Big Data for Engineers (Spring 2020)

Summary

Author:
Yannick Merkli

18 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.

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 valid value for the attributes.
Atomic integrity: 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.
- 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**	$\rightouterjoin$
Cartesian Product	$\times$	**Union**	$\cup$	**right outer Join**	$\leftouterjoin$
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:

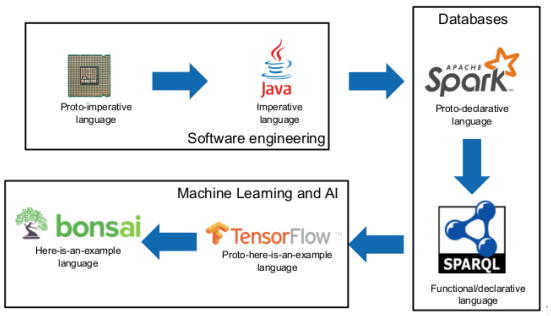
1. Take the Cartesian product $R \times S$
2. select those tuples satisfying condition θ

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)



(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).

- 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.

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!

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:

```
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 $\Longrightarrow$ **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:

```
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*:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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 $\implies$ the `HAVING` clause then follows the `GROUP BY`. Example:

```
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 $\implies$ already saw example in set operations.

- Subqueries can return a single constant $\implies$ 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 $\iff$ R is not empty
- `s IN R` is true $\iff$ s is equal to at least 1 value in R. Can also use the `NOT IN` operator
- `s > ALL R` is true $\iff$ s greater than every value in R.
- `s > ANY R` is true $\iff$ 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)` $\implies$ 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:

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

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

```

```
-- 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.

```
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:

```
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? $\Longrightarrow$ inconsistencies. Same happens with aggregations. This leads to the following theorem about views in SQL:

Theorem 3.1. A SQL View is updateable $\Longleftrightarrow$ 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.

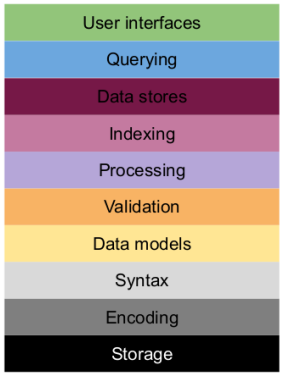


Figure 2: The data stack

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, Elastic-Search, 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.

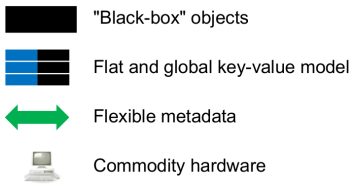


Figure 3: Object storage

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.

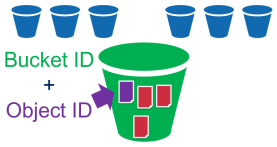


Figure 4: S3 model

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)

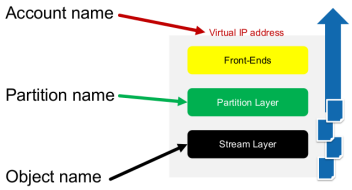
Azure thus has one more layer of indirection: account, container, blob. Further, Azure doesn't view an object as a blackbox, it let's you see inside and differentiate between 3 types of blobs: block, append, page. The account name maps to a virtual machine which is responsible for my data. The partition 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:

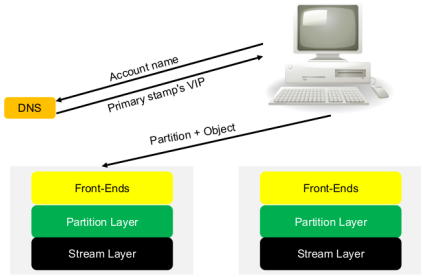
- Intra-stamp replication (synchronous): Replication within the streaming later (i.e. within the same location (data center)).
- Inter-stamp replication (asynchronous): Replication between different partition layers (i.e. between different locations (data centers))

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

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,...) → a lot of small services that each do one particular thing.
- Azure mindset: Azure has bigger services that do more → 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 → lots of files, but single files are not extremely big, just large (e.g. 5TB for S3).
  → Object storage + Key-value model
- **A large amount of huge files** (millions of PB files): We need something new for this...
  → 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[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:

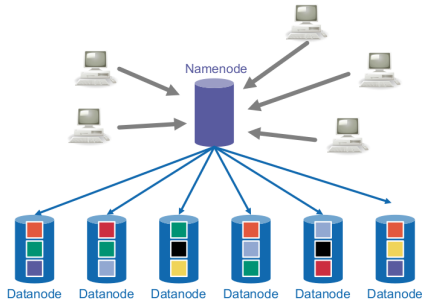
- Distributed File System (HDFS) (inspired by Google's GFS)
- MapReduce (inspired by Google's MapReduce)
- Wide column store (HBase) (inspired by Google's BigTable)

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 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.
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 $file \rightarrow \{blocks\}$.
- 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

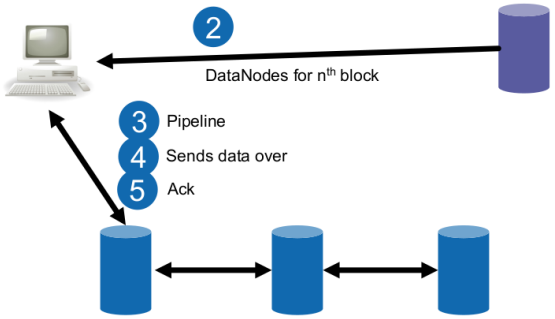
Data transfer protocol: The data transfer protocol handles *client-datanode* communication and is used by clients to read actual block content. The client knows which datanode to contact for a given block - it got that information from the namenode.
Client reads a file:

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 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 replicates further asynchronously

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



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).

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. 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

5.8 Using HDFS

References