Big Data for Engineers (Spring 2020)

Summary

Author:
Yannick Merkli

53 Pages

Contents

1 Introduction

1.1 History

Databases have always existed in some way as a way to preserve information. It started with speaking and singing, went on with stone engraving and printing. Even before computers, tables were the primary format to represent data. Things changed drastically with the introduction of computers. Database management systems (DBMS) started with file systems (1960s), then we entered the relational era (1970s) and finally progressed into the NoSQL era (2000s) with the upcoming of big data.

1.2 Big Data

Big Data is a buzzword that goes across many disciplines (distributed systems, high-performance computing, data management, algorithms, statistics, machine learning, etc.) and that involves a lot of proprietary technology (AWS, Google Cloud, Microsoft Azure, etc.) which is simply a result of the need of companies to have efficient data systems.
The big in big data: **three Vs**

- Volume: Nowadays we have lots of sources of data (web, sensors, proprietary, scientific). Storage has become so cheap that we often just store data because we can. Further, data carries value; data is worth more than the sum of its parts (data totality: one must have complete data).
- Variety: We have different **data shapes**: tables, trees, graphs, cubes, text.
- Velocity: Data is generated automatically, Data is a realtime byproduct of human activity

Prefixes (International System of Units)

kilo (k)	1,000 (3 zeros)	kibi (ki)	1,024 (2^{10})
Mega (M)	1,000,000 (6 zeros)	Mebi (Mi)	1,048,576 (2^{20})
Giga (G)	1,000,000,000 (9 zeros)	Gibi (Gi)	1,073,741,824 (2^{30})
Tera (T)	1,000,000,000,000 (12 zeros)	Tebi (Ti)	1,099,511,627,776 (2^{40})
Peta (P)	1,000,000,000,000,000 (15 zeros)	Pebi (Pi)	1,125,899,906,842,624 (2^{50})
Exa (E)	1,000,000,000,000,000,000 (18 zeros)	Exbi (Ei)	1,152,921,504,606,846,976 (2^{60})
Zetta (Z)	1,000,000,000,000,000,000,000 (21 zeros)	Zebi (Zi)	1,180,591,620,717,411,303,424 (2^{70})
Yotta (Y)	1,000,000,000,000,000,000,000,000 (24 zeros)	Yobi (Yi)	1,208,925,819,614,629,174,706,176 (2^{80})

There are three paramount factors to big data:

- Capacity: "How much data can we store?"
- Throughput: "How fast can we transmit data?"
- Latency: "When do I start receiving data?"

Capacity has improved incredibly much over the past 60 years (1956: huge HDD had 5MB storage, 2020: there are palm sized 20TB HDDs). Capacity has increased by a factor $200 * 10^9$ (per unit of volume). However, throughput and latency have only improved by a factor $10'000$ and 200 respectively. This discrepancy creates problems: the throughput no longer scales to the amount of data and the latency no longer scales to the throughput. Solution:

- Capacity-throughput discrepancy: parallelization
- Throughput-latency discrepancy: batch processing

What is big data?
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.

2 Lessons learnt from the past

Data Independence:
An underlying principle that has been valid for a long time is the principle of **data independence** (developed by Edgar Codd): Data Independence is defined as a property of DBMS that helps you to change the Database schema at one level of a database system without requiring to change the schema at the next higher level. Data independence helps you to keep data separated from all programs that make use of it.
This means we could e.g. change the physical storage (e.g. iPad instead of HDD) *without* changing the logical data model.

Data shapes: Text, trees, tables, graphs, cubes.
Overall architecture
The overall architecture of a DBMS consists of:

- Language (e.g. SQL)
- Model (e.g. Tables (old), graphs, trees, cubes (new)
- Compute (e.g. single CPU (old), hadoop cluster (new))
- Storage (e.g. HDD (old), distributed storage (new))

A data model essentially describes *what data looks like* and *what you can do with the data.*

2.1 Basic concepts

- Table (Collection): A set of rows (= business object, item, entity, document, record) and each row has attributes (= columns)
- Attribute (column, field, property): A certain attribute of a row
- Primary key (row ID, name): a unique identifier of a row

2.2 Relational Algebra

We can look at tables as relations or as partial functions, mapping property to value
($f \in \mathbb{S} \nrightarrow \mathbb{V}$), e.g. $city \mapsto Zurich$.

2.2.1 Relations (the math, for database scientists)

A relation R is made of:

- A set of attributes: $Attributes_R \subseteq \mathbb{S}$
- An extension (set of tuples):

$$Extension_R \subseteq \mathbb{S} \nrightarrow \mathbb{V} \quad s.t. \quad \forall t \in Extension_R, support(t) = Attributes_R$$

Tabular integrity: Holds if all rows have the same attributes and have a value for the attributes.
Atomic integrity (1st normal form): No tables in tables.
Domain integrity: All attribute values are of the specified type (e.g. an attribute *Name* of type string can't be an integer).

In SQL, tabular integrity, atomic integrity and domain integrity all hold. In NoSQL however, none of these three properties hold.

2.3 The relational model of data

- Data Models: A data model is a notation for describing the structure of the data in a database, along with the constraints on that data. The data model also normally provides a notation for describing operations on that data: queries and data modifications.
- Relational Model: Relations axe tables representing information. Columns are headed by attributes; each attribute has an associated domain, or data type. Rows are called tuples, and a tuple has one component for each attribute of the relation.
- Schemas: A relation name, together with the attributes of that relation and their types, form the relation schema. A collection of relation schemas forms a database schema. Particular data for a relation or collection of relations is called an instance of that relation schema or database schema.
- Keys: An important type of constraint on relations is the assertion that an attribute or set of attributes forms a key for the relation. No two tuples of a relation can agree on all attributes of the key, although they can agree on some of the key attributes.
- Semistructured Data Model: In this model, data is organized in a tree or graph structure. XML is an important example of a semistructured data model.
- SQL: The language SQL is the principal query language for relational database systems. The current standard is called SQL-99. Commercial systems generally vary from this standard but adhere to much of it.
- Data Definition: SQL has statements to declare elements of a database schema. The CREATE TABLE statement allows us to declare the schema for stored relations (called tables), specifying the attributes, their types, default values, and keys.

- Altering Schemas: We can change parts of the database schema with an ALTER statement. These changes include adding and removing attributes from relation schemas and changing the default value associated with an attribute. We may also use a DROP statement to completely eliminate relations or other schema elements.
- Relational Algebra: This algebra underlies most query languages for the relational model. Its principal operators are union, intersection, difference, selection, projection, Cartesian product, natural join, theta-join, and renaming.
- Selection and Projection: The selection operator produces a result consisting of all tuples of the argument relation that satisfy the selection condition. Projection removes undesired columns from the argument relation to produce the result.
- Joins: We join two relations by comparing tuples, one from each relation. In a natural join, we splice together those pairs of tuples that agree on all attributes common to the two relations. In a theta-join, pairs of tuples are concatenated if they meet a selection condition associated with the theta-join.
- Grouping: Aggregate multiple rows by some attribute.
- Sorting: Sort rows by some attribute.
- Constraints in Relational Algebra: Many common kinds of constraints can be expressed as the containment of one relational algebra expression in another, or as the equality of a relational algebra expression to the empty set.

2.4 Relational queries

The following table shows the operators used in Relational Algebra.

Selection	σ	**Set Minus**	$-$	**right Semi-Join**	$\rtimes$
Projection	π	**Relational Division**	$\div$	**left Outer Join**	⟕
Cartesian Product	$\times$	**Union**	$\cup$	**right outer Join**	⟖
Join	$\bowtie$	**Intersection**	$\cap$		
Rename	ρ	**left Semi-Join**	$\ltimes$		

Projection: The projection operator is used to produce from a relation R a new relation that has only some of R's columns. The value of expression $\prod_{A_1,...,A_n}(R)$ is a relation that only consists of columns for the attributes $A_1, ..., A_n$.
Selection: The selection operator applied to R produces a new relation with a subset of R's tuples, namely those who meet some condition C. This operation is denoted by $\sigma_C(R)$.
Cartesian product: The Cartesian product of relations R, S, denoted $R \times S$, simply concatenates every possible combination of tuples $r \in R, s \in S$. If R, S have attributes in common: rename them. In practice rarely used without join operators.
Natural Join: The natural join of relations L, R, denoted $L \bowtie R$, pairs only those tuples from L and R that agree in whatever attributes they share commonly. Natural Join is associative! Other variants:

- Left outer join: natural join & unmatched tuples from L
- Right outer join: natural join & unmatched tuples from R
- Full outer join: natural join & unmatched tuples from both L and R
- Left semi join: tuples from L that match with some tuple in R
- Right semi join: tuples from R that match with some tuple in L

Theta-Join: A theta join $\bowtie_\theta$ allows to join tuples from two relations R, S based on an arbitrary condition θ rather than solely based on attribute agreement. We get this new relation by:

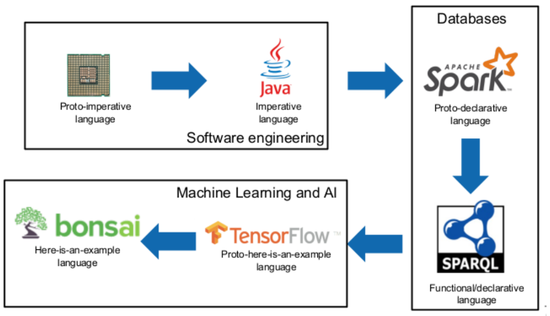
1. Take the Cartesian product $R \times S$
2. select those tuples satisfying condition θ

Union, Intersection, Set Minus: requires: both relations have the same schema $\rightarrow$ then consider set of tuples, do corresponding set operations. Note that $R \cap S = R - (R - S)$
Rename:

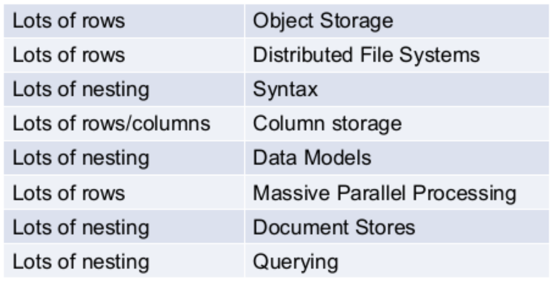
- to change the name of the relation R to S, we write $\rho_S(R)$.
- to rename attributes of R, we use the operator $\rho_{(A_1,..,A_n)}(R)$ where the attributes in the result relation S are called $A_1, ..., A_n$, respectively.

2.5 Terminology

Data: Data Manipulation Language (DML) (Query, insert, remove rows) Schema: Data Definition Language (DDL) (Create or table/schema, drop it)


Proto-imperative language
Java
Imperative language
Software engineering
Databases
Apache Spark
Proto-declarative language
SPARQL
Functional/declarative language
Machine Learning and AI
bonsai
Here-is-an-example language
TensorFlow
Proto-here-is-an-example language

(a) Language landscape



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

(b) Data Scale-Up

2.6 Transactions

The good old times of databases: ACID (Atomicity, Consistency, Isolation, Durability). This is a feature of traditional database since it is only achievable on small DBMS. ACID ensures that transactions are correct (e.g. bank transaction: either I receive money and bank updates balance or nothing at all).

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

With big data, we drop the ACID principle.

2.7 Performance

Optimize for read vs. write intensive:

- OnLine Transaction Processing (OLTP): Write-intensive
- OnLine Analytical Processing (OLAP): Read-intensive

There is no such thing as ”one size fits all” - data shape matters.

2.8 Data scale-up

Data can have lots of rows, lots of columns and lots of nesting. For the rest of this lecture, we are concerned with exactly this: scaling up!

2.9 SQL

SQL stems from Sequel (Structured Englisch QUEry Language) and is a declarative language.

3 SQL

SQL is a family of standards, namely it includes a data definition language for schemas, a data manipulation language for updates and a query language for reads. Note that SQL is case-insensitive.

3.1 DDL: Data Definition Language

The following code snippet shows the most important data definitions in SQL:

```sql
character (n), char (n) -- String of length n
varchar(n) -- String of variable length but at most n characters
numeric (p,s) -- decimal value with precision p and scale s
integer -- numeric values
blob, raw -- large binaries
date -- date value
clob -- large string tables
string1 || string2 -- concatenation operator for strings

CREATE TABLE Professor -- create a new table
    (PersNr     integer not null,
     Name       varchar (30) not null,
     Status     character (2) default "AP");
DROP TABLE Professor -- delete table
ALTER TABLE Professor add column(age integer); -- adds new age column to table

CREATE INDEX myIndex on Professor(name, age); -- Index for performance tuning
DROP INDEX myIndex -- delete index
```

3.2 DML: Data Manipulation Language

A real database cannot be manually populated in a tuple-by-tuple manner $\implies$ **ETL: Extract, Transform, Load** is used to extract data from some file, which is then transformed to the data types supported by the database and then loaded into to database as a *bulk* operation. Manual updates work as follows:

```sql
insert into Student (Legi, Name)        -- standard insertion syntax
    values (16-940-165, 'Jens Eirik Saethre');

-- can also use a nested query to determine the tuples to be inserted.
insert into attends (Legi, 'Databases')
    select Legi
    from Student
    where semester > 2;

delete Student -- standard deletion syntax
where Semester > 13;

update Student -- update all tuples in Student relation table
    set Semester = Semester + 1;

-- there are sequence types for automatic increment (e.g. useful for unique IDs)
create sequence PersNr_seq increment by 1 start with 1;
insert into Professor(PersNr, Name)
    values (PersNr_seq.nextval, "Roscoe");
```

3.3 Simple Queries in SQL

Projection & Selection

The simplest query form in SQL selects tuples from a single relation with some selection criteria and selects only the intersting columns. This query thus combines both *Selection* and *Projection*:

```sql
select   PersNr, Name                -- use * to select all columns
from         Professor
where    Status = 'FP';
```

One can leave out the `where` clause to get the entire table, one can sort the table by appending the line `order by Status desc, Name asc` or select only distinct entries with the keyword `select distinct`.

One can also rename attributes by writing `select PersNr as ProfNr` and use expressions in the `SELECT` (note that SQL is *case-insensitive*) statement to modify the concerned column. The following example of a query in a movie database combines all of the aforementioned special cases:

```
SELECT DISTINCT  release AS releaseDate, length*0.16667 AS runtime
FROM             Movies
WHERE            genre = 'Thriller' AND (rating >= 9.0 OR rating <= 2.0);
```

It returns the release date and the runtime in hours of the best and worst-rated movies in the Thriller genre stored in the database. Note that if there are multiple movies that satisfy the condition with the same (ReleaseDate, runtime) pair, only one tuple will appear in the returned relational table.
$\Longrightarrow$ like this, one can also add constant columns to the output by appending to the `SELECT` clause the following: `'hrs.' AS inHours`. Now the table will have three columns, the last one being the string 'hrs.' for all tuples.

Note: `<>` is SQL syntax for "not equal to" and `=` for "equal to".

SQL relates to relational algebra in the sense that a query `SELECT L FROM R WHERE C` is equivalent to the relational algebra expression $\prod_L(\sigma_C(R))$

String Comparison

- comparisons of `varchar` and `char` only compares the actual string and not the padding.
- comparison with an operator like $>$ compare the string's lexicographical order.

Pattern Matching

We can use the pattern matching operator `s LIKE p` to compare a string s to a pattern p:

- ordinary characters in p match only themselves in s
- a _ symbol in p matches one arbitrary character in s
- a % symbol in p matches an arbitrary sequence of characters in s, also with length 0.

Dates and Times

- **Dates** are represented in the format `DATE '1996-12-06'`
- **Times** are represented in the format `TIME '15:23:05.4'`.
- One can combine the two to get a new type `TIMESTAMP '1996-12-06 15:23:05.4'`

`NULL` value in SQL

The *null value* in SQL can either mean that the value is *unknown, inapplicable* or *withheld*.

- arithmetic operations that include `NULL` evaluate to `NULL`
- comparisons that include `NULL` evaluate to the truth value `UNKNOWN`
- the `group by` operator returns a group for `NULL`

Note: `NULL` is **not** a constant, we cannot use `NULL` explicitly as an operand.

`UNKNOWN` value in SQL

Assume `TRUE = 1, FALSE = 0, UNKNOWN = 1/2`. Then we have the following rules for logical operators:

- **AND:** minimum of the two values
- **OR:** maximum of the two values
- **NOT:** $1 - v$ where v was the previous value.

When selecting tuples, only those whose truth value to the query is `TRUE` are picked for the resulting relation. $\Longrightarrow$ the following query does not necessarily return all movies:

```sql
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120
```

If a movie's length is in fact UNKNOWN, than that tuple will not be returned.

Sorting of Tuples

Can use the ORDER BY <list of attributes> clause to order the returned tuples. Ordering is applied before the SELECT part, so it is possible to order on attributes that do not appear in the final output. Impose an order by using ASC, DESC, ascending being default.

3.4 Queries on multiple Relations

Set Operations

We use the keywords UNION, INTERSECT, and EXCEPT for $\cup, \cap, -$, respectively. The syntax is generally as follows: (Query1) INTERSECT (Query2);.

Products

To get the Cartesian product of two relations, simply provide both as arguments to the FROM clause, e.g. SELECT * FROM Movies, Stars.

Joins

Use the product learned above and define in the WHERE clause which attributes should match, e.g. the following query returns the names of the producer of 'Star Wars', when the movie name and the producer name are stored in different relations:

```sql
SELECT  name
FROM    Movies, MovieProducers
WHERE   title = 'Star Wars' AND producerNr = certificateNr;
```

If attributes of different relations have the same name, we reach non-ambiguity by using the *dot-notation* on the attributes: RelationName.AttributeName. This is generally good practice, even when there is no ambiguity (yet).

Tuple Variables

If a query involves two ore more tuples from the same relation, we need an **alias** to refer to them individually. This is achieved by the following syntax:

```sql
SELECT      Star1.name, Star2.name
FROM        MovieStar Star1, MovieStar Star2
WHERE       Star1.address = Star2.address AND Star1.name < Star2.name;
```

Note: 1.) even though Star1 appears in the statement before it is defined, this is okay since the FROM part is evaluated earlier. 2.) the string comparison is necessary to avoid pairing people with themselves. Further, comparison by <> would produce each pair twice.

A problem with the semantics may arise when we think that we can use simple SQL queries for set operations like $R \cap (S \cup T)$. Assume R, S, T are all unary relations with attribute A. One would think that the following query works perfectly fine:

```sql
SELECT R.A  FROM R, S, T      WHERE R.A = S.A OR R.A = T.A;
```

However, if T were to be empty, one would expect the result to be $R \cap S$, but the query returns nothing. This is due to the semantics of multi-relational queries and how they are implemented.

3.5 Full-Relation Operations

Some operations act on relations as a whole an not just on single tuples. We have already looked at the DISTINCT keyword to eliminate duplicates in query results. Further, set operations eliminate duplicates by default. If this is not desired $\Longrightarrow$ use the ALL keyword, e.g. for relations R, S:

```sql
R INTERSECT ALL S
```

returns the intersection of "bags"

Grouping and Aggregation

We often want to partition the tuples into groups on which we can then apply an aggregation operator, like one of the following:

- avg, max, min, count, sum

We can then do grouping by the clause GROUP BY which follows the WHERE clause. We can e.g. sum all the lengths of movies given studios have produced by the query:

```sql
SELECT studioName, SUM(length)
FROM Movies
GROUP BY studioName;
```

Note that NULL values are ignored in any aggregation (e.g. in avg it would make a difference, in count(*) it only counts non-NULL values). However, NULL is treated as an ordinary group.

HAVING clauses

This clause is used when we want to choose our groups based on some aggregate property of the group itself $\Longrightarrow$ the HAVING clause then follows the GROUP BY. Example:

```sql
SELECT  name, SUM(length)
FROM    Movies, MovieProducers
WHERE   producerNr = certificateNr
GROUP BY name
HAVING MIN(year) < 1930;
```

This query prints the names of producers with their total movie lengths which have at least produced one movie prior to 1930.

Note: one can aggregate any attribute appearing in the relations declared in the FROM clause, but one can only use attributes that are in the GROUP BY list in an unaggregated way.

3.6 Subqueries

A query can be part of another query itself $\Longrightarrow$ already saw example in set operations.

- Subqueries can return a single constant $\Longrightarrow$ use in WHERE clauses
- Subqueries can return relations used in WHERE clauses
- Subqueries can appear in FROM clauses followed by a tuple variable

Conditions involving Relations

We first assume unary = one-column relations for simplicity:

- EXISTS R is true $\Longleftrightarrow$ R is not empty
- s IN R is true $\Longleftrightarrow$ s is equal to at least 1 value in R. Can also use the NOT IN operator
- s > ALL R is true $\Longleftrightarrow$ s greater than every value in R.
- s > ANY R is true $\Longleftrightarrow$ s not the smallest value in R.

Note: if R returns zero rows, then any comparison with ALL clause returns true.

Conditions involving Tuples

A tuple in SQL is a parenthesized list of scalar values e.g. (name, address, networth) $\Longrightarrow$ if tuple t has same number of components as relation R, we can compare it by using e.g. t IN R or t <> ANY R, the latter asking if there exists any tuple in R that is other than t.

To find all producers of movies where Leonardo DiCaprio starred in, one can use the following query:

```sql
SELECT  name
FROM    MovieProducers
WHERE   certificateNR IN
    (SELECT producerNR
     FROM   Movies
     WHERE  (title, year) IN
        (SELECT movieTitle, movieYear
         FROM   StarsIn
         WHERE  starName = 'Leonardo DiCaprio'));
```

Queries like these should be analysed from inside out.

Correlated Subqueries

In a subquery, we can refer to e.g. an attribute from the relation of the `FROM` clause from the super-query. We can refer to it by using an *alias*. This subquery then needs to be executed every time for each tuple in the super-query. An example would be:

```sql
SELECT  title
FROM    Movies Old
WHERE   year < ANY
    (SELECT year
     FROM   MOVIES
     WHERE  title = Old.title);
```

This query finds all movie titles that appeared multiple times. The subquery needs to be executed for every tuple in Movies that we are looking at. *Scoping rules* work similar to other mainstream programming languages.

Subqueries in `FROM` clauses

Can use a parenthesized query instead of a stored relation $\implies$ use an *alias* for the result of the query to be able to refer to it.

Problem with Universal Quantification

SQL does not support relational division directly, that's why one has to play tricks, namely using the following:

$$\forall t.P(t) \equiv \neg\exists t.\neg P(t) \qquad \text{and} \qquad R \implies T \equiv \neg R \lor T$$

Syntactic Sugar

```sql
SELECT  *
FROM    STUDENT
WHERE   semester between 1 and 6; -- variant 1
WHERE   semester in (2,4,6);      -- variant 2

-- case statement can be implemented like this. note that only one clause executes,
   meaning no need for break
SELECT  legi, (CASE WHEN grade >= 5.0 THEN 'gut'
                    WHEN grade >= 4.0 THEN 'ausreichend'
                    else 'nicht bestanden' END)
FROM        Tests;
```

Joins in SQL

```sql
-- Cartesian Product
Movies CROSS JOIN StarsIn;

-- Theta Join (still has all the columns from both relations -> Preceed by a SELECT
   statement to reduce redundancy)
Movies JOIN StarsIn ON
    title = movieTitle AND year = movieYear; -- theta condition

```

```sql
-- Natural Join (joins on all attributes with the same name)
MovieStar NATURAL JOIN MovieProducer

-- OuterJoins (FULL, LEFT or RIGHT are interchangeable in this example)
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

SnapShot Semantics

Updates on relations are carried out in two steps, first one marks the tuples to be updated (also deleted) and in a second step, one actually does the update. This is to prevent inconsistencies.

3.7 Views

A view is a *logical relation* in a DBMS. It allows for easier access of data that is complicatedly stored and could only be accessed with both deep knowledge about the underlying way the data is stored and highly sophisticated queries.

```sql
CREATE VIEW <name of view> AS <sql query>;
```

Now one can use `<name of view>` just like a table. This allows for privacy and better usability resulting in simpler queries. Views get evaluated by automatic *query rewriting*. Example:

```sql
CREATE VIEW StudProf(Sname, Semester, Title, Pname) AS
    SELECT  s.Name, s.Semester, l.Title, p.Name
    FROM    Student s, attends a, Lecture l, Professor p
    WHERE   s.Legi = a.Legi AND a.Nr = l.Nr AND l.PersNr = p.PersNr;

-- now one can use this View in queries:
SELECT DISTINCT Semester
FROM    StudProf
WHERE   Pname = 'Alonso';
```

One can also use Views for "is-a" relationships by providing base tables and creating Views for frequently used concepts.

Updates

Generally, it is not possible to update a view, since the table it represents is not physically stored, and would lead to anomalies in the physical table, e.g. when a view combines only some columns of multiple different relations. What would a change to a view incur? Would all the relations that have some attributes in the view need to change? $\implies$ inconsistencies. Same happens with aggregations. This leads to the following theorem about views in SQL:

Theorem 3.1. A SQL View is updateable $\iff$ the view involves only one base relation, the view involves the key of that base relation and the view does **not** involve aggregates, group-by or duplicate elimination.

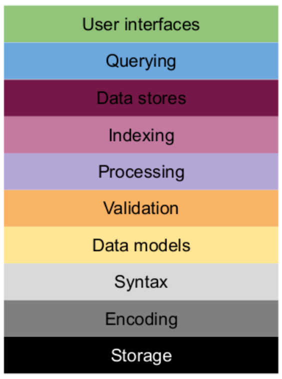


Figure 2: The data stack

4 Object Storage

Data needs to be stored somewhere. The common notion nowadays is to just put the data in the cloud - this section describes how to do this.
As we've seen, classic relational databases that fit on a single machine are still very important today - if the data fits in a relational database, you should choose a relational database. However, petabytes of data do not fit on a single machine. We need to somehow break the monolithic relational database down and rebuild while reusing the good parts of relational databases (e.g. SQL language).

What can we adapt from relational databases:

- Relational algebra: selection, projection, grouping, sorting, joining
- Language: SQL, declarative language, functional language, optimizations, query plans, indices
- What is a table made of: table, rows, columns, primary key

What we throw out of the window:

- Consistency constraints: Tabular integrity, Domain integrity, Atomic integrity (1^{st} normal form), Boyce-Codd normal form.
  - With NoSQL, we now newly have: Heterogeneous data, Nested data, Denormalized data
- Transactions - ACID: Atomicity, Consistency, Isolation, Durability
  - With NoSQL, we now newly have: Atomic Consistency, Availability, Partition tolerance, Eventual Consistency

4.1 The stack

- **Storage** - the actual data needs to be physically stored somehow: Local filesystem, NFS, GFS, HDFS, S3, Azure Blob Storage
- **Encoding** - we need to somehow represent and encode data: ASCII, ISO-8859-1, UTF-8, BSON
- **Syntax** - represent data as text: Text, CSV, XML, JSON, RDF/XML, Turtle, XBRL
- **Data models** - provide a level of abstraction over the syntax (only ever dealing directly with syntax would be very tedious): Tables (Relational model), Trees (XML Infoset, XDM), Graphs (RDF), Cubes (OLAP). All data upwards from the 'data model' level in the stack has the form of tables, trees, graphs, cubes,... **not** encoded data → level of abstraction over data.
- **Validation** - Check that the data is correct (e.g. check that date format is DD-MM-YYYY), data cleaning can be very expensive: XML Schema, JSON Schema, Relational schemas, XBRL taxonomies
- **Processing**: Two-phase processing (MapReduce), DAG-driven processing (Tez, Spark, Flink, Ray), Elastic computing (EC2)
- **Indexing** - make processing faster by building structures: Key-value stores, Hash indices, B-Trees, Geographical indices, Spatial indices
- **Data stores** - the final product: RDBMS (Oracle/IBM/Microsoft), MongoDB, CouchBase, ElasticSearch, Hive, HBase, MarkLogic, Cassandra. **But** this is not yet a database, just a data store; the data store is still low level, we need a high level language for it to be a database.
- **Querying**: SQL, XQuery, JSONiq, N1QL, MDX, SPARQL, REST APIs
- **User interfaces (UI)** - user doesn't even need to use a query language: Excel, Access, Tableau, Qlikview, BI tools, voice assistants (Siri, Alexa,...)

This section talks about the storage.

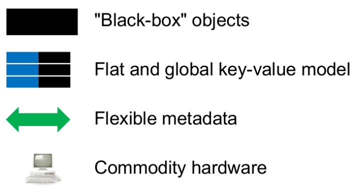


Figure 3: Object storage

4.2 Storage: from a single machine to a cluster

Data needs to be stored somewhere. Back in the 70s, data was rather limited and we could just store databases locally on a single HDD on a single machine. In a classic file storage, files are organized in a hierarchy (the file system).
Typically, a file is made of:

- File metadata: information about the file such as access rights, creation date, etc. File metadata has the data shape *table*.
- File content: The actual content is stored in blocks on disk (not just as one large junk). E.g. FFS: files are represented by inodes which point to various data blocks that make up the file.

Issues with local storage:

- Local storage is possible on a local machine and on the LAN (e.g. a NAS). However, local storage is not usable for the WAN - we can't share a local drive with 100s of millions of people. We'd like to find a way to make this possible.
- Scaling issues: 10^3 or even 10^6 files fit on a local storage. However, we can't fit 10^9 files on a single machine.

So how do we make this scale?

- Use explicit block storage for better performance (expose singe blocks to users). This is not very convenient but performs better.
- Throw away the hierarchy and use a **flat** file system.
- Make metadata flexible - don't force metadata to be a table - it can be any data shape.
- Make the data model trivial - just assign a name to every object, i.e. an ID for each file, no more structure - essentially a **key-value store**.
- Use commodity hardware - scalability principle: take lots of simple, known instances (i.e. local machine).

... and we get Object Storage.

4.3 Scale

One single machine is not good enough - how do we scale?

- Approach 1: Scaling **UP** - more cores, more memory, etc. Scaling up gets extremely expensive very fast (exponentially) - e.g. RAM: 32GB RAM is ok, 64GB RAM is ok, 128GB is still ok, ..., 6TB RAM exists but is exponentially more expensive. 50TB RAM doesn't even exists, would need to be developed.
- Approach 2: Scaling **out** - instead of buying faster, bigger machines, just buy more of the same machine! Scaling up is much more scalable in terms of hardware cost - the cost essentially just increases linearly with the number of machines we buy.
- Approach 3: Be smart (this is the first thing you should always do)

4.4 Scaling out

Data centers: All these single machines need to live somewhere - in a data center. A data center is essentially a collection of *1'000 - 100'000 servers*, each of which has *1-100 cores*. Having more than 100'000 servers in a single DC is hard because of coordination but mainly due to energy consumption for power and cooling. Each server has *1-20TB of storage* and *16GB - 6TB of RAM*. Servers are connected and the network achieves *1-100 GB/s throughput* - high network throughput is very important since servers send data among each other. Servers have the form of *rack servers* which allows to efficiently stack them in *rack units* (RU). A DC has multiple RUs. Racks are modular - they can contain servers, storage, routers etc.

Take away message: how to scale out? Simplify the model, buy (lots of) cheap hardware, remove schemas.

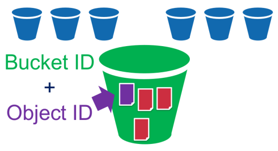


Figure 4: S3 model

4.5 Amazon S3

Amazon's S3 model is very simple, it's essentially: *objects in buckets*. S3 has no tables, cubes,... and no nesting (no buckets in buckets). We have many buckets and each bucket has a *bucket ID*. Each object inside a bucket has an *object ID*. Having buckets makes things cleaner and makes it easy to assign them to machines. An object can be anything (picture, video, etc.) and every object can be accessed by the tuple $(bucket\_id, object\_id)$. The *maximum object size is 5TB* and we can have up to 100 buckets per account (more upon request).
Service level agreement (SLA): SLA assures the quality of the service, e.g. 99.999999999% durability (loss of 1 in 10^{11} objects in a year) and 99.9% availability (1h/year downtime). Very high SLAs are *very* hard to achieve (e.g. 99.99999% SLA) corresponds to 4seconds/year outage - almost impossible. Amazon has a different approach to SLAs: response time < 10ms in 99.9% of the cases.

4.6 APIs

API (application programming interface) is a set of rules and mechanisms by which one application or component interacts with the others. API can return data that you need for your application in a convenient format (e.g. JSON or XML). We need to somehow interact with the objects:

- Driver (Java Database Connectivity (JDBC)...) for various programming languages
- SOAP
- REST (lightweight version of SOAP)

4.7 Properties

We want certain properties for databases. But:
CAP theorem: One cannot achieve consistency, availability and partition tolerance all at the same time.
Proof: Imagine a system (e.g. bank) that is in a partitioned state. At right that time, someone requests the service (e.g. ATM). The system can now either serve the user - giving up partition tolerance (service will change state which won't be propagated globally due to the partition) *OR* the system can not serve the user - giving up availability.

- (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.

4.8 REST APIs

REST (representational state transfer) provides data presentation for a client in the format that is convenient for it and is typically based on the HTTP protocol. REST is not a standard or protocol, it is an approach to or architectural style for writing API! REST is based on the client-server model.
HTTP is used to request resources over a network. Resources are addressed through a URI (uniform resource identifier) e.g. *http://www.ethz.ch/*, *http://www.mywebsite.ch/api/collection/foo/object/bar*, *mailto:sheldon.lee.cooper@ethz.ch*.
Let's dissect a URI: http://www.mywebsite.ch/api/collection/foo/object/bar?id=foobar#head

- http: gives the protocol
- www.mywebsite.ch: the authority (domain)
- api/collection/foo/object/bar: the path (often this is the exact path on the server, e.g. for static websites)
- ?id=foobar: The query
- #head: The fragment (directly jumps to specific part of the page)

HTTP methods

- GET: get the resource (side-effect free)
- PUT: create a new resource and store it (idempotent)
- DELETE: delete a resource (if you send DELETE then GET on same object → 404 page not found)
- POST: update the corresponding resource with information provided by the client, or create this resource if it does not exist (not idempotent)

All requests you make have their HTTP status codes. There are a lot of them and they are divided into 5 classes. The first number indicates which of them a code belongs to:

- 1xx - informational
- 2xx - success
- 3xx - redirection
- 4xx - client error
- 5xx - server error

4.8.1 S3

REST with S3: buckets and objects: http://*bucket*.s3.amazonaws.com/*object-name*

S3 REST API:

- Bucket: {PUT, DELETE, GET} bucket
- Object: {PUT, DELETE, GET} object

Folders: is S3 a file system? The physical file system in S3 is flat - there are no folders/hierarchies. But we can simulate a hierarchy by naming objects as if they were in a hierarchy. This will be displayed as a hierarchy. Thus, on the logical level (browsing), S3 looks like a hierarchical file system but on the physical level (object keys) it's really a flat file system. (Again, it is important to distinguish between the physical and logical part). The objects would then have names such as */food/fruits/orange, /food/vegetables/tomato* which simulates the hierarchy.
S3 can host various things such as static websites or datasets. Datasets are just lots different files put into objects. This is different from relational DBs where we split data by having different tables for different instances (e.g. order, customer table). Here we don't have a file for order and a file for customer.

4.9 More on storage

Replication: We want fault tolerance (faults happen). This can be achieved by replication (if you replicate the file, loosing it totally is less likely). Faults can happen locally (node failure) and regionally (natural catastrophe).

- Local fault: replicate over multiple local machines
- Regional fault: replicate over multiple DCs in various regions. This gives us better resiliency to natural catastrophes and better latency.

Cloud providers offer different storage classes, trading availability for cost:

- Standard: High availability
- Standard - Infrequent Access: Less availability, Cheaper storage, Cost for retrieving
- Amazon Glacier: Low-cost, Hours to GET

4.10 Azure Blob Storage

Azure blob storage is different from S3:

	S3	Azure
Object ID	Bucket + Object	Account + Container + Blob
Object API	Blackbox	Block (like bucket)/Append (for logs)/Page (for VMs)
Limit	5TB	4.78 TB (block), 195 GB (append), 8TB (page)

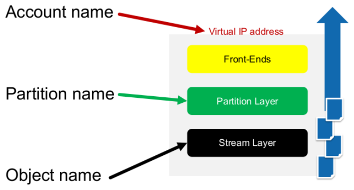
Azure thus has one more layer of indirection: account, container, blob. Further, Azure doesn't view an object as a blackbox, it let's you see inside and differentiate between 3 types of blobs: block, append, page. The account name maps to a virtual machine which is responsible for my data. The partition name is a chunk of data and the stream layer streams data over to me. In the Azure datacenter (i.e. one storage stamp) we have 10-20 racks, each with 18 storage nodes. This totals to 30PB ($= 30 * 10^{15} bytes$) per datacenter - over the whole world, Azure reaches an exabyte range of storage. Azure keeps the usage of the datacenter below 70-80% storage capacity since dealing with full storage is annoying.

Storage replication:

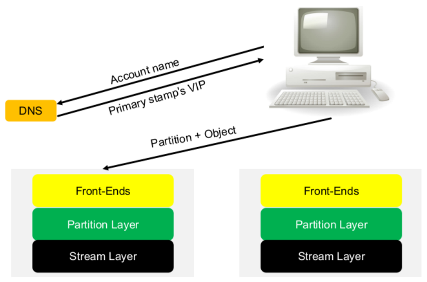
- Intra-stamp replication (synchronous): Replication within the streaming layer (i.e. within the same location (data center)).
- Inter-stamp replication (asynchronous): Replication between different partition layers (i.e. between different locations (data centers))

Location services: Azure has globally distributed data centers for load balancing and latency optimization. The location service works as follows:

1. Location service sends request with account name to DNS server.
2. DNS server returns a virtual IP that is mapped to the account name. The virtual IP points to a primary storage stamp and also to a backup storage stamp.



(a) Azure storage stamp



(b) Azure location services

Stream Layer: Stores the bits on disk and is in charge of distributing and replicating the data across many servers to keep data durable within a storage stamp. Can be thought of as a distributed file system layer within a stamp.
Partition Layer: Built for (a) managing and understanding higher level data abstractions (Blob, Table, Queue), (b) providing a scalable object namespace, (c) providing transaction ordering and strong consistency for objects, (d) storing object data on top of the stream layer, and (e) caching object data to reduce disk I/O.
Front-End Layer: Set of stateless servers that take incoming requests.

4.11 Mindsets

Amazon and Azure have different mindsets in their cloud services:

- Amazon mindset: Amazon thinks a lot in terms of modules. They have 200 different services they offer (S3, domain management, AWS,...) $\rightarrow$ a lot of small services that each do one particular thing.
- Azure mindset: Azure has bigger services that do more $\rightarrow$ fewer services that do several things.

5 Distributed file systems

Where does data come from?

- Raw data: sensors, measurements, events, logs (e.g. CERN sensor measurements)
- Derived data: aggregated data, intermediate data (e.g. computational results on sensor measurements)

Big Data isn't big data - there are different cases:

- **A huge amount of large files** (billions of TB files): This is what S3 is $\rightarrow$ lots of files, but single files are not extremely big, just large (e.g. 5TB for S3).
  $\rightarrow$ Object storage + Key-value model
- **A large amount of huge files** (millions of PB files): We need something new for this...
  $\rightarrow$ Block storage + file system

Google had just this idea: have a FS that looks and feels like a normal file system *but* lives on a distributed cluster. Google went on and developed GoogleFS.

5.1 Fault tolerance and robustness

We have different paradigms for fault tolerance:

- Local disk: the disk *might* fail, we can just keep a backup *in case.*
- Cluster with 100s to 10'000s machines: nodes *will* fail ($P[\text{at least one node fails}] = 1 - (1-p)^n$, with n: #nodes, p: failure probability of a single node). We thus need a stronger property for clusters, we can't restore from backups every single day.

How to achieve fault tolerance and robustness on clusters:

- Fault tolerance (system keeps working under faults)
- Automatic Recovery (can't recover manually with lots of disks)
- Error detection (know about failed disks)
- Monitoring (keep an overview over what's working)

5.2 File system specifications

File read/update model:

- Random access (can read at any position in a file): this is hard to do in clusters

- Sequential access (scan the file from the beginning (reading)/ append to file (update): this is easier for distributed clusters, that's why (most) do it like this

Appends: You can only append, can't go back and change something. This is suitable for sensor data, logs and also intermediate data. **Note:** Since we have a distributed FS, we have 100s of clients in parallel reading/writing → we want atomicity.

Performance requirements:

Our top priority is throughput (how fast do you read/write). Secondary, we want latency (time until we start reading/writing).

Remember: we have a huge discrepancy between storage capacity, throughput and latency. Over the last 60 years, storage capacity improved by 200'000'000'000, throughput improved by 10'000 and latency improved by 8. The solution for the capacity-throughput discrepancy is to parallelize. The solution for the throughput-latency discrepancy is to do batch-processing.

Latency is usually not a problem in big data since the data is so large, the read/write time dominates the latency.

A similar discrepancy between throughput and latency can be observed in websites. In the 90s, a website started loading slowly, elements appeared one after another → throughput was the issue. Today, website is blank for 1s and then the whole website appears → latency is the issue.

5.3 Hadoop

Hadoop is primarily:

- Distributed File System (HDFS) (inspired by Google's GFS)
- MapReduce (inspired by Google's MapReduce)
- Wide column store (HBase) (inspired by Google's BigTable)

5.4 Distributed file systems: the model

Again, rememeber data independence: separate the logical model from the physical model.

File system (logical model): In distributed file systems, we have a **file hierarchy** (unlike the key-value model in object storage which is flat).

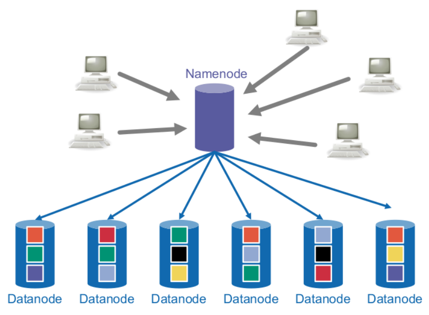
Block storage (physical storage): In distributed file systems, we have block storage (unlike object storage (S3) where we have a blackbox model).

Terminology: HDFS: Block, GFS: Chunk. We thus have a hierarchy of files where each file is associated with a chain of blocks.

Why blocks? 1) The files are bigger than a disk (PBs) , there is no way the files fit on a single machine → need blocks. 2) Simple level of abstraction and blocks are easy to distribute over multiple nodes.

Block size: Every file is a multiple of blocks. In a simple file system, we have a block size of 4kB → good size for local machines, good compromise. However, in distributed file systems, things are different: blocks travel over the network and we have very large (PB) files. Due to the throughput-latency discrepancy, we want larger blocks (for small blocks, the latency would outweigh the transfer time). Further, large blocks lead to less blocks being read per file. The block size in distributed file systems is **64MB - 128MB**. This is a good compromise - not too many blocks for big files, but also small enough to have several blocks on one machine.

5.5 HDFS Architecture



We need to connect the many machines somehow. One possible way would be peer-to-peer, but this is not ideal. HDFS uses a master-slave architecture: the namenode (has the names of the files) is the master and the datanodes (have the actual data) are the slaves.

How it works from the file perspective: The file is divided into 128MB chunks. The chunks are then stored in datanodes. Each chunk is replicated 3 times (the # of replicas can be specified).

5.5.1 Namenode

The namenode will be concurrently accessed by multiple clients. The namenode is responsible for all system-wide activity:

- File namespace (+Access Control): keep track of hierarchy of files. This is actually rather small (hierarchy doesn't contain the actual file data, just the hierarchy).
- File to block mapping: every file is associated with a list of blocks. The namenode keeps track of the mapping $file \to \{blocks\}$.
- Block location: for every block, the namenode needs to know on which 3 (default) datanodes the block is stored.

5.5.2 Datanode

A datanode is a machine with multiple local disks. Blocks are stored on these local disks. Datanodes are responsible for failure detection. Each datanode has its own local view over its disks - proximity to hardware facilitates disk failure detection. Each block has a *block ID (64bit)*. It is also possible to access blocks at a subblock granularity to request parts of a block.

5.5.3 Communication

Client protocol: The client protocol handles *client-namenode* communication. Clients sends metadata operations (e.g. create directory, delete directory, write file, append to file, read file, delete file) and the namenode responds with the datanode location and the block IDs.

DataNode protocol: The datanode protocol handles *datanode-namenode* communication. The datanode always initiates the connection. The following types of datanode-namenode communication exist:

- registration
- heartbeat: datanode tells namenode every 3s that it's still alive
- blockreport: every 6h, datanode sends full list of blocks (not the contents of the blocks) to the namenode
- blockReceived

Data transfer protocol: The data transfer protocol handles *client-datanode* communication and is used by clients to read actual block content. The client knows which datanode to contact for a given block - it got that information from the namenode.
Client reads a file:

1. Client asks namenode for file
2. Namenode sends block location (multiple datanodes for each block, sorted by distance) to client
3. Client reads data from datanode via input stream

Client writes a file:

1. Client sends create to namenode
2. Namenode sends datanodes (all replicas) for first block
3. Client organizes pipeline by contacting one datanode and tells the datanode which other datanodes to forward the data to
4. Client then sends the data over to the datanode
5. Datanode sends Ack
6. Namenode sends datanodes for second block (writing happens block by block, for every block the client contacts a datanode which will also forward to other datanodes (note: if the blocks were very small, this would have large overhead))
7. ...
8. Client sends close/release lock
9. Datanodes check with namenode for minimal replication (datanode protocol)
10. Namenode sends Ack to client
11. Namenode can tell datanodes to replicate further asynchronously

This block-by-block writing is all done simultaneously under DFSOutputStream (streaming through), checksums are used to provide data integrity.

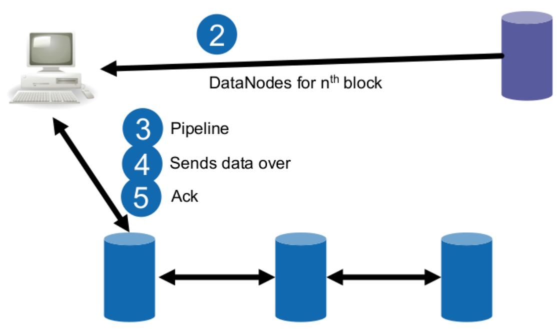


Figure 6: Client writes a file to HDFS

5.6 Replicas

The number of replicas per file can be specified (default: 3). The replicas need to be placed in a smart way. Have the topology of a data center in mind: we have a cluster consisting of multiple racks, each rack consisting of multiple nodes. We add a notion of distance between nodes $D(A, B)$.
The replicas are placed as follows:

- Replica 1: same node as client (or random), rack A
  (Note: in practice, the actual client is not on my laptop but on a node in the data center and my laptop connects to it. The 1st replica goes on that node because it's very efficient).
- Replica 2: a node in a different rack B (put in a different rack from 1st replica - racks can fail)
- Replica 3: in same rack B but on a different node
- Replica 4 and beyond: random, but if possible:
  - at most one replica per node
  - at most two replicas per rack

What if we placed replicas 1&2 on the same rack? If you always put the first two replicas on the same rack as my client node $\rightarrow$ 2/3 of replicas of all blocks written by the client will be on the same rack (the client stays on the same node) $\rightarrow$ worse replication factor.

We could also put all first 3 replicas on 3 different racks, but this would need more data being sent between racks (takes longer).
Distance: HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing up their distances to their closest common ancestor. A shorter distance between two nodes means that the greater bandwidth they can utilize to transfer data. The distance between two nodes a, b in racks A, B is:
$d(A, B) =$ #hops to TOR switch A + #hops to cluster switch + #hops to TOR switch B + #hops to node b

5.7 Performance and availability

The NameNode is responsible for file namespace, file-block mapping, block locations and is a single point of failure. We want the file namespace and file-block mapping to persist; the block locations don't need to persist since datanodes will tell us the blocks they have. HDFS thus puts the namespace file and an edit log onto persistent storage (edit log allows to not rebackup the whole hierarchy and mapping, only the things that change). We further also backup the persistent storage with the namespace file and the edit log to shared drives/ backup drives/ etc.
What if the namenode fails? We need to start it up again:

- Restore the initial hierarchy/ block mapping and then 'play' the edit log and apply changes to the file systems (restore last logged version)
- Receive the block locations from the block reports, which are periodically sent by the datanode (can also be manually requested)

This startup takes 30min. Can we do better?

- Checkpoints: periodically play the edit log and reconstruct a more recent namespace file. This way, we don't have to replay the *whole* edit log upon failure.
- High Availability (HA): Standby NameNodes. Have standby machines that keep the exact same state and immediately take over in case the active namenode crashes.
- Federated DFS: dedicated namenodes for different top directories.

5.8 Using HDFS

We can interact with HDFS as with a normal POSIX file system:

```
$ hadoop fs -ls
$ hadoop fs -cat /dir/file
$ hadoop fs -rm /dir/file
$ hadoop ds -mkdir /dir2
```

HDFS Shell: upload and download

```
$ hadoop fs -copyFromLocal localfile1 localfile2 /user/hadoop/hadoopdir
$ hadoop fs -copyToLocal /user/hadoop/file localfile
```

Populating HDFS:

- Apache Flume: Collects, aggregates, moves log data (into HDFS)
- Apache Sqoop: Imports from a relational database

5.9 Reading: The Hadoop Distributed File System [1]

- All servers are fully connected and communicate with each other using TCP-based protocols.
- DataNodes in HDFS do not use data protection mechanisms such as RAID - they use replication.
- Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file).
- Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp.
- Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance.
- An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

6 Syntax

Now, we can store data - both structured and unstructured. But: how do the files of data actually look like?

6.1 Introduction

Remember: we have data shapes text, trees, tables, cubes and graphs. In this section, we are concerned with the layers *Encoding, Syntax* of the data stack as given in figure 2. The Syntax stack is:

- Text: just text (i.e. the characters)
- Table: CSV
- Tree: XML, JSON
- Graph: RDF/XML, Turtle
- Cube: XBRL

Semi-Structured Documents

- Structured: tables
- Semi-structured: trees (JSON, XML)
- Unstructured: text

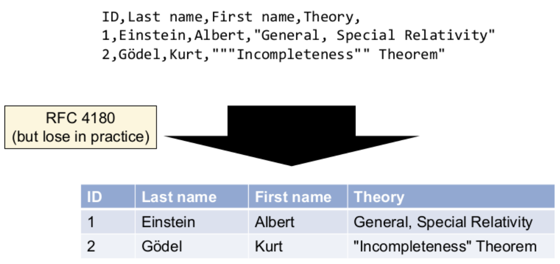
Why do we need syntax? Well-formedness: One syntax = one language, decide whether $D \in L$

The denormalizing road from SQL to NoSQL:

- Relational database: Homogeneous collection of flat items. Homogeneous meaning that on every row has the same columns and every row has some value for every column (even if null)
- Document store: Heterogeneous collection of arborescent items. Heterogeneous meaning that we have missing fields for some data points, you can even have type mismatch for same header in different data points.

6.2 CSV (tables)

In CSV (Comma separated values), we essentially have rows of text, where the top row is a headers row and all following rows are data rows where values are separated by commas. This is very easy - but it works, can have billions of rows. Note: there is an RFC for CSV, but in practice there are multiple 'flavors' of CSV (e.g. how do you escape commas).



```
ID,Last name,First name,Theory,
1,Einstein,Albert,"General, Special Relativity"
2,Gödel,Kurt,"""Incompleteness"" Theorem"
```

ID	Last name	First name	Theory
1	Einstein	Albert	General, Special Relativity
2	Gödel	Kurt	"Incompleteness" Theorem

Remember: In big data, we break several paradigms from relational databases:

- 1^{st} normal form (atomic integrity): no tables in tables
- Boyce-Codd Normal Form: have plenty of small tables that do one thing instead of one big table

Relational databases are highly normalized whereas in big data have data denormalization (NoSQL). A highly normalized DB should be write-intensive and should avoid update anomalies. A highly denormalized DB should be read-intensive and should avoid joins.
Nesting (i.e. tables in tables) is *not* possible with CSV.

6.3 JSON

JSON is a syntax like CSV but it's based on tuples and we repeat the headers in each new object. Syntax:

```
{
"product" : "Phone",
"price" : 800,
"quantity" : 1
}
```

JSON can represent a normal table (collection of tuples) *but* also nestedness (collection of tuples in tuples). CSV can't do this, in CSV we can only separate single values by comma.

JSON data types:

- Strings: `"foo"` (*has* to be double quotes), escaping is possible: `"foo\nbar\u005f"`
- Number: `3.1414, -1.2345E+5`
- Boolean: `true, false`
- Null: `null`
- Array: list of anything you want
  ```
  [ 3.1414, true, "This is a string", {"foo": false}, null ]
  ```
- Object: key-value store (key must be a string, value can be any data type)
  ```
  {
    "foo": 3.14159265368979,
    "bar": true,
    "str": "This is a string",
    "obj": { "school" : "ETH"},
    "Q": null
  }
  ```

JSON well-formedness:

- Keys need to be strings (i.e. require double-quotes "")
- Keys (should syntactically but really:) need to be distinct

6.4 XML

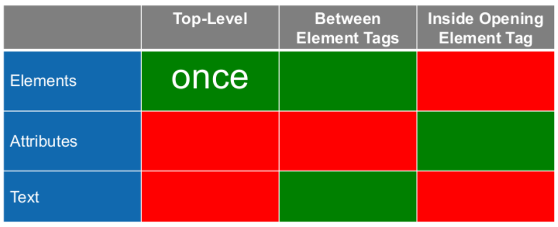
XML, the Extensible Markup Language, is a W3C-endorsed standard for document markup. XML is really similar to JSON but much more complex. Syntax:

- XML Element: `<foo>[more XML]</foo>`, `<bar/> = <bar></bar>`
  `<foo>` is the opening tag, `</foo>` is the closing tag, `[more XML]` is the element's content.
  `<bar/>` is an empty tag. Tag names are *case-sensitive.*
- XML Attribute: `<a attr="value"/>`
  Key-value pair. Keys can't be repeated in an element. Key: not quoted, value: *must* be quoted (single or double quotes).
- XML Text: `<a>This is text</a>`

- Text declaration: `<?xml version="1.0" encoding="UTF-8"?>`
- Comments: `<!-- I need to verify and update these links when I get a chance. -->`
  The double hyphen `--` must not appear anywhere inside the comment until the closing `-->`. In particular, a three-hyphen close like `--->` is specifically forbidden.
- Processing Instructions: XML provides processing instructions as an alternative means of passing information to applications that may read the document. A processing instruction begins with `<?` and ends with `?>`. Immediately following the `<?` is an XML name called the target (the name of the application for which this processing instruction is intended or possibly just an identifier for the processing instruction). example: `<?php [PHP code] ?>`
- XML Declaration: XML documents should (but do not have to) begin with an XML declaration.
  `<?xml version="1.0" encoding="ASCII" standalone="yes"?>`

XML well-formedness:

- There can only be *one* top element (root element), i.e. can't have:
  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <foo/>
  <bar/>
  ```
- Can't have text outside of top element
- Can't have an element inside an element tag. Only between tags is possible.
- Can't have same key twice in the same element
- Need to respect nesting order of opening and closing tags (like parenthesis order)
- No `<` in text (need to escape) - confuses the parser



	Top-Level	Between Element Tags	Inside Opening Element Tag
Elements	🟩 once	🟩	🟥
Attributes	🟥	🟥	🟩
Text	🟥	🟩	🟥

Figure 7: What Appears Where?

Escape characters (XML: Entity References)

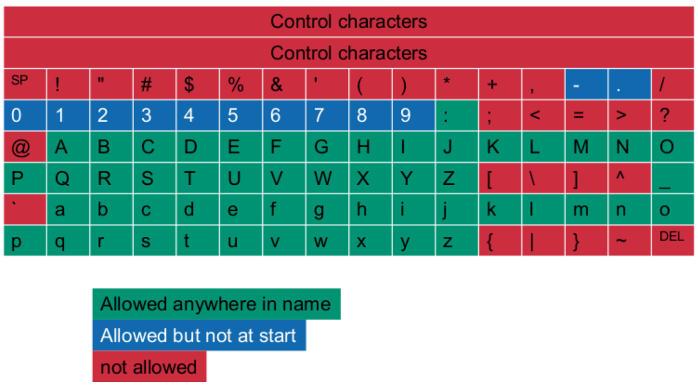
- `<`: `&lt;`
- `>`: `&gt;`
- `'`: `&apos;` (Note: single quote inside double-quotes string is ok)
- `"`: `&quot;`
- `&`: `&amp;`

All these characters need to be escaped. Special characters can also be inserted (see XML character references).

XML names: XML names may contain any alphanumeric character. This includes A-Z, a-z, 0-9, non-English letters, numbers, and ideograms, such as ö, ç, Ω, the three punctuation characters '_', '-', '.'. XML names may not contain other punctuation characters such as quotation marks, apostrophes, dollar signs, carets, percent symbols, and semicolons. The colon is allowed, but its use is reserved for namespaces.
XML names can't start with numbers and we can't have `<` in a name. All names beginning with the string "XML" (in any combination of case) are reserved for standardization in W3C XML-related specifications.

How do you tell well-formedness? An editor (oXygen, ...) will tell you. Note: Indentation does *not* matter in XML - it's only useful to make it more readable.

CDATA sections: If you don't want to escape every single $<$ and & (e.g source code) you can enclose sample of literal code in a CDATA section. A CDATA section is set off by `<![CDATA[ and ]]>`. Everything between the `<![CDATA[ and the ]]>` is treated as raw character data. Less-than signs don't begin tags. Ampersands don't start entity references. Everything is simply character data, not markup.



🟥 Control characters															
🟥 Control characters															
🟥 SP	🟥 !	🟥 "	🟥 #	🟥 $	🟥 %	🟥 &	🟥 '	🟥 (	🟥)	🟥 *	🟥 +	🟥 ,	🟦 -	🟦 .	🟥 /
🟦 0	🟦 1	🟦 2	🟦 3	🟦 4	🟦 5	🟦 6	🟦 7	🟦 8	🟦 9	🟩 :	🟥 ;	🟥 <	🟥 =	🟥 >	🟥 ?
🟥 @	🟩 A	🟩 B	🟩 C	🟩 D	🟩 E	🟩 F	🟩 G	🟩 H	🟩 I	🟩 J	🟩 K	🟩 L	🟩 M	🟩 N	🟩 O
🟩 P	🟩 Q	🟩 R	🟩 S	🟩 T	🟩 U	🟩 V	🟩 W	🟩 X	🟩 Y	🟩 Z	🟥 [	🟥 \	🟥]	🟥 ^	🟩 _
🟥 `	🟩 a	🟩 b	🟩 c	🟩 d	🟩 e	🟩 f	🟩 g	🟩 h	🟩 i	🟩 j	🟩 k	🟩 l	🟩 m	🟩 n	🟩 o
🟩 p	🟩 q	🟩 r	🟩 s	🟩 t	🟩 u	🟩 v	🟩 w	🟩 x	🟩 y	🟩 z	🟥 {	🟥 |	🟥 }	🟥 ~	🟥 DEL

🟩 Allowed anywhere in name
🟦 Allowed but not at start
🟥 not allowed

Figure 8: ASCII characters allowed in XML names

6.4.1 XML as a data format

Why Use XML for Data? Before XML, individual programmers had to invent a new data format every time they needed to save a file or send a message. Programmers thus often just used the most convenient format to store data - this made understanding the data format and loading it into any other program extremely difficult. The unique strengths of using XML as a software data format include: simple syntax, support for nesting, easy to debug, language- and platform-independent.
XML can be used in various case:

- Communication protocols: REST APIs, RPC (XML-RPC), SOAP
- Object serialization: store the state of persistent objects
- File formats: XML as a format to store application data (e.g. game store, spreadsheet, transaction,...)
- Databases: XML can play a role in the communications between databases and other software, providing information in an easily reusable form.

6.5 YAML - the "Python of JSON"

```
%YAML 1.2
---
Country:
code: 'CH'
name: 'Switzerland'
population: 8014000
currency:
name: 'Swiss Franc'
code: 'CHF'
confederation: true
president : Ueli Maurer'
capital: null
cities:
- 'Zurich'
- 'Geneva'
- 'Bern'
description: 'We produce very good chocolate.'
```

7 Wide column stores

In the relational model we have a schema that specifies attributes & types and once this is specified, the table can be filled. Issues with relational databases (RDBMS) are smale scale and the fact that they run on a single machine - PBs of data don't fit!

Column-Oriented Databases: Column-oriented databases save their data grouped by columns. The reason to store values on a per-column basis instead is based on the assumption that, for specific queries, not all of the values are needed. HBase is **not** a column-oriented database in the typical RDBMS sense, but utilizes an on-disk column storage format.

Can we fix RDBMs? The performance of RDBMSes is well suited for transactional processing, it is less so for very large-scale analytical processing

- Scaling up (more storage etc.) helps a little but doesn't solve the underlying problem → we still can't handle PBs of data.
- Scaling out (cluster, replicate): People tried this 20 years ago → very tedious and it didn't really work. It was further very hard to setup and had high maintenance cost.

Database (De-)Normalization: Design schemas differently - Denormalization, Duplication, and Intelligent Keys (DDI). It is about rethinking how data is stored in Bigtable-like storage systems, and how to make use of it in an appropriate way.

Solution: **HBase**:

- an open-source tech that allows you to store a relational table on a cluster
- By design running on a scalable cluster of commodity hardware
- By design running on a scalable cluster of commodity hardware
- HBase uses HDFS to store data (use a distributed FS to run a DB in a distributed way)

7.1 Wide column stores: data model

We first look at the underlying logical model of wide column stores. The founding paper was Google's BigTable. The problem with the tabular model is *expensive joins*. In the tabular model we have lots of tables (BCD normal form) which requires lots of joins → very expensive, especially in the distributed setting.

As such, the design paradigm of BigTable is: *store together what is accessed together* and avoid the whole join complexity (e.g. if customer & order info is accessed together, put it together). This breaks Boyce-Codd normal form → this is bad practice in relational databases. In big data, this is good practice. Keep together what belongs together. This is a shift of paradigm: *denormalize.*

7.1.1 Logical model of Wide column stores (”key-value model with columns on top”)

The logical model of HBase is *big tables with lots of rows (billions).*

- Rows: Each row has a unique identifier (similar to key-value model). The user has full control over the row IDs → can specify e.g. country-code at beginning of row ID to group countries together, or use the AHV number. All rows are always sorted lexicographically by their row key.
- Columns: HBase has columns (key-value doesn't) and columns are grouped into families. Lots of columns can be in a family (wide-column store... we can have millions of columns per family). Column families must be known in advance and thus can't be pushed too much.
- Column families must be known in advance but columns can be added on the fly.

Access to row data is atomic and includes any number of columns being read or written to. There is no further guarantee or transactional feature that spans multiple rows or across tables. The atomic access is also a contributing factor to this architecture being strictly consistent, as each concurrent reader and writer can make safe assumptions about the state of a row.

7.1.2 Primary queries

Unlike in the RDBMS landscape, there is no domain-specific language, such as SQL, to query data. Access is not done declaratively, but purely imperatively through the client-side API.

- GET: get row by ID (similar to key-value model)
- PUT: put new row with id & columns into table
- SCAN: ask HBase to completely scan the table (typically do something on every row, e.g. comparison) (doesn't exist in KV-model)
- DELETE: delete row. When a user deletes a value in an HBase table, this does *not* remove a KeyValue in the MemStore and/or in the HFile where it was stored. HDFS does not support random access. Deleting a value on the logical level creates, on the physical level, a new version of the value flagged as deleted. That way, the next time a query potentially includes this value, it will be omitted from the output by HBase.

Some terminology:

- Key-value model: entries are accessed by a key and have some value
- Column-oriented storage: instead of storing table row by row, you store the columns (e.g. store all city names in one location)
- Wide column stores: millions of columns & billions of rows → quadrillions of cells. This doesn't fit in a single DC?! → Wide column stores are typically sparse, thus they fit in a DC.

Examples of wide column stores: Google's BigTable, Apache HBase (open-source), Cassandra.

7.2 HBase: physical level

Physical layer: How should we split & distribute over machines? Split table horizontally across machines and split regions vertically across files.

- regions (= continuous set of rows in a range): partition by rows. Rows are sorted by row Id. This allows to specify regions by a range [min-incl.,max-excl.) (open max.). Regions are dynamically split by the system when they become too large.
- column families (= column family of one region) are stored together (i.e one file on HDFS)

HBase uses the same model as HDFS: master-slave model. Each region is served by exactly one region server, and each of these servers can serve many regions at any time. Splitting and serving regions can be thought of as autosharding, as offered by other systems.

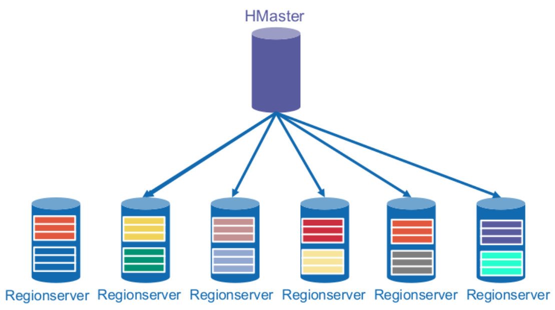


Figure 9: HBase master-slave model

7.2.1 HMaster & RegionServers

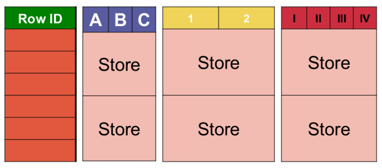
There are three major components to HBase: the client library, one master server, and many region servers. The region servers can be added or removed while the system is up and running to accommodate changing workloads. The master is responsible for assigning regions to region servers and uses Apache ZooKeeper, a reliable, highly available, persistent and distributed coordination service, to facilitate that task.

- DDL (data definition language) operations: create table, delete table, etc.
- **HMaster** splits table into regions and assigns regions to RegionServers. Someone could keep inserting more & more rows into a region. Regions must thus be split when they grow too large. HMaster will split large regions into two new regions. The master is not part of the actual data storage or retrieval path.
- **Region servers** are responsible for all read and write requests for all regions they serve, and also split regions that have exceeded the configured region size thresholds. Clients communicate directly with them to handle all data-related operations.
- If one RegionServer starts having too many regions, HMaster will rebalance them, i.e. assign region to new RegionServer.
- HMaster handles Regionserver failovers. If HMaster sees that a RegionServer failed, he tells another RegionServer that he'll now has to takeover the regions of the failed RegionServer.
- Upon a RegionServer failure, we won't loose data! The RegionServers don't actually store data, they are responsible for the regions. The actual data is stored in HDFS. HDFS has replication and is thus failure-resistent. The RegionServer from HDFS perspective is just a client. A single machine runs both a DataNode process and a RegionServer process. The HMaster and NameNode may also run on the same machine.

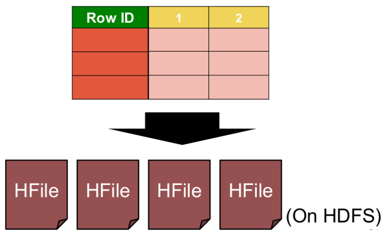
Why do we need HBase when we have HDFS? HBase is there s.t. users don't need to worry about low-level details of HDFS → data-independence. Shield users from details of storage.

7.2.2 Physical storage

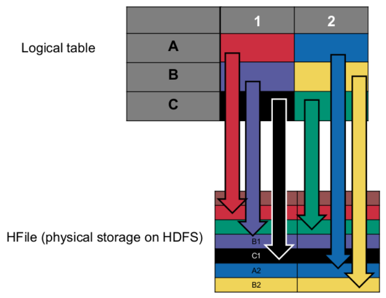
A region is handled by one RegionServer.
Store: Each region is split into column families. A column family in one region is called a **Store = (Region, Column family)**. A store is essentially the cartesian product between regions and column families. A store is made of cells. A store is mapped to **HFiles** on drives. At the beginning (small store) we only have 1 HFile. When the store grows, we'll have multiple HFiles.
HFile: We have a mapping of table to files. From the perspective of HDFS, this is just any file. If the HFile gets too large (> 128MB) HDFS splits the HFile into multiple blocks. This doesn't concern HBase. From HBase's view. it's just another file in HDFS → modularity.
Inside, the HFile works as follows: we essentially break the store into cells and store cells one after another by their key in the HFile. That's actually a sorted list of key-value pairs - keys: row-column(-version) Ids, value: cell-content (e.g cell in row B column 2: key B2).
Versioning: We can store multiple old versions of a cell. In GET, we'd just get the latest version. This is used for historical purposes (e.g. price history).
KeyValue in HBase: A KeyValue in HBase is the smallest unit of physical storage, indexed by row, column and version, and sharded by regions and column families.



(a) Physical storage



(b) Store = column family



(c) HFile

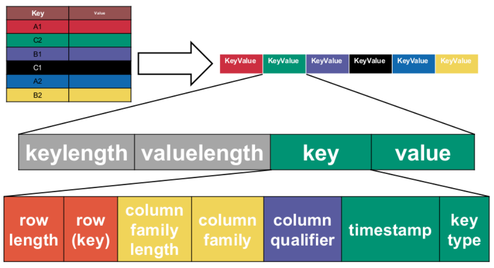
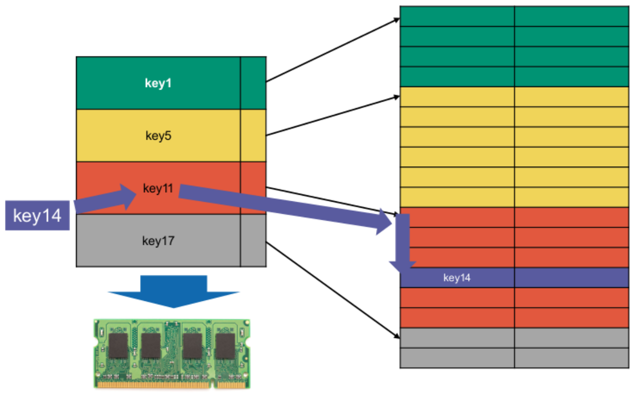
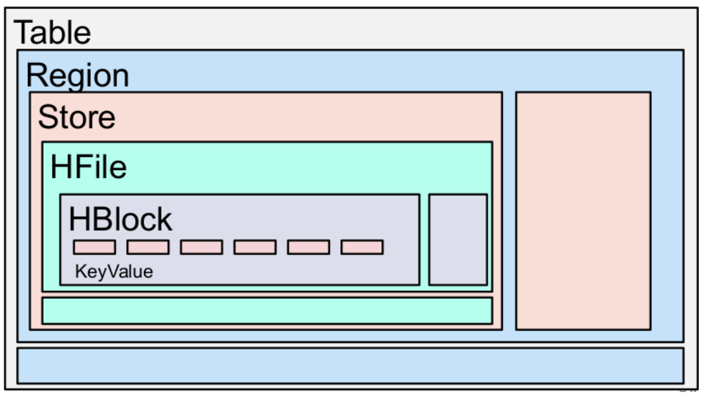


Figure 11: HFile: KeyValue

HBlocks: Key-values can be very small (bytes). We don't want to read byte-by-byte in HDFS. But we also don't want to read the whole 128MB block at a time → lower grouping of key-values. *HBlocks* are a "Quantity" of KeyValues that get read at a time (default size **64kb**). Inside the HFile, the key of the first row of the HBlock is used as key to the HBlock. Since keys are sorted, this way we easily find the HBlock in which the wanted key-value pair is located in.



(a) HFile: key lookup in HBlocks



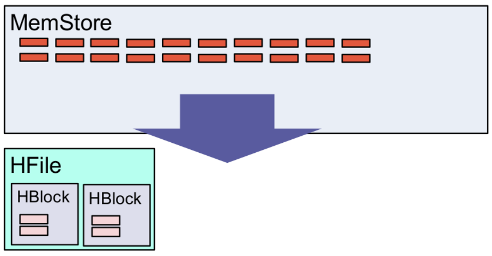
(b) Levels of physical storage

Problem: What about inserting new keys? We can only write key-values in sorted order. We need to maintain the order of keys. *But* in HDFS we can only append, we can't insert in between past lines! Solution: sort the new set of keys, completely flush the old file and create a new file.

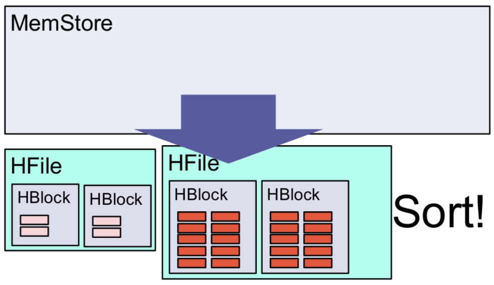
Writing new cells

We have multiple HFiles, each with multiple HBlocks (filled with key-values) on disk. When a user writes (inserts) a new key-value pair to a HFile, the pair is first stored in memory - not yet in the HFile. At some point the memory is full (when I insert a whole row, a lot of key-value pairs are created - each key-value pair is a cell in the table). Once the memory is full, all the key-values pairs in memory are flushed to HDFS. I.e. all keys from HDFS and memory are sorted and then put into new HFiles.

So essentially, instead of writing to HDFS every time we insert a new key-value pair, we aggregate new key-value pairs in memory and then flush simultaneously.



(a) Writing new cells



(b) Flush cells in memory and sort

Reading from a Store:

If someone wants to read, we have to look in HFiles *and* in MemStore (since recently added key-value pairs could not have been flushed yet).

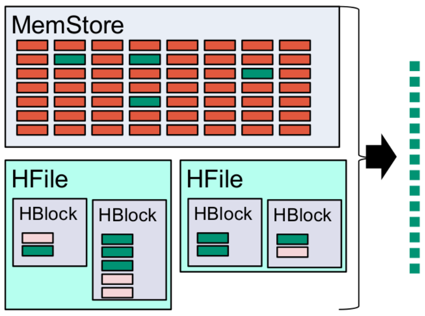


Figure 14: Reading from a Store

Compaction:

If we do a lot of flushing, we might end up with lots of HFiles. This is practically bad since in a query we need to look in every single HFile. We can compact them into a single HFile. Compaction uses MergeSort to compact the various HFiles into one HFile. MergeSort is used since it doesn't need to load all data into memory.

Seeks vs. transfer: What data structure should we use:

- B+-trees: solve latency issues (store indices on disk). Used by classical RDBMS. Latency-bound.
- LSM-Trees: solve throughput issues. Used by wide column stores. Throughput-bound.

Log-structured Merge-Trees procedure: We have lots of cells in memory which causes the memory to be full → flush the memory. This creates a new HFile. If we flush over and over, we get many HFiles. Compaction then merges two HFiles into one larger HFile. This is done over and over.

HBase Bootstrap: How to start up

If someone wants to start querying, how does he (or the HMaster) know which RegionServer to contact → meta table. Meta table stores where regions are stored/ on which RegionServer.

Resolution:

1. Client contacts HMaster to Create/delete/update table
2. Client contacts RegionServer that stores the meta table and asks it for a region.
3. The RegionServer responds with the RegionServer location(s)
4. Client queries the RegionServer

7.3 HBase: Underlying APIs

HBase is implemented in Java and offers a Java and REST API.

7.4 HBase: caching

HBase uses HDFS and HDFS can be slow. Thus, HBase will be slow. In order to improve this, HBase uses caching to speed up. The cache contains copies of some cells (according to some caching policy).
When NOT to use the cache:

- Batch processing: If you keep scanning the entire table the cache is useless. Cache is only useful if you access a few hot items.
- Random access: If you access random places, the cache will just keep replacing.

As a general notion, HBase is fast (even though it runs on top of HDFS, which is slow) because:

- HBase has KeyValues in the MemStore as well as in caches
- HBase shortcircuits DataNodes: A RegionServer process is on the same machine as a DataNode process. Thus, the files of a RegionServer are essentially stored 'locally' on disk in HDFS blocks. If the RegionServer knows that the blocks of the HFile it wants to read are directly on the same machine, the RegionServer will *NOT* go through the whole HDFS process of contacting the NameNode, contacting the DataNodes, etc. and instead short circuit that process and directly read the blocks from its local hard drive. This is much faster because you don't have the network overhead.

7.5 Data Locality

HBase vs HDFS: How do HDFS and HBase interact with eachother?
When you create an HFile, HDFS will create and store it. It will split the HFile into HDFS blocks (128MB). If a RegionServer creates a new HFile, it will communicate with the HDFS NameNode → NameNode answers where blocks should be stored in DataNodes. The first replica is placed on the machine where the client is on - the machine that runs the RegionServer (HDFS client) process also runs a DataNode process! Thus at least one replica of each HDFS block that corresponds to an HFile of a RegionServer is stored on the same physical machine as the RegionServer.

HDFS may redistribute blocks which could cause some blocks to be on different DataNodes... RegionServer could tell NameNode to put its blocks back on the DataNode that runs on the machine it is running on - but this isn't necessary! *HFile compaction brings back locality* - the new compactified HFile will be located on the DataNode that is on the same machine as the RegionServer.

7.6 Best practices

- Number of rows:
  - Millions: RDBMS
  - Billions: HBase
- Number of nodes: HBase needs many nodes to be useful (> 5). With just 1 node: use a RDBMS.
- Row IDs and column names: Keep them short. Why? Keys are made of row IDs and column names. The keys are stored in every key-value pair → long keys use just too much memory.

7.7 Summary

HBase is a distributed, persistent, strictly consistent storage system with near-optimal write—in terms of I/O channel saturation—and excellent read performance, and it makes efficient use of disk space by supporting pluggable compression algorithms that can be selected based on the nature of the data in specific column families.
There is no declarative query language as part of the core implementation, and it has limited support for transactions. Row atomicity and read-modify-write operations make up for this in practice, as they cover most use cases and remove the wait or deadlock-related pauses experienced with other systems.
HBase handles shifting load and failures gracefully and transparently to the clients. Scalability is built in, and clusters can be grown or shrunk while the system is in production. Changing the cluster does not involve any complicated rebalancing or resharding procedure, but is completely automated.

8 Data models

Syntax vs. data model:

- Data model (Logical view): table, tree, etc.
- Syntax (Physical view): CSV, JSON, etc.

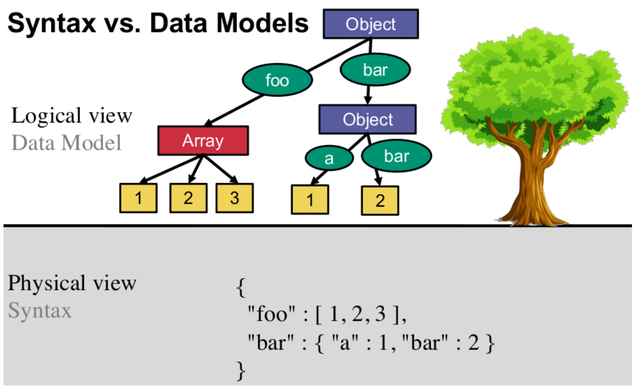
A syntax (e.g. CSV) is associated with a data model (e.g. table). In RDBMS, we first defined the data model (table) and *then* the syntax (CSV). With XML and JSON, we did the opposite. We first defined the syntax (XML, JSON) and after the tree data model.

8.1 JSON Data Model

Two types of values:

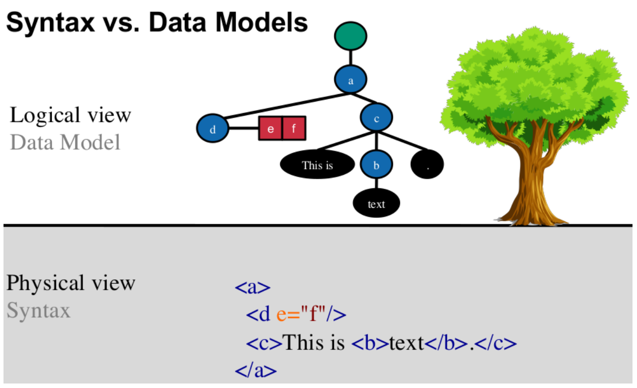
- Atomic values: strings, numbers, booleans, null
- Structured values: objects (String-to-Value map), arrays (List of values)
  $\rightarrow$ both of these offer recursion (i.e. array in array or object in object)

As we know, JSON is a **tree-based** model:



```
{
  "foo" : [ 1, 2, 3 ],
  "bar" : { "a" : 1, "bar" : 2 }
}
```

(a) JSON: Syntax vs. Data Models



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

(b) XML: Syntax vs. Data Models

The same holds for XML.

The difference between JSON and XML regarding the visual model is that for JSON, the labels are on the edges and for XML, the labels are on the nodes.

8.2 XML Information Set

The 11 XML Information Items:

- **Document**
- **Element**
- **Attribute**
- (Processing Instruction)
- **Character**
- (Comment)
- (Namespace)
- (Unexpanded Entity Reference)
- (DTD)
- (Unparsed Entity)
- (Notation)

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
<metadata>
    <title
        language="en"
        year="2019"
    >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Document Information Items:

- Document Information Item: doc
- `[children]` Element Information Item: metadata
- `[version]`: 1.0

Element Information Items:

- Element Information Item: metadata
- `[local name]` metadata
- `[children]` Element Information Items: title, publisher
- `[attributes]` `<empty>`
- `[parent]` Document Information Item: doc

- Element Information Item: title
- `[local name]` title
- `[children]` Element Information Items: Systems Group
- `[attributes]` Attribute Information Items: language=en, year=2019
- `[parent]` Element Information Item: metadata

- Element Information Item: publisher
- `[local name]` publisher
- `[children]` Element Information Items: ETH Zurich
- `[attributes]` Attribute Information Items `<empty>`
- `[parent]` Element Information Item: dc:metadata

Attribute Information Items:

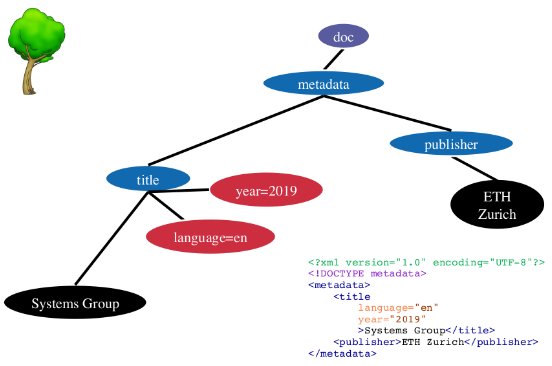
- Attribute Information Item: year=2019
- `[local name]` year
- `[normalized value]` 2019
- `[owner element]` Element Information Item: title

- Attribute Information Item: language=en
- `[local name]` language
- `[normalized value]` en
- `[owner element]` Element Information Item: title

Text Information Items:

- Text Information Item: Systems Group
- `[characters]` S y s t e m s `<space>` G r o u p
- `[owner element]` Element Information Item: title

- Text Information Item: ETH Zurich
- `[characters]` E T H `<space>` Z u r i c h
- `[owner element]` Element Information Item: publisher



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
<metadata>
    <title
        language="en"
        year="2019"
        >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Figure 16: XML Infoset - the tree

8.3 Types

Type systems: Almost all type systems (Java, SQL, PSVI, JDM, Protocol buffers, Avro, Parquet, and so on) share the following properties:

- Distinction between **atomic** types and **structured** types
- **Same categories** of atomic types
- **Lists** and **maps** as structured types
- Sequence type **cardinalities**

Types (general): *Atomic types vs. structured types*

8.3.1 Atomic types

Strings: Character sequences with monoid structure

Numbers:

- Interval-based integer types: exist as signed and unsigned, different lengths (16-bit, 32-bit, etc.)
- Arbitrary precision decimals (and integers): this needs some library (e.g. in Java big int, big float). These offer any precision and any scale (can have 1000s of digits after comma)
  $\rightarrow$ in XML & JSON, this works! There is no size limit on numbers, just put all digits

- Float and Double (IEEE 754 standard)
  - single precision: 32bit, ca. 7 digits, 10^{-37} to 10^{37}
  - double precision: 64bit, ca. 15 digits, 10^{-307} to 10^{308}

Booleans: true, false (different names exist: t/f, y/n, on/off, etc.)
Dates and Times: (quite important but overlooked):
- date (a certain day, Gregorian calendar): Year + Month + Day, e.g. 2020-March-30
- time: Hours + Minutes + Seconds, e.g. 10:31:15.109378
- timestamp (date+time): Year + Month + Day + Hours + Minutes + Seconds, e.g. 2020-March-30 10:31:15.109378

→ date and time are *still* not completely standardized

Time Intervals: Two duration kinds
- years + months, e.g. 2 years and 4 months
- hours + minutes + ..., e.g. 3 hours and 14 minutes

→ *never* mix months and days in duration, because a month can be 28,29,30,31 days!
Binaries: just a bunch of 0s and 1s
Null: exists in many schemas, languages, etc.

Lexical space vs. value space:
- Value space: the mathematical value (concept of value), e.g. the number 10
- Lexical space: the representation in syntax of the value, e.g. $10_{dec}, 1010_{bin}, A_{hex}, +10E00_{sci}$...

Subtypes:
This is a similar concept to classes and subclasses in OO languages. Example of subtypes: integer is a subtype of decimal. Abstract perspective: value spaces. A type's value space is all values a type can have. A subtype means that the value space of the subtype is a subset of the supertype's value space.

8.3.2 Structured Types

Structured types are almost always the same: *Maps (Key-value model!) & Lists*

Data Structure	Examples
Maps (Key-value model!)	JSON Object, Set of XML Attributes, Protobuf Message
Lists	JSON Array, XML Element, Protobuf repeated field

Cardinality: not available in all languages

How many?	Common sign	Common adjective
One		required
Zero or more	*	repeated
Zero or one	?	optional
One or more	+	

8.4 Protocol Buffers

In the 70s, the 'typical' way to define something was to first define a class that defines an object. Then, you instantiate the class as an object. This is the way it's done in RDBMS and in most programming languages - and this is also how it's done in Protobuf.
Protobuf is language neutral and will convert messages to different languages. This allows to share data between different languages. A Protobuf message defines a **schema**:

```
message Person {
  required string last_name = 1;
  repeated string first_name = 2;
  optional Title title = 3;
  optional Person boss = 4;
}
```

Scalar types:
- double, float
- int32, int64 and variants
- bool
- string
- bytes

Enums: enumeration

```
enum Title {
  MR = 1;
  MS = 2;
  MRS = 3;
}
```

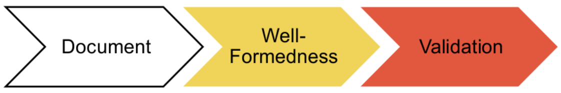
e.g. in C++: `person.boss().first_name()`
This is essentially a 'query' for the firstname of the boss person. Protobuf builds methods, functions etc. in a specified language (e.g. C++) → one can then access this.
JSON/XML vs. Protobufs:

- With schema (homogeneous, e.g. Protobuf): always the same schema - RDBMS has this - always the same fields and types must match.
- No schema (heterogeneous, e.g. JSON, XML): This is the new thing - get rid of schema! We just have a syntax (JSON, XML) but don't specify a schema. The objects in the syntax can vary *widely*. **But**: schemas are still useful. We just worry about the schema later. First, create the data, then compare it to a schema.

Not having a schema can be very powerful, e.g. when we just want to dump data. But without a schema, things can get messy. One can also enforce schemas on a heterogeneous syntax! You can introduce a schema to make objects look a certain way. A syntax should then be well-formed *AND* should fulfill some additional properties (schema). E.g. in JSON doc, all values for key "b" are boolean.

8.5 Validation

Many applications need a more powerful and expressive validation method.



- Document: just text
- Well-formedness: is text well-formed, i.e. valid syntax (well-formed JSON, XML,...)
- Validation: is text valid against a particular *schema* → *validation always needs a schema*. There is no absolute notion of validity.

$$\text{valid XML} \neq \text{well-formed XML}$$

Validation can only be checked on a well-formed document! We thus have to first check well-formedness and then validity. A well-formed document can be valid against schema A and invalid against schema B.
Validation vs. Annotation:

- Validation: check text against schema. Validity can only be checked on a well-formed document.
- Annotation: in XML, we only have text, no int, bool, etc. We can 'stamp' text in XML as int, bool, etc.

8.6 XML Schema

An XML Schema is an XML document containing a formal description of what comprises a valid XML document. An XML document described by a schema is called an instance document.

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

This is well-formed XML. We can define a schema on this XML document such that element `foo` has type string.

```
<?xml version="1.0" encoding="UTF-8"?>         <?xml version="1.0" encoding="UTF-8"?>
<xs:schema                                     <foo>
  xmlns:xs="http://www.w3.org/2001/XMLSchema">   142857
  <xs:element name="foo" type="xs:integer"/>   </foo>
</xs:schema>
```

Simple Types: Built-in

Strings	string, anyURI, QName
Numbers	decimal, integer, float, double, long int, short byte, positiveInteger, nonNegativeInteger..., unsignedLong unsignedInt...
Booleans	boolean
Dates and Times	dateTime, time, date, gYearMonth, gMonthDay, gYear, gMonth, gDay, dateTimeStamp
Time Intervals	duration, yearMonthDuration, dayTimeDuration
Binaries	hexBinary base64Binary
Null	-

8.6.1 User-defined types

- Restriction, e.g. string of char length 3
- Union (not atomic), e.g. integers or booleans
- List (not atomic), e.g. list of strings

8.6.2 Complex types

- Empty, e.g. `<foo/>`
- Simple content, e.g. `<foo>text</foo>`
- Complex Content, e.g.
  ```
  <foo>
    <a/>
    <b/>
  </foo>
  ```
- Mixed Content: character data can appear along elements in body of associated element, e.g.
  ```
  <foo>
    Text<a/>Text<b/>
  </foo>
  ```

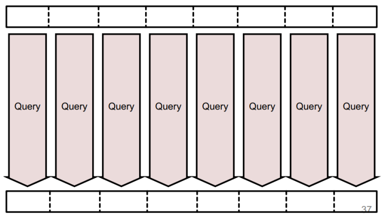
9 Distributed Computations I: MapReduce

Basic idea of MapReduce at an example: count pokemons. Map phase: Distribute sets of Pokemons to many people. Each person counts how many of each Pokemon type he has (i.e. map some part of the input to some intermediate representation). Phase 2: Each person is responsible for counting a set of specific Pokemon types. Each person asks all other for their counts of the Pokemons they are responsible for and then adds all these counts up (i.e. reduce the intermediate representation).

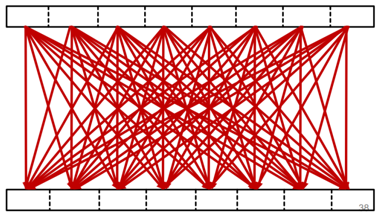
MapReduce is the processing stage - needs data. Our data can be in HDFS, HBase, S3, etc. and can have various shapes (text, tables (CSV), trees (XML, JSON), binary files, etc.). The common scenario is HDFS with data split over many HDFS files. The data is only useful for MapReduce if we can query it in parallel - data is spread over many machines and many CPUs → since storage is already distributed, we also want queries to be in parallel.

9.1 MapReduce

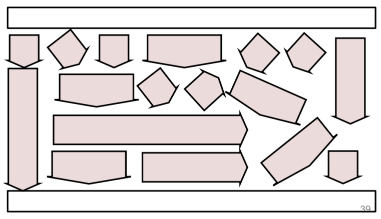
In data processing, the input data is typically sharded (comes in chunks), therefore the output is typically sharded as well. In the ideal case each output shard depends on one single input shard, this allows every machine to just work on its own data; however, this rarely happens in real life. The worst case is when every output chunk depends on every input chunk; this needs lots of communication, which is slow. The typical case is a mix of the two.



(a) Ideal case



(b) Worst case



(c) Typical case

Figure 17: Data processing cases

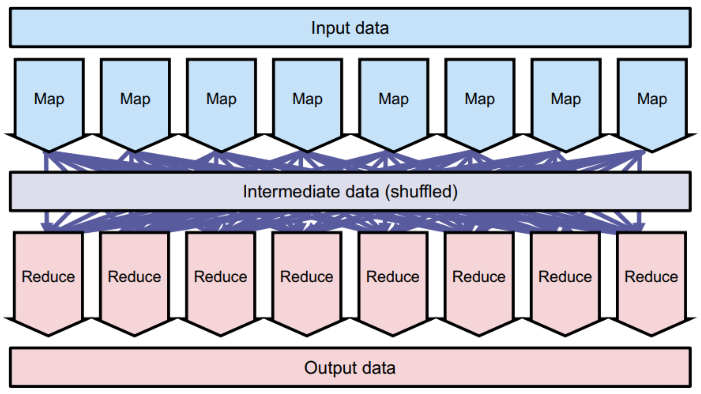
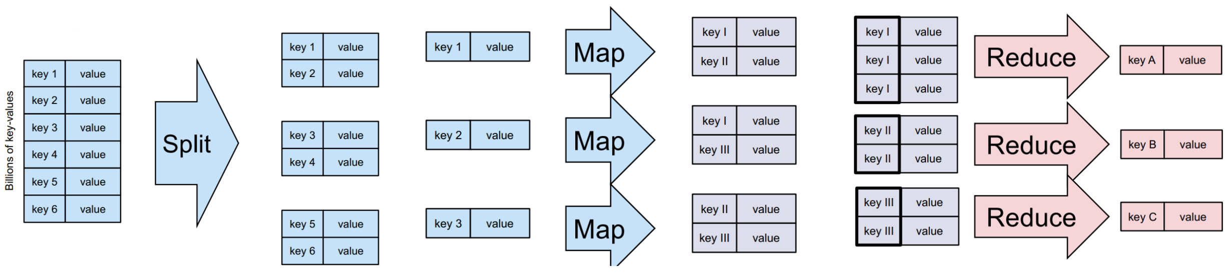


Figure 18: Data Processing: Data Model

The idea of MapReduce is to do the ideal and worst case (mapping and shuffling) in an ordered way.

9.1.1 Logical walkthrough

1. **Splitting:** The key-value input (converted from some data input) is split into logical chunks.
2. **Mapping:** Associate one key-value to one or more key-values through some mapping function. Every key is mapped in parallel. We have splits → within each split, I call mapping on each key-value in the split and this is done in parallel for all splits.
3. **Put it together:** Put all intermediate key-values together and sort them by key. Then repartition the keys *but* make sure that all key-values with the same key are in the same partition! We can have several key-values in one partition but not one key-value in several partitions.
4. **Reduce:** Take intermediate partitions and produce the output by applying the reduce function to the IR key-values in parallel. The output of one key-value set can be one or more key-values (typically just one).



(a) Splitting (b) Mapping (c) Reduce

Figure 19: MapReduce stages

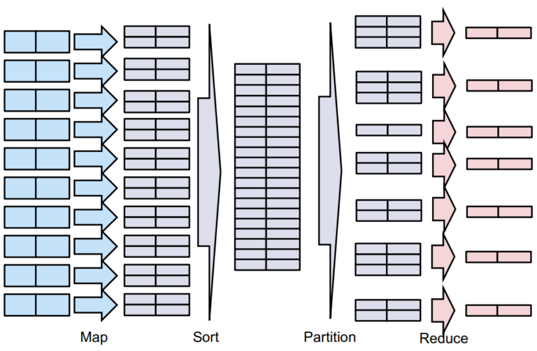


Figure 20: MapReduce overall

9.2 Architecture

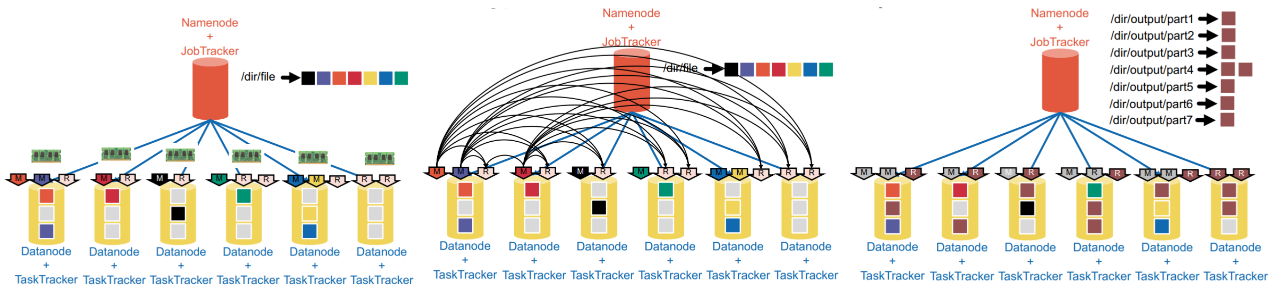
MapReduce can run on different storage layers: Local file system, HDFS (most common), S3, Azure Blob storage,... Numbers: we have several TBs of data and 1000s of nodes. What kind of architecture do we have?

Mater-slave architecture (again): Just like in HDFS and HBase, we also have a Master-slave architecture for MapReduce. If we use HDFS, HDFS runs on the same machines as MapReduce!

- JobTracker (Master): The JobTracker assigns map and reduce tasks, tells respective keys to reducers, etc. The JobTracker typically runs on the same machine as the NameNode in HDFS.
- TaskTracker (Slave): A TaskTracker runs on each one of the nodes that run DataNodes, and spawns 0,1, or more mappers as well as 0,1, or more reducers according to pre-allocated map and reduce slots. Mappers mostly read their input locally. When all mappers have completed their tasks, the reducers start requesting the intermediate output relevant to them (shuffle). When the reducers have completed, they mostly write their output locally. The TaskTracker typically runs on the same machine as a DataNode.

Running the TaskTracker on the same machine as the DataNode allows the TaskTracker to work with *local data* → **short-circuiting, bring the query to the data**. We don't need to bring the blocks anywhere, bring the query to the data and ececute on the machine where the blocks are.

Mapping phase: In MapReduce, the input data is *split* (= shard/chunk of the input data). Generally, we have on split per block (128MB) and we only look at one replica per block. The JobTracker's task is to tell the machines which mappings to do where → split the input (HDFS file), arrange s.t. split HDFS block and every mapper is working locally on a block. Occasionally it can happen that a mapper will have to fetch a block remotely over the network. Typically, you may have more mapping tasks than mapping slots (slot = just some reserved resource) → do it sequentially paralll (e.g. 100 tasks, 10 slots → each slot 10 task sequence). During the mapping phase, reducer slots are idle; they need to wait for the mapping to finish. The mapping phase outputs key-value pairs to memory (intermediate represenation) → may overfill the memory. Key-value pairs are thus spilled to disk if necessary (local disk not HDFS), spilling is done with sorted KV-pairs. Why spill to local disk? Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.
Shuffling Phase: After the mapping is done, we have all the intermediate key-value pairs, on the same machines as the mappers that produced them. The reducers have to wait until *all* mappers have finished (very



(a) Mapping Phase (b) Shuffling Phase (c) Reduce Phase

Figure 21: MapReduce phases

strict 2-phase algorithm). Once all mappers have finished, the reducers connect to every TaskTracker and ask for their respective keys. Each TaskTracker runs an HTTP server. The reducers can thus just connect to each TaskTracker HTTP server and request the key they are responsible for (assigned by JobTracker). The reducers then receive all IR key-value pairs for the keys they requested. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition.

Reduce phase: Reduce tasks don't have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. Reduce is not reading locally but from network (shuffling) since intermediate key-values were produce all over the network $\rightarrow$ received over the network. Once the reducer has all key-value pairs from the TaskTrackers (shuffling), they apply the reduce function and output the final key-values to disk - an HDFS file is created for *every* reducer task. The output of the reduce is normally stored in HDFS for reliability. Possible bottleneck: if one key has lots of values, the responsible reducer will have lots of keys while another has only a few. Note: it's possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel.

9.3 Input/output format

How do we create key-value pairs? We can have different inputs: tables (HBase, RDBMS, ...), files (JSON, XML, ...), etc. There is an intrinsic way to convert each data shape into key-value format.

- Tabular: Use primary key (RDBMS), row ID (HBase) as key and the rest of the row (all attribute values) as value $\rightarrow$ key-value pair.
- Text: different possibilities, e.g. use line number as key and text line as value.
- Key-value: already key-value.
- SequenceFile: binary version of a key-value file, stores generic key-values in Hadoop binary format. This is easy since it's already a key-value file, just encoded in binary.

Example: count words in text. Map every line number to text. Even easier: map each single word in text (key) to count 1 (value). Mappers will collect all keys (words) in their splits and sum up the 1-counts. Reducers will then collect all IR counts and sum them up.

Example: filter lines with less than 2 words out. Input: line number as key, line as value. Mapping function: filter all lines with less than 2 words out. Reduce function: identity function.

9.4 Optimization

We can optimize the shuffling phase by "pre-reducing" (=combine) the intermediate key-values values from the input mapping. This reduces the amount of data that needs to be shuffled around, which makes MapReduce more efficient. For example, in the word count use case, we could already count words that appear multiple times in one line and create one KV with the count. This combining can be done when spilling intermediate KVs to disk.

Often (90%) the combine function and the reduce function are equivalent. We thus need two assumptions to hold:

- Key/Value types must be identical for reduce input and output.
- Reduce function must be commutative (a+b=b+a) and associative (a+(b+c)=(a+b)+c).

9.5 MapReduce: the APIs

Hadoop MapReduce supports a native Java API and a streaming API that allows you to write your map and reduce functions in languages other than Java. Hadoop Streaming uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

9.6 Tricky point: blocks and splits

Split vs map tasks: a split is essentially an HDFS block. But in HDFS, a block is just 128MB long and then cut $\rightarrow$ you may separate a value into two blocks. HDFS only sees binary data and does not differentiate key/value borders.

MapReduce solves this by doing the splits on a more logical level, i.e. a split always includes the whole last key-value. The problem now is that the last key-value of one split could be in two different blocks. This is generally not an issue (just read the other block) except when the other block that contains the rest of the last key-value pair is remote (on another machine). This then requires a remote read, which leads to a slight decrease in performance, but MapReduce will take care of this.

9.7 Reading Assignment

9.7.1 Hadoop: The definitive guide

Example: find the maximum global temperature record for each year from weather data. This can be done using MapReduce. The input mapping is just a (line-offset, weather measurement) key-value pair. The mapping function extracts a (year, temperature) key-value pair for each weather record. The reduce function finally returns the maximum temperature in a key-value pair (year, [temperatures]), where [temperature] is an array temperatures for all measurements of that year.

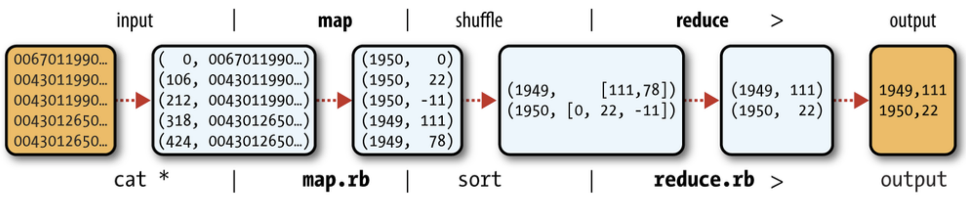


Figure 22: MapReduce example: finding the hottest temperature for each year

Data Flow: A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. The tasks are scheduled using YARN and run on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node. Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user-defined map function for each record in the split. Smaller splits are preferred since they allow for better load balancing and thus more efficient use of resources. On the other hand, if splits are too small, the overhead of managing the splits and map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, which is 128 MB by default, although this can be changed for the cluster (for all newly created files) or specified when each file is created.

9.7.2 MapReduce: Simplified Data Processing on Large Clusters

Fault tolerance:

- Worker failure: The master (JobTracker) pings workers (TaskTracker) periodically. If no response is received from a worker in some amount of time, the worker is marked failed and the map task is reset back to idle.
- Master failure: It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state.

10 Resource Management

There are some issues with MapReduce 1:

- Scalability: MapReduce 1 only works with <4000 nodes and < 40'000 tasks.
- Bottleneck on the JobTracker: the JobTracker has too many responsibilities. The JobTracker needs to do resource management, scheduling, monitoring, job life-cycle, fault tolerance - a lot of things and there is only one JobTracker.
- Jack of all trades, master of none: JobTracker is not specialized and thus not efficient since it needs to do so many different things.
- Static allocation, fixed size: Map & Reduce slots are assigned statically at configuration time (i.e. before the job starts), which leads to them often being idle, which does not fully use cluster resources.
- Not fungible: Reduce slots are idle during map phase and map slots are idle during reduce phase.

10.1 YARN (Yet Another Resource Manager)

YARN is a resource management tool that separates the JobTrackers responsibilities. The responsibilities are split into:

- Resource Manager: Scheduling, Application management
- Application masters (many): Monitoring

YARN can be used for many applications: MapReduce, DAG distributed processing, message passing interface, graph processing. Further, YARN scales better (10'000 nodes, 100'000 tasks).

10.1.1 YARN architecture

Again, we use a master-slave architecture. We have the ResourceManager (RM) as master and NodeManagers (NM) as slaves. The ResourceManager acts as the central authority arbitrating resources among various competing applications in the cluster (decides who gets which container). The NodeManager is the "worker" daemon in YARN. It authenticates container leases, manages containers' dependencies, monitors their execution, and provides a set of services to containers. The NM will also kill containers as directed by the RM or the AM. We further have the notion of slots again, here called *containers*. Containers do the virtualization of resources, mapping and reducing will be done in containers. This allows for more fine grained resource requests (memory and CPU only for now). Application Masters (AM) are the "head" of a job, managing all lifecycle aspects including dynamically increasing and decreasing resources consumption, managing the flow of execution (e.g., running reducers against the output of maps), handling faults and computation skew, and performing other local optimizations. The AM is the process that coordinates the application's execution in the cluster, but it itself is run in the cluster just like any other container.

YARN process: When a client submits a job, it is first sent to the RM. The RM then distributes the containers. The RM then allocates an AM (can be different for every machine) - the Application Master takes care of monitoring. The AM requests resources (containers) from the RM. Typically, an AM will need to harness the resources (CPUs, RAM, disks etc.) available on multiple nodes to complete a job. To obtain resources (containers), AM issues resource requests to the RM. The NM only assigns containers locally, the Application Master needs the authorization from the Resource Manager. When a resource is allocated on behalf of an AM, the RM generates a lease for the resource, which is pulled by a subsequent AM heartbeat. At the beginning, the AM only requests mapping resources. Once mapping is done, only then does it request reducing resources.

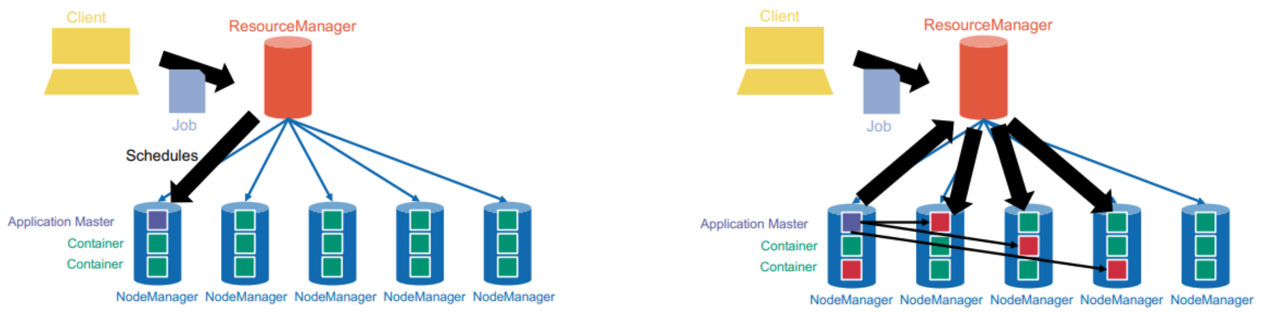


Figure 23: YARN

The resulting process is then essentially MapReduce with YARN around it.

The ResourceManager is a pure scheduler, it does not monitor tasks and it does not restart upon failure. Different scheduling strategies exist:

- FIFO scheduler: First applications has priority. Problem: long running applications can block small ones.
- Capacity scheduler: Have several queues, every queue gets separate part of the cluster.
- Hierarchical queues (Steady Fair Share)
- Fair scheduler: idle clusters are temporarily reassigned while idle. Preempted when original job needs resources.

Note: people say MapReduce and Spark are slow. But: these technologies are not built to minimize latency but to maximize cluster optimization.

11 Distributed Computations II: Spark

11.1 Introduction to Spark

Both MapReduce 1 and MapReduce 2 (YARN) use a rather restricted data flow model: data is first mapped to an intermediate representation and the reduce to an output. This is a very specific topology (*not* DAG-based). YARN is capable of much more because it supports any DAG data flow. Let's build something much more general.

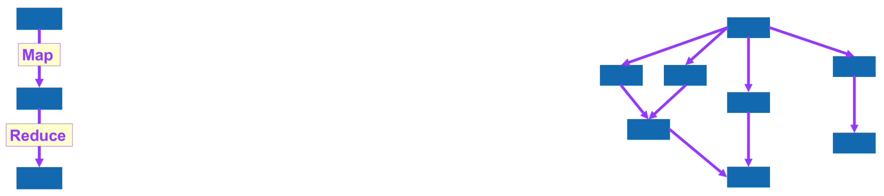
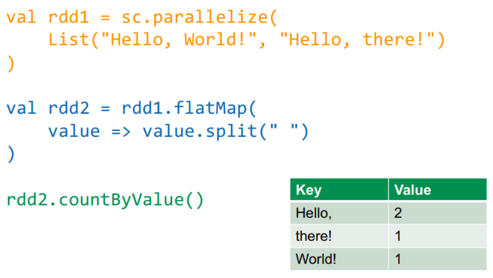


Figure 24: MapReduce dataflow vs. DAG dataflow

Spark uses RDDs (Resilient Distributed Dataset) as data format. An RDD is a big collection of values (not limited to key-values!) that can be recomputed and is spread over machines. RDDs are partitioned (like MapReduce uses splits).

RDD lifecycle: RDDs can be created on the fly from data in HDFS - RDD is just the data read from RDD and partitioned along HDFS blocks. RDDs can be transformed into other RDDs (many-to-many transform) (like mapping/reducing in MapReduce). An RDD is only returned (output) once an action is taken on the RDD. RDDs and their transforms can be represented by a directed acyclic graph (DAG) where nodes are RDDs and egdes are transformations. This lineage graph use lazy evaluations - nothing happens (no transformations are executed) until an action is requested at some RDD. Only then will the data flow through the graph and be transformed and output.

Spark can be executed in application or shell.



```
val rdd1 = sc.parallelize(
    List("Hello, World!", "Hello, there!")
)

val rdd2 = rdd1.flatMap(
    value => value.split(" ")
)

rdd2.countByValue()
```

Key	Value
Hello,	2
there!	1
World!	1

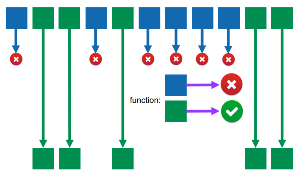
Figure 25: Example program in Spark

11.2 Spark: overview of transformations

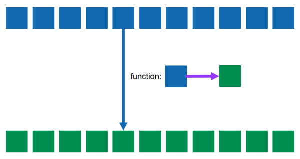
As we've seen, RDDs can be transformed. Many different (100) transformations exist.

Transformation on one RDDs:

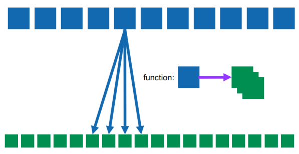
- filter: eliminate RDDs that don't fulfill some test (= SELECTION in rel. algebra). Narrow dependency.
- map: map each value in input to some value in output (#input values = #output values). Narrow dependency.
- flatMap: Map map to multiple outputs, but outputs are flattened. Narrow dependency.
- distinct: eliminate duplicates/ Wide dependency.
- sample: pick inputs randomly.



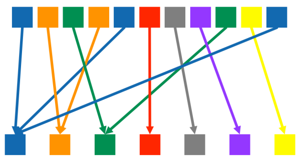
(a) filter



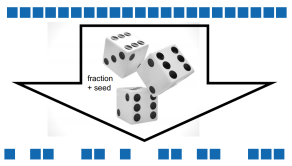
(b) map



(c) flatmap



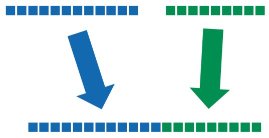
(a) filter



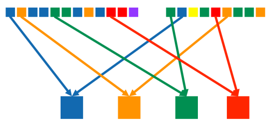
(b) map

Transformation on two RDDs:

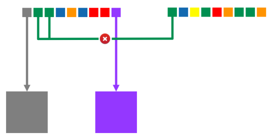
- union
- intersection
- substract
- cartesian product



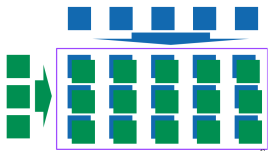
(a) union



(b) intersection



(c) substract



(d) cartesian product

11.3 Spark: overview of actions

- collect: put RDDs into local memory/ disk. This only works when RDDs are small enough to even fit in memory/ disk (risky).
- count
- count by value
- take: collect a specified number of values
- top: return top values
- takeSample: random values taken
- reduce

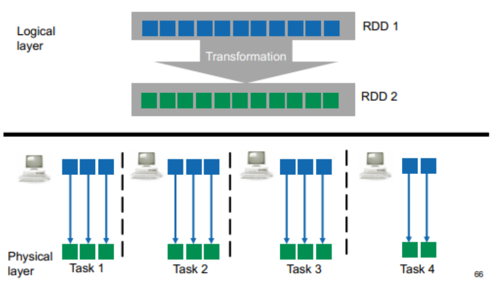
11.4 Spark: pair RDDs

We can have RDDs of values that are pairs (like MapReduce with key-values).
Transformation:

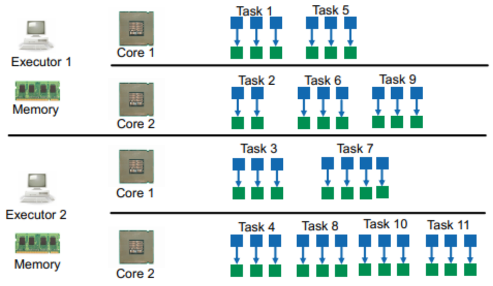
- keys: get all keys. Narrow dependency.
- values: get all values. Narrow dependency.
- reduce by key: group by key and output some aggregate value per key. Like reduce in MR. Wide dependency.
- group by key: keep all values per key and group them together. Wide dependency.
- sort by key (like GROUP BY). Wide dependency.
- map values: map each value to another value. Narrow dependency.
- join: combine values with same keys from two RDDs.
- substract by key
- count by key
- lookup

11.5 Physical layer

We again want parallel execution. The input data is split (mostly according to HDFS blocks, short circuiting) and we have one task per block. Executors will then execute the tasks. We don't want to assign one single tasks per executor: some tasks may take longer, so that most executors will be idle waiting for the last one(s) to complete. We thus want smaller tasks that can be spread over executors such that each executor has multiple small tasks. If one executor finishes, it can just continue with the next task. Tasks are thus spread over executors and spread over CPUs within an executor.

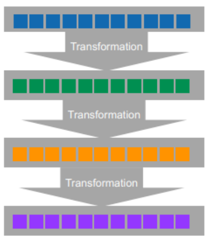


(a) Suboptimal parallel execution

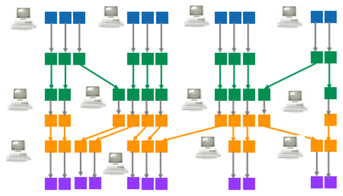


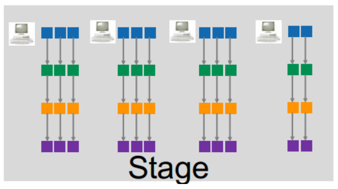
(b) Improved parallel execution

Sequence of (parallelizable) transformations: We can further improve parallelization by grouping together series of transformations on the same level (not ancestor/child of each other) with narrow dependencies. We can group these transformations together as if they were a single transformation. This can reduce the amnount of data that needs to be shuffled around. We should try to execute transformations that have dependencies on a machine on that machine, as long as the transformations allow this - compute locally as much as possible.



(a) Sequence of transformations


(b) Inefficient allocation to executors



(c) Efficient allocation to executors

Stage: We call a set of transformations that all have narrow dependencies (i.e. the transformations are parallelizable, don't require shuffling) and that can all be executed on the same executor a stage. Stages can be spread over executors/ cores.
Once we encounter a *wide dependency*, the stage is finished, we need to shuffle around. A wide dependency requires all input values, which requires shuffling (send over the network). A new stage needs to be set up.

Idea of Spark: do as many narrow dependency transformations in one stage as possible, then wait until all tasks in a stage have finished and then shuffle once the stage is done. A spark job as a whole is a *sequence/DAG of stages.*

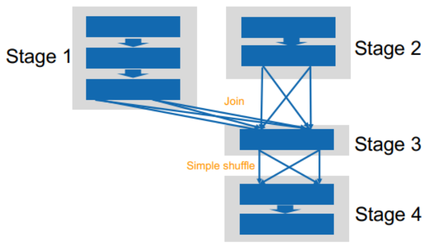


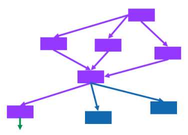
Figure 31: Spark job

Terminology

- Vertical grouping: group transformations into stage
- Horizontal splitting: split stage into tasks
- Sequence: arrange stages into Spark job.

11.6 Performance tuning

Consider any DAG with multiple output RDDs. Remember that evaluation is lazy - transformation are only executed once an action is applied on an RDD. If we apply an actions to multiple RDDs that all have common ancestor RDDs, the ancestor RDDs are calculated multiple times equivalently. This is very inefficient since we keep recomputing the same RDDs over and over.


(a) Inefficient recomputation of RDDs

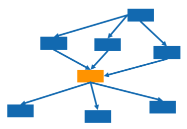

(b) Spark persisiting RDDs

Figure 32

Persisting RDDs: We can instruct Spark to persist some RDDs (cache them). Considering Figure 32b, an action on output RDDs then only requires one single transformation.
Avoiding wide dependencies: As we've seen, wide dependencies require shuffling, which takes time. Spark splits RDDs into partitions. What if we pre-partition the input s.t. shuffling is no longer necessary. We can tell Spark to proactively partition the data. The amount of work is the same, we just partition differently and as a result have less shuffling. Spark partitions key-values s.t. all KV pairs with the same key are already on the same machine and no shuffling is necessary.

11.7 Data Frames

RDDs are a low-level concept, it's just a collection of values. Doing processing on RDDs using Spark and Python is not simple. That's why we introduce DataFrames - we basically already know what they are: tables. Most important concept: data independence. Since with DataFrames we now have tables, we don't have to use transformations and actions. We can use SQL like queries which can be translated by Spark to transformations and actions. You can still use transformations on DataFrames, but SQL is generally easier to use.
If each value in an RDD is a dataset row and all values have the same set of attributes (columns), we can just interpret the RDD as a table, i.e. a DataFrame.

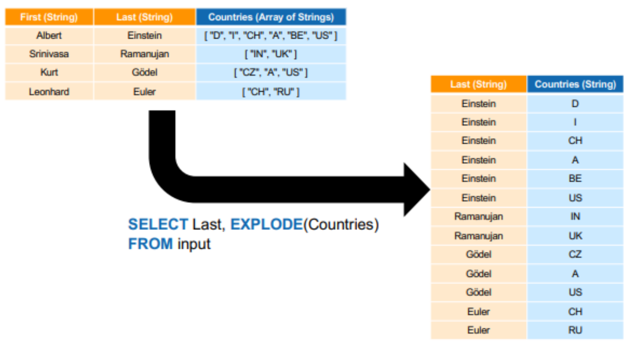
Columnar storage: Spark can store columns together. In a column, all values typically have the same datatype. This allows for much more efficient storing in memory. For example, a int32 column can be stored as just a multiple of 32bits → highly structured and efficient. DataFrames thus have a much lower memory footprint.
Schema inference: Spark automatically infers types (works for csv, json, ...).

It is possible to convert back and forth between DataFrames and RDDs, however in practice it is best to just stick with one and don't mix.
Limits of DataFrames: Heterogeneity
In practice, we don't have uniform data, e.g. column *foo* might be int somewhere but an array somewhere else. Then forcing the values into a column will produce all strings (e.g. "1" and "[1,2]").

11.7.1 Dealing with nestedness

As we've seen, JSON and XML don't always fit into tables since they allow nested data. Thus, transformations are still useful since if our data is not in tables, we can't use SQL. Then we need transformations.
Spark also offers additional optimizations to deal with nested data:

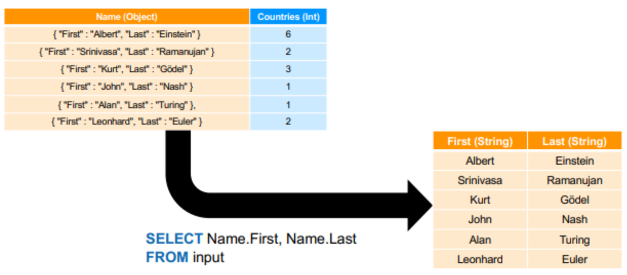
- Arrays: Spark offers an *explode()* function that automatically expands an array into normal form. We essentially get one single row for each value in the array.
- Objects: Spark allows accessing fields of objects using SQL, using a '.' notation (like class attribute). This allows us to explore tables with objects in them.



First (String)	Last (String)	Countries (Array of Strings)
Albert	Einstein	["D", "I", "CH", "A", "BE", "US"]
Srinivasa	Ramanujan	["IN", "UK"]
Kurt	Gödel	["CZ", "A", "US"]
Leonhard	Euler	["CH", "RU"]

```
SELECT Last, EXPLODE(Countries)
FROM input
```

Last (String)	Countries (String)
Einstein	D
Einstein	I
Einstein	CH
Einstein	A
Einstein	BE
Einstein	US
Ramanujan	IN
Ramanujan	UK
Gödel	CZ
Gödel	A
Gödel	US
Euler	CH
Euler	RU



Name (Object)	Countries (Int)
{ "First" : "Albert", "Last" : "Einstein" }	6
{ "First" : "Srinivasa", "Last" : "Ramanujan" }	2
{ "First" : "Kurt", "Last" : "Gödel" }	3
{ "First" : "John", "Last" : "Nash" }	1
{ "First" : "Alan", "Last" : "Turing" },	1
{ "First" : "Leonhard", "Last" : "Euler" }	2

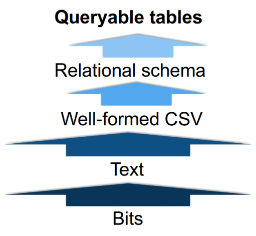
```
SELECT Name.First, Name.Last
FROM input
```

First (String)	Last (String)
Albert	Einstein
Srinivasa	Ramanujan
Kurt	Gödel
John	Nash
Alan	Turing
Leonhard	Euler

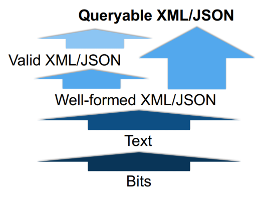
12 Document stores

Documents are just trees. Document stores are often referred to as NoSQL (= not only SQL).

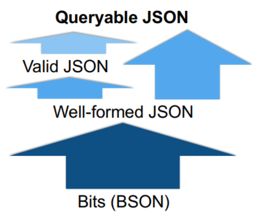
Remember: relational model
Typical relational stack: bits → text → well-formed CSV → relational schema → queryable tables.
Relational tables fulfill relational integrity (all rows have the same columns), atomic integrity (no nested tables), domain integrity (strict type per column). Relational algebra provides an way to query data with SQL being an easy query language.
Remember: We want to rebuild that stack for clusters. Up until now, we have rebuilt the stack for tables on clusters. Now: we want to do the same for the tree data model.



(a) The structured stack



(b) The semi-structured stack



(c) The MongoDB stack

12.1 Making trees fit into tables

Why not reuse things? We could try to fit trees into tables.
For flat trees, this is definitely possible (e.g. a tree with just leafs is just a row). We can natively fit several flat trees as several rows into a table, for both JSON and XML. However, all flat trees need the same fields (columns) and the same data type for each field. How do we ensure this domain integrity? We can use schemas! We can validate JSON or XML by using schemas and thus make sure that the trees are flat trees with all the same fields and domain integrity.

We thus can fit trees into tables, but we need to assume a lot: relational integrity (all same fields), atomic integrity (flat trees), maybe domain integrity (all same type in one column). For trees, this is generally not the case.
In general, we have **nestedness** and **heterogeneity**. Could we solve this...?
Nested arrays: there are ways to fix it, use multiple tables, explode row.
Heterogeneity: put NULL in missing fields.
However, this is not a good solution. **Impedance mismatch:** data to store does not have the same shape as the storage. This is why we have document stores.

12.2 Document stores (MongoDB)

Characteristics:

- Document stores don't scale as well as MapReduce, Spark, HDFS, etc. MongoDB works on a few machines (not 1000s).
- A document store is a collection of trees (large number of small trees). You can have bigger documents (e.g. picture, MB size), but not documents in the GB or TB region.
- Document stores store millions or billions of such small documents. AND: the documents don't need to look alike (but often they do).
- Tree vs. flat: document stores natively support nestedness where as rel. DBs are flat.
- Homogeneous vs. Heterogeneous: document stores natively support heterogeneity (every column/field can look different) where as rel. DBs are homogeneous (everything needs to be equivalent).
- Document stores support SQL principles such as Projection, Selection, Aggregation. BUT: joins are not ideal, very expensive (you can still implement your own join). This is not really an issue though because the data is denormalized → we already "precomputed" the joins by nesting the data.
- NoSQL: validation *after* the data was populated → validation on read (compared to RDBMS where schema needs to specified first → schema on write).

Many document stores implementations exist (MongoDB, CouchDB, elasticsearch, etc.) We focus on MongoDB. MongoDB uses JSON (document stores using XML exist as well). But MongoDB supports more types than JSON (dates, binary, etc.). MongoDB stores JSON documents as binary, which is more efficient: BSON (binary JSON). MongoDB thus does not have text, it stores directly into BSON.

12.3 Querying a document store

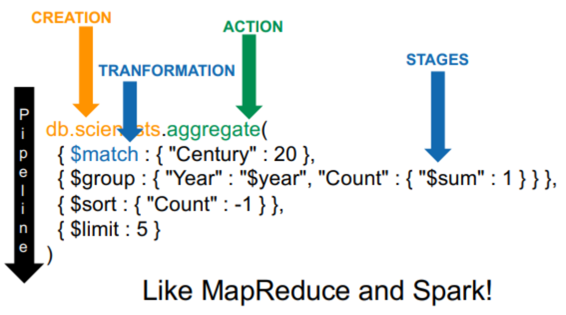
Ideally, we’d want to query document stores with a language like SQL. On a lower level, we need a CRUD (create, read, update, delete) based access to the document store. SQL (query language) can then build on top of this and provide easy querying for users.

Desc	Low level query	SQL equivalent
Read	db.scientists.find({”Theory” : ”Relativity”})	SELECT * FROM scientists WHERE Theory = ”Relativity”
Read: projection	db.scientists.find({ ”Theory” : ”Relativity” }, { ”Name” : 1, ”Last”: 1 })	SELECT Name, Last FROM scientists WHERE Theory = ”Relativity”
Read: AND	db.scientists.find({”Theory”: ”Relativity”, ”Last”: ”Einstein”})	SELECT Name, Last FROM scientists WHERE Theory = ”Particle Physics” AND Last = ”Einstein”
Read: OR	db.scientists.find({$or : [{ ”Last” : ”Newton” }, {”Last”: ”Einstein” }]})	SELECT Name, Last FROM scientists WHERE Last = ”Newton” OR Last = ”Einstein”
Read: Comparison	db.scientists.find({ ”Publications”: {$gte: 100}})	SELECT Name, Last FROM scientists WHERE Publications ¿= 100

These low level queries further support heterogeneity, i.e. they still work if we have missing fields or different types. Further, these queries support nestedness, which SQL does not.

Desc	Low level query
Read: nestedness (objects)	db.scientists.find({”Name.First”: ”Albert”})
Read: nestedness (arrays)	db.scientists.find({”Theories”: ”Special relativity”})
Read: other operators	db.scientists.find({”University”: {$in: [”ETH Zurich”, ”EPFL”]}})
Count	db.scientists.find({”University”: {$in: [”ETH Zurich”, ”EPFL”]}}).count()
Sort (-1: descending)	db.scientists.find({”University”: {$in: [”ETH Zurich”, ”EPFL”]}}).sort({”Founded”: -1})
Limit and offset	db.scientists.find({”University”: {$in: [”ETH Zurich”, ”EPFL”]}}).sort({”Founded”: -1}).skip(30).limit(10)
Duplicates	db.scientists.distinct(”name”)

Possible confusion: `db.scientists.find({"Name.First": "Albert"})` finds all documents where the ”First” field of the ”Name” object is ”Albert”. `db.scientists.find({"Name": {"First": "Albert"}})` however searches for the *exact* object ”Name” of from `{"First": "Albert"}`.
Note on *Read: nestedness (arrays)*: If we have an array, just searching for `{"Theories": "Special relativity"}` works, this will check if ”Special relativity” is in the array ”Theories”.
Many other operators exist: $nin, $eq, $ne, $gt, $lt, $lte,...
There also exist aggregation and pipelines (like in Spark). But there is an easier way to do this (see section 13).



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

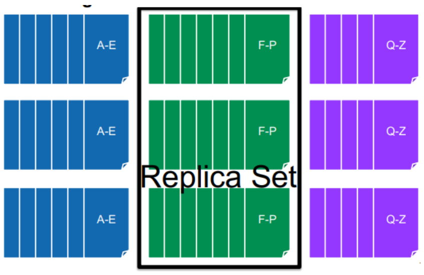
Figure 35: Aggregation and pipelines

Desc	Low level query
Insert	db.flights.insertOne({"Name": "Einstein", "Theory": "Relativity"})
Update	db.scientists.updateMany({"Name": "Einstein"}, $set: "Century": "20")
Remove	db.scientists.deleteMany({"century": "15"})

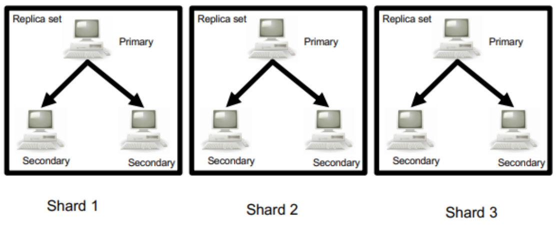
Data defintion: Observe that we can find existing docs and update them. This is not possible in Spark and HDFS.
Document stores are document write atomic (like HBase: row atomic). Two users can't write to the same document simultaneously.

12.4 Architecture

Principles of big data: Shard the data, Replicate the data, Buy lots of cheap hardware. MongoDB does this as well. However, typically we don't have lots of shards and only few machines.



(a) Clustering model



(b) Clustering on the physical level

We, again, have a master-slave architecture. One master replica and several slave replicas. An important feature of document stores is the ability to modify data (not possible in Spark, MapReduce, ...). When writing, the write goes to the master, which will execute it and forward the write to all slave replicas. The slaves send ACKs. However, in order to speed writing up, the master only waits for a subset of slaves to send their ACK (the number of ACKs to wait for can be specified in MongoDB).

12.5 Indices

Imagine a collection of billions of objects and we start a query. Query can be of two different types:

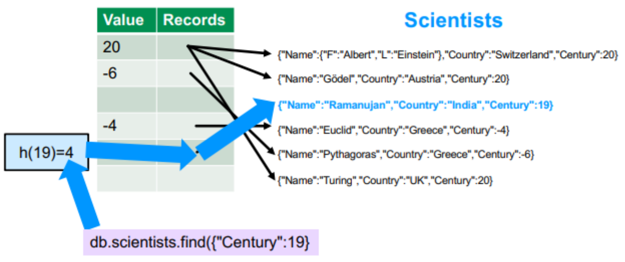
- Point query (= highly-filtering queries): returns only very few objects from the collection.
- Not highly-filtering queries: returns lots of objects from the collection.
- Range queries: return objects that fit into some range.

Indices are mainly interesting for point and range queries since returning many objects is slow anyways. Thanks to indices, point queries can run in milliseconds. On Spark, which does not have indices, this would take a while (seconds to minutes).

Indices - the principle: Indices are like an index in a book. They directly point to an entry (object) and thus make lookups faster. Two kinds of indices exist in MongoDB: hash indices and B+ trees.

12.5.1 Hash indices

Hash indices are the fastest and can be created on some field by `db.scientists.createIndex({"Century": "hash"})`. The procedure works as follows: the value of the specified field is hashed for every object. In a hash table, we then map the hashed value to an array of records that point to all objects that have the specified field equal to the value of that hash table entry. This allows to quickly find all objects that have a specified value in the field (e.g. all objects with "Century" 19 → lookup h(19)).



```
{"Name":{"F":"Albert","L":"Einstein"},"Country":"Switzerland","Century":20}
{"Name":"Gödel","Country":"Austria","Century":20}
{"Name":"Ramanujan","Country":"India","Century":19}
{"Name":"Euclid","Country":"Greece","Century":-4}
{"Name":"Pythagoras","Country":"Greece","Century":-6}
{"Name":"Turing","Country":"UK","Century":20}
db.scientists.find({"Century":19}
```
Figure 37: Hash indices (the fastest)

Limitations of hash indices:

- No support for range queries
- Hash function not perfect in real life
- Space requirements for collision avoidance

12.5.2 B+ trees indices

B+ trees are another way to index and they solve some issues of hash indices, but they are slower. Consider the B+ tree example in Figure 38 where we group words together such that we have single disk read → less latency. B+ trees have several keys per node. These values define intervals (e.g. 2 keys: 3 intervals = 3 children, 4 keys: 5 intervals = 5 children). For each interval, there is a child. In B+ trees, all leaves have the same depth and all non-leaf nodes have between d+1 and 2d+1 children (the root can have less).

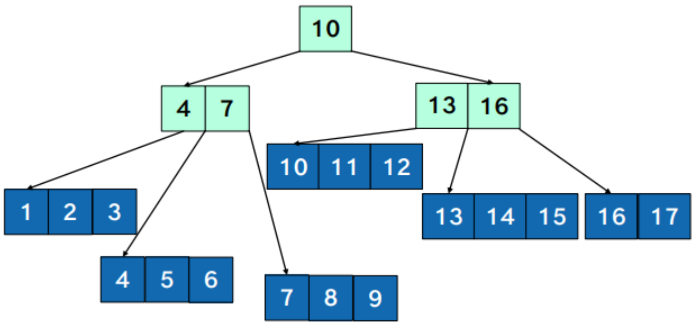


Figure 38: B+ tree example

Procedure: We keep inserting into an empty tree. First, all values just go in the root. Once we have more than 4 values, we split the root and create two leafs and the last value before splitting is added as new key. This is done recursively - every time a leaf has too many values, it is split. If the parent has not yet too many keys, a new key is added to the parent and the leaf is split into two. If the parent already has full keys, the parent is split and a new root is added (see slides for example construction).
B+ trees can then be used to build an index, where the values for a field are entered in the tree and each value point to a set of objects. When executing a query, we just have to walk the tree. Further, B+ trees naturally encode intervals, which speeds up range queries.

MongoDB has a special field `_id`, which always exists for any object. MongoDB always builds an index on this field, which allows for very fast lookups on the _id field. We thus have two types of indices in MongoDB:

- Primary: _id
- Secondary: other fields. MongoDB will only build an index for other fields if instructed to do so.

Querying without an index requires scanning through the entire collection in memory, which is very slow. With an index, we can directly *prefilter* the collection without even looking at the collection. Building the index takes a while in the beginning, however once it is done, queries are very fast.

12.6 Examples

Index (hash): db.scientists.createIndex({"Name.Last": "hash"})
Query: db.scientists.find({"Name.Last": "Einstein"}) - very fast.

Index (hash): db.scientists.createIndex({"Name.Profession": "hash"})
Query: db.scientists.find({"Profession": "Physicist", "Theories": "Relativity"}) - very fast, collection can be prefiltered on "Profession" (using indices) and then postfiltered in memory on "Theories".

Index (compound (only B+-tree!)): db.scientists.createIndex({"Birth": 1, "Death" : 1})
Query: db.scientists.find({"Birth": 1887, "Death": 1946}) - very fast.

Index (compound (only B+-tree!)): db.scientists.createIndex({"Birth": 1, "Death" : -1})
Query: db.scientists.find({"Birth": 1887}) - faster as well, even though it only looks at "Birth" not "Death".

Index (compound (only B+-tree!)): db.scientists.createIndex({"Birth": 1, "Death" : -1})

Query: db.scientists.find({"$gte" : 1980}) - very fast.

Index (compound (only B+-tree!)): db.scientists.createIndex({"Birth": 1, "Death" : -1})
Query: db.scientists.find("Death" : 1887) - *not fast* because the index first sorts on "Birth", then on "Death". The query only checks the second field.

13 Querying trees

In document stores, we have large collections of small trees and we can query with nested and heterogeneous data. We now want a querying layer above the data store (document store) in the stack.
Data independence: We want a logical data model with a query language independent of the physical implementation. The physical layer can be changed while the logical layer stays. For ecample, Spark SQL allows to query Spark using an SQL-like language, while on the physical level, Spark translates the SQL queries to transformations and actions. Due to data independence, this translation needs to be done only once, not everytime by the user.

However, we've seen that nested and heterogeneous data is still a problem for SQL. Using DataFrames, Spark SQL provides possible ways to deal with nestedness (explode arrays) or heterogeneity (convert everything to string) but the approach is suboptimal. We want a language that is similar to SQL but handles nestedness and heterogeneity natively → **JSONiq** (others exist).

Declarative language: declare *what* you want (e.g. I want a cake) - vs. imperative language: declare how you want it (e.g. put sugar, flour, etc. in and bake for 30min).
Function language: everything transforms (functions) several instances into an instance.
Set-based languages: always returns collections (e.g. rows).

Language ecosystem:

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

And more recently, we have languages that query collections of trees: UnQL, N1QL, GraphQL, SQL++, ArangoDB Query Language (AQL), MRQL, Asterix Query Language (AQL).

13.1 JSONiq Data Model (JDM)

JSONiq is a functional language that always operates on sequences of items (∘, ∘, ∘, ∘, ∘, ∘, ∘, ∘, ∘).
Sequence of items:
- Heterogeneous: each item can be different (deviation from SQL!)
- Denormalized: some items in the sequence may be another tree.
- Sequence of one item = item itself: (∘) = ∘ (different from e.g. Python where $1 \neq [1]$)
- Sequences are flat: ((∘, ∘), ∘) = (∘, ∘, ∘). You can only concatenate sequences, not nest them. Nesting is only possible with items.

Items:
- Atomic (int, bool, date, ...)
- Object (mapping)
- Array (ordered list)
- Function (encodes a function)

13.2 JSON Navigation (JSONiq, Rumble)

Query	Description
json -doc(”myfile.json”)	Open a JSON doc and returns an object (i.e. an item), which is a sequence of one item.
json -doc(”myfile.json”).countries	Navigate the object using ”.”, returns array = item = seq. of items
json -doc(”myfile.json”).countries[]	Unbox an array with []. Returns a sequence of all array items.
json -doc(”myfile.json”).countries[].name	Returns sequence of country name strings
json -doc(”myfile.json”).countries[].code	Returns sequence of country code strings
json-doc(”myfile.json”).countries[][$$.code = ”CH”]	$$: special variable for filtering. $$.code: look at code field for every object and compare it (= SELECTION)
json-doc(”myfile.json”).countries[][$$.code = ”CH”].name	Selection and Projection

Table 1: JSONiq navigation

13.3 Construction

- Strings: ”foo”, escaping: ”This is a line\n and this is a new line” (like in JSON).
- Numbers: integer: 42, decimal: 3.1415926, double: -6.022E23. (float exists as well).
- Booleans: true, false
- other numbers (cast type): long(”1234567890123”), int(”12345678”), short(”32000”), byte(”200”), non-NegativeInteger(”42”)
- Dates and times: date(”2013-05-01Z”), dateTime(”2013-06-21T05:00:00”), time(”05:00:00-08:00”), dateTimeStamp(”2013-06-21T05:00:00Z”), gYear(”2019”), gMonth(”–11Z”), gYearMonth(”2019-11”), gMonthDay(”–11-26+02:00”), gDay(”—26”)
- Durations: dayTimeDuration(”PT1D2H”), yearMonthDuration(”P1Y2M”), duration(”P1Y3MT3D”)
- Binaries: hexBinary(”0CD7”), base64Binary(”RGFzc2Vs”)

Construction:

- JSON (objects and arrays): `[{"foo": "bar"}, {"bar": [1, 2, true, null]}]`. You can just copy any well-formed JSON to JSONiq for construction.
- Sequences with a comma: `[2, 3], true, "foo", {"f": 1 }`
- Sequences with a range: `1 to 100`

13.4 JSONiq: Basic Operations

Arithmetics: All operators have the form of *seq. of items* ∘ *seq. of items*: 1 + 1, 42 - 1-, 6 * 7, 42.3 div 7.2 (division), 42 idiv 9 (integer division), 42 mod 9 (modulo). Type conversion is automatically taken care of. Don’t use /. % for div and mod!
Empty sequence: () + 1 = ()
Types: ”foo” + 2 ERROR
Cardinality: (3,4) + 2 ERROR
Strings:

- concatenation: ”foo” —— ”bar”, concat(”foo”, ”bar”), string-join((”foo”, ”bar”, ”foobar”), ”-”)
- sub-string: substr(”foobar”, 4, 3)
- length: string-length(”foobar”)

If you want to e.g. concat ”foo” and 2, you need to first cast 2 to string. JSONiq has a full string library.
Value Comparison

- equality: 1 eq 2 (=false)
- inequality: ”foo” ne ”bar” (=true)
- greater than: 234 gt 123 (=true), 234 ge 123 (=true)
- less than: 42.3 lt 7.2 (=false), 42.3 le 7.2 (=false)
- empty sequence: () le 2 = ()
- types: ”foo” le 2: ERROR

General Comparison

- Existential quantification (tells if at least one item in the sequence is equal to the righthand item): (1,2,3,4,5) = 1 (=true) ((1,2,3,4,5) eq 1 would throw an error)

- General comparison: (1, 2, 3, 4, 5) ¡ (2, 3, 4, 5, 6) (=true, at least one item in the left sequence is smaller than at least one item in the right sequence: 1¡3), (1, 2, 3, 4, 5) ¿= (6, 7, 8, 9, 10) (=false, no item in the left sequence is greater or equal at least one item in the right sequence), (1, 2, 3, 4, 5) ¿= ("foo", 7, 8, 9, 10) (ERROR)

This structure is driven by practical use cases, which allows us to quickly check if a large collection contains a match.

Logics

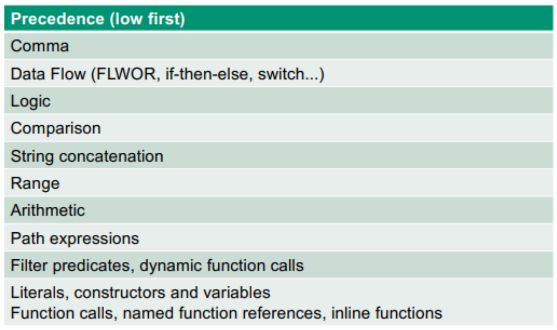
- conjunction: 1+1 eq 2 and 2+2 eq 4 (true and true)
- disjunction: 1+1 eq 2 or 2+2 eq 4 (true or true)
- not: not(100 mod 5 eq 0) (not true)
- Effective boolean value of non-booleans:
  - true: "foo", 42, ({"foo": "bar"}, 1)
  - false: "", 0, ()

13.5 Composability

Rule of thumb: Any Expression can be the operand of any other expression.

- (1 + (({"b":["a": **2+2}**]).b[].a)) to 50
- (1 + (({"b":[{"a": 4}]})**.b**[].a)) to 50 (access field b and get value (unboxed array))
- (1 + ([{"a": 4}][].a)) to 50

Presedence: Use parentheses to override or when in doubt!



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

Figure 39: Precedence

JSON construction using functions:
`{"attr": string-length("foobar"), "values": [for $i in 1 to 10 return long($i)]}` becomes `{"attr": 6, "values": [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}`.

13.6 Data Flow (XQuery, JSONiq)

Conditional Expressions:

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

Switch expressions:

```
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"
```

Try Catch Expressions:

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

13.7 FLWOR Expressions

Let clauses: bind seq. of items to variable. This is not an assignment! After the return, $x no longer exists.

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

For clauses: $x is first bound to 1, then to 2, then to 4, ... Return is sequence of integers (left), sequence of objects (right), ...

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

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

Where clauses: The where clause is applied to each item in the for clause

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

Order by clauses: Each object is bound to $x and then sorted by some field (default: ascending)

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

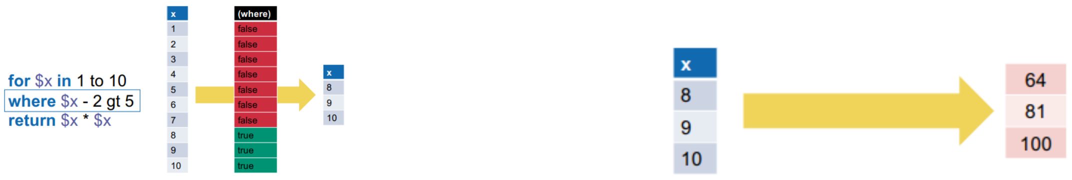
Group by clauses: Group countries by continent. Note: you don't have to aggregate if you group! In SQL, we have to.

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

13.7.1 Tuple stream semantics

JSONiq was developed by people that developed SQL → JSONiq is more precise and solves imperfections of SQL.

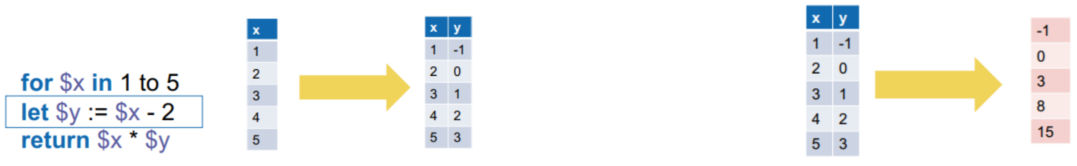
Tuple stream semantics: Each item in a for loop is bound to the variable, which gives us a set of tuples. We essentially have a binding of $x to all items over which $x iterates. Expressions such as *where* or *return* can thus be evaluated against the set of tuples.



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

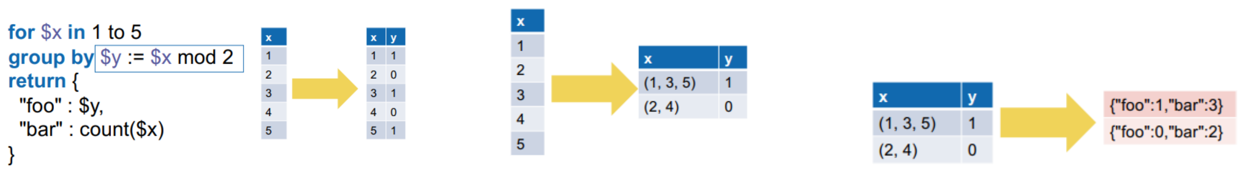
x: 1 2 3 4 5 6 7 8 9 10
(where): false false false false false false false true true true
x: 8 9 10
x: 8 9 10
64 81 100

Figure 40: Each item is bound to $x, i.e. we have tuples (x,1), (x,2), ... The where clause is then evaluated against each tuple and returns tuples that fulfill the expression. The return then evaluates against these tuples.



```
for $x in 1 to 5
let $y := $x - 2
return $x * $y
```

Figure 41: For each tuple (x,1), (x,2), ..., we get a new tuple for y. **Observe:** the data may be heterogeneous, but the bindings are always homogeneous!



```
for $x in 1 to 5
group by $y := $x mod 2
return {
  "foo" : $y,
  "bar" : count($x)
}
```

Figure 42: After the group by, we have x tuples with sequence of all items with the same mod result.

13.8 Types

Simple types: by name e.g. integer
Complex types: by kind e.g. object

Cardinality:

- integer: exactly one
- boolean? at most one
- array+: at least one
- object*: any

Type usage:
You can (but don't have to) specify a type for a variable. For example:

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

```
let $x as object := { "foo" : 1, "bar" : 2 }
return $x.foo + $x.bar
```

You can check if a type is matched in a return (throws an error if type is not matched). For example:

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

You can change the type of a return:

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

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

You can check if an item is of instance or can be casted to an instance:

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

```
3.14 castable as double
```

You can define functions and specify the types of function arguments (not necessary though, the default type is item*)

```
declare function is-big-data($threshold as integer, $objects as object*) as boolean {
  count($objects) gt $threshold
};
is-big-data(1000, collection("countries"))
```

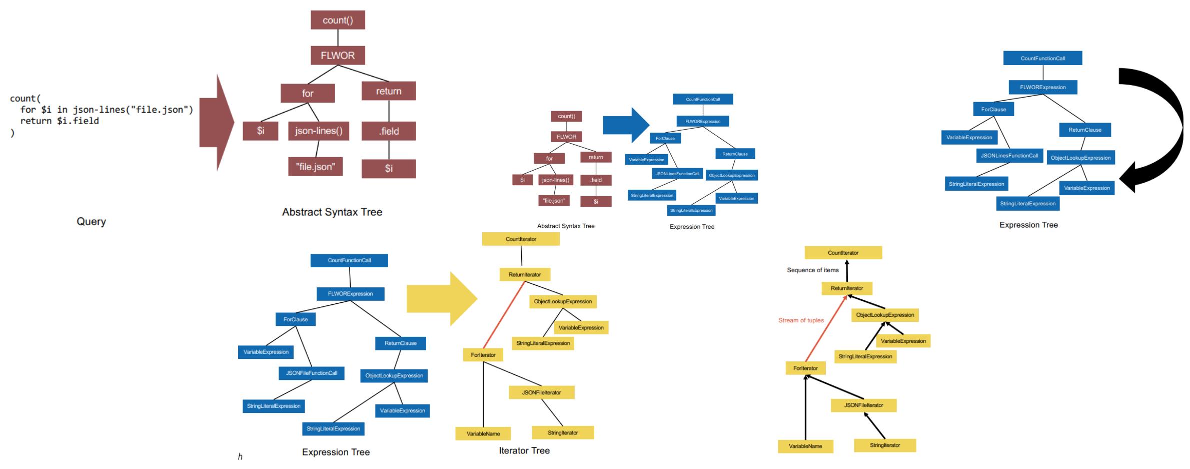
13.9 Architecture of a query processing engine

A query needs to be compiled (=text of code needs to be translated into an AST (abstract syntax tree)). Steps of compiling a JSONiq query:

1. Parsing: Parse the text query and translated it into an AST (abstract syntax tree).
2. Translating: visit the AST and turn it into an expression tree.
3. Optimization: optimize the expression tree.
4. Code Generation: Turn the expression tree into a runtime iterator tree (actual execution). Iterator: we have a language on sequences, we thus always iterate on sequences.
5. Execution: Core tuntime of engine

The actual execution can be of two types:

- Materialized execution: The entire sequence is computed synchronously before returning. For example, in a for loop, every single loop is executed and each tuple is produce. Only once all tuples are produce is the return applied. Imagine a for loop over 10^9 objects.. This takes lots of memory and is very inefficient. Streaming is more efficient.
- Streamed execution: Directly return after materializing one single object and then forget it. For example, in a for loop produce one tuple and directly return. In order to make this sequential process even faster, one can use parallelization. We can split the data into shards and do the execution in parallel. This



```
count(
  for $i in json-lines("file.json")
  return $i.field
)
```

Figure 43: Architecture of a query processing engine.

parallel execution strategy is orthogonal to the runtime iterator tree (is just a query plan). Rumble is constantly switching between local and parallelized. **Note:** this is not always possible! Selection, projection is ok, but group by, order by is not possible with streaming. In some cases, we have no choice but to use materialized execution.

14 Performance at large scales

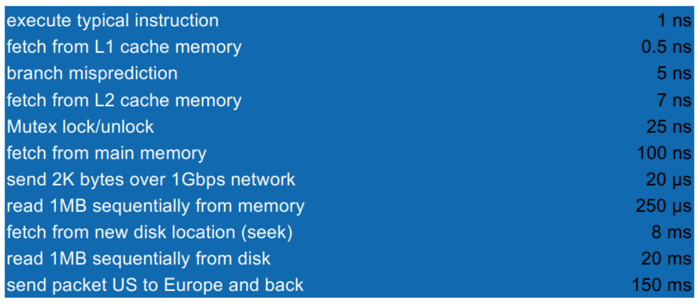
14.1 Speed up

How can we make jobs faster? We need to identify **bottlenecks**! Find the overloaded part in the execution (bottleneck) and improve that part.
Many different sources for bottlenecks exist: memory (full memory, paging), CPU (100% CPU load), disk I/O (slow read/write), network I/O (slow transmission). Typically, you can look into a cluster and identify the overloaded components. There is no point in improving a component that is not overloaded when another component is. E.g. no point in adding more memory when the CPU is the bottleneck.

MapReduce and Spark solve the (disk) I/O-bound bottleneck. If other bottlenecks are present, we might not need MapReduce or Spark.

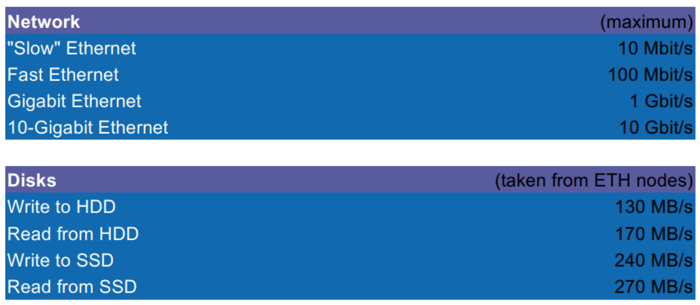
14.2 Measurements

Latency = "When do I start receiving data?"
Throughput = "How fast can we transmit data?"
Total response time = Latency + Transfer time



execute typical instruction	1 ns
fetch from L1 cache memory	0.5 ns
branch misprediction	5 ns
fetch from L2 cache memory	7 ns
Mutex lock/unlock	25 ns
fetch from main memory	100 ns
send 2K bytes over 1Gbps network	20 µs
read 1MB sequentially from memory	250 µs
fetch from new disk location (seek)	8 ms
read 1MB sequentially from disk	20 ms
send packet US to Europe and back	150 ms

(a) Typical latencies



Network	(maximum)
"Slow" Ethernet	10 Mbit/s
Fast Ethernet	100 Mbit/s
Gigabit Ethernet	1 Gbit/s
10-Gigabit Ethernet	10 Gbit/s

Disks	(taken from ETH nodes)
Write to HDD	130 MB/s
Read from HDD	170 MB/s
Write to SSD	240 MB/s
Read from SSD	270 MB/s

(b) Typical transfer rates

$Speedup = \frac{Latency_{old}}{Latency_{new}}$

Amdahl's law: When you fix a bottleneck, you only fix one part of the system. The rest of the system that doesn't depend on this will not improve. $Speedup = \frac{1}{1-p+p/s}$

Gustafson's law: Gustofson assumes that once we have a speedup, people will just come with more data (more realistic). What stays constant is the computing power. $Speedup = 1 - p + sp$

with p how much of the execution is parallelizable and s the speedup on the parallelizable part.

Example ($p = 0.3, s = 2$): Amdahl: $Speedup = \frac{1}{1-0.3+\frac{0.3}{2}} = 1.18$, Gustafson: $Speedup = 1 - 0.3 + 2 * 0.3 = 1.3$

14.3 Tuning

What can we act on?

- Number of machines (scale out): easy way out but expensive
- Machine features (CPU, Memory, ...) (scale up): expensive
- Code: most important and efficient tuning
- Size of chunks
- Storage format
- Network usage
- Architecture

How to avoid unnecessary tuning?

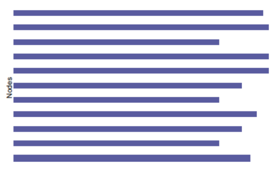
- Avoid memory scale-up: Search for classes that are instantiated billions of times and make code more efficient. Are there unused fields? Can I find more memory-efficient data structures (TreeMap, HashMap, ArrayList, LinkedList)? Do I have redundancy? Do I really need inheritance and RTTI (mechanisms in OO languages at runtime)?
- Avoid CPU scale-up: Search for gigantic loops (e.g., functions passed to transformations). Am I (over)using method calls? (spaghetti bolognese of one-line functions?). Am I calling overriden methods? (switch is faster!). Am I using class hierarchies? (interfaces are faster). Am I using using instanceof tests and casting all over the place? Am I using exceptions in normal setups?
- Avoid Disk I/O scale-up: Use efficient formats (e.g. BSON)! (except if one-off query). Use compression (It's read out of the box). In the storage format =, only the actual number of bits matter... compromise efficiency over reusability.
- Avoid network I/O scale-up: batch process, pre-filter, pre-project, pre-aggregate. Avoid/minimize shuffling and minimize the amount of data shuffled around by preparing it.

Network usage:
Example with Spark: 8 nodes, each with 16 cores and 128GB of memory. How many resources per container?

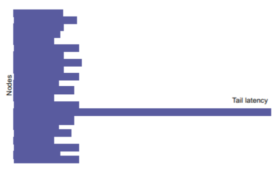
1. 1 container per machine (keep 1 core for application master)
2. 1 container per core (16 conrainers per machine)
3. Middle ground: 3 containers per machine (5 cores, 30GB memory). This is usually a sweet spot.

Code: Improve code
Size of chunks: if we have large chunks, in real life there will always be some executor that takes much longer than the others. If we have large chunks, we can't split them up and all executors have to wait for the last one to finish. Solution: smaller chunks (tasks). Then, fast executors can execute more tasks. *BUT:* don't push it too far. If the chunks are too small, the latency will dominate and make things inefficient again.
Network usage: Network bandwidth is limited. Too much transmitted data leads to a bottleneck. Thus, keep shuffling to a minimum.

14.4 Tail latencies



(a) Ideal execution times



(b) Execution times in reality

In reality, not all executors finish at the same time, there is always going to be one that takes a lot longer (when working with large scales). Why?

- Resources are shared on a machine (lots of daemons run on one machine).
- Resources are shared globally.
- Background daemon creates hiccups.
- Queues
- Power limits
- Garbage collection
- Energy management

All this can happen and we cannot control it.

Example: SLA, 10ms typical latency, <1s in 99% of the cases.

- 1 node: very high probability that it's fast (0.99), and we can just retry if it's not.
- 10 nodes: it's bad if at least *one* of the 10 nodes fails. Probabilities sum up: $10 * 0.01 = 0.1$, thus one node fails with probability 10% already.
- 100 nodes: fails with 63% probability
- 1000 nodes: fails with 99.9999% probability

A better SLA will not help, the phenomena will just reappear with a large amount of nodes. The tail latency phenomena is a physical issue and will thus always appear.

Hedge requests: just execute every task twice at the same time and pick the faster one. For the tasks that are too slow, just launch a new task.

15 Wrap up

15.1 10 Design Principles of Big Data

1. Learn from the past: Tables won't disappear, data shapes matter
2. Keep the design simple: Simplicity allows to scale up and design large systems. Best example: key-value model.
3. Modularize the architecture: Build components on top of each other. Build a stack layer by layer. Every data shape follows the same principles (paradigm, data model, syntax, query language, etc.)
4. Homogeneity in the large: We mostly look at very large collections of small objects. This makes it easy to shard.
5. Heterogeneity in the small
6. Separate metadata from data
7. Abstract logical model from its physical implementation. Allows to reuse things, e.g. store a shape on top of a different shape (cube on top of tables).
8. Shard the data
9. Replicate the data
10. Buy lots of cheap hardware

15.2 Design choices

You may have to:

- Learn an existing product
- Choose which product to use
- Design a whole new technology or product

It's very fundamental to be aware of your data: data shape, how much data, what do I want to do with my data.
Do I *really, really* need to reinvent the wheel...? Learn from the past.

15.3 Data in 2030?

- MapReduce and Spark (or the like) will be invisible. Only engineers will directly use it. The rest of people will use a higher level query language.
- Data shapes will be central (and new shapes will be discovered?)
- Querying all data shapes will be standardized (and maybe even unified?)
- UIs will be the norm for end users (maybe even superseded by ML and AI)
- We will literally talk to our databases (Siri, Alexa, etc.)
- Data will be fully fungible (Web of Data 2.0)
- Data and money will have converged
- People will control and manage their own data like their own property (already happening)
- Machine Learning will leverage cutting edge database technologies (already happening)
- Databases will leverage cutting edge ML technologies (already happening)
- Quantum computing will become ordinary
- Mankind will turn the entire universe into a big database (after all, it's already a computer)

References

[1] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)* (2010), Ieee, pp. 1–10.