterns.

Because of the HDFS placement policy, blocks are usually (exceptions are possible if node is full or after a failover) placed on the Regionserver that is responsible for them (first replica is stored on the same machine). This allows short-circuiting: The Regionserver does not need to contact the NameNode if it already knows that the blocks are stored locally. HFile compaction brings back locality (because a new HFile is created) if some blocks are stored on different DataNodes for the reasons mentioned before.

HBase uses bloom filters to avoid disk reads when it can guarantee that a key is not in an HFile. Bloom filters are a data structure used to speed up queries, useful in the case in which it is likely that the value we are looking for does not exist in the collection we are querying. Their main component is a bit array with all values initially set to 0. When a new element is inserted in the collection, its value is first run through a certain number of (fixed) hash functions, and the locations in the bit array corresponding to the outputs of these functions are set to 1. Therefore, if any of the locations is set to 0 when querying, the element is not present in the collection (no false negatives). However, there can be false positives if all locations are set to 1.

Chapter 9

Document Stores

One possibility to store trees is to make them fit into tables. For flat trees, this is easy. When the trees contain nested arrays, it is still doable if the structure is regular, but heterogeneity is hard to handle (NULL values everywhere, casting to string, etc...)

A solution are document stores that are optimized for storing collection of trees (XML/JSON documents). The documents are typically small, but there can be billions of them. Document stores typically provide projection, selection, and aggregation but no joins because they are too slow (need to be implemented by the user if they are needed). Validation is done after the data was populated.

9.1 MongoDB

The basic unit of data is a document (ordered set of keys with associated values), a collection contains many documents and a single instance can have multiple databases with its own collections. The system is type-/case-sensitive and no duplicate keys are allowed. Collections should not start with `system` (reserved) or contain the `$` character. A convention is to create subcollections that are separated by the `.` character. This is for organizational purposes only and imposes no technical relations. Every database has its own permissions and is stored in separate files on disk. There are some reserved database names (`admin`, `local`, `config`).

For efficiency reasons (faster parsing and storage efficiency), MongoDB encodes JSON in the binary format BSON. Besides the JSON data types, MongoDB supports 4-byte/8-byte integers, dates (stored as milliseconds without time zone information), regular expressions, object IDs (unique 12-byte IDs, per default consisting of a timestamp, machine ID, PID, and

increment), binary data, and code. It supports the CRUD (create, read, update, delete) operations over a custom DML. To read all documents in a collection `db.scientists.find({})` or `db.scientists.find()` is used. One can also provide a filter (selection):

```
db.scientists.find(
  { "Theory": "Relativity" }
)
```

Projecting is done via the second argument (where 1 indicates that a field should be in the output and 0 means it should not). With exception of the `_id` field (which is included by default), inclusion/exclusion cannot be combined in projection documents.

```
db.scientists.find(
  { "Theory": "Relativity" },
  { "Name": 1, "Last": 1 }
)
```

And in a selection is done by providing multiple key/value pairs, for or the `$or` operator is used:

```
db.scientists.find(
  {
    "Theory" : "Relativity",
    "Last" : "Einstein"
  }
)
```

```
db.scientists.find(
  {
    $or : [
      { "Last" : "Newton" },
      { "Last" : "Einstein" }
    ]
  }
)
```

Different comparison operators (`$lte`, `$lt`, `$ne`, ...) can be used:

```
db.scientists.find(
  { "Publications" : { $gte : 100 } }
)
```

Furthermore, the metaconditional `$not` can be applied on top of any other criteria. `null` is used to query missing fields and fields with a value of `null`, there is also the `$exists` operator to query only missing fields. For nested objects, the `.` is used. E.g. this query will return:

```
db.scientists.find(
  { "Name.First" : "Albert" }
)
```

```
{
  "Name" : {
    "First" : "Albert",
    "Last" : "Einstein"
  },
  "Theories": [ "Relativity" ]
}
{
  "Name" : {
    "First" : "Albert"
  },
  "Theories": [ "Unification" ]
}
```

However, the following query will only return the second document (checks for exact match with the object).

```
db.scientists.find({
  "Name" : { "First" : "Albert" }
})
```

If a value for an array field is provided, MongoDB will check if the array contains the field. There are also the `$in`/`$nin` operators to provide multiple values to check a field against, `$all` when multiple values must appear in an array, `$size` for the array size, `$slice` for array subsets, and `$elemMatch` to force comparison of all clauses with each array element (default behavior is that one element must match per clause). For instance, to find all restaurants where a rating has a grade C with a score greater than 50:

```
db.restaurants.find({
    grades : {$elemMatch : {grade : "C", score : {$gt : 50}}}
})
```

Simply combining the two conditions would also include restaurants where one rating has a grade C and another one a score greater than 50.

`$where` allows the execution of arbitrary JavaScript-code but is generally slow.

Besides that, MongoDB supports `.count()`, `.sort({ "Field": 1 })` (or -1), `.skip(n)` (which can be slow for large values of n), `.limit(n)`, `.distinct("field")` on collections. Note that the order does not matter, i.e. if e.g. skip and limit is interchanged, the result stays the same (the query

is only executed when `hasNext` is called on the cursor). It also supports aggregation and pipelines similar to Spark/MapReduce:

```
db.scientists.aggregate(
  { $match : { "Century" : 20 },
  { $group : { "Year" : "$year", "Count" : { "$sum" : 1 } } },
  { $sort : { "Count" : -1 } },
  { $limit : 5 }
)
```

Projection in these pipelines can be done with `$project` (with different pipeline expressions like `$add`, `$sub`, `$year`, `$dayOfMonth`, `$substr`, `$concat`, `$cmp`, ...) different operators (`$sum`, `$avg`, `$min`, `$max`, `$addToSet`, ...) are supported for grouping, and `$unwind` is used to turn array fields into separate documents. MapReduce can also work on top of MongoDB and is integrated into the system:

```
mr = db.runCommand({"mapreduce" : "foo", "map" : map, "reduce" : reduce})
```

`db.collection.insertOne({ ... })` (or `batchInsert`) is used for inserts. If a batch update fails, the documents up to that documents will be inserted. `db.collection.deleteMany({ conditions })` is used for deletions and for updates:

```
db.scientists.updateMany(
  { "Name" : "Einstein" },
  { $set : { "Century" : "20" } }
)
```

Updates are atomic, the first parameter is the query document and the second a modifier document. The modifier document can fully replace the existing document or contain special atomic update modifiers such as `$inc`, `$set` (field will be added if not present), `$push` (adds elements to arrays/creates them if not present), `$addToSet`, `$pop` (remove from start/end), or `$pull` (remove specific elements) The positional operator allows to update on the matched array element:

```
db.blog.update({"comments.author" : "John"},
  {"$set" : {"comments.$.author" : "Jim"}})
```

If the third parameter is set to true, upserts are performed. `findAndModify` allows atomic get/set operations. When documents grow bigger because of updates, they are moved to another part of the collection (this can lead to documents appearing multiple times when iterating over a cursor, which is solved by `.snapshot()`). With write concerns, it is possible to control if writes should be acknowledged before continuing.

The granularity of atomicity in MongoDB is one document (no two writes to a document in parallel).

9.1.1 Architecture

Documents are sharded based on field values (e.g. range A-E in one shard, F-P in another, ...). A replica set is a copy of one shard among multiple hosts with one primary host and multiple secondary hosts (that usually do not answer queries). In a normal setup, writes go through the master of the replica set and as soon as the write is concerned by a configurable number of secondaries, it is acknowledged.

MongoDB provides different type of indices, among others hash indices (for point queries) and range indices using B+-trees. Limitations of hash indices are that they do not support range queries, the hash function is not perfect in real life and the space requirements for collision avoidance are high. Indices can be sparse, meaning they do not include every document as an entry. An index is created like this:

```
1 db.scientists.createIndex({
2   "Century" : "hash"
3 })
```

Where "hash" can be replaced with 1/-1 (ascending/descending range index). Every row has a special field `_id` that is always indexed using a unique index.

If fields without an index are queried, scanning/filtering in memory is needed. If queries contain indexed and unindexed fields, the indexed fields can be used for pre-filtering and post-filtering in memory only needs to be done on a smaller set. Compound indices can always be used to query prefixes of the fields, if we have an index on `A.B.C`, it does therefore not make sense to create an additional one on `A` or `A.B`. Some queries can be answered by only considering the index (covered indices). Indexing whole subdocuments will only help queries that are querying the whole subdocument, but it is also possible to index individual fields of subdocuments. Indexing array fields indexes each element of the array (without the position information). MongoDB's query optimizer will select a subset of likely indices and run the query once with each plan, in parallel. The first plan to return 100 results is used and the other plans' executions are halted. The plan is then cached with certain conditions (index creation, inserts, age) for reevaluation.

The system adds dynamic padding to documents and preallocates data files to trade extra space usage for consistent performance. Whenever

possible, the server offloads processing and logic to the client side.

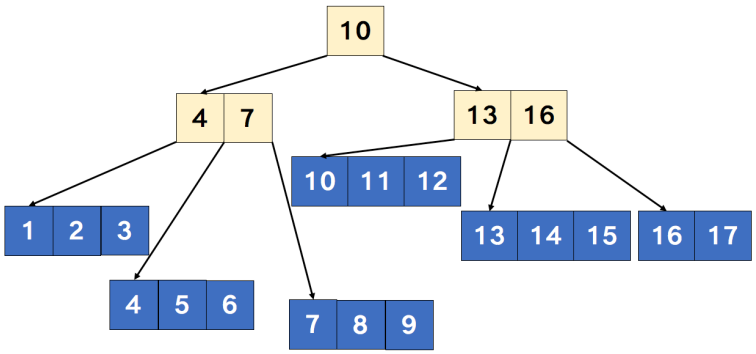
9.2 B+-trees

A $(d+1)-(2d+1)$ B+-tree is a tree where the number of children is between $d+1$ and $2d+1$ for any node (except the root node, it can only have two children) and all leaf nodes are at the same depth. A node with $d+1$ children has d keys, therefore the number of keys (per node) is between d and $2d$.

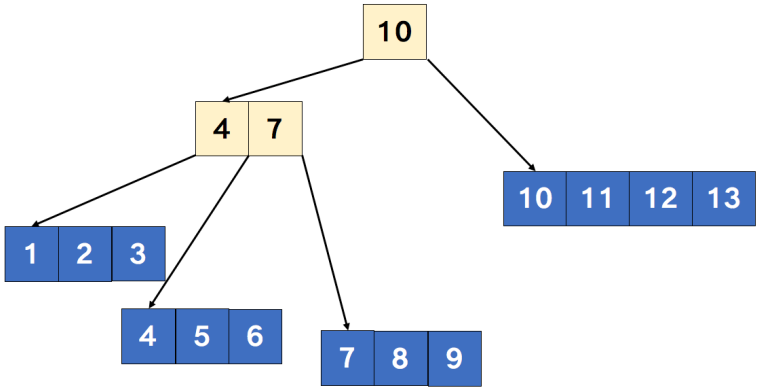
In a B+-tree, the values are only at the leaves, whereas in a B-tree, they are also stored at intermediate nodes.

An advantage of B+-trees is that the tree without the leaf nodes can be kept in memory, whereas the values are stored on disk. Because disks favor sequential accesses, this increases the read performance.

Example 9.1 The following tree is a 3-5 B+ tree:



The following is NOT, because not all leaf nodes are at the same depth:



Example 9.2 How many values can a 4-7 B+ tree of depth 3 contain at most?

For the first 3 levels, the nodes on each level can have at most 7 children. The leaf nodes can consist of 6 keys at most, we therefore have: $7^3 * 6 = 2058$

How many values can a 4-7 B+ tree of depth 3 contain at least?

The root node can only have two children, the nodes on the second / third level have at least 4 children and the leaf nodes have to consist of at least 3 keys, therefore: $2 * 4 * 4 * 3 = 96$.

Chapter 10

Graph Databases

Relational databases have expensive joins and are not that efficient at relationships. This is solved by graph databases/triple stores. The principle (for databases using native graph storage/native graph processing) is index-free adjacency: Data is directly linked in memory using pointers. Similar to other NoSQL approaches, iterative evolution of the data model/domain is possible. Because many domains are naturally modelled as graphs, there is no disconnection between the conceptual world and the way data is physically laid out (as with relational databases and ER schemas). Because many queries are local, graph databases often exhibit good scaling behaviour as the runtime of local queries is independent of the overall database size.

Logically, graphs consist of nodes (records) and edges and are either directed or undirected. In labeled property graphs (the data model that is used by Neo4j), nodes/edges can additionally have properties (key/value pairs) and labels (classification/types). We differentiate between property graphs that use LPGs (such as Neo4j) and triple stores, e.g. using RDF.

10.1 RDF

The model of the Resource Description Framework is the triple-based graph. One triple has a subject (node), object (node), and a property (property of the edge, e.g. "is located in"). In RDF, we have IRIs (Internationalized Resource Identifiers), literals (XML schema types), and the blank node that can appear in different places:

	Subject	Property	Object
IRI			
Literal			
Blank node			

10.1.1 Syntax

There are many RDF formats, among others RDF/XML, Turtle, and JSON-LD. A RDF/XML document looks like this:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:geo="http://www.example.com/geography#">
 <rdf:Description rdf:about="http://www.ethz.ch/#self">
  <geo:isLocatedIn
   rdf:resource="http://www.example.com/Switzerland"/>
  <geo:population>20000</geo:population>
 </rdf:Description>
</rdf:RDF>
```

In those documents, QNames are interpreted as concatenation, e.g. `geo:isLocatedIn` (the property) is interpreted as `http://www.example.com/geography#isLocatedIn` (an IRI). Because the `type` property is very common, it can be specified in the element name, which makes those documents equivalent:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:geo="http://www.example.com/geography#">
 <rdf:Description rdf:about="http://www.ethz.ch/#self">
  <geo:isLocatedIn
   rdf:resource="http://www.example.com/Switzerland"/>
  <rdf:type
   rdf:resource="http://www.example.com/geography#school"/>
  <geo:population>8000000</geo:population>
 </rdf:Description>
</rdf:RDF>
```

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:geo="http://www.example.com/geography#">
 <geo:school rdf:about="http://www.ethz.ch/#self">
  <geo:isLocatedIn
   rdf:resource="http://www.example.com/Switzerland"/>
  <geo:population>20000</geo:population>
 </geo:school>
</rdf:RDF>
```

The same graph in JSON-LD looks like this:

```
{
  "@context": {
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "geo": "http://www.example.com/geography#"
  },
  "@id" : "http://www.ethz.ch/#self",
  "rdf:type": "geo:school",
  "geo:isLocatedIn": "http://www.example.com/Switzerland",
  "geo:population" : 8000000
}
```

And in Turtle (which is a simple list of subject property object):

```
@prefix geo: <http://www.example.com/geography#> .
@prefix countries: <http://www.example.com/> .
@prefix eth: <http://www.ethz.ch/#> .
eth:self geo:isLocated countries:Switzerland .
eth:self geo:population 8000000 .
```

Semicolons can be used to avoid repetition of subjects, commas to avoid repetition of subjects and properties:

```
@prefix geo: <http://www.example.com/geography#> .
@prefix countries: <http://www.example.com/> .
@prefix eth: <http://www.ethz.ch/#> .
eth:self geo:isLocated countries:Switzerland,
                       countries:Europe ;
         geo:population 8000000 .
```

10.1.2 SPARQL

SPARQL is the RDF query language. A basic query looks like this:

```
PREFIX geo: <http://www.example.com/geography#>
PREFIX countries: <http://www.example.com/>
SELECT ?s
WHERE {
  ?s geo:isLocatedIn countries:Switzerland .
  ?s :deliversDiplom :bachelor .
}
```

It is also possible to sort, limit, and reference the property via a variable:

```
SELECT ?s ?name
WHERE {
  ?s geo:isLocatedIn countries:Switzerland .
  ?s :deliversDiplom :bachelor .
  ?s :hasName ?name .
}
```

```
7 ORDER BY ?name
8 LIMIT 10
```

10.1.3 Semantics

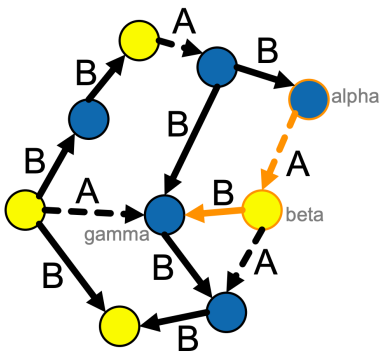
RDF itself has no semantics, but there is RDF schema which introduces a set of classes (`rdfs:Resource`, `rdfs:Class`, `rdfs:Property`, `rdfs:type`...) that allow to model statements such as "a resource is of type resource". Entailment regimes (such as OWL) on top of RDFS allow to derive true statements based on RDFS (e.g. deriving which nodes are animals after describing the "class structure" (using the special classes) in RDF) and can be used to formally represent metadata (semantic web).

10.2 Neo4j

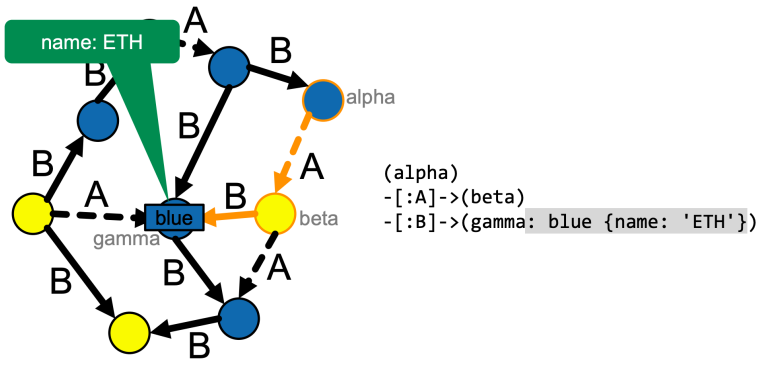
Neo4j is a graph database using the labeled property graph model.

10.2.1 Cypher

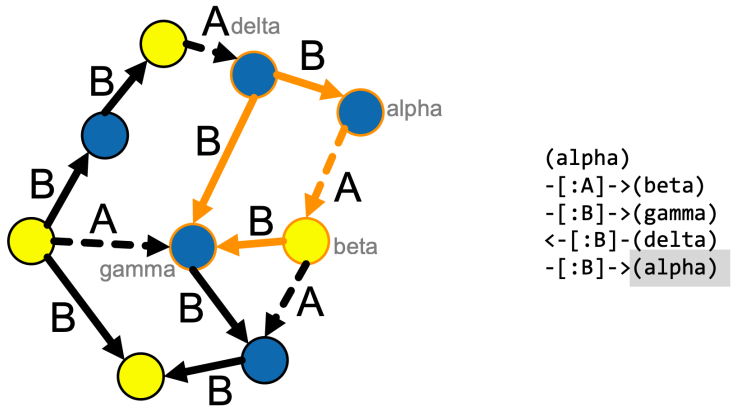
Cypher is Neo4j's query language. It is based on the idea of "query by example", the user provides structures to search for in the graph:



It is also possible to specify labels and/or properties:



Furthermore, variables can be reused:



With `(alpha)-[*1..4]->(beta)`, it is possible to query for a variable length path.

These pattern are parts of other clauses, for instance `MATCH` to return specific nodes:

```
1 MATCH (alpha {name: 'Einstein' })-[:A]->(beta)-[:B]->(gamma)
2 RETURN gamma
```

Cypher also has a `WHERE` clause, which makes the previous example equivalent to:

```
1 MATCH (alpha)-[:A]->(beta)-[:B]->(gamma)
2 WHERE alpha.name = 'Einstein'
3 RETURN gamma
```

Clauses like `STARTS WITH` and regular expressions (with =~) are also supported:

```
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE p.name STARTS WITH "Tom"
RETURN m.title
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE p.name =~ "Tom.*"
RETURN m.title as mtitle ORDER BY mtitle DESC
```

Neo4j does not eliminate duplicates by default, this is done using `RETURN DISTINCT`. There is also an expression for the shortest path:

```
MATCH
    (p1:Person { name: "Keanu Reeves" }),
    (p2:Person { name:"Tom Cruise" }),
    p = shortestPath((p1)-[*..15]-(p2))
RETURN p, length(p)
```

`WITH` allows query parts to be chained together (similar to SQL subqueries), e.g.:

```
MATCH (p1:Person),(p2:Person), p = shortestPath((p1)-[*]-(p2))
WHERE p1.name = "Kevin Bacon" AND p1 <> p2
WITH length(p) AS length
RETURN max(length) AS maxLength
```

With `CREATE`, the desired structure can be created (or using `MERGE`, which ensures the pattern exists, i.e. only creates it if it does not already):

```
CREATE (einstein:Scientist {name: 'Einstein', first: 'Albert' }),
       (eth:University {name: 'ETH Zurich' }),
       (einstein)-[:VISITED]->(eth)
```

10.2.2 Architecture

Up to recently, there was no sharding because this would make graph traversal much slower (memory pointers point directly to objects in memory which is not possible with sharding) and sharding without application-level intervention is generally NP hard (requires a minimum point cut). However, neo4j fabric introduced sharding (according to custom properties/attributes) where some nodes are duplicated in different shards. Generally, there is a master/slave architecture and slaves store replicas of the graph (data loss prevention and better performance because the master and slaves can response to queries). Writes to the slave are propagated by the slaves to the master (and they block until this is done). On disk, there are different stores (with fixed size records) for nodes, relationships, labels, and properties. Labels and properties are stored as linked list. To

allow efficient traversal, the source/target, s-previous/s-next (pointers to the previous/next relation in the source), and t-previous/t-next (pointers to the previous/next relation in the target) is stored for every relationship, i.e. relationships are stored as doubly linked list and each node contains a pointer to these lists.

Besides Cypher, Neo4j provides lower level APIs for JVM languages (the low-level Core API or the declarative Traversal Framework). Furthermore, transactions with ACID properties are supported. Similar to RDBMS, they are implemented using a write-ahead log and locking.

Chapter 11

Data Warehousing

There are two query processing paradigms: OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing). In OLTP, we have consistent and reliable record-keeping and typically lots of transactions on small portions of data. Data is normalized to avoid anomalies. OLAP is for data-based decision support. We query large portions of the data with possibly many joins and have a few long heavy queries. Example applications are web or sales analytics. In OLTP, we have a lot of writes whereas there are many reads in OLAP. The main differences are listed here: A

	OLTP	**OLAP**
Source	Original (operational)	Derived (consolidated)
Purpose	Business tasks	Decision support
Interface	Snapshot	Multidimensional views
Writing	Short and fast, by the 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

data ware house is a subject-oriented (on a specific topic, e.g. sales), integrated (one database/location with all the data), time-variant (time is often important, e.g. statistics per year), nonvolatile (only loading, no updates) collection of data in support of management's decision-making process.

We have denormalized, materialized (data is copied from source systems

because their heterogeneity would result in a lot of work otherwise) views. The basic principle is extract, transform, load (ETL). Data is extracted from different source systems (incremental updates with triggers, gateways, or log extraction), transformed (derivation of values, value transformation, cleaning, and tasks such as filter, split, merge, join) and loaded (where integrity constraints are checked, data is partitioned and sorted and indices can be built).

11.1 Data Cubes

The data model of data warehouses are multidimensional hypercubes. A fact is a single point in the cube (usually a single value, but there are also solutions with multiple measures per point) and there are many dimensions, e.g. the person, time, currency, etc... In the fact table, there is one column per dimension (the compound key of the table) and another column (or multiple if we have multiple measures) for the fact.

Basic operations are slicing and dicing. Slice is the act of picking a rectangular subset of a cube by choosing a single value (anywhere in the hierarchy, can therefore also correspond to multiple values in a lower hierarchy level) for one of its dimensions, creating a new cube with one fewer dimension. The dice operation produces a subcube by allowing the analyst to pick specific values of multiple dimensions. A roll-up involves summarizing the data along a dimension and the contrary drill-down refers to the exploration of more detailed data, starting from summary data at a higher level in the hierarchy.

11.1.1 Implementation

There is ROLAP (relational OLAP on top of relational databases) and MOLAP (memory OLAP, native implementation). In ROLAP, there is a fact table with either a star schema (which means more information about each dimension is stored in one satellite table per dimension) or snow-flake schema (with normalized, i.e. multiple, satellite tables per dimension). SQL is used to query those cubes. If information from the satellite tables is needed, they are simply joined. Slicing is implemented as selection (with where clauses). Aggregating (rolling up) is implemented with group by, for the complete rollup we need to union different subqueries. Because this is cumbersome, there is `GROUP BY GROUPING SETS` and `GROUP BY ROLLUP`, i.e. the following queries are equivalent:

```
1 (SELECT t.Year, p.Brand, SUM(s.Quantity)
2 FROM Sales s, Time t, Product p
```

```
WHERE s.Date = t.Date
  AND s.Product = p.Name
GROUP BY t.Year, p.Brand)
UNION
(SELECT t.Year, null, SUM(s.Quantity)
FROM Sales s, Time t
WHERE s.Date = t.Date
GROUP BY t.Year)
UNION
(SELECT null, null, SUM(s.Quantity)
FROM Sales s)
```

```
SELECT t.Year, p.Brand, SUM(s.Quantity)
FROM Sales s, Time t, Product p
WHERE s.Date = t.Date
  AND s.Product = p.Name
GROUP BY GROUPING SETS (
  (t.Year, p.Brand),
  (t.Year),
  ()
)
```

```
SELECT t.Year, p.Brand, SUM(s.Quantity)
FROM Sales s, Time t, Product p
WHERE s.Date = t.Date
  AND s.Product = p.Name
GROUP BY ROLLUP (t.Year, p.Brand)
```

There is also `GROUP BY CUBE`, which produces for two columns (A, B), the grouping sets $(A, B), (A), (B), ()$ (wheres the rollup only produces $(A, B), (A), ()$), which makes the following queries equivalent:

```
(SELECT t.Year, p.Brand, SUM(s.Quantity)
FROM Sales s, Time t, Product p
WHERE s.Date = t.Date
  AND s.Product = p.Name
GROUP BY t.Year, p.Brand)
UNION
(SELECT t.Year, null, SUM(s.Quantity)
FROM Sales s, Time t
WHERE s.Date = t.Date
GROUP BY t.Year)
UNION
(SELECT null, p.Brand, SUM(s.Quantity)
FROM Sales s, Product p
WHERE s.Product = p.Name
GROUP BY p.Brand)
UNION
(SELECT null, null, SUM(s.Quantity)
FROM Sales s)
```

```sql
SELECT t.Year, p.Brand, SUM(s.Quantity)
FROM Sales s, Time t, Product p
WHERE s.Date = t.Date
  AND s.Product = p.Name
GROUP BY GROUPING SETS (
  (t.Year, p.Brand),
  (t.Year),
  (p.Brand),
  ()
)
```

```sql
SELECT t.Year, p.Brand, SUM(s.Quantity)
FROM Sales s, Time t, Product p
WHERE s.Date = t.Date
AND s.Product = p.Name
GROUP BY CUBE (t.Year, p.Brand)
```

A common visualization for roll-ups are cross tabulations:

Year	Brand		
	Apple	Samsung	
2017	5	6	11
2016	9	5	14
2015	5	4	9
Total	20	14	**34**

In MOLAP, special languages such as MDX (MultiDimensional eXpressions) are used for querying. Dimension values are organized in hierarchies and there is special syntax for slicing/dicing.

11.1.2 Syntax

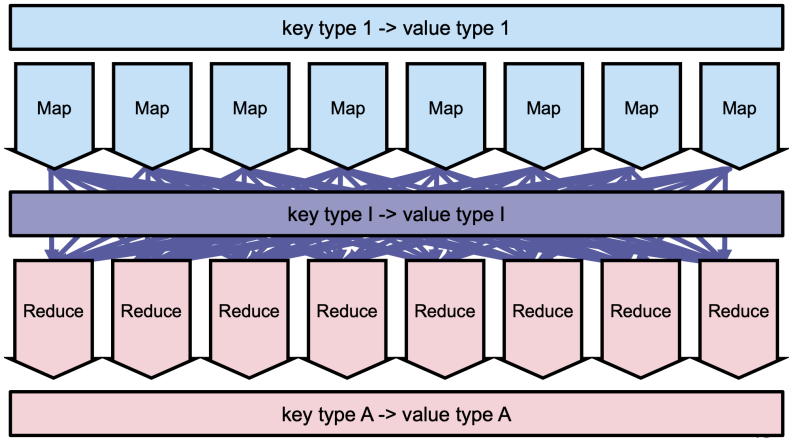
XBRL (eXtensible Business Reporting Language) is an XML-based syntax for cubes. It has a discoverable taxonomy set (schema for the cubes) and is based on technologies such as XML Schema, XML Names, etc...

Chapter 12

Distributed Computations

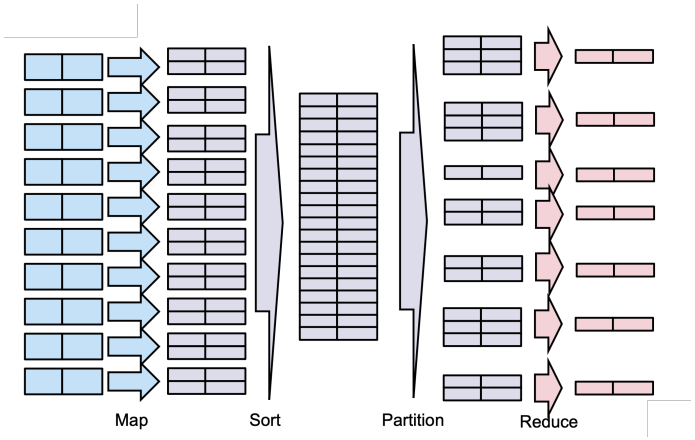
12.1 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets (not for a small amount of data as the overhead would be too large) with a parallel, distributed algorithm on a cluster. Input data is mapped, shuffled, and finally reduced to produce the output data. Input, intermediate, and output data are key-value pairs with potentially different types (although the type of the intermediate and output data is often identical in practice).



First, the key/value pairs are split and the splits are assigned to nodes. They apply the mapping function and produce intermediate pairs (where one input pair can produce multiple or none intermediate pairs). The intermediate pairs are partitioned/grouped by key and all pairs with the same key are assigned to a reducer. The reducer applies the reduce func-

tion and produces one or more (although more is uncommon) output key/-value pairs. Overall, the steps therefore look like this:



Physically, MapReduce supports many storage layers including HDFS. Hadoop MapReduce introduces JobTrackers (master nodes, usually alongside NameNodes) and TaskTrackers (slave nodes containing one or multiple slots for the execution of tasks, alongside DataNodes). Whole jobs are sent to the JobTracker that splits them into multiple tasks (map/reduce tasks). The JobTracker assigns splits to the TaskTracker. Ideally (for performance reasons/to allow short-circuiting), one split should correspond to an HDFS block (note that this is often not completely possible as HDFS blocks are fixed-size but splits need to be at key/value boundaries, therefore nodes usually need to do a small remote read for a remainder of the last key/-value pair). There is one map task per split and to bring the query to the data, the DataNodes/TaskTrackers that host the blocks are usually assigned the map task (although this colocation is not always possible). In the mapping phase, the pairs are kept in memory, but can be spilled to disk (sorted, to allow easier access afterwards) if necessary. The reduce tasks need to wait for all map tasks to complete (because they require all intermediate pairs with the same key). In the shuffle phase, they connect to all TaskTrackers that execute map tasks over HTTP and get the key/-value pairs with their keys from them. After reducing, output is written to disk with one file per reducer.

MapReduce supports many input/output formats. For instance from tables (RDBMS, HBase, etc...) where each row is mapped to a key/value pair (usually primary key as key and row values as values). When reading text, there is a certain impedance mismatch. Text is usually converted to key/value pairs by using the line offset as key and the line content as value (with the NLine format, each key contains n lines).

Combiner are one optimization technique. They are an additional function that is provided by the programmer with the intent to reduce the number of intermediate results. For instance, values with the same key can already be summed up (pre-aggregated) instead of producing multiple key/value pairs with a 1. They are not guaranteed to be used by the framework, usually combining is done when spilling to disk. Often, the combine function is identical to the reduce function. For that, the following must hold:

1. Key/Value types must be identical for the reduce input and output.
2. The reduce function must be commutative and associative.

Therefore, the average function could not be used as a combiner (not associative).

MapReduce provides a Java and Streaming (for other programming languages, e.g. Python) API. Map functions, Reduce functions and whole Jobs can be easily expressed with the API:

```
1  import org.apache.hadoop.mapreduce.Mapper;
2  public class MyOwnMapper extends Mapper<K1, V1, K2, V2> {
3          public void map(K1 key, V1 value, Context context) throws
               IOException, InterruptedException {
4                  ...
5                  K2 new_key = ...
6                  V2 new_value = ...
7                  context.write(new_key, new_value);
8                  ...
9          }
10 }
```

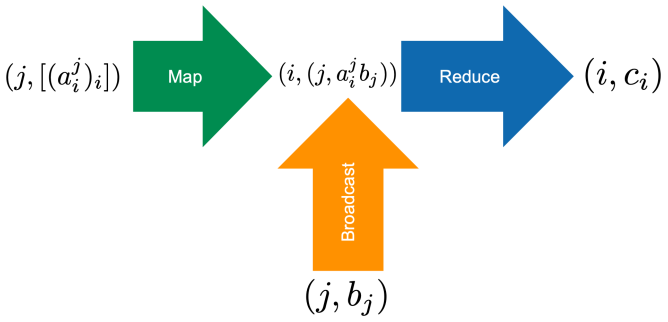
```
1  import org.apache.hadoop.mapreduce.Reducer;
2  public class MyOwnReducer extends Reducer<K2, V2, K3, V3> {
3          public void reduce(K2 key, Iterable<V2> values, Context context)
               throws IOException, InterruptedException {
4                  ...
5                  K3 new_key = ...
6                  V3 new_value = ...
7                  context.write(new_key, new_value);
8                  ...
9          }
10 }
```

```
1  import org.apache.hadoop.mapreduce.Job;
2  public class MyMapReduceJob {
3          public static void main(String[] args) throws Exception {
4                  Configuration conf = new Configuration();
5                  Job job = Job.getInstance(conf, "word count");
```

```
6                   job.setMapperClass(MyOwnMapper.class);
7                   job.setCombinerClass(MyOwnReducer.class);
8                   job.setReducerClass(MyOwnReducer.class);
9                   FileInputFormat.addInputPath(job, ...);
10                  FileOutputFormat.setOutputPath(job, ...);
11                  System.exit(job.waitForCompletion(true) ? 0 : 1);
12          }
13  }
```

12.1.1 Example Applications

- Counting Words: The map function produces pairs where the word is the key and 1 the value. The reducers simply reduce by summation.
- Filtering Lines (Selection): The map function outputs the line if the condition is met and nothing otherwise. The reduce function is the identity function.
- Projection: The map function parses (e.g.) the JSON and output the desired columns, the reduce function is the identity function.
- Matrix Multiplication $Ab = c$: The inputs are the columns (with the column number as keys), the map function produces a pair for every row $(i, a_i^j b_j)$ (optionally with the column in the value, but this is not really needed) where the vector elements b_j are broadcasted to the TaskTracker. The reduce function then produces c_i by summation:



  For repeated matrix multiplication (e.g. in the page rank algorithm), MapReduce is not ideal as it requires a job for every multiplication (in contrast to Spark).

12.2 Resource Management

A problem in MapReduce v1 is the JobTracker. There are multiple issues with it in the design:

1. **Scalability**: Only one node, supports up to 4,000 nodes and 40,000 tasks.
2. **Bottleneck**: A single JobTracker for the whole cluster can become the bottleneck.
3. **Jack of all Trades**: The JobTracker is responsible for Resource Management, Scheduling, Monitoring, the job lifecycle and fault-tolerance.
4. **Static Allocation**: Each slot of the TaskTrackers is either allocated for mapping or reducing statically.
5. **Not fungible**: The reduce slots are idle until the map slots are complete, reduce/map is therefore not interchangeable.

MapReduce v2 uses YARN for resource management, which optimizes utilization (but not necessarily performance/execution time of individual jobs).

12.2.1 YARN

YARN (Yet Another Resource Negotiator) introduces a ResourceManager for scheduling and application management and multiple Application Masters for monitoring (it is therefore a master/slave architecture as well). The master is called ResourceManager, the slaves (one per node) NodeManager. The NodeManager divide their resources (CPU, memory, disk, network) dynamically into containers (similar to slots in Hadoop) based on the application requirements. Clients submit jobs to the ResourceManager, which allocates an Application Master for the job (a container that is dedicated to it). The Application Master then dynamically requests (from the ResourceManager) containers for the execution of the jobs. If the job is e.g. a MapReduce job, it first only requests containers to map and then to reduce. The Application Master is responsible for the monitoring (and handling of failures) of the containers, the ResourceManager only needs to monitor the Application Master.

The ResourceManager provides a client service (application start and end, queue information, statistics) and an admin service (node list refresh, queue configuration). NodeManagers regularly send a heartbeat and their resource usage to the ResourceManager. Application Master contact the ResourceManager for registration (via the application master service), heartbeats and container requests. Requests are answered with tokens that grant access to resources (application masters are not trusted which is solved by the token mechanism). The ResourceManager is a pure scheduler, it does not monitor tasks and does not restart upon failure.

As application masters are per application and application-specific, YARN can be used for many different applications with the framework-specific application masters (MapReduce, Spark, MPI, Graph processing, ...).

12.2.2 Scheduling

There are four main types of scheduler:

1. **FIFO Scheduler**: Simple first-in-first-out scheduling policy. Usually, a job gets the whole cluster but as an improvement, multiple jobs at the same time may be allowed if it is permitted by the resources (but the implementation of this is hard in practice).

2. **Capacity Scheduler**: There are multiple queues with a certain priority (can be hierarchical, i.e. the queues can be further divided and different scheduling policies can be used on different hierarchy levels). Resources are assigned based on the queue priority. Queue elasticity can be configured that allows the usage of resources that belong to another queue when they are not needed up to a certain degree.

3. **Fair Scheduler**: There is the steady fair share (corresponding to the priority in the capacity scheduler) and the instantaneous fair share. For the calculation of the instantaneous fair share, only active queues are considered. The difference to the (instantaneous or usual) fair share is calculated and resources are allocated to the queue with the largest (absolute or relative) difference. Preemption of tasks in other queues is possible under certain conditions.

4. **Dominant Resource Fairness Scheduler**: When there are multiple resources, the highest percentage per application (the percentage of the dominant resource) is taken as the percentage for the calculations (and the percentages are normalized). Otherwise, the scheduler behaves identical to the fair scheduler. The DRF algorithm looks like this:

Algorithm 1 DRF pseudo-code

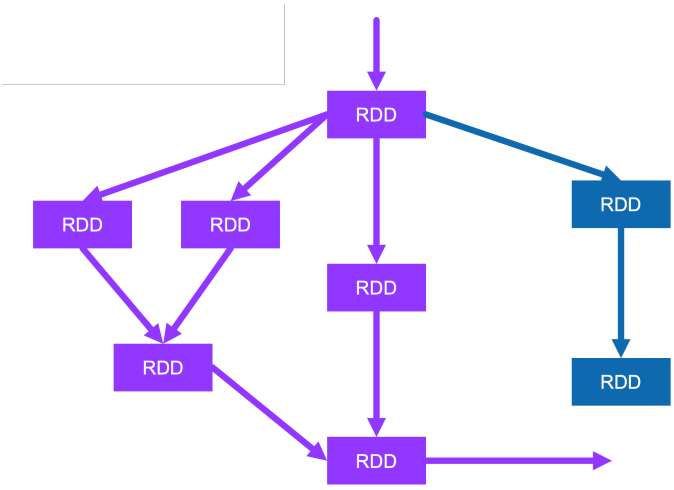
$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
$s_i \ (i = 1..n)$ ▷ user i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ (i = 1..n)$ ▷ resources given to user i, initially 0

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $C + D_i \leq R$ **then**
    $C = C + D_i$ ▷ update consumed vector
    $U_i = U_i + D_i$ ▷ update i's allocation vector
    $s_i = \max_{j=1}^{m} \{u_{i,j}/r_j\}$
else
    return ▷ the cluster is full
end if

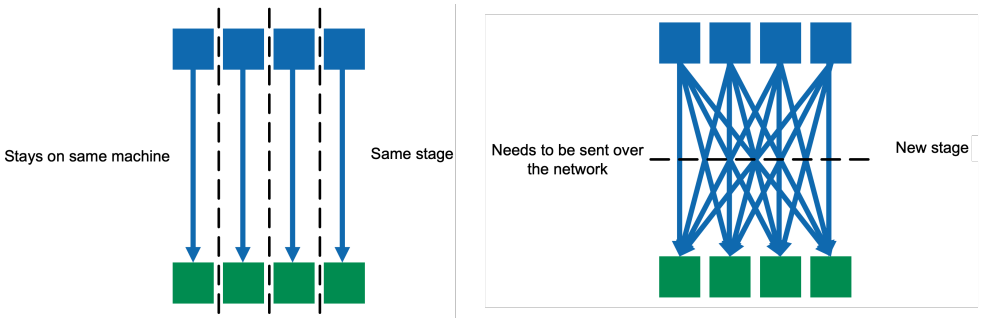
12.3 Spark

Spark enables full-DAG query processing (in contrast to MapReduce which enforces a specific topology). The architectural foundation is the resilient distributed dataset (RDD), a partitioned collection of data items (can be anything, e.g. numbers, key/value pairs, etc...) which acts as a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. There are three operations in the RDD lifecycle: Creation (from a filesystem, HDFS, any other Hadoop input source such as HBase, etc...), transformations (coarse-grained; i.e. the same operation applied to many data items), and actions (operations that return a value to the application or export data to a storage system).

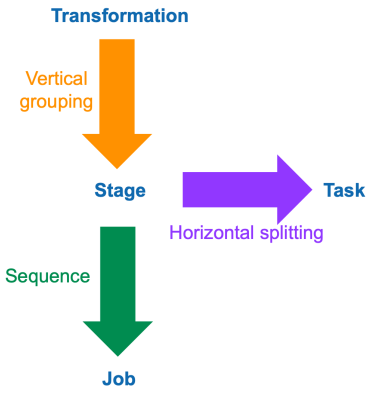
The lineage graph is the logical execution plan. RDDs (and individual partitions) can always be recreated based on it (although checkpointing is possible to prevent costly recalculations). Spark uses lazy evaluation, an action triggers an evaluation (where only the needed RDDs/transformations are evaluated) by creating a job and not all RDDs need to be materialized. Slow nodes (stragglers) can be mitigated by running backup copies of slow tasks (as in MapReduce).



A task is the assignment to map a partition of an RDD to the output RDD. The default is one task per HDFS block. A sequence of parallelizable transformations is called a stage. The transformations are parallelizable if the dependencies are narrow, i.e. if the item in the output RDD depends on one item in the input RDD (e.g. map, flatMap, filter). On the other hand, transformations with wide dependencies start a new stage:



For performance reasons (to minimize data movement if it is not needed because there are only narrow dependencies), stages instead of transformations are split into tasks, i.e. we have the following hierarchy:



Executors (corresponding to YARN containers) execute the tasks, an executor can have multiple cores that execute tasks (using multiple threads) with shared memory. When using `spark-submit`, the number of executors is specified by `-num-executors`, the number of cores/amount of memory with `-executor-cores` and `-executor-memory`. The scheduler assigns tasks based on data locality.

Intermediate RDDs can be persisted (in memory by default, can spill to disk if needed) to prevent recalculation if multiple actions need the results. Furthermore, wide dependencies can often be avoided by pre-partitioning the data (e.g. putting key/value pairs with the same key on the same machine). Internally, RDDs are represented through an interface that exposes the following information: A set of partitions, a set of dependencies on parent RDDs, a function for computing the dataset based on its parents, and metadata (about the partitioning schema and data placement).

To use Spark, developers write a driver program that connects to a cluster of workers. The code on the driver also tracks the RDD's lineage.

12.3.1 Transformations

There are many transformations, the most important unary ones are:

- Filter (e.g. `rdd.filter(lambda x: x % 2 == 0)`): Filtering by a predicate (i.e. relational algebra selection)
- Map (e.g. `rdd.map(lambda x: (x, 1))`): Mapping a single value to a single value
- flatMap (e.g. `rdd.flatMap(lambda x: [(x, x), (x, x)])`): Mapping a single value to potentially multiple values (i.e. the same as the map in MapReduce) and flattening the results.

- Distinct `.distinct()` in PySpark): Duplicate Elimination
- Sample (e.g. `rdd.sample(False, 0.1)`), where the first parameter determines if sampling is performed with replacement and the second the fraction/sample size): Randomly keep/drop elements

The most important binary transformations are:

- Union (e.g. `rdd1.union(rdd2)`)
- Intersection (e.g. `rdd1.intersection(rdd2)`)
- Subtraction (e.g. `rdd1.subtract(rdd2)`)
- Cartesian Product (e.g. `rdd1.cartesian(rdd2)`)

Spark is aware of key/value pairs, there are special transformations for them:

- Keys/Values (get the keys or values; `.keys()` and `.values()` in PySpark)
- Reduce by key (e.g. `rdd.reduceByKey(lambda x, y: x + y).collect()`)
- Group by key (e.g. `rdd.groupByKey().mapValues(list)`, without the `mapValues`, a `ResultIterable` per key is returned)
- Sort by key (`.sortByKey()` in PySpark)
- Map values
- Join (inner join where all the values are kept; `x.join(y)` with `x=[(a, 1), (b, 4)]` and `y=[(a, 2), (a, 3)]` results in `[(a, (1, 2)), (a, (1, 3))]`)
- Subtract by key (Returns each (key, value) pair in the first RDD that has no matching key in the second RDD; `.subtractByKey()` in PySpark).

12.3.2 Actions

The most important actions are:

- Collect (`.collect()`)
- Count (`.count()`)
- Count by value / Count by key (`.countByKey()` / `.countByValue()`)
- Take (`.take(n)` for the first n elements, `.takeOrdered(n)` if the RDD should be first sorted in ascending order)

- Top (`.top(n)` for the first n elements of the sorted RDD in descending order, i.e. the counterpart to `.takeOrdered(n)`)
- takeSample (sample n elements with `.takeSample(False, n)` where the first parameter specifies if sampling should be performed with replacement)
- Reduce (e.g. `rdd.reduce(lambda x, y:  x + y)`)

There are also pair actions like count by key or lookup.

12.3.3 Data Frames

DataFrames are a collection of structured records/rows (with the same columns), they therefore correspond to a table. They use columnar storage because it is more efficient if only some columns are queried and the type is identical per column which allows efficient storage/retrieval. The memory footprint of DataFrames is generally much smaller (for instance with JSON, the column name is always repeated whereas it is stored only once for DataFrames). The schema is automatically inferred, but data should be homogeneous (otherwise, the inferred type is often impractical, i.e. string if integers and lists are mixed for the same column). As with other type systems, there are atomic and structured (array, struct, map) types. DataFrames can be built from RDDs or other data formats (JSON, Parquet, CSV, Text, JDBC, Avro, user-defined data sources, ...) and they are lazy (but the analysis of logical plans is eagerly, i.e. errors can be reported before execution).

DataFrames allow queries with Spark SQL. The DataFrame API of Spark SQL lets users mix procedural and relational code, using data frame DSL. Queries are automatically translated to a physical query (and the query optimizer catalyst, which includes an extensible framework for transforming trees and rule-based/cost-based optimizations, optimizes the logical and physical plan). Because the engine understands the structure of the data (in contrast to RDDs) and the semantics of user functions (in contrast to traditional Spark with arbitrary code), more optimizations are possible. Users can create user-defined types based on catalyst's built-in types and catalyst tries to push predicates down into the data sources whenever possible. Hot data can be materialized/cached in memory using columnar storage. Nestedness can be handled with special commands such as `EXPLODE()` (for arrays) or `object.first` (for objects). The SQL interface can also be accessed through JDBC/ODBC and UDFs (user-defined functions) can be integrated in this way with e.g. BI tools.

For instance, the following three ways to calculate the monthly count per crime type and order them by month number (ascending) and number of crimes per type (descending) are identical, expressed as RDD transformations, in the data frame DSL and with Spark SQL:

```
dataset = sc.textFile("crimes.csv")
dataset.map(lambda s: ((s.split(',')[2][:2], s.split(',')[5]), 1)).
    reduceByKey(lambda x, y: x + y).sortBy(lambda x: (x[0][0], -x[1]))

from pyspark.sql.functions import desc
df = spark.read.option("header", True).csv("crimes.csv")
df.groupby([df.Date.substr(0, 2), "Primary Type"]).agg({"*": "count"}).
    orderBy("substring(Date, 0, 2)", desc("count(1)"))

df.createOrReplaceTempView("crime")
spark.sql("SELECT substring(date, 0, 2) AS month, `Primary Type`, COUNT(*)
    FROM crime GROUP BY month, `Primary Type` ORDER BY month, count(1) DESC
    ")
```

Chapter 13

Performance

The main sources of bottleneck and possible elimination strategies are:

- **Memory**: Search for classes that are instantiated billions of times and check if there are unused fields, more memory-efficient data structures, and potential redundancies. Furthermore, inheritance/RTTI (runtime type information) can be a overhead.
- **CPU**: Search for gigantic loops (e.g. functions passed to transformations) and check if methods calls are overused, methods overridden (switch is faster), instanceof/casting is often used, or exceptions are used in normal setups. Furthermore, interfaces can be faster than class hierarchies.
- **Disk I/O**: Can be alleviated by efficient formats and compression.
- **Network**: Shuffling should be kept to a minimum. Furthermore, pre-filtering, pre-projection, and pre-aggregate (e.g. combiner) can be used to reduce the exchanged data volume.

The right chunk size can also improve performance (too large chunks can lead to too much load imbalance, too small to a large overhead).

According to Amdahl's law (constant problem size), if we have a parallelizable part of p with a speedup of s, the overall speedup is:

$$S = \frac{1}{1 - p + \frac{p}{s}}$$

According to Gustafson's law (constant computing power), the speedup is:

$$S = 1 - p + sp$$

13.1 Tail Latencies

In large scales, some nodes will have much longer runtimes than the average runtime because resources are shared on a machine and globally (network switches, distributed file system). Furthermore, background threads can cause hiccups, there are power limits, queues, energy management, or garbage collection which can slow down a node. A solution is to hedge requests: Tasks are duplicated and the first that is done wins. A smarter solution are deferred hedge requests: A duplicate task is only created when e.g. the 95% percentile (of the runtime distribution) is reached.