Big Data – from clay tablets to integrated lakehouses

Book · May 2021

CITATIONS
0

READS
687

1 author:

Ghislain Fourny
ETH Zurich
30 PUBLICATIONS **108** CITATIONS

SEE PROFILE

Big Data

From clay tablets to integrated lakehouses

by Ghislain Fourny

September 12, 2022

Big Data by Ghislain Fourny

Revision history for the First Edition
2021-05-27 First Release

ISBN 979-*********

Contents

Chapter 1

Introduction

1.1 Scale

Humankind noticed very early the Sun, the Moon and the stars in the sky. It also noticed that some celestial bodies, which it called planets, were moving on strange paths. But it is only relatively recent in our history that we understood that the planets are orbiting around the Sun, just like the Earth, that planets have satellites too (Phobos and Deimos for Mars, etc).

The Solar system alone forces us to experience larger scales: meters for us humans, kilometers when we drive cars, Megameters when we consider the size of our Planet, Gigameters when we consider the distance to the Sun. Jupiter brings us to a Terameter, and the entire Solar system fits in a Petameter.

Even more recently, we discovered that the spiral-shaped nebulas in the sky are actually galaxies just like this large line of stars in the sky we called Milky Way. It pushed us even farther: not just Exameters, but even Zettameters just for one galaxy and about 400 Yottametters for the entirety of what we are able to see.

I am still not sure today what is the most amazing: is it that the visible universe is large beyond anything we can imagine with its 400,000,000,000,000,000,000,000,000 meters in size? Or is it that this printed number, on this page, is so small and that our standardized prefixes are more than enough to express this size?

Our experience with the growth of data, somehow, is different. We grew accustomed to bytes, kilobytes, Gigabytes, Terabytes in our everyday life without realizing that, with this analogy with astronomy, computers went to Jupiter in just a few decades of existence.

Astrophysicists going to such large scales, however, quickly noticed that, to understand the Universe at such large scales, they also need to understand it at much smaller scales.

Data Science is not really different from Physics, in this and many other respects. Data at the scale of Exabytes can only be understood and tamed if we also model it at the tiniest scales.

Another aspect of Data Science that makes it akin to Physics is that, unlike Mathematics and Computer Science, Data Science is about studying what the world *actually is*, as opposed as *how it could have been*, a phenomenon also called contingency in modal logic theory. Many processes in Data Science (e.g., hypothesis testing) are much closer to physics and the discovery of theories than one would think.

This book provides an introduction to the beautiful world of Big Data as I currently teach it in the lectures *Big Data* and *Big Data for Engineers* at ETH Zürich.

1.2 A short history of databases

1.2.1 Prehistory of databases

Let us now quickly sketch a short history of databases.

Data storage, in fact, predates us: all living beings have at the core of their cells DNA, which is nothing else than data storage based on bits with 4 possible values (A, T, G, C) rather than 2.

But the first data storage that was in the control of humans was simply their brains. Humans recollect events, and transfer this knowledge from generation to generation by simply speaking, telling and singing stories to the younger generations. This, of course, is prone to distortions in the information, loss of information, and the introduction of errors in the information.

The first revolution happened thousands of years ago with the invention of writing. Writing was first made on clay tablets. Once dry, information written thereon can survive for at least thousands of years. This is why we have much more knowledge about humankind after the mastery of writing: we could decode them and have access to pristine information from these older times.

But what is truly amazing about clay tablets is that they were not only used for writing texts: we found clay tablets with... relational tables, the earliest known example being Plimpton 322, which is 3,800+ years old. This alone illustrates how natural relational tables are to human beings, and was an omen for what was to come centuries later.

The second revolution was the invention of the printing press. Before it, making duplicates of documents was very costly and, above all, required manual work, copy by copy. This also slowed down the spread of knowledge and information. With the printing press, it became straightforward to make large numbers of copies of the same text, which led in particular to the Golden age of the written press.

The third revolution was the invention of modern, silicon-based computers. Computers accelerated the processing of data.

1.2.2 Modern databases

The birth of database management systems in our modern understanding is often placed in 1970. This is the year where a seeding paper by Edgar Codd was published, introducing the concept of data independence. In the early days of computers, people were managing data on storage devices themselves, which was very demanding in resources. Edgar Codd suggested that a usable database management system should hide all the physical complexity from the user and expose instead a simple, clean model. He further suggested that this model should be based on tables, which gave birth to the relational model and the relational algebra.

More recently though, the explosion in the quantity of the data we produce brought this model to its limits a few decades later, with the emergence of modern systems such as key-value stores, wide column stores, document stores and graph databases.

1.3 The three Vs of Big Data

The recent evolution in the domain of large-scale data processing over the past two decades is often explained in terms of the three Vs: Volume, Velocity, Variety.

1.3.1 Volume

First, the volume of data stored worldwide is increasing exponentially. The total amount of data stored digitally worldwide is estimated to be getting close to 100 ZB as of 2021 (zettabytes). This is a 1 followed by 23 zeros. There are many reasons for this: we have automated the collection of data (sensors, logs, etc), and we also have all the necessary space to store it without needing to delete it. Many companies actually keep all their data just in case they might need it later, although this is likely to become less commonplace in the future with new regulations such as the European GDPR.

Prefixes have been standardized for large powers of ten, all the way to 10^{24}. They should be known by heart:

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

They can be used in particular for bytes (B), i.e., 1,000,000 B = 1,000 kB = 1 MB, but also in conjunction with any other units in the international system.

Computer Scientists have often used prefixes to expressed powers of 2 rather than powers of 10, using the coincidence that 2^{10} is very close to 10^3. For example, 1 kB actually meaning 1,024 B and not 1,000 B, although this was an abuse of notation. In order to make it official, prefixes have been defined for this purpose, even though many people nowadays continue to confuse the two series. It is, of course, useless to remember the exact values of the powers of 2, but one needs only understand that each prefix corresponds to a 2^{10n}.

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

1.3.2 Variety

Second, new shapes of data have emerged. While Edgar Codd suggested to focus on data organized in tables because it is the most intuitive for human beings, more recent systems involve data models relying on four more shapes:

- trees: trees correspond to denormalized data often found in formats such as XML and JSON, but also more recently such as Parquet, Avro, etc.
- unstructured: a lot of data is simply accumulated in a raw form such as text, pictures (pixels), audio, video, etc.
- cubes: in the 1990s, data cubes became very popular in the field of business analytics, with the primary use case of analyzing sales data and producing reports for the top management and strategic decision making.

- graphs: there are database systems (such as neo4j, Oracle PGX, etc) that expose data as a graph with nodes and vertices. Graph shapes are especially useful when the use case is focused on the efficient traversal of data (equivalent to joins in tables, which are rather slow and expensive in comparison to other operations).

The relevance of these new shapes is increasing, and it is important to keep in mind that it is desirable to keep a clean, logical and abstract view of the data for all of them, not only tables. It is commonly said for example that tree-like data undesirably exposes physical layouts to the user, but this statement is inaccurate and this misconception is due to the fact that the principle of data independence has not yet found its way in all database management systems with a focus on tree shapes (e.g., document stores), which unlock immense power to normalize and denormalize data at will.

1.3.3 Velocity

Third, a distortion has appeared between how much data we can store in a given volume, how fast we can read it and with which latency. This distortion carried so many challenges that this drove the emergence of many data processing technologies that we know today (MapReduce, Apache Spark, etc.).

Today, data is generated automatically: by sensors, but also by logging how people use websites or applications: it has simply become a byproduct of human activity.

In order to understand what happened in the past 20 years, we need to look at three factors:

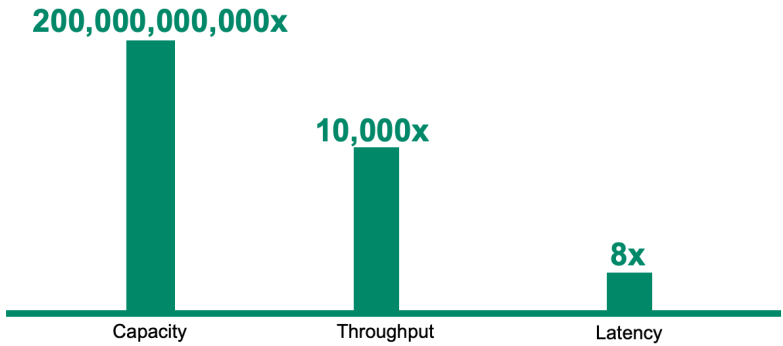
- Capacity: how much data can we store per unit of volume?
- Throughput: how many bytes can we read per unit of time?
- Latency: how much time do we need to wait until the bytes start arriving?

The earliest commercially available hard drive dates back to 1956 and was the IBM RAMAC 350. It had a capacity of 5 MB, a throughput of 12.5 kB/s and a latency of 600 ms. Its dimensions were 1.7m by 1.5m by 70cm, which would be enormous for today's standards.

Nowadays, we can hold a thousand times this much information in our fingers with a UBS stick. But more precisely, let us look at the largest available hard drive in 2021, which is as far as we are aware the Western Digital Ultrastar DC HC650. It has a capacity of 20 TB, a throughput of 250 MB/s and a latency of 4.16 ms. Its dimensions are 10.1cm by 14.7cm by 2.6cm.

Now, if we compare how these quantities have evolved:

- Capacity per unit of volume has increased by a factor of 200,000,000,000.
- Throughput has increased by a factor of 10,000.
- Latency has decreased by a factor of a mere 150.



Let us make an analogy: take a book with 600,000 words, which an average human can read with, say, 1,000 words a minute. The book can be read in 10 hours.

Let us now imagine that, in two centuries from now (which is more than the 70 years considered for hard drives), the size of books is multiplied by the same factor of 200,000,000,000, making it 120,000,000,000,000,000. And that the speed at which we read it increased by the same factor of 10,000, making it 10,000,000 words per minute. Now, it will take 22,800 years to finish the book.

Yes. We Are In Big Trouble.

However, a key insight is: imagine we spread the read to 20 million people over a social network: then, if everybody takes one page and reads it in parallel, we are back to 10 hours. This is called parallelization, and this is the first technique used by modern data processing platforms.

The second technique addresses the growing discrepancy between throughput and latency: batch processing. Rather than processing records one by one, batches are created and the processing is done batch by batch. Also this is used by modern data processing platforms, where batches are commonly known as tasks.

Which leads us to my attempt to define 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.*

1.4 Applications

There are numerous applications to Big Data. One of the paramount consumers is the field of High-Energy Physics: at CERN, 50 PB of data is produced every year. There are a billion collisions happening every second. CERN has to-date about 15,000 servers with in total 230,000 cores. In fact, the quantity of raw data generated is so enormous that most of it is filtered out right away and only a tiny part is actually stored on persistent storage. Then, more passes and quality filters will lead to the publication of smaller and more manageable, curated datasets.

And this might surprise you: such vast quantities of data are often archived, in 2021, on tape. This might seem outdated, but this remains the cheapest storage technology available for long-term archiving, if you are fine with waiting for a few hours until you can get your hands on the data.

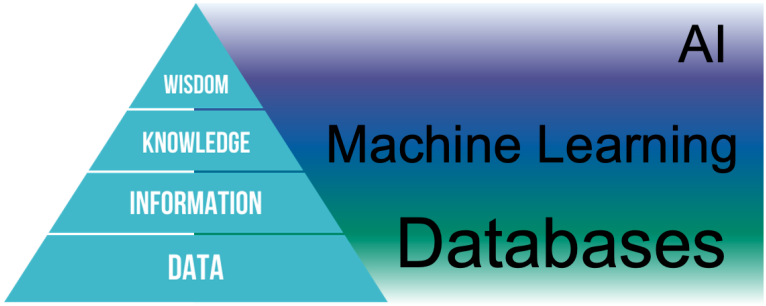
Another use case for large amounts of data is the attempt by the Sloan Digital Sky Survey (SDSS) to catalogue all visible objects in the sky. Phase IV has finished last year, and this dataset is the most detailed 3D map of the universe ever made. 200 GB of data was generated every night.

Another storage support gaining in importance is DNA: a major breakthrough to edit it was made with the CRISPR-Cas9 technique. Also, everybody on the planet now knows about the messenger RNA technique (which is, in fact, a few decades old but had not been tested as such large scales before), which resembles Computer Science and assembly code development more than biology[1].

1.5 Scope of this book

The scope of data science is very fast: from data we extract information, which we turn into knowledge and finally wisdom.

[1] see https://berthub.eu/articles/posts/reverse-engineering-source-code-of-the-biontech-pfizer-vaccine/ for more detailed explanations



Data storage and management is only the first stage of this pyramid, and is completed with Machine Learning and Artificial Intelligence. In this course, we focus on data storage, management and processing at large scales but keeping in mind that the separation from Machine Learning is becoming more blurry every day and in several aspects.

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

We now start our journey by brushing up on relational database management systems and SQL.

Chapter 2

Lessons learned from the past

2.1 Data independence

In spite of the many revolutions that happened in the past two decades, it is very important not to lose sight of the fundamental aspects of data management. The fundamental socle on which modern data management is based is *data independence*, following the seeding paper by Edgar Codd in 1970.

Data independence means that the logical view on the data is cleanly separated, decoupled, from its physical storage. In its original version, it was suggested that the most natural logical view on data is in form of a table. Everybody understands tables and, as we say, they are thousands of years old.

A relational database management system (RDBMS) exposes this logical model, together with logical building blocks for manipulating it, on top of a physical layer. In the 1970s, this physical layer was an individual computer, and the data was stored on the hard drive, in a format that is irrelevant to the user of the RDBMS. Then, the RDBMS would get updates, for example, every year, and the format on the disk might evolve. But the user does not have to change anything in the way it interacts with the system: their queries will continue to work. However, as products get updates and machines get more powerful, the user might notice that the queries get automagically faster. Of course, new features can be added on the logical layer, but this is always with long-term thinking.

Furthermore, if these features get standardized, the user can even reuse their queries with different vendors. Vendors generally have an interest to cooperate with each other and standardize their models and query languages for interoperability across different systems.

A database management system stack can be viewed as a four-layer stack:

- A logical query language with which the user can query data;
- A logical model for the data;
- A physical compute layer that processes the query on an instance of the model;
- A physical storage layer where the data is physically stored.

What happened in the past two decades is that the physical layer was changed: the same logical model and query language can now run on clusters with thousands of machines, rather than a single machine. Such large-scale clusters of machines will be the focus on our lecture. What is important to remember in this chapter is that the look and feel of querying data should remain the same to the end user, whether we are talking about GBs of data on a single machine or of PBs of data on a large cluster.

Of course, we will see that there are other data shapes out there than tables: there are trees, there are cubes, there are graphs, and there is unstructured data. But data shapes should be seen as a logical concept: data independence continues to apply no matter what data shape is supported by the database management system.

This is the reason why we will now take the time to look into the lessons learned from 50 years of relational database management systems, and insist that they are still very relevant today even for non-relational database management systems.

2.2 Formal prerequisites

Before we dive into the relational model and algebra, let us take a brief detour through some mathematics formalism and notations that are typically taught at the Bachelor's level.

2.2.1 Sets and relations

set $x \in S$ denotes that a set S contains the element x. $x \notin S$ denotes that it does not. As it only relies on the notion of containment, a set is thus unordered, and has no duplicates.

inclusion A set A is included in another set B if $\forall\, a \in A, a \in B$. This is denoted $A \subseteq B$.

Cartesian product $A \times B$ denotes the cartesian product of the set A with the set B, i.e., it is a set that contains all pairs made of one element of A and one element of B. $\times$ is associative and one can compute cartesian products over any number of sets: $A \times B ... \times Z$

relation A relation R on a family of sets $(A_i)_{i=1..n}$ is a subset of their cartesian product: $R \subseteq A_1 \times ... \times A_n$. A relation is thus a list of tuples. The set of all relations on $(A_i)_{i=1..n}$ is the powerset of the cartesian product: $\mathcal{P}(A_1 \times .. \times A_n)$.

partial function A partial function p between two sets A and B is a relation that does not associate any element of A to more than an element of B:

$$\forall (x_A, x_B), (y_A, y_B) \in p, x_A = y_A \Rightarrow x_B = y_B$$

Rather than writing $(a, b) \in p$ (with the semantics of p being a relation), we write $p(a) = b$ or sometimes $p_a = b$ or $p : a \mapsto b$.

The set of all partial functions from A to B is denoted $A \nrightarrow B$. A is called the domain of p, B its codomain.

The subset of A with the elements that do get associated to an element of B is denoted $support(p)$. It is the inverse image of B.

$$support(p) = \{a \in A \mid \exists\, b \in B, p(a) = b\} = p^{-1}(B)$$

function A function f between two sets A (the domain) and B (the codomain) is a relation that associates every element of A to exactly one element of B:

$$\forall\, a \in A, \exists !(x_A, x_B) \in f, x_A = a$$

A function is nothing else than a partial function of which the support is the entire domain A.

The set of all functions from A to B is denoted $A \to B$ or sometimes B^A.

Rather than writing $(a, b) \in f$ (with the semantics of f being a relation), we write $f(a) = b$ or sometimes $f_a = b$ or $f : a \mapsto b$.

Functions can be injective, surjective, bijective, but we do not need these concepts here.

2.2.2 Sets commonly used

There are a few standard mathematical sets such as $\mathbb{N}$ (natural integers), $\mathbb{Z}$ (relative integers), $\mathbb{D}$ (decimals), $\mathbb{Q}$ (rational numbers), $\mathbb{R}$ (real numbers).

We also use the set of the two booleans, $\mathbb{B}$ and denote the set of all strings $\mathbb{S}$. $\mathbb{S}$ is a monoïd when augmented with a concatenation operation.

Finally, we use the notation $\mathbb{V}$ to denote all values, which includes the union of all sets above, but also many other possible values such as dates, times, timestamps, URLs, sequences of bits, but also not excluding data structures such as sets, lists, trees, maps and so on. Ideally, it would be the set of 'everything', but this is a concept that leads to contradictions and kept a lot of mathematicians busy in the 19th century. However, considering that the storage capacity of a single machine, but also even of the visible universe is finite, it is reasonable to consider that $\mathbb{V}$ contains anything that can be stored on persistent storage or in memory as a sequence of 0s and 1s and according to a convention to interpret these bits.

$\mathbb{V}$ is thus, as far as we are concerned a finite set, and we also consider $\mathbb{S}, \mathbb{N}$, etc, to be limited to values that fit on a machine.

We also use the set $\mathbb{A} \subset \mathbb{V}$ that contains all atomic values. Atomic values are values that are not structured, i.e., this excludes objects, arrays, lists, sets, trees, bags, etc. but only includes strings, integers, Booleans, dates and so on.

2.3 Relational database management systems

2.3.1 Main concepts

Relational database management systems (RDBMS) are based on a tabular data format. The core of the relational model is thus the concept of table.

On a high level, a table is quite simple and the first class citizens in this model are:

- The *table* itself, which can be seen as a collection of records. For example, in an employee table, each record would be a person, in a products table, each record would be a product, etc. Thus, in some products with models that generalize tables, the term *collection* is used instead.

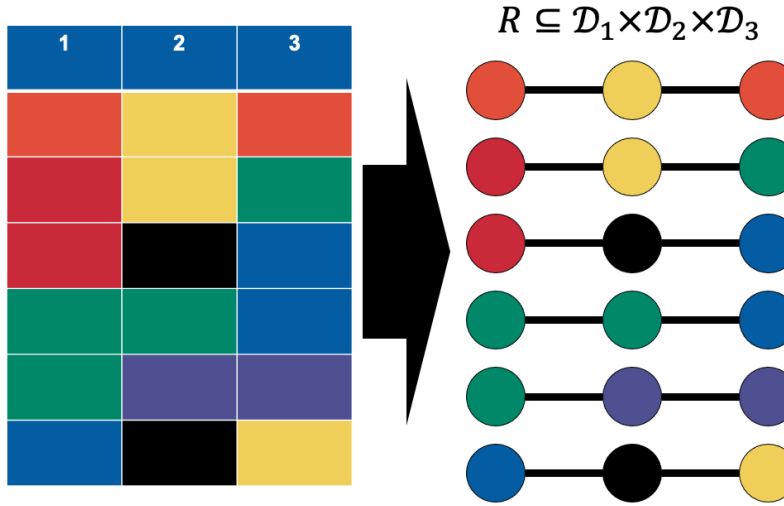
- The *attribute*, which is a property that records can have. For example, if a record is an employee, then an attribute can be their last name, their city of residence, etc. Other terminologies commonly found as synonyms of *attribute* are: *column, field, property, key.*

- The *row*, which is a record in a collection. A row associates properties with the values applicable for the record it represents. Other terminologies commonly found as synonyms of *row* are: *record, entity, document, item,* and even the classy *business object*, which seems quite popular with people who have an MBA.

- The *primary key*, which is a particular attribute or set of attributes that uniquely identify a record in its table. For example, the social security number (AHV in Switzerland) can identify a person, or a code (HG, CAB) can identify a building at ETH.

2.3.2 The formalism behind the relational model

Tables can be defined formally, that is, as purely mathematical objects. Most database textbooks introduce tables as mathematical relations over the domains associated with each attribute. A mathematical relation over several domains is defined as a subset of the Cartesian product of the domains. Each element in a mathematical relation, that is, a record in the table, is thus a tuple made of values picked in each domain.



For example, let us consider a table with four attributes:

- Name, the domain of which is a string (let us denote this set $\mathbb{S}$);
- First name, the domain of which is a string ($\mathbb{S}$);
- Physicist, the domain of which is $\mathbb{B}$, the set of Booleans (true and false);
- Year, the domain of which is $\mathbb{N}$, the set of natural integers.

A relational table is thus a subset of $\mathbb{S} \times \mathbb{S} \times \mathbb{B} \times \mathbb{N}$. An example of tuple that could belong to this subset (and thus to the table) is (Einstein, Albert, true, 1905), where each value is implicitly associated with a attribute like so:

- Name: Einstein
- First name: Albert
- Physicist: true
- Year: 1905

And a relational table is simply a set of such tuples. In fact, this is where the names *relational* as well as *relation* come from.

Expressed formally, given a family of attributes $(A_i)_{1 \leq i \leq n}$ and their associated domains

$$(Domain(A_i))_{1 \leq i \leq n}$$

A table T is a relation over these domains, i.e.

$$T \in Domain(A_1) \times Domain(A_2) \times ... \times Domain(A_n)$$

.

However, there are several issues with the above definition:

- The values in a mathematical tuple are indexed by natural integers, whereas in the logical relational model, they are indexed by strings: the attributes they are associated with. For example, Albert is associated with "First name" and not with the integer 2, its position.
- Mathematical tuples are ordered, whereas in a relational table, the order of the columns is irrelevant.
- Mathematical relations can only be sets, which makes it very difficult to easily relax and generalize to alternative forms of the relational model based on lists or bags.

These issues make the bridge between the math and the logical relational model a bit blurry, which also raises a lot of questions on smaller details that can distract from the content actually being taught and usable in practice.

A collection of records is a set of partial functions from $\mathbb{S}$ to $\mathbb{V}$. We denote the set of collections $\mathcal{C}$.

$$\mathcal{C} = \mathcal{P}(\mathbb{S} \nrightarrow \mathbb{V})$$

Each record is thus modelled as a partial function mapping strings to values, e.g. with our example, if we call our record f:

- f("Name") = "Einstein"
- f("First name") = "Albert"

- f(“Physicist”) = true
- f(“Year”) = 1905
- f is undefined for any other string.

In practice and in the context of databases, we prefer to use subscript notations for the function calls, and the quotes of the attributes are omitted if they are simple (e.g., no spaces):

- f_{Name} = “Einstein”
- $f_{\text{“First name”}}$ = “Albert”
- $f_{\text{Physicist}} = \mathit{true}$
- $f_{\text{Year}} = 1905$
- f is undefined for any other string.

We can also represent the partial function extensively:

Name	↦	Einstein
First name	↦	Albert
Physicist	↦	true
Year	↦	1905
$\mathbb{S}$		$\mathbb{V}$

However, most data scientists feel more comfortable with a visual:

$\mathbb{S}$	Name	First name	Physicist	Year
$\mathbb{V}$	Einstein	Albert	true	1905

The definition with partial functions reflects more accurately the fact that attributes are unordered, and avoids “hand-waving” explanations with ordered tuples indexed by integers.

The concept of collection does not fully reflect what a table is. For a table, we need to throw in three additional constraints: relational integrity, domain integrity and atomic integrity.

Relational integrity

A collection T fulfils relational integrity if all its records have identical support:

$$\forall\, t, u \in T, support(t) = support(u)$$

If a collection, in particular a table, has relational integrity, then this common support is a property of the table and contains the attributes of the table T: $Attributes_T$.

The extension of the table, sometimes denoted $Extension_T$ when used together with $Attributes_T$, is its actual content, which is T itself. We use T or $Extension_T$ interchangeably depending on the context.

This collection does not respect relational integrity:

	Name	First name	Physicist	Year
	Einstein	Albert	true	1905
Country		Alan	false	1936
A	Gödel	Kurt		1931

This one does:

Name	First name	Physicist	Year

Name	First name	Physicist	Year
Einstein	Albert	true	1905

Name	First name	Physicist	Year
Turing	Alan	false	1936

Name	First name	Physicist	Year
Gödel	Kurt	false	1931

in which case we typically "merge" the attributes in the display:

Name	First name	Physicist	Year
Einstein	Albert	true	1905
Turing	Alan	false	1936
Gödel	Kurt	false	1931

Domain integrity

A collection T fulfils domain integrity if the values associated with each attribute are restricted to a domain. We define these domains with a function D mapping strings (the attributes) to domains (unused attributes are just associated with empty domains, i.e. this does not need to be a partial function):

$$D \in \mathcal{P}(\mathbb{V})^{\mathbb{S}}$$

A collection T fulfils the domain integrity constraint specified by D if, for each row, the values are in the specified domains:

$$\forall\, t \in T, \forall\, a \in \mathit{support}(t), t.a \in D(a)$$

Typically, the domains $(D(a))_a$ will be standard sets such as integers, strings, booleans, dates and so on.

Concretely, the domain mapping D is called a *schema*, i.e., the names of the columns, and the domain of each column (string, integer, date, boolean, and so on).

This collection respects domain integrity:

Name	First name	Physicist	Year
String	String	Boolean	Integer
Einstein	Albert	true	1905
Turing	Alan	false	1936
Gödel	Kurt	false	1931

This one does not:

Name	First name	Physicist	Year
String	String	Boolean	Integer
Einstein	Albert	true	1905
Turing	Alan	0	1936
Gödel	Kurt	false	thirty-one

Note that the definition of domain integrity still allows records with missing values. When combining domain integrity with relational integrity, we typically only consider the case when the support of the schema (attributes associated with non-empty domains) matches exactly the support common to all records; otherwise, this is still sound, however the extra domains in the schema are unused and thus useless.

Atomic integrity

A collection T fulfils the atomic integrity constraint if the values used in it are only atomic values, i.e.

$$T \subseteq \mathbb{S} \nrightarrow \mathbb{A}$$

This means that the collection does not contain any nested collections or sets or lists or anything that has a structure of its own: it is "flat."

This collection respects atomic integrity:

Legi	Name	Lecture ID	Lecture Name	City	State	PLZ
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	263-3010-00L	Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	263-3010-00L	Big Data	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	123-4567-89L	Set theory	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	123-4567-89L	Set theory	Pfäffikon	ZH	8330

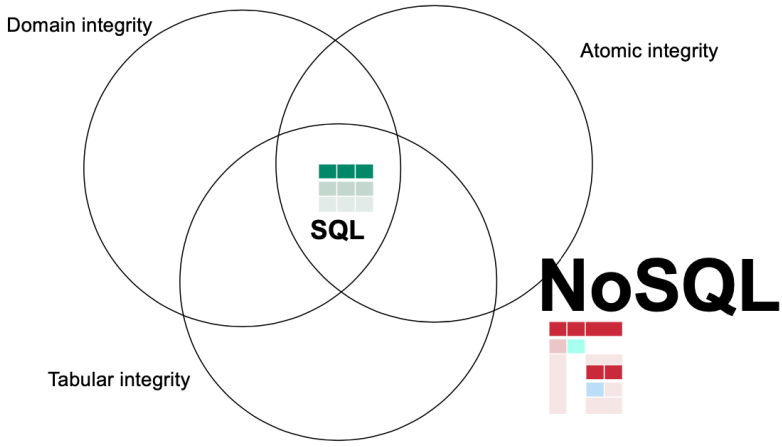
This one does not:

Legi	Name	Lecture	City	State	PLZ
32-000-000	Alan Turing	Lecture ID: xxx-xxxx-xxX, Cryptography; 263-3010-00L, Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	Lecture ID: 263-3010-00L, Big Data; 123-4567-89L, Set theory	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	Lecture ID: 123-4567-89L, Set theory	Pfäffikon	ZH	8330

The relational model and beyond

A relational table is defined as a collection that fulfils all three fundamental integrity constraints: all records have the same support, there is a schema assigning a domain to all attributes and the records respect the schema, and the values are all atomic.

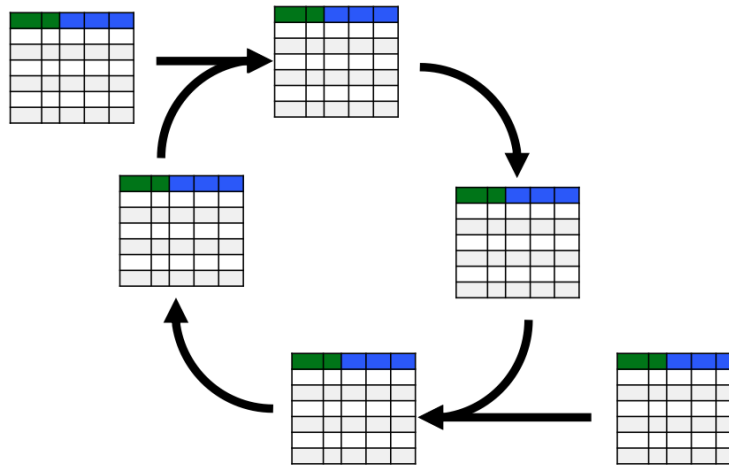
Being able to express these three integrity constraints separately from the core definition of a collection has a high pedagogical value, because it raises the awareness of these constraints for tables in the relational world, but also will allow us, in this course, to selectively relax one, two or all three constraints as we see fit. When these constraints are relaxed, we enter the beautiful world of NoSQL databases, and we will see that this comes down to manipulating collections of trees, not tables.



Also, as the definition of collections is explicitly based on a *set* of partial functions, it can very easily be extended to bag semantics, or even list (ordered rows) semantics based on the needs of the relational algebra: all we need to do is substitute "set" for "bag" or "list" in the definition.

2.3.3 Relational algebra

Once relational tables have been formally defined, it is possible to manipulate them. Mathematically, the framework that allows for this is called the relational algebra. Like numbers can be manipulated with addition, multiplication, relational tables can be manipulated with many different operators.



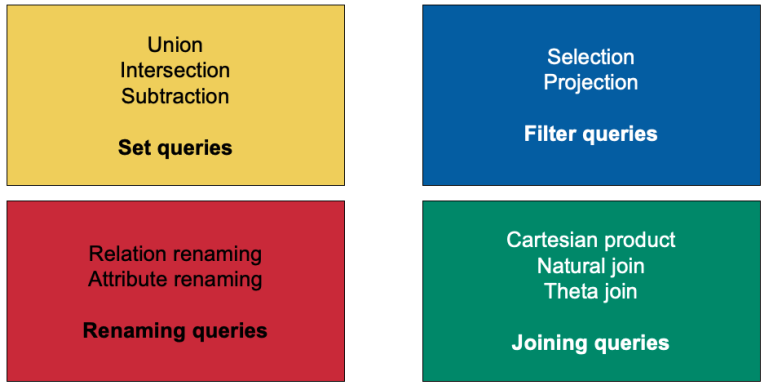
These operators can be classified in four broad categories:

- Set queries act on relational tables as sets (as previously described): one can take the union or the intersection of two sets, or subtract a set from another. These operators directly and naturally translate to relational tables.

- Filter queries. These operators take a portion of a table: some or all columns, some or all rows, etc. They are known as projection and selection. There also exists a fancy operator called the "extended projection" that can be used to add more computed columns.

- Renaming queries. These operators can rename columns.

- Joining queries. These operators can take the Cartesian product of two tables, potentially filtering to match values from both sides. The latter is called a join.



Let us a give a few examples with the most common operators.

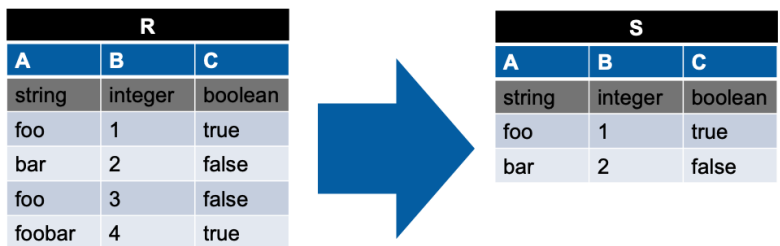
2.3.4 Selection

A selection takes a subset of the records belonging to the table. It takes a parameter, which is a predicate on the attributes. This predicate is evaluated for each record: if it is true, the record is kept, if it is false, it does not appear in the selection.

The notation used is the σ letter. For example,

$$S = \sigma_{B \leq 2}(R)$$

corresponds visually to:



R

A	B	C
string	integer	boolean
foo	1	true
bar	2	false
foo	3	false
foobar	4	true

S

A	B	C
string	integer	boolean
foo	1	true
bar	2	false

Predicates commonly involve arithmetics (addition ($+$), subtraction ($-$), multiplication ($\times$), division ($/$), integer division (e.g., n), modulo ($\%$), comparison ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$), logic ($\wedge$ (and), $\vee$ (or), $\neg$ (not)), constant literals (hard-coded strings like "foo", numbers like 3.14, etc) and attributes (the names of the column of which to take the value)). Pay attention to precedence, or use parentheses to avoid ambiguities. Predicate syntax can be extended at will, but of course it is important to carefully document any extension so other people can understand it.

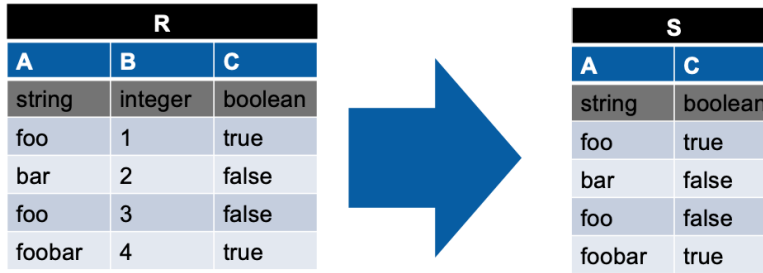
2.3.5 Projection

A projection keeps all records, but removes columns. It takes as a parameter the names of the attributes to keep in the projection.

The notation used is the π letter. For example,

$$S = \pi_{A,C}(R)$$

corresponds visually to:



R		
A	B	C
string	integer	boolean
foo	1	true
bar	2	false
foo	3	false
foobar	4	true

S	
A	C
string	boolean
foo	true
bar	false
foo	false
foobar	true

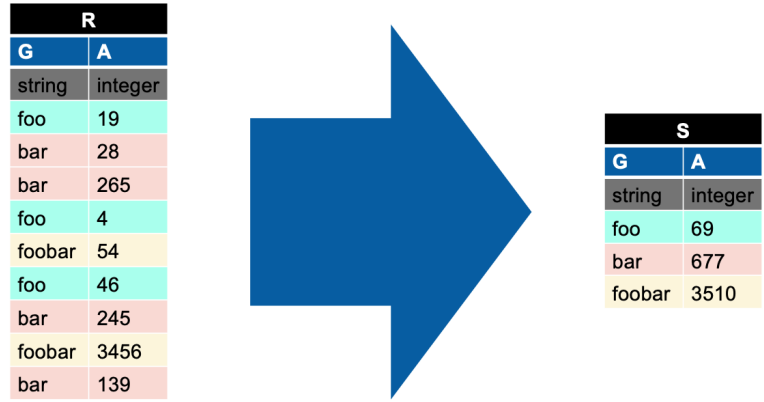
2.3.6 Grouping

A grouping, also called aggregation, merges records by grouping on some attributes, and aggregating on all others.

The notation used is the γ letter. For example,

$$S = \gamma_{G,SUM(A)\to A}(R)$$

groups by G and aggregate the values in column A (within the same group) with a sum, and corresponds visually to:



R	
G	A
string	integer
foo	19
bar	28
bar	265
foo	4
foobar	54
foo	46
bar	245
foobar	3456
bar	139

S	
G	A
string	integer
foo	69
bar	677
foobar	3510

It is easy to get things wrong with grouping. A general recommendation is to make sure to carefully classify each attribute as either a grouping attribute (it appears on its own in the γ subscript, like G above) or a non-grouping attribute (it must then either be left out, or appear in an aggregation function, with an arrow pointing to the new name of the aggregated column, $SUM(A) \rightarrow A$ in our example). The new column may have a different name than the original one. Do not use an attribute both as grouping and non grouping. Grouping with no grouping key is possible and results in an aggregation of the entire table to a single value (for example, to take a max or a sum of all values in an attribute).

The most common aggregation functions are COUNT, SUM, MAX, MIN, AVERAGE but there are many others (ask a statistician!). Please note that, in the relational algebra, aggregation functions must aggregate to a single value, otherwise it leads to nested structures that break atomic integrity (you may have guessed that we will, of course, be doing this sort of thing in the course, but this is then NoSQL).

2.3.7 Renaming

Renaming a column changes the name of a column and is denoted with ρ, i.e.,

$$S = \rho_{A \rightarrow D}(R)$$

2.3.8 Extended projections

Extended projections are less common in the formalism of the relational algebra, however they are used very often in practice, so that it is worth being aware of its existence.

An extended projection can compute values with syntax similar to that used in selection predicates, and then assign the result to a new column denoted with an arrow, like the rename operator.

$$S = \pi_{A+C \times 2 \rightarrow D}(R)$$

It is possible to mix several such computations, also with pure projections:

$$S = \pi_{B, A+C \times 2 \rightarrow D, 2 \times B \rightarrow E}(R)$$

You need to be careful not to create several attributes with the same name.

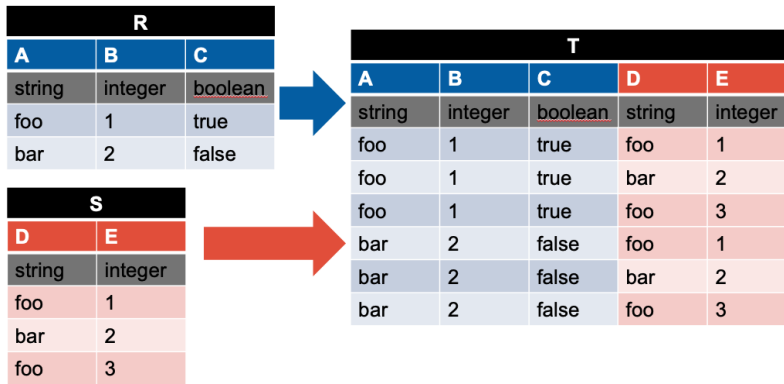
2.3.9 Cartesian products

A Cartesian product combines each tuple from the left relational table with each tuple from the right relational table.

The notation used is the $\times$ symbol. For example,

$$T = R \times S$$

is the Cartesian products of R with S and corresponds visually to:



R

A	B	C
string	integer	boolean
foo	1	true
bar	2	false

S

D	E
string	integer
foo	1
bar	2
foo	3

T

A	B	C	D	E
string	integer	boolean	string	integer
foo	1	true	foo	1
foo	1	true	bar	2
foo	1	true	foo	3
bar	2	false	foo	1
bar	2	false	bar	2
bar	2	false	foo	3

Cartesian products should be handled with care: imagine the size of the resulting table if the table on the left has a billion records and that on the right has a million records (tables with millions or billions of records are quite common in the real world).

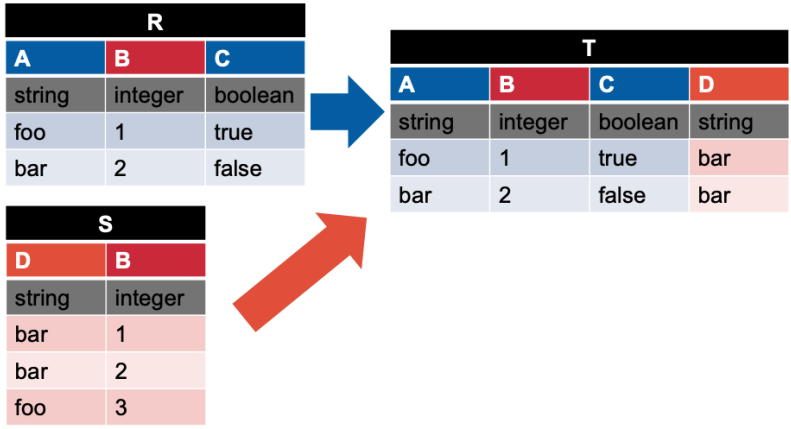
2.3.10 Joins

Often, we do not use Cartesian products but joins, which are a "filtered" Cartesian product in which we only combine directly related tuples and omit all other non-matching pairs.

The notation used is the $\bowtie$ symbol. For example,

$$T = R \bowtie S$$

joins R and S but only keeps records that coincide on the attributes common to both sides (here, B must match):



The joins as above, called natural joins, are very common in relational databases and are less expensive than Cartesian products, but remain more expensive than simpler operators such as projection and selection.

There are other kinds of joins:

- Theta joins explicitly specify the joining criterion instead of matching the common attributes. The criterion is then supplied as a predicate in the subscript of $\bowtie$ with the same syntax as σ, e.g. $T = R \bowtie_{A=D} S$. Note that, in this case, common attributes require care and, if expressed with mathematical formulas, renames should be used to keep the formulas consistent, e.g. $T = R \bowtie_{B=E} \rho_{B \to E} S$.

- Outer joins keep records with no match on the other size, keeping the other attributes absent. Note that this breaks relational integrity in its strictest sense, as these joined records will not have the full support of the resulting relational table. Relational integrity can also be preserved by picking a default value in the domains of the attributes that are missing.

- Semi-outer joins are like (full) outer joins but only keep unmatched records on the left, or only on the right.

2.3.11 Combining operators

Relational algebra operators can be combined at will and nested in more complex formulas. It is important to make sure you can write and read formulas involving relational algebra operators, explain what a formula does in English terms, or design one when given instructions in English.

2.4 Normal forms

When creating databases, the schemas, that is, the columns and their domains, should be designed with care. If some rules are not respected, things can go wrong as the data can easily become inconsistent because it is duplicated:

- deletion anomalies occur in poorly designed databases when deleting a record makes the database inconsistent. For example, deleting the only order of a specific phone in an online shop might completely delete the phone data.
- insertion anomalies occur in poorly designed databases when inserting a new record makes the database inconsistent. For example, inserting a new order might cause duplicate and inconsistent data to appear on the product.
- update anomalies occur in poorly designed databases when updating a record makes the database inconsistent. For example, updating data on a phone in an order might make it inconsistent with the data in other orders.

In order to avoid such anomalies, best practice dictates to follow so-called normal forms. Normal forms are typically taught in Bachelor-level database lectures in Computer Science curricula in many universities (in particular in German-speaking countries). A complete theory of normal forms would demand much more space, so that we only quickly give a small survey to develop an intuition.

The first normal form was already covered earlier: it is in fact atomic integrity.

This table does not respect the first normal form: there are tables nested in it.

Legi	Name	Lecture	City	State	PLZ
32-000-000	Alan Turing	Lecture ID: xxx-xxxx-xxX, Cryptography; 263-3010-00L, Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	Lecture ID: 263-3010-00L, Big Data; 123-4567-89L, Set theory	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	Lecture ID: 123-4567-89L, Lecture Name: Set theory	Pfäffikon	ZH	8330

This table does respect the first normal form: all values are atomic.

Legi	Name	Lecture ID	Lecture Name	City	State	PLZ
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	263-3010-00L	Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	263-3010-00L	Big Data	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	123-4567-89L	Set theory	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	123-4567-89L	Set theory	Pfäffikon	ZH	8330

The second normal form takes it to the next level: it requires that each column in a record contains information on the *entire* record.

This table is not in second normal, because the column "Name" refers to a student (identified with their Legi), but the whole record semantically corresponds to an attendance of a student to a lecture; we say that the Legi and Lecture ID, together, are a primary key of the record, but "Name" does not functionally depend on the full primary key, only on half of it:

Legi	Name	Lecture ID	Lecture Name	City	State	PLZ
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	263-3010-00L	Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	263-3010-00L	Big Data	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	123-4567-89L	Set theory	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	123-4567-89L	Set theory	Pfäffikon	ZH	8330

These three tables are in second normal form, in fact, they are how the previous table can be fixed: by cleanly separating information on the attendance from information specific to the lecture or specific to the student in separate tables:

Legi	Name	City	State	PLZ
32-000-000	Alan Turing	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	Pfäffikon	ZH	8330

Legi	Lecture ID
32-000-000	xxx-xxxx-xxX
32-000-000	263-3010-00L
62-000-000	263-3010-00L
62-000-000	123-4567-89L
25-000-000	123-4567-89L

Lecture ID	Lecture Name
xxx-xxxx-xxX	Cryptography
263-3010-00L	Big Data
123-4567-89L	Set theory

If you paid attention, you will have noticed that the former table can be obtained by joining the latter three tables. Normalizing, in fact, can be seen as the opposite of joining (which is also called denormalizing as we will see in this course).

The third normal form additionally forbids functional dependencies on anything else than the primary key, for example the following table is not in third normal form because "PLZ" functionally depends on "City"+"State", but this is not a primary key (column "Legi" is).

Legi	Name	City	State	PLZ
32-000-000	Alan Turing	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	Pfäffikon	ZH	8330

This can be fixed, again, by separating the data into two tables, with the data on "City"+"State" moved out to another table, like so:

Legi	Name	City	State
32-000-000	Alan Turing	Bletchley Park	UK
32-000-000	Alan Turing	Bletchley Park	UK
62-000-000	Georg Cantor	Pfäffikon	SZ
62-000-000	Georg Cantor	Pfäffikon	SZ
25-000-000	Felix Bloch	Pfäffikon	ZH

City	State	PLZ
Bletchley Park	UK	MK3 6EB
Bletchley Park	UK	MK3 6EB
Pfäffikon	SZ	8808
Pfäffikon	SZ	8808
Pfäffikon	ZH	8330

If this sounds complicated, the following will probably come as good news: in Big Data, we often have to throw away normal forms and denormalize our data for better scalability and performance. More in the next chapters.

2.5 The SQL language

SQL was a follow up on the original idea of data independence by Edgar Codd. It was contributed by Don Chamberlin and Raymond Boyce (who gave his name to a normal form, too: the Boyce-Codd normal form).

SQL was originally named SEQUEL, for Structured English QUEry Language. The first commercial relational database that implemented it was called System R and was born in IBM Almaden (San Jose). Their first customer was Pratt & Whitney. However, due to a trademark issue, the name SEQUEL had to be changed to SQL. This explains why many people, including yours truly, pronounce SQL see-kwel while others pronounced es-kew-el.

In 1977, a company named Software Development Laboratories was created with another implementation. It was renamed in 1979 to Relational Software and then in 1982 as Oracle, who is one of the major RDBMS players today alongside IBM and Microsoft.

SQL is a declarative language, which means that the user specifies what they want, and not how to compute it: it is up to the underlying system to figure out how to best execute the query.

It is also a set-based language, in the sense that it manipulates sets of records at a time, rather than single values as is common in other languages.

It is also, to some limited extent, a functional language in the sense that it contains expressions that can nest in each other (nested queries). We will see, however, that the extent to which it can be done is limited

and that new-generation querying languages that go beyond data in normal form are even more functional.

It is very important that you learn SQL as this is a fundamental building block in the area of databases. It is very common nowadays for many to use Python or R with a DataFrame API such as pandas. Even worse, it is also common to use spreadsheets to store and query sensitive data (with disastrous consequences regularly being reported in the news and causing damage to the reputation of any institution who gets it wrong – and it is easy to get things wrong on the consistency of data stored in a spreadsheet). However in many cases, when dealing with structured data, SQL is the language of choice (and can also be nested in host languages such as Python).

We will give an example of a query that has many of the relational operators:

```
SELECT c.century AS cent,
       COUNT(c.name) AS num_captains,
       SUM(s.id) AS ships
FROM captains c, ships s
WHERE c.id = s.captain
GROUP BY century
HAVING COUNT(c.name) >= 3
```

The FROM clause selects from which tables to read the data, in this case two tables, "captains" and "ships". Implicitly the Cartesian product is computed.

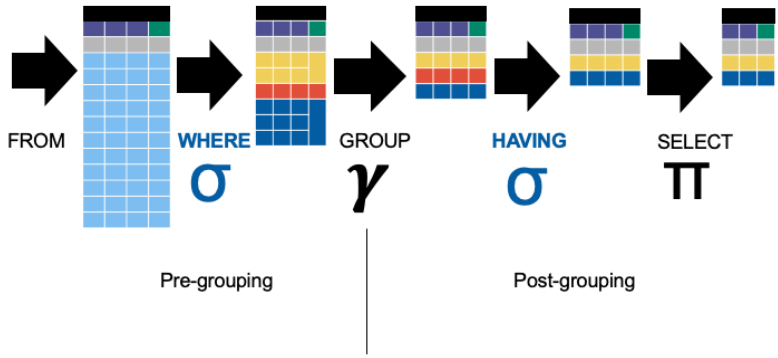
The WHERE clause performs a selection: it only keeps the records for which the captain (from the captains table) is the captain of the ship (from the ships table). If you pay attention, you will recognize that this filter together with the Cartesian product are actually a theta join. Any reasonable SQL implementation will be smart enough to detect this and evaluate this query efficiently (joins can be computed in linear time rather than quadratic!).

The GROUP BY clause performs an aggregation, with century as a grouping key. Aggregations on the captain name (COUNT) and the ship id (SUM) are done in the SELECT clause.

The SELECT clause is also where projections are made: it lists the columns to include in the results. Renames are also made in this clause with AS.

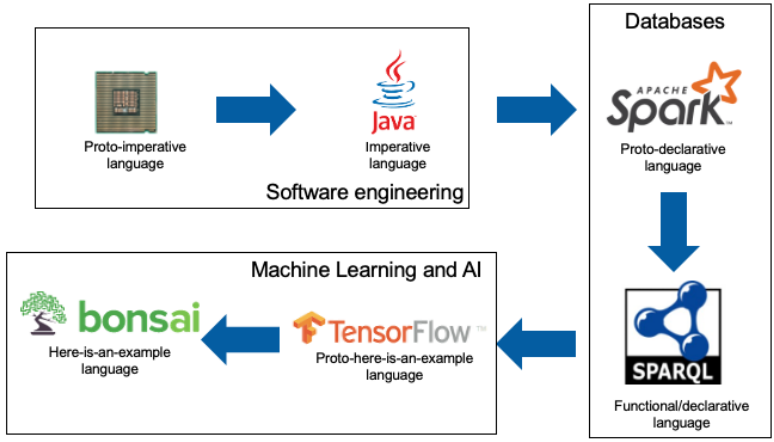
The HAVING clause is in fact like the WHERE clause, but performs a selection after, rather than before, the grouping.

Internally a SQL query will be converted to a query plan that is closely related to the relational algebra, for example like so:



2.6 Languages

In the bigger picture, there are six main categories of languages: assembly code, later (fortunately) superseded by higher-level imperative programming languages (C, C++, Java, ...). Functional and declarative query languages (SQL, JSONiq, SPARQL...), often co-habiting with lower-level APIs (DataFrame APIs such as in Apache Spark, ...). And Machine Learning frameworks (tensorflow, pytorch, keras...), which are slowly (and hopefully increasingly) supplemented with higher-level declarative languages for ML:



In this course, we are interested in relational, functional query languages as well as lower-level APIs such as DataFrames.

2.7 Transactions

Relational databases also offer convenient guarantees that prevents things from going wrong. There are four main properties (often called ACID) that the good old systems provide:

- Atomicity: either an update (called a transaction if it consists of several updates) is applied to the database completely, or not at all;
- Consistency: before and after the transactions, the data is in a consistent state (e.g., some values sum to another value, another value is positive, etc);
- Isolation; the system "feels like" the user is the only one using the system, where in fact maybe thousands of people are using it as well concurrently;
- Durability; any data written to the database is durably stored and will not be lost (e.g., if there is an electricity shortage or a disk crash).

We will see that many systems who scale beyond one machine have to make compromises on these guarantees, even though a lot of progress has recently been made.

2.8 Scaling up and out

So how are the limits of traditional relational database management systems reached? In many ways:

- there can be lots of rows: beyond a million records, a system on one machine can start showing signs of weakness; even though more recent system manage to push it a bit higher on a single machine (e.g., close to a billion);
- there can be lots of columns: beyond 255 columns, a system on one machine can start showing signs of weakness or even not support it at all;
- there can be lots of nesting: many systems do not support nested data or, if they do, do so only in a limited fashion and it becomes quickly cumbersome;

This concludes our brush-up, as in fact most of what follows will be directly related to data that has lots of rows, lots of columns, or lots of nesting:

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

In the next chapter, we will start with storage systems that massively scale on the number of records that can be stored.

Chapter 3

Object storage

In Chapter 1, we mentioned the Sloan Digital Sky Survey dataset, which is the result of several years of work. It is the most detailed 3D map of the universe ever made.

If you browse through the dataset, you will see that it has 273 TB of data, organized as 176,000,000 files in 680,000 directories. This data contains 260 millions of stars, 208 millions of galaxies, in total 1,231,000,000 objects on 4 spectra.

This is much more than any single disk in 2021 can store, which means that a solution with a single machine cannot possibly work.

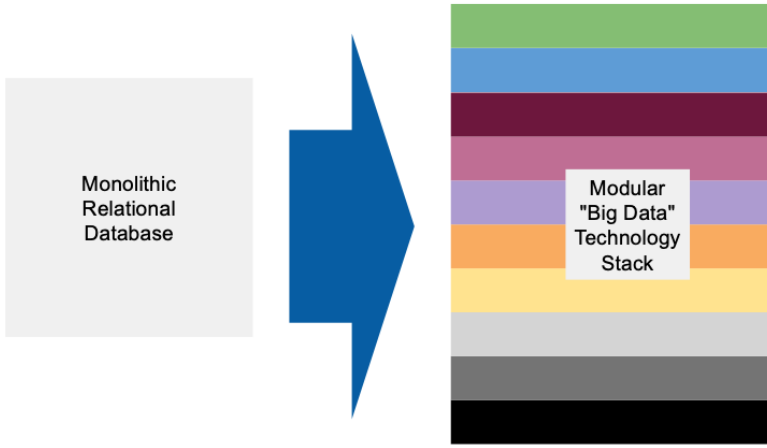
So, how do we deal with 273 TB of data organized as 176,000,000 files in 680,000 directories?

This is the question we answer in this chapter.

3.1 Storing data

In the previous chapter, we had a quick survey of relational databases. Traditionally, a relational database management system fits on a single machine. But Petabytes of data do not fit on a machine.

As a consequence, in this course, we will have to rebuild the entire technology stack, bottom to top, with those same concepts and insights that we got in the past decades, but on clusters of machines rather than on a single machine.



Indeed, the good news is that 99% of what we learn in the past 50+ years in the field of SQL and relational database management systems can be directly reused.

The relational algebra: selection, projection, grouping, sorting, joining, ... still makes sense on clusters.

Language features: SQL, declarative languages, functional languages, optimizations, query plans, indices... still make sense on clusters.

The table layout: tables, rows, columns, primary keys... still makes sense on clusters.

But not everything. We will have to let go, partially or fully, of consistency constraints. In the Big Data world, relational integrity may or may not hold. Domain integrity may or may not hold. Atomic integrity (also known as the first normal form) may or may not hold. And all normal forms may or may not hold.

Indeed, in what is called the NoSQL universe, these concepts are replaced with:

- Nested data: data that is not in first normal form;
- Heterogeneous data: data that does not fulfil domain integrity (it may not even have a schema!) and also not relational integrity.

Collectively, this alternate design is referred to as *data denormalization*. At large scales, it becomes desirable to sometimes denormalize, rather than normalize data, in order to get better performance.

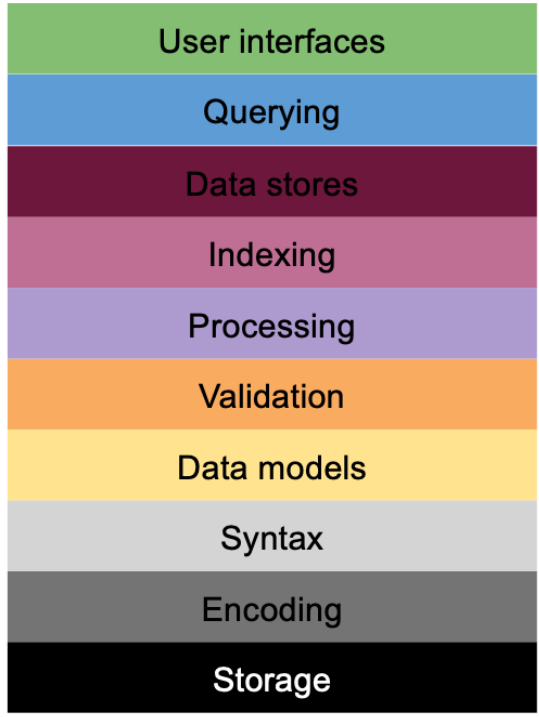
The same goes for the ACID paradigm: Atomicity, Consistency, Isolation, Durability are very challenging to obtain on large systems (even though there is a lot of research going in this direction). As a consequence, this concept is replaced with thinking in terms of the so-called CAP theorem:

- (Atomic) Consistency
- Availability
- Partition tolerance

and in particular of a weaker condition of eventual consistency instead of full atomic consistency. But we will shortly come back to this. Before this, let us have a look again at our technology stack.

3.2 The technology stack

The technology stack we are going to rebuild goes from a level very close to the hardware (storage) and all the way to interaction with the end user (query language or even voice assistant).



Examples of storage systems involve the local file system on your laptop, network file systems (NFS), distributed file systems (HDFS from Hadoop, GFS from Google), as well as hosted and fully managed cloud storage (S3, Azure Blob Storage...).

Encodings are used to convert data (for textual syntax) to bits, as this is what computers understand. In the early days, ASCII was used,

later on ISO-8859-1 for latin characters but the recommendation today is to stick with Unicode, for example UTF-8, for full compatibility also across (human) languages). There are also binary encodings such as BSON (which is a binary format corresponding roughly to JSON, used in MongoDB).

Syntax is used when data is stored as (human-readable) text. This includes among others raw text (for natural language processing), CSV (for tables), XML and JSON (for trees), RDF/XML and Turtle (for graphs) and XBRL (for trees). Data can also be stored in binary form (protocol buffers, BSON, Parquet...), in which case it cannot be read by a human in a text editor.

Models are a higher-level representation of the data not necessarily tied to a particular syntax or binary format. For tables, it is the relational model. For trees, there are several (the XML infoset, the JSONiq data model...) as well as for graphs (RDF triples and labeled property graphs). For cubes (often "abusively" referred to as OLAP), the most common model uses dimensional coordinates with dimensions organized as hierarchies.

Validation involves languages to add constraints to data: XML Schema, JSON schema, JSound (which are optional for trees), relational schemas (in relational database systems), XBRL taxonomies (for cubes), etc.

Processing frameworks come in different forms and are mostly (at least on the theoretical level) orthogonal to data shapes: the older, two-stage forms (MapReduce) and newer generations that handle generic Directed Acyclic Graphs of dataflows (Tez, Apache Spark, Apache Flink, Ray...). There are also lower level frameworks at the level of the virtual machine, accessed via a command-line shell (Amazon EC2...).

Indices accelerate data lookups. They are mostly orthogonal to data shapes: hash indices, B+-trees, geographical or spacial indices, etc.

Data stores are the generic product providing all the functionality to end users: a relational database management system (RDBMS), document stores (e.g., MongoDB, CouchBase, ElasticSearch), higher-level wrappers on data processing (Hive, HBase, Databricks, Cassandra...) or more general integrated products (Snowflake, MarkLogic...).

Higher-level languages (to be contrasted with data processing APIs like Pandas or Spark's DataFrames) are supported by some but not all data stores. SQL is the undisputed choice for tables. For trees, there are 20+ competing languages among which JSONiq, XQuery, N1QL, PartiQL... for graphs there is mostly Cypher and SPARQL. REST APIs (e.g. with GraphQL) also provide a (lower-level) access to a data store but are meant to be consumed by a host language, whereas query languages can be used on their own.

And finally, user interfaces can hide even the query language from the end user: spreadsheet software, Business Intelligence tools (Tableau,

QLikview, Access...), and even voice assistants (Siri, Alexa...).

But for this chapter, we are focusing on the storage level and in particular, with cloud storage services that require no installation.

3.3 Databases vs. Data lakes

It is obvious, but needs to be said: data needs to be stored somewhere. There are two main paradigms for storing and retrieving data.

On the one hand, data can be *imported* into the database (this is called ETL, for Extract-Transform-Load. ETL is often used as a verb). This causes more work for the user, as they need to import all their data before they are able to query it. Relational database management systems require an ETL of the data as well as some NoSQL products (MongoDB, HBase...). The reason is that the data is internally stored as a proprietary format that is optimized to make queries faster. This includes in particular building indices on the data.

On the other hand, data can also just be stored on some file system (whether local or distributed or in the cloud) and queried in place (in situ) by a data processing engine either with an API (Pandas, Apache Spark...) or query language (RumbleDB...). This paradigm is called the *data lake* paradigm and gained a lot of popularity in the past two decades. It is slower, however users can start querying their data without the effort of ETLing. Data lakes are best used in a context in which a full scan of the data is needed (e.g. with MapReduce or Apache Spark).

3.4 From your laptop to a data center

3.4.1 Local file systems

Typically, on a single machine, the data is stored on the local file system. This is possible if it is no more than a few Terabytes, depending on the disk capacity. There are also newer generations of disks called SSDs (Solid State Disks) that are more expensive and usually a bit smaller, but there exist SSDs of 1 TB nowadays.

When a file is stored on the local filesystem, it consists of both content of metadata. The content is made of the bits of the file itself (e.g., the text encoded to bits in UTF-8 if it is a JSON or XML file). The content is also sliced in blocks that are roughly 4 kB of size (this varies with the file system, but the order of magnitude does not change) and a bit is never read individually: files are read block by block. This is to optimize the balance between throughput and latency (more on this in the next chapter).

The metadata can be seen as a small relational table whose attributes are: the name of the file, access rights, the owner, the group of the owner, the last modification time, the creation time, the size, etc.

Furthermore, files are organized in a hierarchy of files and directories familiar to most people.

3.4.2 More users, more files

How does this scale to more users? A disk can be made accessible via the network (LAN for Local Area Network) and shared with other people. There exists also larger networks (WAN for Wide Area Networks), however it is difficult to scale concurrent access and problems can arise already when two people work on a file at the same time (e.g., for sharing an Excel file, a special mode must be enabled to avoid issues, a feature that text files surely do not have).

The other scalability problem is that a local file system can easily support 1,000 files or even 1,000,000 files, but can hardly make it to billions of files.

In order to make this scale, the approach for large scales, namely here, object storage, is to:

- throw away the hierarchy: there are no directories;
- make the metadata flexible: attributes can differ from file to file (no schema);
- use a very simple, if not trivial, data model: a flat list of files (called objects) identified with an identifier (ID); blocks are not exposed to the user, i.e., an object is a blackbox;
- use a large number of cheap machines rather than some "supercomputer";

3.4.3 Scale up vs. scale out

When a system is too slow on a single machine, there are several ways out.

First, one can buy a bigger machine: more memory, more or faster CPU cores, a larger disk, etc. This is called *scaling up*.

Second, one can buy more, similar machines and share the work across them. This is called *scaling out*.

Scaling up has its limits: while increasing memory from 8 GB to 16 or 32 GB works, 64 GB or 128 GB quickly becomes more expensive, 6 TB is probably beyond what a private individual or small company wants to have, and 1 PB of working memory would require billions of not trillions of dollars of investment to hire a team of dozens of researchers in a Research and Development unit to maybe achieve this

in 20 years: in other words the costs of scaling up grow up exponentially, if not hyperbolically (vertical asymptote), making it impractical.

Thus, scaling out is the better approach, and it is also the one taken in Big Data systems: this is why, in a cluster within a data center, there are thousands or even tens ot thousands of servers, of all which have performance numbers similar, or slightly better, in order of magnitude, to your personal laptop.

There is also a third way: optimizing your code. In fact, you should always first try to improve your code before scaling out. The vast majority of data processing use cases fit on a single machine, and you can save a lot of money as well as get a faster system by squeezing the data on a machine (possibly compressed) and writing very efficient code.

Scaling out in the context of data processing is typically worth it in just two cases: either because the data does not fit on your laptop, e.b., you are looking at 50 TB of data, or because your laptop cannot read the data fast enough, i.e., your bottleneck is the disk I/O.

3.4.4 Data centers

So in a cluster, in a data center, there are thousands to tens of thousands of machine. Why not more? Well first, it is difficult to know because companies do not always fully communicate on their numbers. However, it is difficult to go beyond this because of pragmatic reasons having to do with power and cooling. A data center consumes as much electricity as an entire airport, and it becomes almost impossible to handle 100,000+ machines together. In fact, the trend at places like CERN is to have less servers, but more cores per server.

A server is also called a node in the context of a data center. Servers typically have between 1 and 64 cores and the number keeps going up.

The working memory available on a node ranges ranges to 16 GB to 6 TB as of today.

The local storage available (SSD or HDD) ranges from 1 to 20 TB.

The network bandwidth goes from 1 to 100 Gb/s but higher speeds are also possible especially in the context of High-Performance Computing (HPC). Bandwidth is the highest within the same cluster and can be slower across clusters or data centers, even though there also exist high-speed connections across data centers.

Nodes are typically flat, rectangular boxes that are piled up in what is called a rack, which looks like a tower. A cluster is just a room filled with racks put next to each other.

Each module in a rack can be a server, or pure storage (many disks), or a network switch, etc. The height of a module is standardized and measured in so-called rack units (RU). Each module has typically between 1 and 4 RU.

Data centers are typically concentrated with a few players, called cloud providers. The big tree are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. Most companies on the planet rent resources from these players, typically not directly physically but via the creation and "deletion" of virtual machines and/or of high-lever services. Object storage is one of them.

3.5 Object stores

Now that we have the hardware in place and have out our disposal thousands of cheap commodity servers, we can come back to the problem at hand: how do we store plenty of data?

Most cloud providers offer a solution for cloud storage, which is typically called object storage. These solutions are typically as-a-service, meaning that users can directly go ahead and use them rather than have to install anything.

3.5.1 Amazon S3

Amazon's object storage system is called Simple Storage Service, abbreviated S3. From a logical perspective, S3 is extremely simple: objects are organized in buckets. Buckets are identified with a bucket ID, and each object within a bucket is identified with an Object ID.

Thus, any object in S3, worldwide, is uniquely identified with a Bucket ID and an Object ID. "Object", in fact, is only a synonym for "file" and can thus be a text file, a picture, a video, a dataset (CSV, JSON...) etc. But in the context of a object storage, we say "object" rather than "file". The main reason is that files are organized in hierarchies, but object stores have no hierarchy: just a flat list of objects organized in buckets.

An object can be at most 5 TB (to remember this: it is typically something that fits on a single disk). However, it is only possible to upload an object in a single chunk if it is less than 5 GB – otherwise, the upload must be done in several blocks. We will cover blocks in the next chapter and here focus our attention on objects seen as black boxes.

By default, users can have up to 100 buckets but can ask for more on request.

3.5.2 Azure Blob Storage

3.6 Guarantees and service level

3.6.1 Service Level Agreements

Most cloud services come with a cost, but also a promise. This promise is a contract between the provider and the user called a Service Level Agreement, abbreviated as SLA.

For example, S3 promises that it loses, each year, less than one object in 100 billion. This is formulated as a durability of 99.999999999%.

Another promise is availability: S3 is available 99.99% of the time, meaning that it will be down less than 1 hour per year.

SLAs very often contain numbers with a lot of 9s. 99% of availability means at most 4 days a year of downtime, 99.9% less than 10 hours, 99.999% six minutes, 99.9999% 32 seconds and 99.99999% less than 4 seconds.

Response times are also expressed in terms of the 99.9% percentile, for example one can promise that 99.9% of the requests will be served in less than 300ms. However, S3 does not make promises in terms of latency: it varies depending from where you request an object, and usually objects are large enough so that the bottleneck is in the time to download the object over the network, not in the latency. Rather, the promise is made in terms of throughput, which in this context is the number of objects read or written per second: typically 5,500 reads/s and 3,500 writes/s.

3.6.2 The CAP theorem

Early relational database management system relied on ACID properties (see Chapter 2. In order to scale out, many distributed systems have to make a compromise on the transactional guarantees that they offer. This is best explained with the so-called CAP theorem.

The CAP theorem is basically an impossibility triangle: a system cannot guarantee at the same time:

- (atomic) Consistency: at any point in time, the same request to any server returns the same result, in order words, all nodes see the same data;
- Availability: the system is available for requests at all times (SLA with very high availability);
- Partition tolerance: the system continues to function even if the network linking its machines is occasionally partitioned.

The CAP theorem is, in fact, not formally proven and it would be more appropriate to call it a conjecture. An intuitive way of explaining it is as follows.

When an update is made via some server, this update must propagate to other nodes to the extent that a specific replication factor for the data is met.

If there is a partition of the network leading to two disconnected sub-networks and the propagation of some data did not have a chance to complete, then there are two possible design decisions:

1. either the system is temporarily put to a halt (synchronous propagation) and thus becomes unavailable until the network partition is resolved and the propagation can be completed. In this case the system is atomically consistent but not available (CP).

2. or the servers continue to serve requests, but the data served by the two disconnected sub-networks will be different. In this case the system is available but not atomically consistent (AP). Such systems propagate updates asynchronously and are also often called *eventually consistent* in the sense that, if one hypothetically would no longer receive updates, there exists a time at which the system will be consistent. In practice of course, as updates keep arriving, the system rarely becomes fully consistent – a common misconception about eventual consistency.

A third possibility is for the system not to be partition tolerant: in this case, the system is available and consistent (AC) but only for as long as no network partition occurs. If a network partition occurs, this is unknown territory and no further guarantees exist.

Beware of systems that claim to be available, atomically consistent and partition-tolerant at the same time: this is often just marketing. In practice, the companies who claim such properties (ab)use the fact that a partition of their network is very rare. But when a partition happens, they have in fact to make a choice between A or C, even though users will barely notice.

3.7 REST APIs

Data stores, be it object stores, key-value stores, document stores, wide column stores, distributed file systems, etc expose their functionality via an API. A vast majority of them offers a so-called REST API.

REST means REpresentational State Transfer. In fact, it is really just "HTTP done right" in the sense that it closely uses HTTP functionality: methods and resources.

The benefit of offering a REST API is tremendous: it makes an integration with any host language very easy, because HTTP clients

exist in almost any host language (Java, Python, PHP, R...) and it is straightforward to implement a "wrapper API" in the host language that forwards all requests through the REST API to the data store. If you design a new system in the future, it is almost a no-brainer decision to support a REST API.

Other possibilities are a native driver, but this needs to be done for each host language. MongoDB is an example of document store that works with drivers.

Another protocol is the XML-based SOAP protocol.

The HTTP protocol, which REST builds on, was invented back in 1989, thirty years ago, by Sir Tim Berners-Lee, who is also the creator of the Web.

A client and server communicated with the HTTP protocol interact in terms of *methods* applied to *resources*.

A resource can be anything: a document, a PDF, a person, a calendar entry, or even a physical object such as a smart plug ("Web of things"). A resource is referred to with what is called a URI. URI stands for Uniform Resource Identifier[1]. A URI looks like so:

http://www.example.com/api/collection/foo/object/bar?id=foobar#head

where

- "http" is the scheme;
- "//www.example.com" is the authority.;
- "/api/collection/foo/object/bar" is the path;
- "?id=foobar" is the query;
- "#head" is the fragment.

There exist other schemes than http, for example mailto, ftp, hdfs, file, and so on. However, http is the most popular schemes even for resources that are "offline", i.e., not actually reachable via HTTP.

A client can act on resources by invoking methods, with an optional body. The most important methods are:

- GET (without a body): this method returns a representation of the resource in some format (text, XML, JSON...). GET should have no side effects (beyond logging, of course).

[1]In fact, we should normally call them IRI (Internationalized Resource Identifier) for the most recent version, however most people continue to use the term URI. URIs used to be divided in URLs (locators) and URNs (names), however this distinction is now obsolete.

- PUT: this method creates or updates a resource from a representation of a newer version of it, in some format (text, XML, JSON...). PUT has the side effect that a subsequent GET asking for the same format should return the same representation. PUT is idempotent, in that calling PUT with the same resource and body is identical to calling it just once.
- DELETE (without a body): this method deletes a resource. DELETE has the side effect that a subsequent GET asking for a representation of the resource should fail with a not-found (404) error.
- POST: this method is a blank-check, in that it acts on a resource in any way the data store seems fit; the behavior, of course, should be publicly documented. A typical use of POST is to create new resources but letting the REST server pick a resource URI for this new resource.

An example of URI for S3 is "http://bucket.s3.amazonaws.com" for a bucket and "http://bucket.s3.amazonaws.com/object-name" for an object.

3.8 Object stores in practice

Although object stores are based on a flat key-value model, most object storage services emulate a file hierarchy by allowing slash (/) characters in keys, and interpreting these slashes as virtual paths. As far as the storage layer is concerned, slashes are a character like any other. But on the logical level, a logical hierarchy enables more use cases for object stores.

3.8.1 Static website hosting

Object stores such as S3 or Azure Blob Storage can be used for static website hosting in a very straightforward way.

Static website hosting means that there is no dynamically generated content, as would be the case for example with a PHP or Java EE server (Tomcat, etc). In a static website, the HTML pages, pictures, CSS stylesheets, (client-side) JavaScript scripts, videos are delivered *as is*.

It is both very convenient and relatively cheap to drop HTML pages, pictures, CSS stylesheets and JavaScript scripts in an object storage service, and use this as a simple website backend. Most cloud providers provide a very simple way of accessing these objects via HTTP URLs.

It is common to place a content delivery network (CDN) service on top of the storage bucket of a website in order to accelerate and cache these files at multiple places on the planet. This is particularly useful

if very high traffic is expected over short periods of time, for example for delivering a file that many people will rush to download at the same time. For the user, content delivered with a CDN feels instantaneous.

3.8.2 Dataset storage

Object stores are popular as data lakes, in the sense that datasets can be stored as objects in any desired format (CSV, JSON, ...).

As we will see later, a dataset often consists of several objects rather than just one. Hence, even though the size of an object is typically limited (e.g., 5 TB on S3), the size of a dataset is not. This way of splitting a dataset into multiple chunks is called sharding and will play a fundamental role throughout this course, as we will see when we study MapReduce and Spark.

3.9 Azure Blob Storage

Azure Blob Storage is another object storage technology in the Azure cloud.

It is similar in spirit to Amazon S3, in that it is a fully managed cloud storage service. However, it differs from S3 in several ways:

- Its architecture is publicly documented in scientific papers, making it easier to understand how it works.
- Objects are identified with three, rather than two IDs: An Account, a Container (called partition in scientific papers) and a Blob.
- Azure Blob Storage exposes more details to the user. In particular, it exposes that objects are divided in several blocks more prominently than Amazon S3 (in S3, only advanced users would actually know and access blocks within an object, while the frontend hides them).
- it differentiates between Block Blobs, Append Blobs (for logging) and Page Blobs (for storing and accessing the memory of virtual machines).
- the maximum sizes are different and go from 195 GB for an Append Blob to 190.7 TB for a Block Blob.

Block storage will be covered in more details in the next chapter on HDFS.

On the physical level, Azure Blob Storage is organized in so-called storage stamps located in various data centers worldwide. Each storage stamp consists of 10 to 20 racks, with each rack containing around 18

storage nodes (the disks + servers). In all, a storage stamp can store up to ca. 30 PB of data. However, a storage stamp will not be filled more than 80% of its total capacity in order to avoid being full: if a storage stamp reaches capacity, then some data is going to be reallocated to another storage stamp in the background. And if there are not enough storage stamps, well new racks will need to be purchased and installed at the locations that make the most sense.

Chapter 4

Distributed file systems

There is Big Data and Big Data.

In the previous chapter, we saw that cloud storage services provide a global service that is able to store billions, trillions of objects. We also saw that the size of these objects is limited, for example in Amazon S3, an object cannot exceed 5 TB. In other words, they can store huge amounts of large files.

What about large amounts of huge files? Is it possible to store individual objects bigger than a single machine? The answer is yes. It requires a shift in paradigm that:

- brings back the file hierarchy natively;
- supports block-based storage natively.

4.1 Main requirements of a distributed file system

Data goes through a lifecycle: there is the raw data that is directly collected from the real world (e.g., sensors, logs, etc). Later in the pipeline, this data gets processed, analyzed and turned into derived datasets. These datasets need to be written back to a large-scale storage backend. In turn, they might be processed again leading to more derived datasets, etc.

Such a storage backend must first be resilient to failure. While a single hard drive might occasionally fail, as some of us probably already experienced with their laptop, the nodes in a cluster with 1,000 machines and more is *guaranteed* to fail. Not once, not twice, all the time. Thus, the storage technology must be capable of

- monitoring itself;
- detecting failures;

- automatically recovering;
- being, in the end and as a whole, fault tolerant.

The access mode is also an important feature of the storage backend. On a laptop hard drive, random access is paramount: any part of the disk can be read and written at any time, and in any order. Of course, accessing data randomly and back and forth will be slower than reading a file stored contiguously on disk, or writing a file as a stream and in one go. But laptop hard drives are *designed* to be practically usable for random access.

For distributed data storage though, and for the use case at hand where we read a large dataset, analyze it, and write back the output as a new dataset, random access is not needed. Rather, we need to be able to:

- efficiently scan a large file (in its entirety) – for data analysis
- append efficiently new data at the end of an existing large file – particularly for logging and sensors

Furthermore, this must be supported even with hundreds of concurrent users reading and writing to and from the same file system.

Going back to our capacity-throughput-latency view of storage, a distributed file system is designed so that, in cruise mode, its bottleneck will be the data flow (throughput), not the latency. This aspect of the design is directly consistent with a full-scan pattern, rather than with a random access pattern, the latter being strongly latency-bound (in the sense that most of the time is spent waiting for data to arrive).

We saw that capacity increased much faster than throughput, and that this can be solved with parallelism;

We saw that throughput increased much faster than latency decreased, and that this can be solved with batch processing.

Distributed file systems support both parallelism and batch processing natively, forming the core part of the ideal storage system accessed by MapReduce or Apache Spark.

The origins of such a system come back to the design of GoogleFS, the Google File System. Later on, an open source version of it was released as part of the Hadoop project, initiated by Doug Cutting at Yahoo, and called HDFS, for Hadoop Distributed File System. The Hadoop projects further includes other open source releases of MapReduce (with the same name as Google's original MapReduce) and HBase (corresponding to Google's BigTable).

Between 2006 and 2016, in just ten years, the cluster size running HDFS successfully went from 188 nodes to almost 42,000 nodes (and by now probably more), storing hundreds of Petabytes of data.

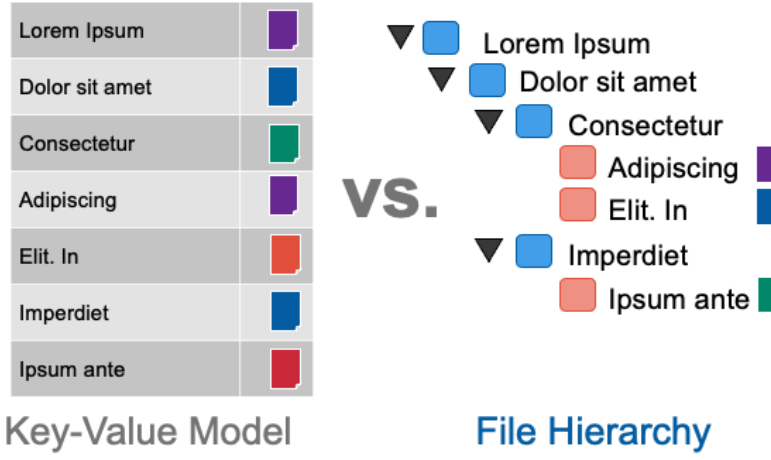
4.2 The model behind HDFS

In almost every chapter of this course, we will carefully introduce the logical level separately from its physical implementation. Thus, we start by describing HDFS on the logical level.

4.2.1 File hierarchy

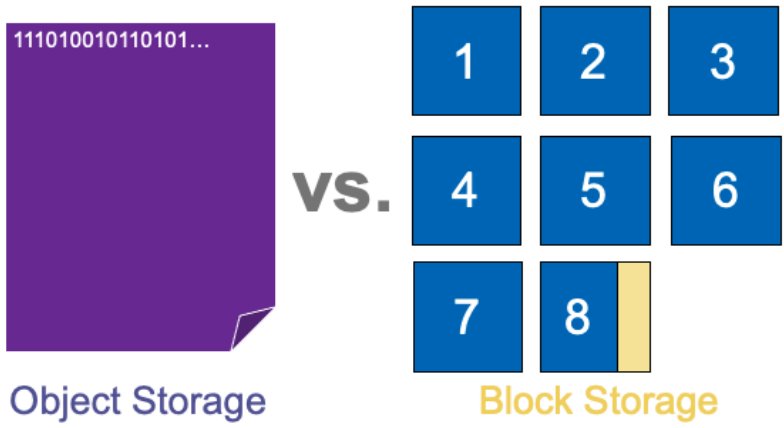
First, in HDFS, the word "file" is used instead of "object", although they correspond to S3 objects logically.

HDFS does not follow a key-value model: instead, an HDFS cluster organizes its files as a hierarchy, called the file namespace. Files are thus organized in directories, similar to a local file system.



4.2.2 Blocks

Unlike in S3, HDFS files are furthermore not stored as monolithic black-boxes, but HDFS exposes them as lists of blocks – also similar to a local file system.

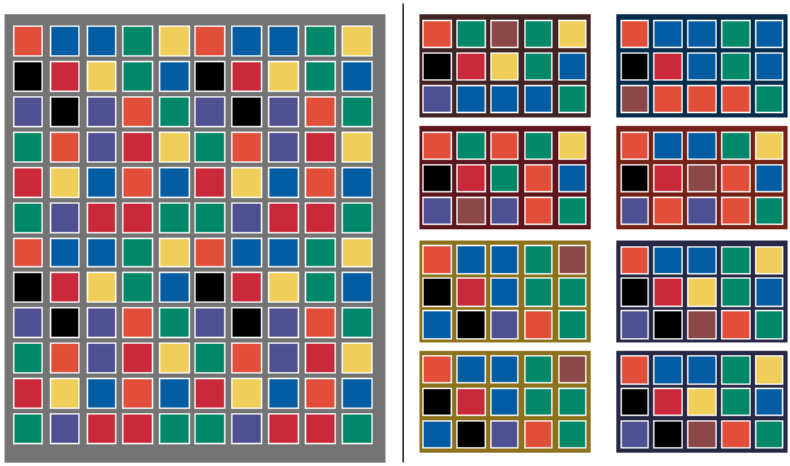


In Google's original file system, GFS, blocks are called chunks. In fact, throughout the course, many words will be used, in each technology, to describe partitioning data in this way: blocks, chunks, splits, shards, partitions, and so on. It is more important to understand that they are almost interchangeable, while it is secondary to learn "by heart" which technology uses which word.

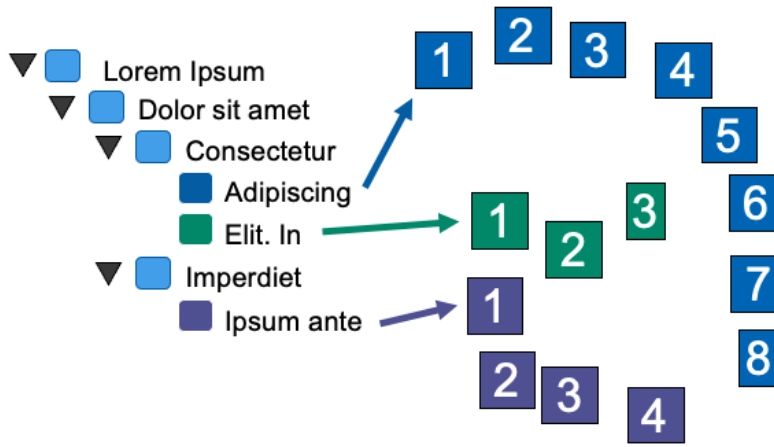
Why does HDFS use blocks?

First, because a PB-sized file surely does not fit on a single machine as of 2022: files have to be split in some way. Second, it is a level of abstraction simple enough that it can be exposed on the logical level.

Second, blocks can easily be spread at will across many machines, a bit like a large jar of cookies can easily be split and shared among several people.



Putting it all together, the following picture summarizes best the logical model of an HDFS cluster:



4.2.3 The size of the blocks

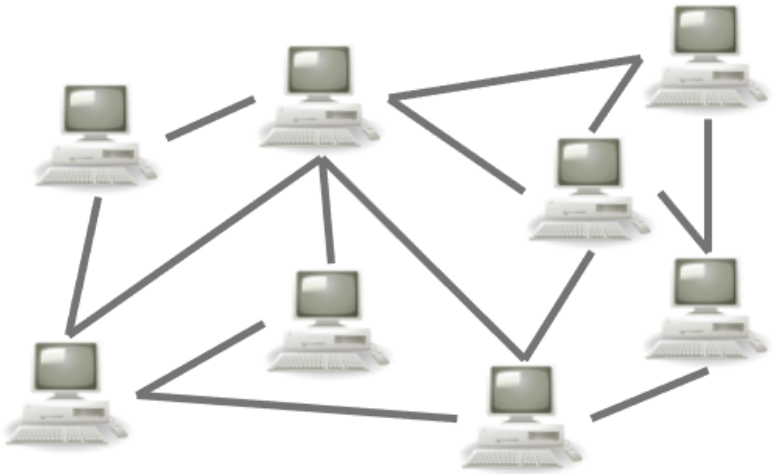
As for the block size: HDFS blocks are typically 64 MB or 128 MB large, and are thus considerably larger than blocks on a local hard drive (around 4 kB). This is because of the prerequisites of HDFS: first, it is not optimized for random access, and second, blocks will be shipped over the network.

Thus, blocks of 128 MB are large enough that time is not lost in latency waiting for a block to arrive, i.e., access to the HDFS cluster will be throughput-bound during a full-scan analysis with MapReduce or Spark. And at the same time, they are small enough for a large file to be conveniently spread over many machines, allowing for parallel access, and also small enough for a block to be sent again without too much overhead in case of a network issue.

4.3 Physical architecture

HDFS is designed to a run on a cluster of machines. The number of machines can range from just a handful of them to thousands of machines. How are they connected?

One to connect machines is called a peer-to-peer, decentralized network. In a peer-to-peer network, each machine talks to any other machine. This architecture, for example, is used in the Bitcoin blockchain and was also popular in the 1990s with the Napster network.



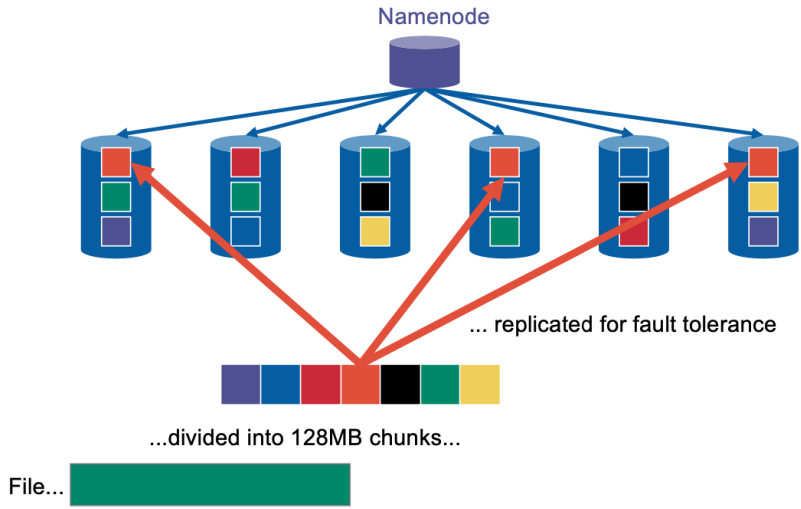
By contrast, HDFS is on the contrary implement on a fully centralized architecture, in which one node is special and all others are interchangeable and connected to it.

The terminology used to describe the nodes in a centralized network architecture varies in the literature. In the case of HDFS, the central node is called the NameNode and the other nodes are called the DataNodes. In fact, more precisely, the NameNode and DataNodes are

processes running on these nodes, and the CamelCase notation often used to write their names down corresponds to the Java classes implementing these processes. By metonymy, we also use these names to describes the nodes themselves.

Now that we have the NameNode and DataNodes in place, how is the logical HDFS model implemented on top of this architecture?

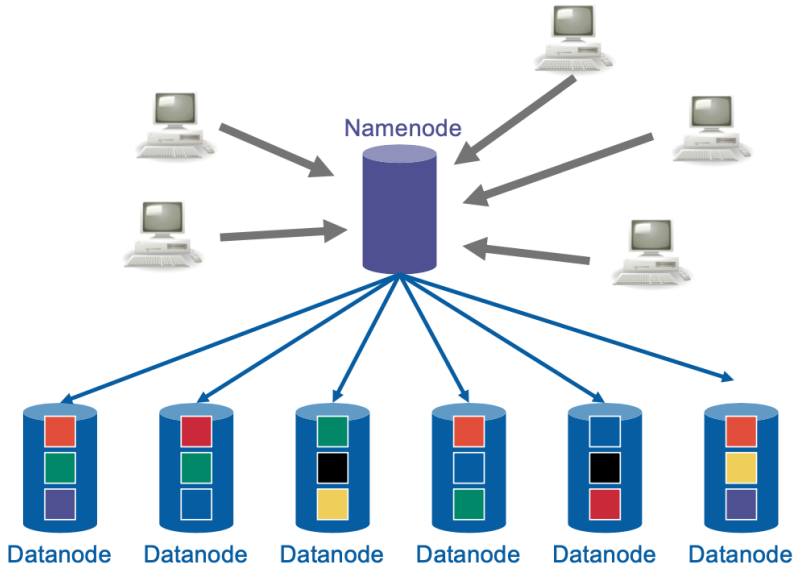
As we saw before, every file is divided into chunks called blocks. All blocks have a size of exactly 128 MB, except the last one which is usually smaller (indeed, what are the odds that the file size would be exactly a multiple of 128 MB?).



Each one of the blocks is then replicated and stored on several DataNodes. How many times? This is a parameter called the *replication factor*. By default, it is 3, but this can be changed by the user.

There is no such thing as a primary replica: all replicas are on an equal footing. In other words, by default, three replicas of each block are stored by default. Overall, in a production instance, there is a very high number of block replicas and they are spread more or less evenly across the nodes, to have a certain balance.

An HDFS cluster is then accessed concurrently by many clients. Typically, the cluster is owned by one company or organization and centrally managed by a specialized team (e.g., site reliability engineers). The clients correspond to users within this organization who read from and write to the HDFS cluster.



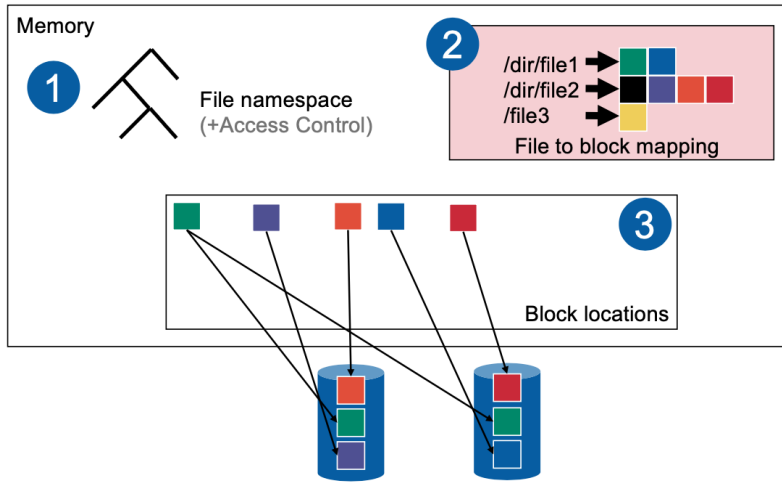
4.3.1 The responsibilities of the NameNode

The NameNode is responsible for the system-wide activity of the HDFS cluster. It store in particular three things:

- the file namespace, that is, the hierarchy of directory names and file names, as well as any access control (ACL) information similar to Unix-based systems.

- a mapping from each file to the list of its blocks. Each block, in this list, is represented with a 64-bit identifier; the content of the blocks is not on the NameNode.

- a mapping from each block, represented with its 64-bit identifier, to the locations of its replicas, that is, the list of the DataNodes that store a copy of this block.



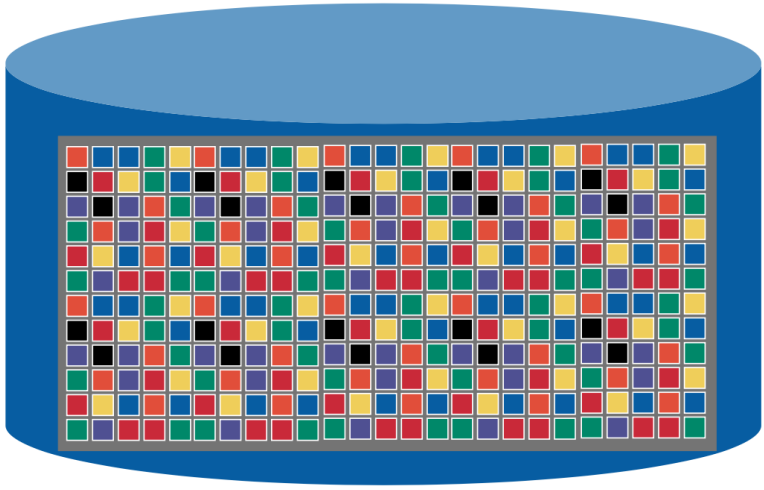
The NameNode updates this information whenever a client connects to it in order to update files and directories, as well as with the regular reports it receives from the DataNodes.

Clients connect to the NameNode via the Client Protocol. Clients can perform metadata operations such as creating or deleting a directory, but also ask to delete a file, read a file or write a new file. In the latter case, the NameNode will send back to the client block identifiers (for reading them), or lists of DataNode locations (for reading and writing them).

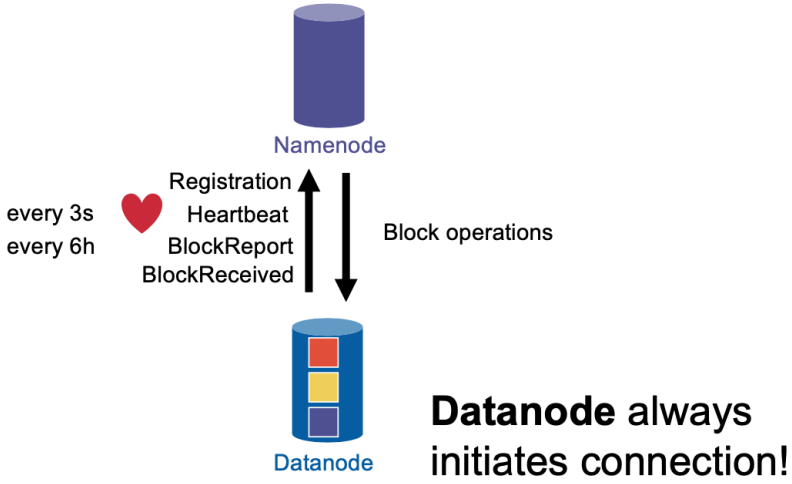
4.3.2 The responsibilities of the DataNode

The DataNodes store the blocks themselves. These blocks are stored on their local disks[1]. Each block is stored as a 128 MB local, physical file on the DataNode. If the block is the last block of an HDFS file and thus less than 128 MB, then the physical file has exactly the size of the block: there is no waste of space. This is different from the physical blocks (4 kB or so) of a local hard drive, which have a constant size.

[1] In a cloud installation, these local disks can also be virtually attached, but this is the same from the perspective of the DataNode process.



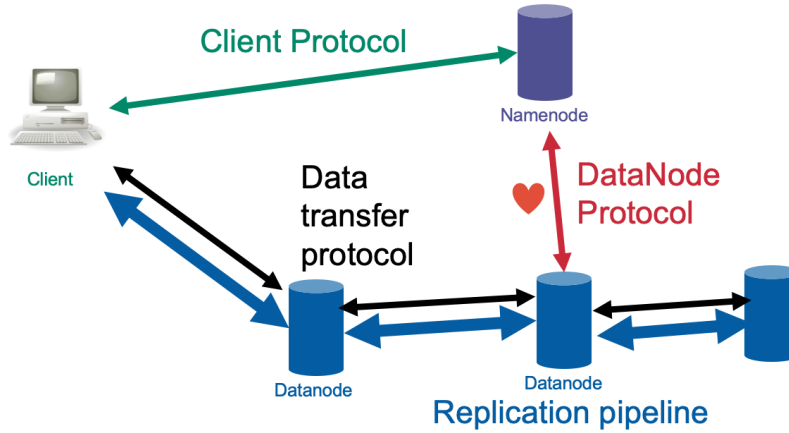
DataNodes send regular heartbeats to the NameNode. The frequency of these heartbeats is configurable and is by default a few seconds (e.g., 3s, but this value may change across releases). This is a way to let the NameNode know that everything is alright. The heartbeat may also contain a notification to the NameNode that a new block was received and successfully stored, and that the DataNode can thus be used as a location for this block. Finally, the DataNode also sends, every couple of hours (e.g., 6h, but this value may change across releases), a full report including all the blocks that it contains.



If there is an issue with the local disk or the node, then the DataNode can report the block as corrupted to the NameNode, which will then ensure, asynchronously, its replication to somewhere else. "Asynchronously," as opposed to "synchronously,", means that this is done in the background at some later time and the DataNode does not idly wait for this to happen. In the meantime, the block is simply marked as underreplicated.

A NameNode never initiates a connection to a DataNode. If the NameNode needs anything from a DataNode, for example, if it needs to request a DataNode to download an underreplicated block from another DataNode and store a new replica of it, then the NameNode will wait until the next heartbeat, and answer to it with this request.

Finally, DataNodes are also capable of communicating with each other by forming replication pipelines. A pipeline happens whenever a new HDFS file is created. The client does not send a copy of the block to all the destination DataNodes, but only to the first one. This first DataNode is then responsible for creating the pipeline and propagating the block to its counterparts.



When a replication pipeline is ongoing and a new block is being written to the cluster, the content of the block is not sent in one single 128 MB packet. Rather, it is send in smaller packets (e.g., 64 kB) in a streaming fashion via a network protocol. That way, if some packets are missing, the client can send them again. The client receives the acknowledgements in a streaming fashion as well.

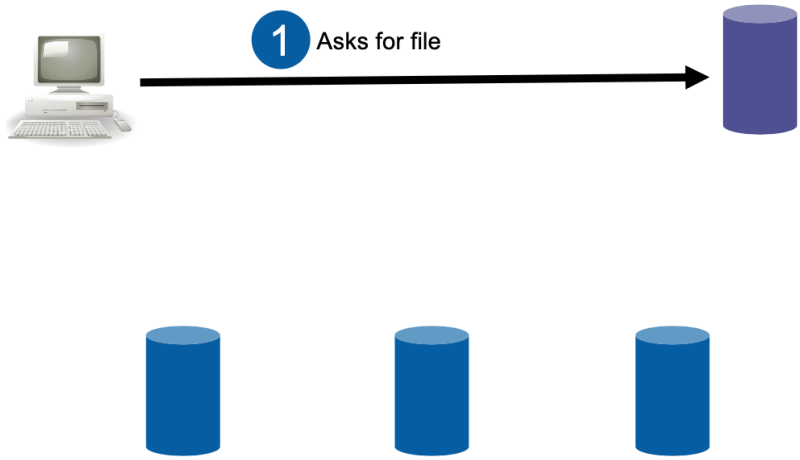
4.3.3 File system functionality

HDFS exposes to the client, via the NameNode, an API that allows typical operations available on a file system: creating a directory, deleting

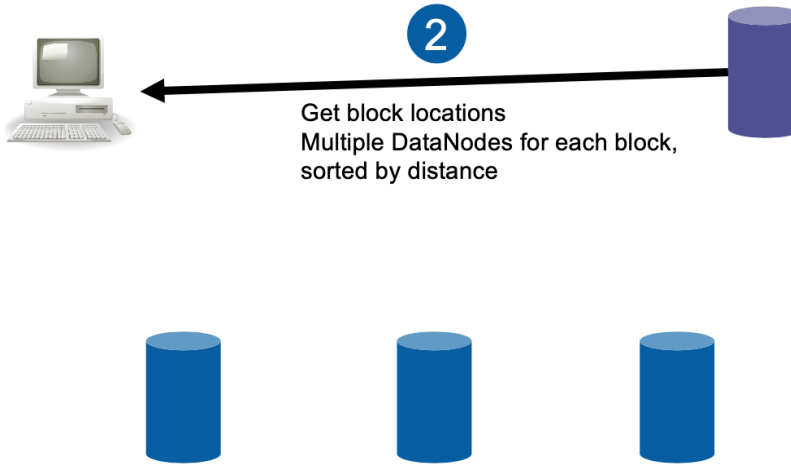
a directory, listing the contents of a directory, creating a file, appending to a file, reading a file, deleting a file.

Many of these involve the NameNode only. However, reading, writing and deleting a file involves communication across the cluster, which we now detail.

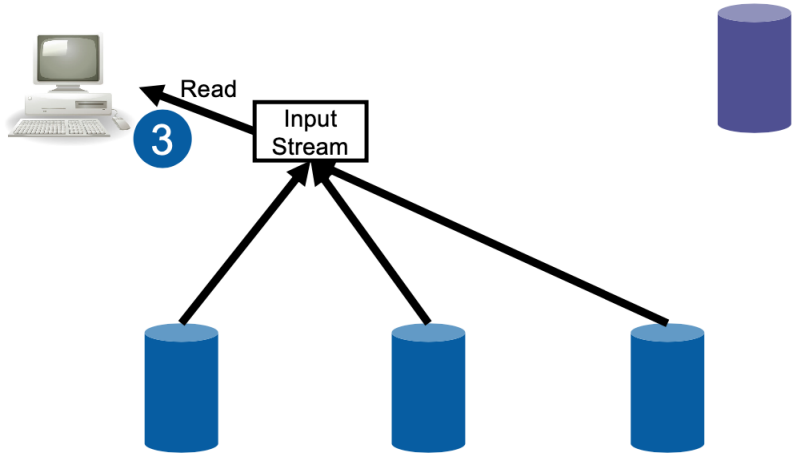
Let us start with reading a file. The client first connects to the NameNode to initiate the read and request info on the file.



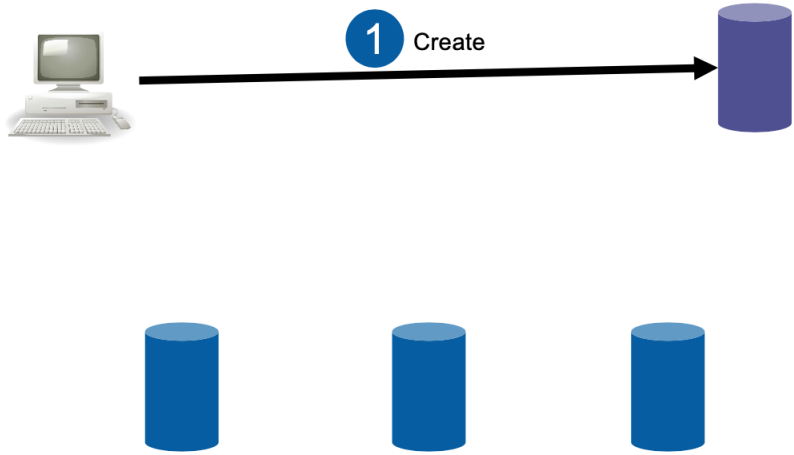
Then, the NameNode responds with the list of all blocks, as well as a list of DataNodes that contains a replica of each block. The DataNodes are furthermore sorted by increasing distance of the client (which is typically itself one of the nodes in the cluster, we will come back to this).



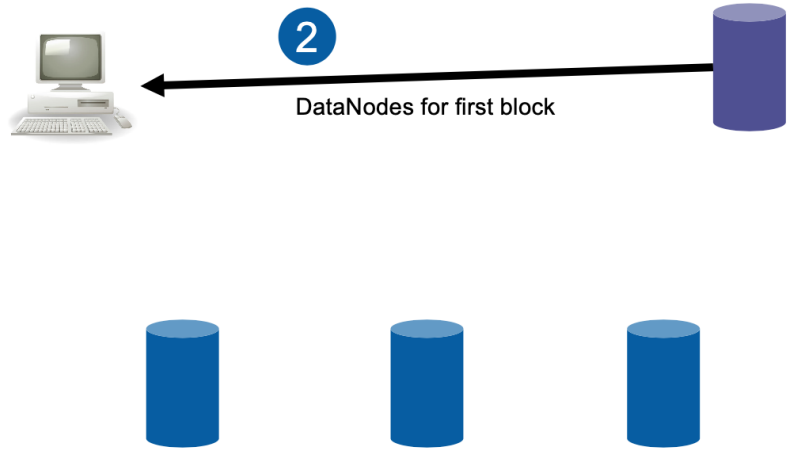
The client then connects to the DataNodes in order to download a copy of the blocks. It starts with the first (closest) DataNode in the list provided by the NameNode for each blocks, and will go down the list if a DataNode cannot be reached or is not available. In the simple case that the client wants to stream its way through an HDFS file, bit by bit, it will download each block in turn. However, this functionality is typically nicely encapsulated in additional, client-side Java libraries that expose this as a "single big input stream" with the InputStream classes familiar to Java programmers. Switching between the DataNodes is hidden and the user just sees a stream of bits flowing through. Note that with streaming, it is possible to process files larger than the working memory, because older blocks can be thrown away from the memory of the client once processed. This is something we will come back to when we look into the architecture of querying engines. Alternatively, the client can also download a multi-block HDFS file and store it as a single big file on its local drive as long as it fits.



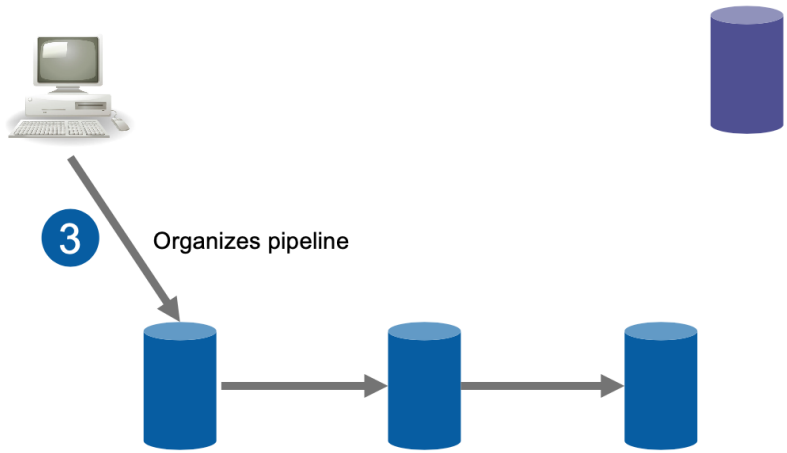
Let us now write a new file to HDFS. This can be either a simple upload of a large local file, or it can be from a stream of bits created on the fly by a program. As for reading, the client first connects to the NameNode formulating its intent to create a new file.



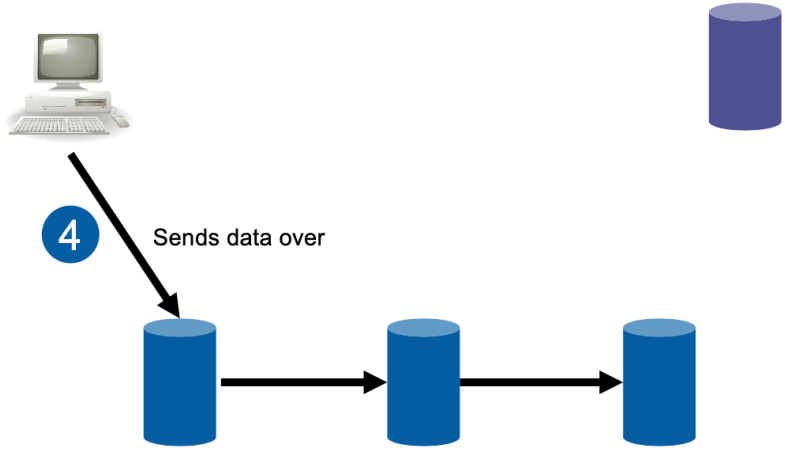
The NameNode then responds with a list of DataNodes to which the content of the first block should be sent. Note that, at that point, the file is not yet available for read for other clients (it is locked).



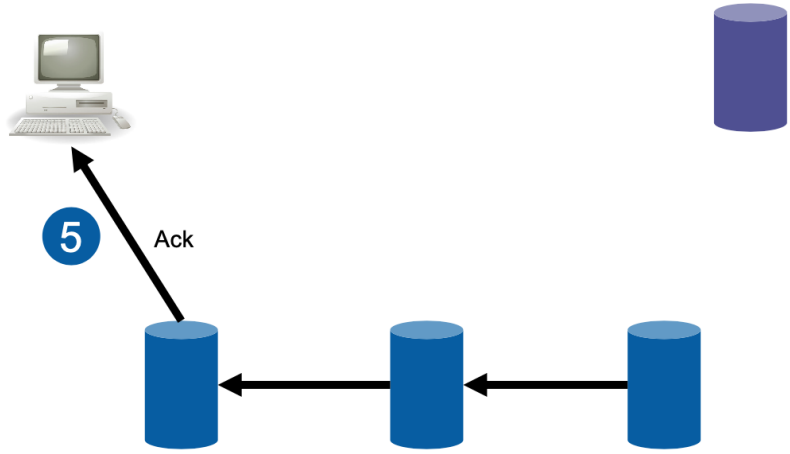
The client then connects to the first DataNode and instructs it to organize a pipeline with the other DataNodes provided by the NameNode for this block.



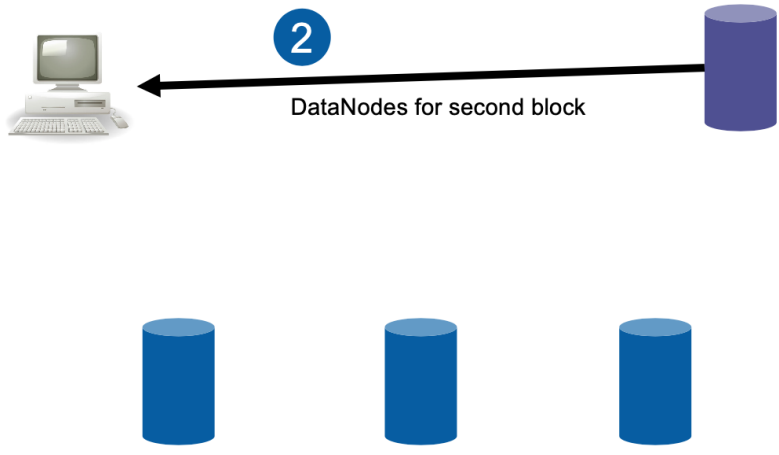
The client then starts sending through the content of that block, as we explained earlier. The content will be pipelined all the way to the other DataNodes.



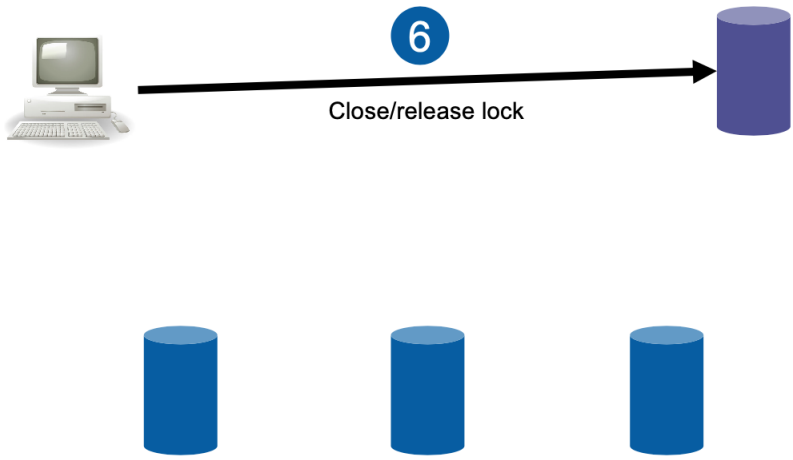
The client receives regular acknowledgements from the first DataNode that the bits have been received.



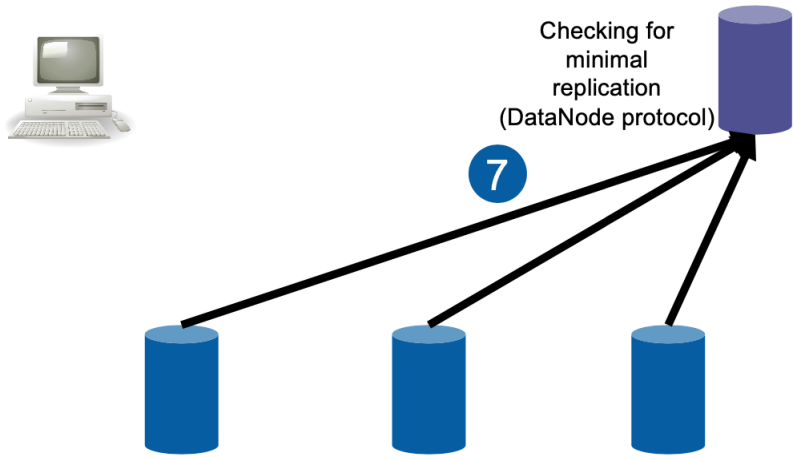
When all bits have been acknowledged, the client connects to the NameNode in order to move over to the second block. Then, the same steps (2, 3, 4, 5) as before are repeated for each block.



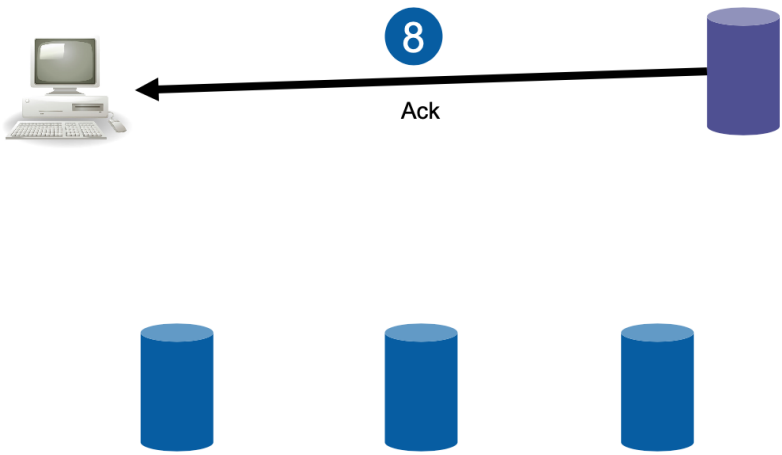
Once the last block has been written, the client informs the NameNode that the file is complete, and asks to release the lock.



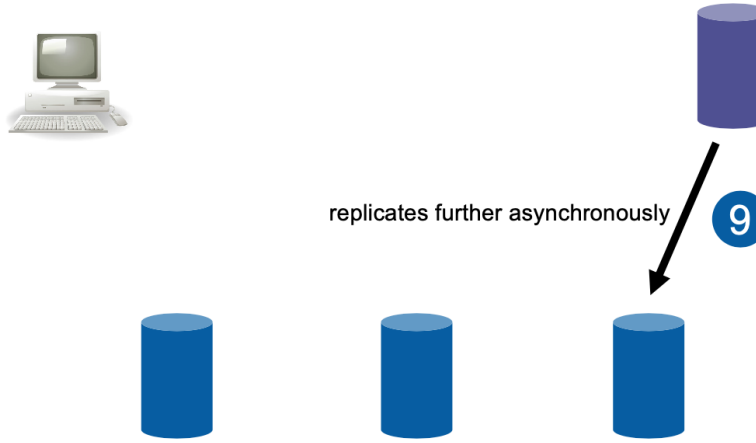
The NameNode then checks for minimal replication through the DataNode protocol.



... and gives its final acknowledgement to the client.



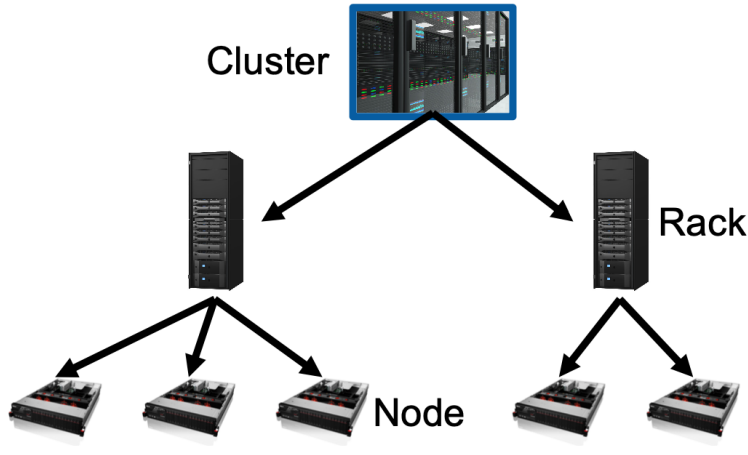
From now on, and separately (this is called "asynchronous"), the NameNode will continue to monitor the replicas and trigger copies from DataNode to DataNode whenever necessary, that is, when a block is underreplicated.



4.4 Replication strategy

By default, three replicas of each block are stored, although this can be changed by users for every file. Note that there is no such thing as a "main replica", there are just several replicas.

The placement strategy of replicas is based on knowledge of the physical setup of the cluster. As you may recall from the previous chapter, servers, called nodes, are organized in racks, and several racks are placed in the same room or data center. This gives a natural tree topology.



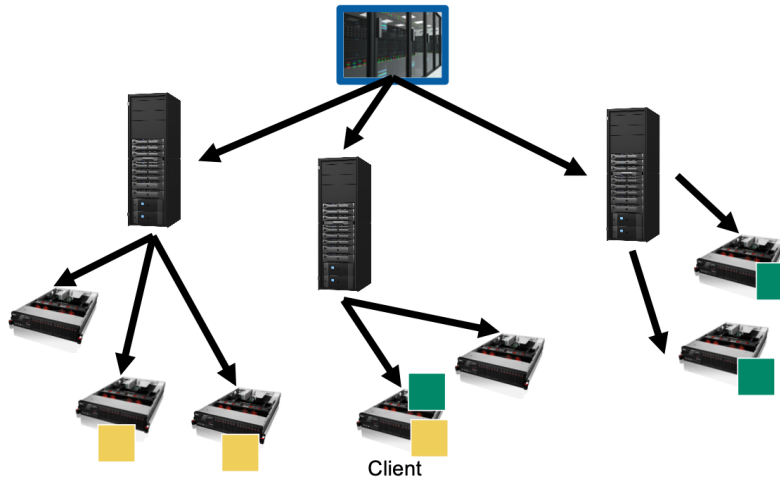
With this topology in mind, one can define a notion of distance between two nodes: two nodes in the same rack have a distance of 2 (one edge from the first node to the rack, and one from the rack to the other node). Two nodes in different racks have a distance of 4 (one edge from the first node to the first rack, one from the first rack to the data center, one from the data center to the other rack, one from the other rack to the other node). It is this distance that can then be used by the NameNode to sort DataNodes by increasing distance from the client.

Now, the strategy for placing replicas of a new blocks is as follows. First, it is important to understand that, in practice, the client that is writing the file is a process running on one of the nodes of the HDFS cluster. This will become obvious when we study massive parallel computing (e.g., MapReduce and Spark), where reading and writing is done in a distributed fashion over the same cluster. But even without parallel computing, when users create clusters in public clouds such as AWS or Azure or Google Cloud, they connect to the machines using SSH (a safe protocol for remotely accessing a machine). Thus, anything they will do will originate from the machine they are connected to, which is in the cluster.

Having this in mind, the first replica of the block, by default, gets written to the same machine that the client is running on – keep in mind that this machine is typically a DataNode (again, this will become obvious when we look into MapReduce and Spark).

The second replica is written on a DataNode sitting in a different rack than the client, that we call B. The third replica is written to another DataNode on the same rack B. And further replicas are written mostly at random, but respecting two simple rules for resilience: at most one replica per node, and at most two replicas per rack.

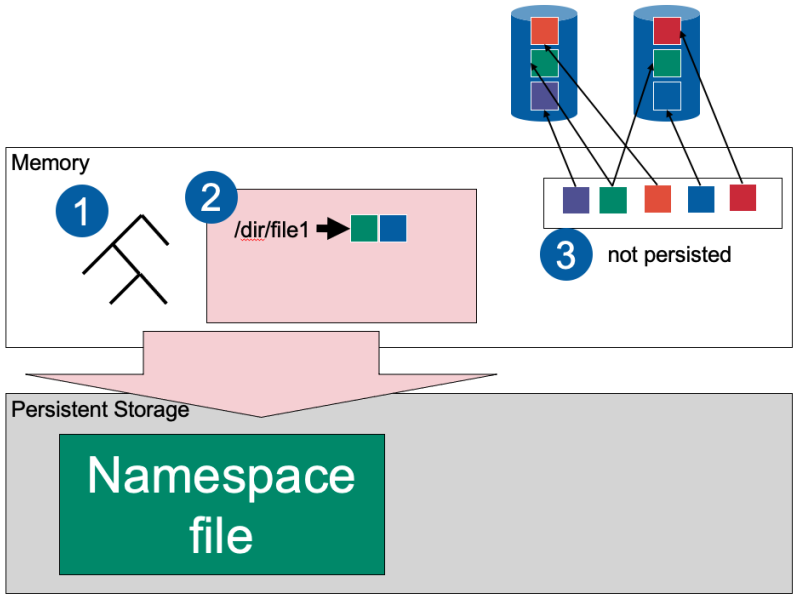
Below is an example with the three replicas of two different blocks, written by the same client.



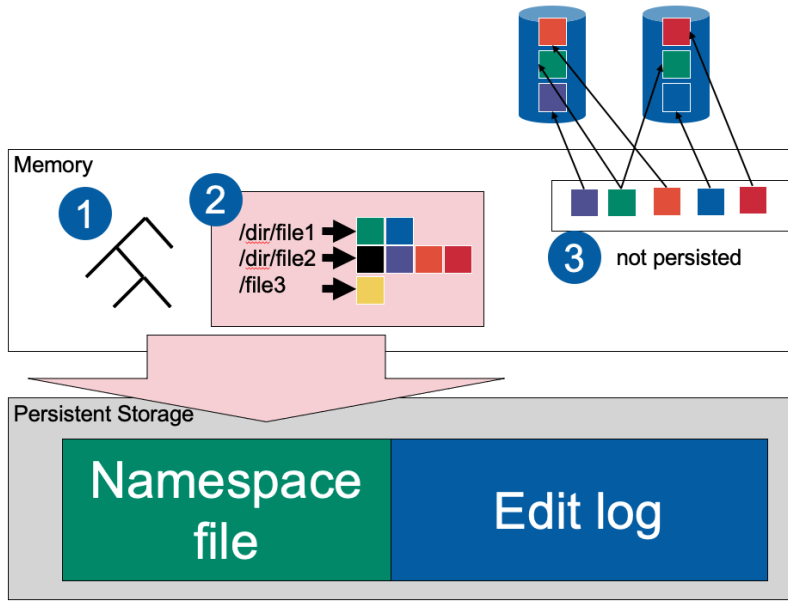
4.5 Fault tolerance and availability

HDFS has a single point of failure: the NameNode. If the metadata stored on it is lost, then all the data on the cluster is lost, because it is not possible to reassemble the blocks into files any more.

For this reason, the metadata is backed up. More precisely, the file namespace containing the directory and file hierarchy as well as the mapping from files to block IDs is backed up to a so-called snapshot. Note that the mapping of block IDs to DataNodes does not require a back, as it can be recovered from the periodic block reports.

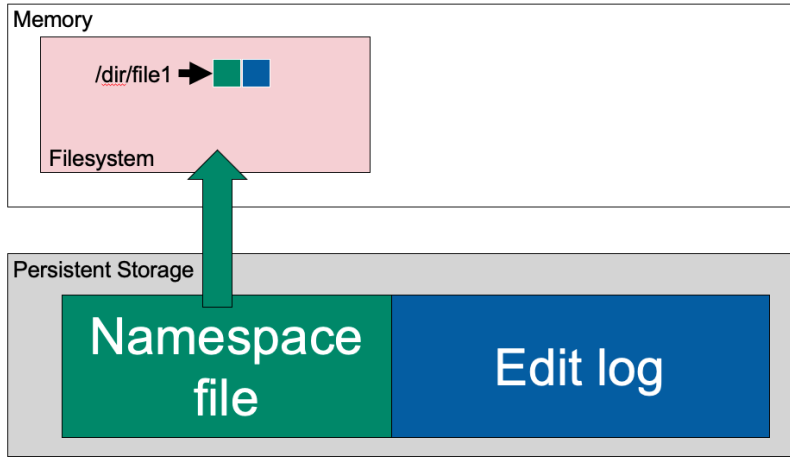


Since the HDFS system is constantly updated, it would not be viable to do a backup upon each update. It would also not be viable to do backups less often, as this could lead to data loss. Thus, what is done is that updates to the file system arriving after the snapshot has been made are instead stored in a journal, called edit log, that lists the updates sorted by time of arrival.

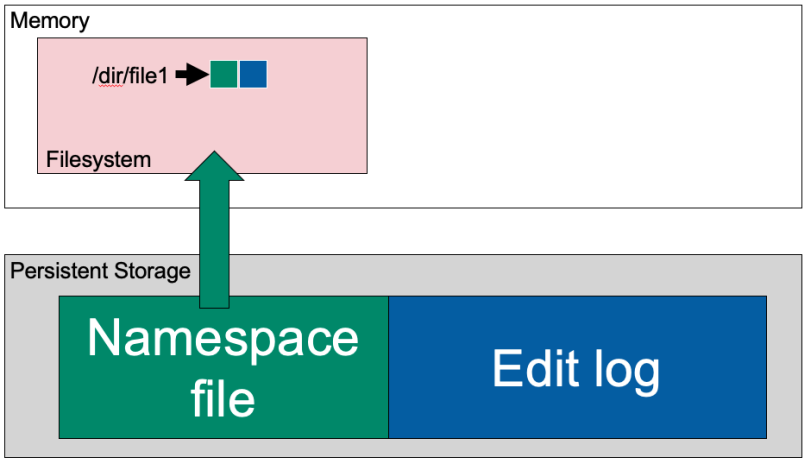


The snapshot and edit log are stored either locally or on a network-attached drive (not HDFS itself). For more resilience, they can also be copied over to more backup locations.

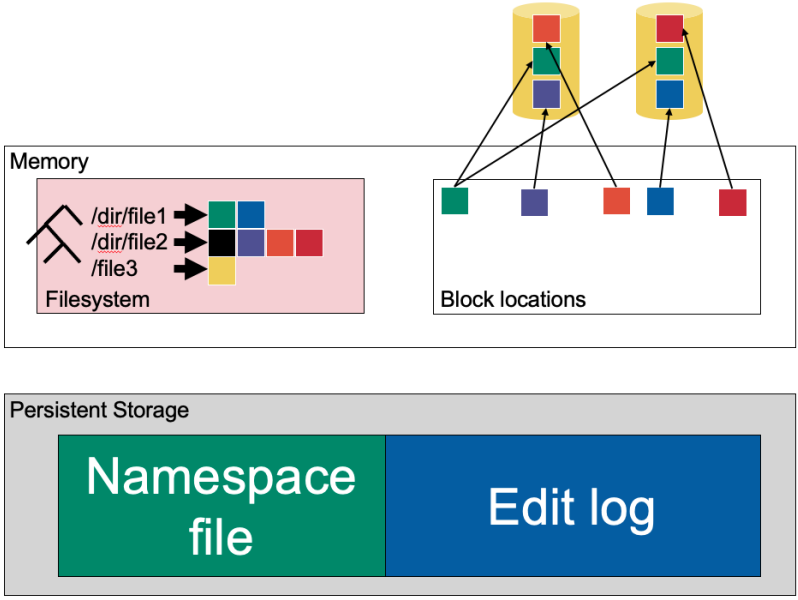
If the NameNode crashes, it can be restarted, the snapshot can be loaded back into memory to get the file namespace and the mapping of the files to block IDs.



Then the edit log can be replayed in order to apply the latest changes.



And the NameNode can wait for (or trigger) block reports to rebuild the mapping from block IDs to the DataNodes that have a replica of them.

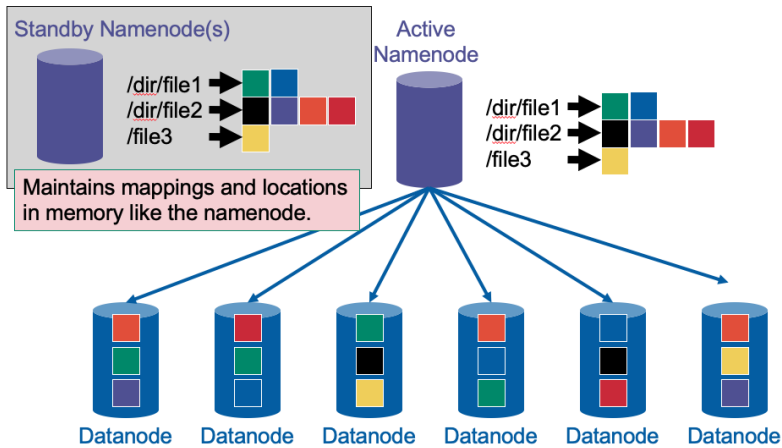


However, as more and more updates are applied, the edit log grows. This can easily lead to a long delay of 30+ minutes to restart the NameNode after a crash. More strategies had to be put in place. This was done incrementally in each HDFS release, and there are many different kinds of additional NameNodes that came into place and succeeded to one another: the Checkpoint NameNode, the Backup NameNode, the Secondary NameNode, the Standby NameNode, etc. We will not go into the details of each one of the version, but summarize the most important take aways.

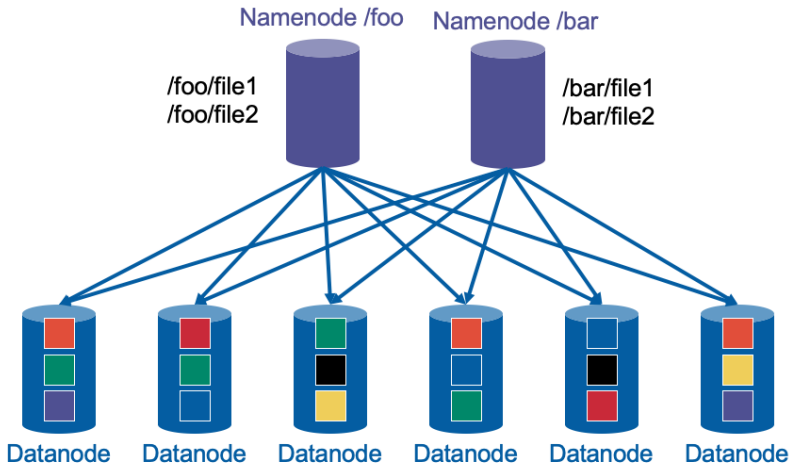
First, the edit log is periodically merged back into a new, larger snapshot and reset to an empty edit log. This is called a checkpoint. This can be done with a "phantom NameNode" (our terminology) that keeps the exact same structures in its memory as the real NameNode, and performs checkpoints periodically.

Second, it is possible to configure a phantom NameNode to be able to instantly take over from the NameNode in case of a crash. This considerably shortens the amount of time to recover.

As of the time of writing, these are called Standby NameNodes, but we do warn that the HDFS architecture will continue to evolve.



Another change to the architecture was made later: Federated NameNodes. In this change, several NameNodes can run at the same time, and each NameNode is responsible for specific (non overlapping) directories within the hierarchy. This spreads the management workload over multiple nodes.



4.6 Using HDFS

One of the ways to access HDFS is via the command line (a shell). The HDFS command line interface is mostly POSIX compliant, which means that the names of the commands are quite similar to those found on Linux or macos.

HDFS command start with

```
hdfs dfs <put command here>
```

Although we recommend to use the hadoop command instead, because it can also connect to other file systems (S3, your local drive, etc).

```
hadoop fs <put command here>
```

You can list the contents of the current directory with:

```
hadoop fs -ls
```

You can print to screen the contents of a file with:

```
hadoop fs -cat /user/hadoop/dir/file.json
```

You can delete a file with:

```
hadoop fs -rm /user/hadoop/dir/file.json
```

You can create a directory with:

```
hadoop fs -mkdir /user/hadoop/dir/file.json
```

You can upload files from your local computer to HDFS with:

```
hadoop fs {copyFromLocal localfile1 localfile2 /user/hadoop/targetdir
```

You can upload a file from HDFS to your local computer with:

```
hadoop fs {copyToLocal /user/hadoop/file.json localfile.json
```

4.7 Paths and URIs

Although in the previous examples we used relative paths, generally, paths are URIs, as seen in the previous chapter. The scheme determines the file system.

This is what an HDFS URI looks like:

```
hdfs://www.example.com:8021/user/hadoop/file.json
```

This is what an S3 URI looks like (there are also the s3a and s3n schemes):

```
s3://my-bucket/directory/file.json
```

In Azure blob storage:

```
wasb://mycontainer@myaccount.blob.core.windows.net/directory/file.jso
```

And on the local file system:

```
file:///home/user/file.json
```

Typically, a default file system can be defined in a configuration file called core-site.xml:

```
<properties> <property>
<name>fs.defaultFS</name>
<value>hdfs://hdfs.mycluster.example.com:8020</value>
<description>NameNode hostname</description>
</property> </properties>
```

Then, the scheme is not needed to access files on this system and an absolute path starting with a slash can be used instead:

```
/user/hadoop/file.json
```

Note that other file systems can still be accessed (if set up properly) by using a URI scheme.

As for relative paths, they are resolved against the working directory, which should be unsurprising to anybody familiar with the command line interface even outside the context of HDFS.

Please mind that the current working directory might be on a different file system than the default file system; then you have situations in which a relative path will be local, while an absolute path will be on cloud, or distributed storage. This can cause unexpected issues. A common mistake is for users to use a relative path within a massive parallel computing framework (MapReduce, Apache Spark) when the working directory is local, which causes the data to be read, or written to the local disk rather than the HDFS cluster. If the job succeeds but the output is nowhere to be seen, it is likely it was just spit all over the local disks of the cluster. You can play treasure hunt and go after each machine to download the results, or you might realize that it will be easier to just rerun your job with the correct paths.

4.8 Logging and importing data

Two tools are worth mentioning: Apache Flume lets you collect, aggregate and move log data to HDFS. Apache Sqoop lets you import data from a relational database management system to HDFS.

This completes the chapter on distributed file systems. The most adventurous readers are encouraged to create a small HDFS cluster (this only takes a few clicks with Amazon EMR or Azure HDInsights), upload a few files, and log onto the machines with SSH to try to locate the physical block replicas as local files on the DataNodes.

Chapter 5

Syntax

5.1 Why syntax

Now that we have a storage layer in place, be it as a public cloud service like S3 or Azure blob storage, or as one's own HDFS cluster, we can start working our way up the Big Data technology stack by looking at what files we actually store.

We already mentioned that the public cloud is very popular for storing pictures, videos (for streaming and video on demand) as well as for static website hosting. But the public cloud is also great for storing datasets. Storing datasets directly in the public cloud or on a distributed file system is also known as a data lake.

What makes a data lake different from the way a classical database management system works is that the latter stores all its data in a proprietary format, hidden from the user. Any data that enters the system must be imported, more precisely, ETLed[1]. ETLing brings a lot of bells and whistles such as faster queries through efficient storage formats, indices, and so on. But ETLing also takes time and efforts and it might not be worth it in all cases. In-situ querying (querying in place) increased in importance in the last decades.

In a data lake, the syntax of the data is exposed to the user. A prominent example is that of CSV files that are known beyond the computer science community. CSV is syntax for tabular data.

[1] ETL is for Extract, Transform, Load

5.1.1 CSV

```
ID,Last name,First name
1,Einstein,Albert
2,Gödel,Kurt
```

ID	Last name	First name
1	Einstein	Albert
2	Gödel	Kurt

CSV is a textual format, in the sense that it can be opened in a text editor. This is in contrast to binary formats that are more opaque.

Each record (a table row) corresponds to one line of text in a CSV file. Having a record per line of text is a common pattern not unique to CSV, as we will see later in this course. This is what makes it possible to scale up data processing to billions of records.

What appears on each line of text is specific to CSV. CSV means comma-separated values.

The main challenge with CSV files is that, in spite of a standard (RFC 4180), in practice there are many different dialects and variations, which limits interoperability. For example, another character can be used instead of the comma (tabulation, semi-colons, etc). Also, when a comma (or the special character used in its stead) needs to actually appear in a value, it needs to be escaped. There are many way to do so; one of them is to double-quote the cell, which implies in turn that quotes within quotes must be escaped. There are many different conventions for doing so.

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

5.1.2 Data denormalization

We saw in a previous chapter that it is desirable to store data in so-called normal forms in a relational database management system. As you may recall, data in the first normal form cannot nest, and data in higher normal forms are split across multiple tables that get joined at query time. As a rule of thumb, normalizing data means joining it back at query time.

In the context of data lake and large-scale data processing, it is often desirable to go exactly the opposite way. This is called data denormalization. This means that not only several tables can be merged

into just one (with functional dependencies that would otherwise have been considered "undesirable"), it also means that we can nest data: tables in tables in tables.

While this is likely to come as a shock to any student who spent hours learning normal forms in a Bachelor's level database lecture (widespread in European curricula, in particular in German-speaking countries), it has to be said that data denormalization should be done with knowledge of normal forms, because one needs to have a deep understanding of what one is doing and why one is doing it.

Data denormalization makes a lot of sense in read-intensive scenarios in which not having to join brings a significant performance improvement. In read-intensive scenarios, we love anything that is linear, which corresponds to a full scan of the dataset. This is as opposed to point queries more commonly found in traditional databases.

Thanks to the way that we defined tables in chapter 2, data denormalization is straightforward to explain. A table is a collection of tuples.

We required identical support (relational integrity), flat rows (atomic integrity, which is also the first normal form), and homogeneous data types within a column (domain integrity). Denormalization simply means that we drop all three constraints (or two, or just one).

Let us dive into this.

A tuple, mathematically, can be formalized as a partial function mapping strings to values:

$$\begin{aligned} \text{product} &\mapsto \text{Phone} \\ \text{price} &\mapsto 800 \\ \text{customer} &\mapsto \text{John} \\ \text{quantity} &\mapsto 1 \end{aligned}$$

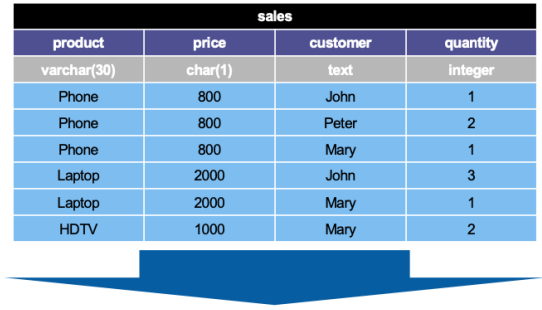
$$\mathbb{S} \qquad\qquad \mathbb{V}$$

As it turns out, a tuple can also be represented in a purely textual fashion (we will see this is called JSON – we will learn JSON in details later in this chapter).

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

The difference with CSV is that, in JSON, the attributes appear in every tuple, while in CSV they do not appear except in the header line. JSON is appropriate for data denormalization because including the attributes in every tuple allows us to drop the identical support requirement.

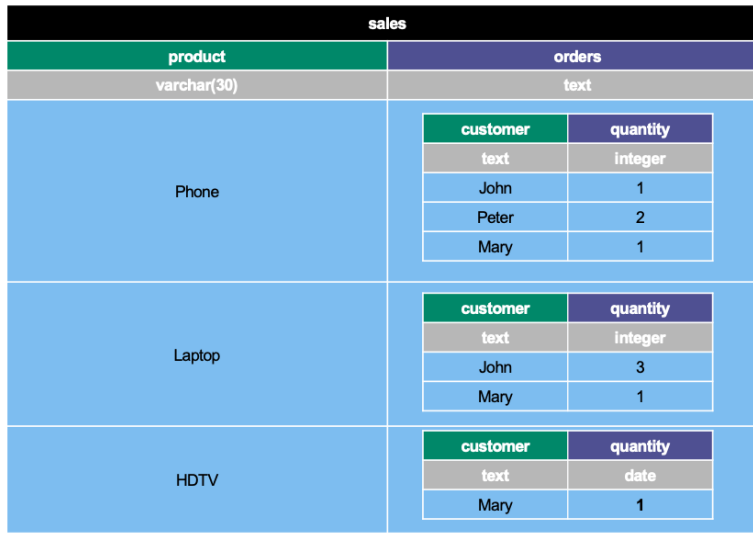
If we now look at a table (which checks all three integrity boxes), we can re-express it in a JSON-based textual format like so:



sales			
product	price	customer	quantity
varchar(30)	char(1)	text	integer
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

```
{ "product" : "Phone", "price" : 800, "customer" : "John", "quantity" : 1 }
{ "product" : "Phone", "price" : 800, "customer" : "Peter", "quantity" : 2 }
{ "product" : "Phone", "price" : 800, "customer" : "Mary", "quantity" : 1 }
{ "product" : "Laptop", "price" : 2000, "customer" : "John", "quantity" : 1 }
{ "product" : "Laptop", "price" : 2000, "customer" : "Mary", "quantity" : 1 }
...
```

Now, if we are to drop relational integrity and allow for nestedness, the table could look like so:

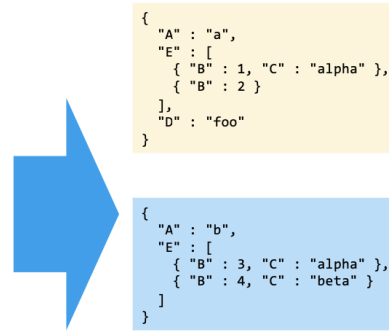


CSV would not be powerful enough to express such data. But JSON is able to. For example, the first tuple of the table above, expressed in JSON, looks like so:

```
{
  "product" : "Phone",
  "orders" : [
    { "customer" : John, "quantity" : 1 },
    { "customer" : Peter, "quantity" : 2 },
    { "customer" : Mary, "quantity" : 2 }
  ]
}
```

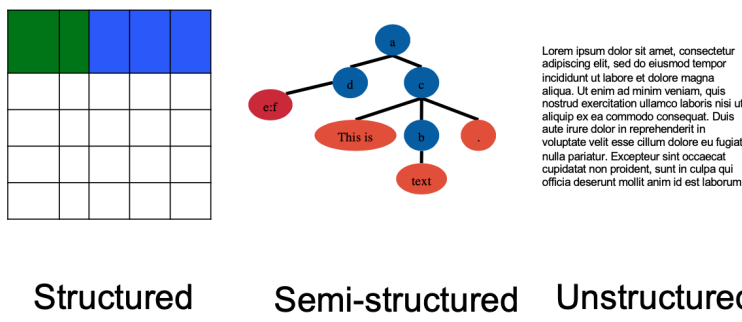
Concretely, data denormalization means that we abandon the paradigm of homogeneous collections of flat items (tables) and instead consider heterogeneous collections of nested items;

A	B	C	D
a	1	alpha	foo
a	2	NULL	foo
b	3	alpha	NULL
b	4	beta	NULL



5.2 Semi-structured data and well-formedness

The generic name for denormalized data (in the same of heterogeneous and nested) is "semi-structured data". Textual formats such as XML and JSON have the advantage that they can both be processed by computers, and can also be read, written and edited by humans. As we will see in Chapter **??**, though, binary formats exist, too, although they are typically less denormalized.



Another very important and characterizing aspect of XML and JSON is that they are standards: XML is a W3C standard. W3C, also known as the World Wide Web consortium, is the same body that also standardizes HTML, HTTP, etc. JSON is now an ECMA standard, which is the same body that also standardizes JavaScript. In fact, the JS in JSON comes from JavaScript, because its look was inspired by JavaScript.

This is what an XML document looks like:

```xml
<?xml version="1.0"?>
<country code="CH">
  <name>Switzerland</name>
  <population>8014000</population>
  <currency code="CHF">Swiss Franc</currency>
  <cities>
    <city>Zurich</city>
    <city>Geneva></city>
    <city>Bern <!-- the Federal City --></city>
  </cities>
  <description>
    We produce <b>very</b> good chocolate.
  </description>
</country>
```

This is what a JSON document looks like:

```json
{
  "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."
}
```

In the rest of the chapter, we will explore both XML and JSON in details. It is commonly believed that XML is losing in popularity and JSON is "the new cool stuff", however this is not fully accurate; while on the research side, the publications on XML have become less widespread, in companies, XML is very popular due to its very mature ecosystem supported by several other W3C standards. A few examples are that the mandatory financial reports of US public companies must

be filed in XML, and in Switzerland, electronic tax statements are also stored in XML. What is important is to understand that none of them is better than the other; this is highly use-case dependent and in some cases XML will be a better fit (this is typically the case in the publishing industry), while in other cases JSON will be a better fit.

Whichever syntax is used, they have in common the concept of well-formedness. For any input text document, one can ask whether it is well-formed XML, or whether it is well-formed JSON. This is a yes-no question: given a specific chosen syntax, either a document is well-formed, or it is not well-formed. In theoretical computer science, XML and JSON would be called languages. Formally, languages are sets of strings (which is the fancy word for "text"): all the strings that belong to this language. A string is said to be well-formed if it belongs to the language. Concretely, when a document is well-formed XML, it means that it can be successfully opened by an editor as XML with no errors, and plenty of bells and whistles kick in such as fancy colours to facilitate the reading by humans, automatic indentation, etc. A well-formed XML document can then be further processed (we will see later how). For well-formed JSON, this works exactly in the same way.

On the other, a non-well-formed document cannot be used and cannot benefit from all the XML and JSON tools until it is fixed and edited into a well-formed document. This is all or nothing.

Since JSON and XML are standards, there is a very large number of tools and libraries that support them out of the box, both for reading and for writing, many being free and open source. Thus, producing well-formed XML and JSON documents is very easy with these tools. It is probably the existence of all these tools that pushes people to use these syntaxes rather than inventing a new one, and having to redesign all the tooling.

5.3 JSON

Let us now dive into the details of the JSON syntax. JSON stands for JavaScript Object Notation because the way it looks like originates from JavaScript syntax, however it is now living its own life completely independently of JavaScript[2].

JSON is made of exactly six building blocks: strings, numbers, Booleans, null, objects, and arrays. Let us go through them.

5.3.1 Strings

Strings are simply text. In JSON, strings always appear in double quotes. This is a well-formed JSON string:

[2] which is not a query language!

```
"This is a string"
```

Obviously, strings could contain quotes and in order not to confuse them with the surrounding quotes, they need to be differentiated. This is called escaping and, in JSON, escaping is done with backslash characters (\). This should be quite familiar to many people with programming experience:

```
"The word \"quoted\" is quoted."
```

There are several other escape sequences in JSON, the most popular ones being:

\\	\
\n	new line
\r	carriage return
\t	tabulation
\u followed by four hexadecimal digits	any character

The last one, in fact, allows the insertion of *any* character via its Unicode code point. Unicode is a standard that assigns a numeric code (called a code point) to each character in order to catalogue them across all languages of the world, even including emojis. The catalogue evolves with regular meetings of the working group. If you know the Unicode code point of a special character, then it is straightforward to escape it, e.g., for the letter Π:

```
"\u03C0"
```

The code point must be indicated in base 16 (digits 0 to 9, plus letters from A to F). Code points can easily be looked up with a search engine by typing a description of what you are looking for, even though more complex strings will typically be created automatically.

5.3.2 Numbers

JSON generally supports numbers, without explicitly naming any types nor making any distinction between numbers apart from how they appear in syntax. The way a number appears in syntax is called a *lexical representation*, or a *literal*. These two words, in fact, also generally apply to many other types as we will see in subsequent chapters.

Generally, a number is made of digits, possibly including a decimal period (which must be a dot) and optionally followed by the letter e (in either case) and a power of ten (scientific notation). Both the number and the optional power of ten can also have an optional sign.

These are a few examples of well-formed JSON number literals:

```
0
1234
12.34
-132.54
12.3E45
12.3e-45
-12.3e-45
```

JSON places a few restrictions: a leading + is not allowed. Also, a leading 0 is not allowed except if the integer part is exactly 0 (in which case it is even mandatory, i.e., .23 is not a well-formed JSON number literal):

```
0.23
```

JSON numbers are unquoted. Otherwise, they would be recognized as strings by the parser and not as numbers.

A warning: the same (mathematical) number might have several literals to represent it.

```
2
20e-1
2.0
```

It is important, as we will see later, to have in mind that the literal, which is the syntactic representation, is not the same as the actual, logical number. The above three literals have in common their "two-ness".

5.3.3 Booleans

There are two Booleans, true and false, and each one is associated with exactly one possible literal, which are, well, true and false.

```
true
false
```

In spite of the fact that there is only exactly one literal for each Boolean, it is also important to distinguish the literal *true*, which is the sequence of letters t, r, u and e appearing in JSON syntax, from the actual concept of "true-ness," which is an abstract mathematical concept. While on this chapter we focus on syntax and literals, when later we actually query data, we will work on the abstract concepts, not on the underlying syntax.

Boolean literals are unquoted. Otherwise, they would be recognized as strings by the parser and not as Booleans.

5.3.4 Null

There is a special value, null, which corresponds to the (unique) literal.

```
null
```

The concept of "null-ness" can be subject to debate: some like to see this as an unknown or hidden value, others as equivalent to an absent value, etc. In this course, on the logical level, we will consider that an absent value is not the same thing as a null value.

Null literals are unquoted. Otherwise, they would be recognized as strings by the parser and not as nulls.

5.3.5 Arrays

Arrays are simply lists of values. The concept of list is abstract and mathematical, i.e., lists are considered an abstract data type and correspond to finite mathematical sequences.

The concept of array is the syntactic counterpart of a list, i.e., an array is a physical representation of an abstract list.

The members of an array can be any JSON value: string, number, Boolean, null, array, or (we will get to it quite shortly) object. They are listed within square brackets, and are separated by commas.

```
[ 1, 2, 3 ]
[ ]
[ null, "foo", 12.3, false, [ 1, 3 ] ]
```

It can also be convenient to let arrays "breathe" with extra spaces, which are irrelevant when parsing JSON (except if they are *inside* a string literal!). In fact, there are plenty of libraries out there that can nicely do this, which is known as "pretty-printing:"

```
[
  1,
  2,
  3
]
[ ]
[
  null,
  "foo",
  12.3,
  false,
  [
    1,
```

```
    3
  ]
]
```

5.3.6 Objects

Objects are simply maps from strings values. The concept of map is abstract and mathematical, i.e., maps are considered an abstract data type and correspond to mathematical partial functions with a string domain and the range of all values.

The concept of object is the syntactic counterpart of a map, i.e., an object is a physical representation of an abstract map that explicitly lists all string-value pairs (this is called an *extensional* definition of a function, as opposed to the way functions are typically defined in mathematics (e.g., $f(x) = x + 1$, which is, by contrast, intensional)).

The keys of an object must be strings. This excludes any other kind of value: it cannot be an integer, it cannot be an object. This also implies that keys must be quoted. While some JSON parsers are lenient and will accept unquoted keys, it is very important to never create any JSON documents with unquoted keys for full compatibility with all parsers.

The values associated with them can be any JSON value: string, number, Boolean, null, array, or (we will get to it quite shortly) object. The pairs are listed within curly brackets, and are separated by commas. Within a pair, the value is separated from the key with a colon character.

```
{ "foo" : 1 }
{ }
{ "foo" : "foo", "bar" : [ 1, 2 ],
  "foobar" : [ { "foo" : null },  { "foo" : true } ]
}
```

It can also be convenient to let objects "breathe" with extra spaces, which are irrelevant when parsing JSON (except if they are *inside* a string literal!). In fact, there are plenty of libraries out there that can nicely do this, which is known as "pretty-printing:"

```
{
  "foo" : "foo",
  "bar" : [
    1,
    2
  ],
  "foobar" : [
    {
```

```
      "foo" : null,
      "bar" : 2
    },
    {
      "foo" : true,
      "bar" : 3
    }
  ]
}
```

The JSON standard recommends for keys to be unique within an object. Many parsers and products will reject duplicate keys, because they rely on the semantics of a map abstract data type. If one downloads a dataset that has duplicate keys and the engine one intends to use to process it does not allow them, then this will require extra work. In particular, one needs to find a JSON library that accepts duplicate keys, and use it to fix the dataset by disambiguating the keys to make it parseable with any engine. It is very important to never create any JSON documents with duplicate keys for full compatibility with all parsers, to avoid creating this extra workload for the consumers.

5.4 XML

XML stands for eXtensible Markup Language. It resembles HTML, except that it allows for any tags and that it is stricter in what it allows.

XML considerably much complex than JSON but, fortunately, most datasets only use a subset of what XML can do. In our course, we will stick to the most common features of XML.

XML's most important building blocks are elements, attributes, text and comments.

5.4.1 Elements

XML is a markup language, which means that content is "tagged". Tagging is done with XML elements.

An XML element consists of an opening tag, and a closing tag. What is "tagged" is everything inbetween the opening tag and the closing tag.

This is an example with an opening tag, some content (which can be recursively anything, as we will see), and then a closing tag. Tags consist of a name surrounded with angle brackets ¡¿, and the closing tag has an additional slash in front of the name.

```
<person>(any content here)</person>
```

If there is no content at all, the lazy of us will appreciate a convenient shortcut to denote the empty element with a single tag. Mind that the slash is at the end:

```
<person/>
```

is equivalent to:

```
<person></person>
```

Elements nest arbitrarily:

```
<person><first>(some content)</first><student/>
<last>(some other content)</last></person>
```

Like JSON, it is possible to use indentation and new lines to pretty-print the document for ease of read by a human:

```
<person>
  <first>(some content)</first>
  <student/>
  <last>(some other content)</last>
</person>
```

Unlike JSON keys, element names can repeat at will. In fact, it is even a common pattern to repeat an element many times under another element in plural form, like so:

```
<persons>
  <person>
    <first>(some content)</first>
    <last>(some other content)</last>
  </person>
  <person>
    <first>(some content)</first>
    <last>(some other content)</last>
  </person>
  <person>
    <first>(some content)</first>
    <last>(some other content)</last>
  </person>
</persons>
```

Some care needs to be put in "well-parenthesizing" tags, for example, this is incorrect and not well-formed XML:

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

because the inner elements must close before the outer elements.

Elements cannot appear within opening or closing tags, they must appear between tags. This is not well-formed XML:

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

At the top-level, a well-formed XML document must have exactly one element. Not zero, not two, exactly one. This is not well-formed XML[3]

```
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
```

5.4.2 Attributes

Attributes appear in any opening elements tag and are basically key-value pairs. In the following examples, we added two attributes with the keys "birth" and "death."

```
<person birth="1879" death="1955">
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

Values can be either double-quoted or single-quoted. This is also well-formed XML:

```
<person birth='1879' death='1955'>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

As well as this:

[3]:

```
<person birth="1879" death='1955'>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

The key is never quoted, and it is not allowed to have unquoted values[4]. This is not well-formed XML:

```
<person birth=1879 "death"=1955>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

Within the same opening tag, there cannot be duplicate keys. This is not well-formed XML:

```
<person birth="1879" birth="1955">
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

Attributes can also appear in an empty element tag:

```
<person birth="1879" birth="1955"/>
```

Attributes can never appear in a closing tag. This is not well-formed XML:

```
<person>
    <first>(some content)</first>
    <last>(some other content)</last>
</person birth="1879" birth="1955">
```

Elements cannot nest without attribute values. This is not well-formed XML:

```
<person birth="<date>1879</date>" birth="1955">
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

It is not allowed to create attributes that start with XML or xml, or any case combination (XmL, etc). This is because this is reserved for another use (namespaces, as we will see shortly).

[4]This is unlike HTML, in which values can be without quotes

5.4.3 Text

Text, in XML syntax, is simply freely appearing in elements and without any quotes (attributes values are not text!). For example, we can have test inside the first and last elements like so:

```
<person birth="1879" birth="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

Text cannot appear on its own at the top level. This is not well-formed XML:

```
Albert <person/> Einstein
```

Within an element, text can freely alternate with other elements. This is called mixed content and is unique to XML, like so:

```
<person>
    <style>His Royal Highness</style>
    The <title>Duke of <location>Cambridge</location></title>
</person>
```

This feature of XML makes it very popular in the publishing industry, where it is very convenient to have books, papers, etc, with the text tagged with extra information.

5.4.4 Comments

Comments in XML look like so:

```
<!-- This is a comment -->
```

but, as we saw, a single comment alone is not well-formed XML (remember: we need exactly one top-level element). This would be well-formed XML with a comment:

```
<person birth="1879" birth="1955">
    <first>Albert</first>
    <last>Einstein</last>
    <!-- He is still famous today -->
</person>
```

Comments can also appear at the top-level though, but under the condition that there is exactly one top-level element.

```
<!-- He is still famous today -->
<person birth="1879" birth="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
<!-- He is -->
<!-- He totally is -->
```

The reason why comments specifically look like this is historical: XML was derived as a simplified subset of an older markup language called SGML. Many of the strange-looking symbols of XML are in fact coming from SGML.

5.4.5 Text declaration

XML documents can be identified as such with an optional text declaration containing a version number and an encoding.

```
<?xml version="1.0" encoding="UTF-8"?>
<person birth="1879" birth="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

The version is either 1.0 or 1.1, but there is no need to understand the difference for this course. It is rather subtle and mostly due to more permissive behaviour with international characters in 1.1. Most XML documents out there are version 1.0.

The encoding is a physical detail and gives information on how the document is stored as bits on the disk. This is also advanced and out of the scope of this course. If in doubt, UTF-8 is the recommended standard as of 2022.

Another tag that might appear right below, or instead of, the text declaration is the doctype declaration. It must then repeat the name of the top-level element, like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE person>
<person birth="1879" birth="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

or like so:

```
<!DOCTYPE person>
<person birth="1879" birth="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

This might be familiar to HTML aficionados:

```
<!DOCTYPE html>
<html>
  ...
</html>
```

Doctype declarations exist also for historical reasons and are part of the DTD validation mechanism, which is out of scope for this course because it was superseded with more modern mechanisms such as XML Schema, which we will look at in Chapter 7.

All in all, the only reason why we showed what text declarations look like and what doctype declarations look like is so you are not surprised when you see them.

5.4.6 Escaping special characters

As you might have guessed, if characters such as < are used in the text, or characters such as ” or ’ are used in attribute values, it will cause problems. This is not well-formed XML:

```
<equation name="the "basic" comparison">
    1 < 2
</equation>
```

Remember that in JSON, it is possible to escape sequences with a backslash character. In XML, this is done with an ampersand (&) character.

There are exactly five possible escape sequences pre-defined in XML:

Escape sequence	Corresponding character
<	<
>	>
"	”
'	’
&	&

For example, the above document can be turned into a well-formed XML document like so:

```
<equation name="the &quot;basic&quot; comparison">
    1 &lt; 2
</equation>
```

Escape sequences can be used anywhere in text, and in attribute values. At other places (element names, attribute names, inside comments), they will not be recognized or will lead to well-formedness errors.

But there are a few places where they are mandatory:

- In text, & and < MUST be escaped. The other characters may, but need not, be escaped.
- In double-quoted attribute values, ", & and < MUST be escaped. The other characters may, but need not, be escaped.
- In single-quoted attribute values, ', & and < MUST be escaped. The other characters may, but need not, be escaped.

5.4.7 Namespaces in XML

When a lot of data is created in the XML format, scaling issues start appearing because people use the same element and attribute names for different purposes. For example, an element named "client" can be used in customer relationship datasets, or in computer network datasets.

Namespaces are an extension of XML that allows users to group their elements and attributes in packages, similar to Python modules, Java packages or C++ namespaces. This is a very natural thing to do.

Namespace URIs

A namespace is identified with a URI. We already studied URIs in the context of REST in Chapter 3. A point of confusion is that XML namespaces often start with *http://*, but are not meant to be entered as an address into a browser! It might sometimes work, but this will only be because the owner of the namespace was kind enough to put a page (often with documentation on the namespace) on the Web at the same URI.

Here are just a few examples of namespaces:

Namespace	use
http://www.w3.org/1999/xhtml	HTML
http://www.w3.org/1998/Math/MathML	MathML (formulas)
http://www.music-encoding.org/ns/mei	Music sheets

If you think about it, this is not so different from Java: Java just uses a different convention with reversed domains and dots instead of slashes. For example the Music package would probably be called org.music-encoding.ns.mei in Java.

An entire XML document in a namespace

Let us start with something easy. It is possible to put all elements of an XML document in a namespace, here http://www.example.com/persons, like so:

```
<person xmlns="http://www.example.com/persons">
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
```

In the above document, the elements person, first and last all live in the namespace http://www.example.com/persons. This is because of what looks like an xmlns attribute, associated with the namespace http://www.example.com/persons as the value. But in fact, xmlns is not an attribute, it is really a namespace declaration. If you remember, we saw that attributes starting with xml are forbidden, and this is because this is reserved for namespace declarations.

The document below is different, because it does not have this declaration. So, the elements person, first and last do not live in any namespace, we say that the namespace is absent for these elements:

```
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
<person>
  <first>(some content)</first>
  <last>(some other content)</last>
</person>
```

Here is another example, this time a MathML document, in the corresponding namespace:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <apply>
    <eq/>
    <ci>x</ci>
    <apply>
    <root/>
      <cn>2</cn>
    </apply>
  </apply>
</math>
```

QNames

What about documents that use multiple namespaces? This is done by associating namespaces with prefixes, which act as shorthands for a namespace. Then, we can use the prefix shorthand in every element that we want to have in this namespace.

This is the same MathML document as previously seen, except that now we explicitly associate the MathML namespace with prefix m. This is done by using xmlns:m instead of just xmlns, and by adding m: in front of every element that we want to have in this namespace, like so:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
  <m:apply>
    <m:eq/>
    <m:ci>x</m:ci>
    <m:apply>
    <m:root/>
      <m:cn>2</m:cn>
    </m:apply>
  </m:apply>
</m:math>
```

What is important to understand is that, semantically, this is the *same* document as the one we saw without the prefix: all elements are, in both cases, in the MathML namespace. The part of the element name that appears on the right of the colon sign (or the entire element name, if it does not have a prefix) is called the local name of the element.

Every element has a local name. An element may have a prefix (on the left of the colon sign, say *foo*), in which case the corresponding namespace is looked up with the appropriate *xmlns:foo* declaration with the same prefix. An element may also not have a prefix, in which case the prefix is said to be absent. In this case the corresponding namespace (called the default namespace) is looked up with the appropriate *xmlns* declaration. If there is no such declaration, then the namespace is said to be absent, too.

So, given any element, it is possible to find its local name, its (possibly absent) prefix, and its (possibly absent) namespace. The triplet (namespace, prefix, localname) is called a QName (for "qualified name").

For the purpose of the comparisons of two QNames (and thus of documents), the prefix is ignored: only the local name and the namespace are compared. The following document, which uses yet another prefix, is again the same document as the previous two MathML documents:

```
<foo:math xmlns:foo="http://www.w3.org/1998/Math/MathML">
  <foo:apply>
    <foo:eq/>
    <foo:ci>x</foo:ci>
    <foo:apply>
    <foo:root/>
      <foo:cn>2</foo:cn>
    </foo:apply>
  </foo:apply>
</foo:math>
```

This document, however, is different, because its elements are in no namespace at all:

```
<math>
  <apply>
    <eq/>
    <ci>x</ci>
    <apply>
    <root/>
      <cn>2</cn>
    </apply>
  </apply>
</math>
```

With the QName machinery, it is possible to have as many namespaces and prefixes as one wants, in this example four of them:

```
<?xml version "1.0"?>
<a:bar
  xmlns:a="http://example.com/a"
  xmlns:b="http://example.com/b"
  xmlns:c="http://example.com/c"
  xmlns:d="http://example.com/d">
    <b:foo/>
    <c:bar>
      <d:foo/>
```

```
      <a:foobar/>
    </c:bar>
</a:bar>
```

Namespaces are quite flexible: xmlns declarations can be put anywhere and it can quickly become messy and out of control. Thus, we highly recommend to stick to several rules that are common practice if you want to keep your sanity:

- only put xmlns and xmlns:prefix declarations in the top-level element. Nowhere else.
- make sure the prefix-namespace mapping is bijective. Do not use twice the same prefix with the same namespace. Do not bind two namespaces with the same prefix (which would in fact be an error).
- do not mix the default namespace (xmlns declaration) with namespaces associated with prefixes in the same document (xmlns:prefix declarations). It is either or: either you have a single namespace for all elements in the entire document that is the default namespace, or you have one or several namespaces that are all associated with a (non-absent) prefix. Mixing the two types of declarations quickly leads to confusion for people looking at the document.

As a counterexample showing bad practice, this is what should be avoided:

```
<foo:bar xmlns:foo="http://example.com/foo">
  <foo:foo/>
  <bar:foobar xmlns:bar="http://example.com/bar">
    <bar:foo/>
    <foo:foo/>
    <foo/>
    <foo/>
  </bar:foobar>
  <foo xmlns="http://example.com/foo"/>
  <foo:bar/>
  <foo:bar xmlns:foo="http://example.com/bar"/>
  <foo:foo/>
</foo:bar>
```

Attributes and namespaces

Attributes can also live in namespaces, that is, attribute names are generally QNames. However, there are two very important aspects to consider.

First, unprefixed attributes are not sensitive to default namespaces: unlike elements, the namespace of an unprefixed attribute is always absent even if there is a default namespace. The attribute attr in this example is in no namespace, although all (unprefixed) elements live in the http://example.com/foo namespace:

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

Second, it is possible for two attributes to collide if they have the same local name, and different prefixes but associated with the same namespace (but again, we told you: do not do that!). The following document is thus not well-formed.

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

5.4.8 Datasets in XML

Let us now look deeper at how to express tabular data in XML, in order to demonstrate that XML subsumes tabular data.

The tuple marked in red here:

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

can be stored in XML as:

```
<sale>
  <product>Phone</product>
  <price>800</price>
  <customer>John</customer>
  <quantity>1</quantity>
</sale>
```

The tuples marked in red here:

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

can be stored in XML using the plural-singular convention as:

```
<sales>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <customer>John</customer>
    <quantity>1</quantity>
  </sale>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <customer>Peter</customer>
    <quantity>2</quantity>
  </sale>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <customer>Mary</customer>
    <quantity>1</quantity>
  </sale>
  <sale>
    <product>Laptop</product>
    <price>200</price>
    <customer>John</customer>
    <quantity>3</quantity>
  </sale>
</sales>
```

Finally this nested tuple:

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

can be stored in XML using nested elements as:

```
<?xml version "1.0"?>
<!DOCTYPE sales>
<sales>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <orders>
      <order>
        <customer>John</customer>
        <quantity>1</quantity>
      </order>
      <order>
        <customer>Peter</customer>
        <quantity>2</quantity>
      </order>
      <order>
        <customer>Mary</customer>
        <quantity>1</quantity>
      </order>
    </orders>
  <sale>
</sales>
```

5.5 XML vs. JSON, or how to troll internet forums

Whether XML or JSON is better is the topic of an intense debate, a bit like vi vs. emacs or mac vs. PC. The reason is simple: neither is. It depends on the use case. Objectively, one can nevertheless say that XML is quite suitable and popular for the publishing industry and "text-oriented", tagged data because of its unique mixed content feature. Data itself can be stored indifferently in XML or in JSON, as we saw. The XML ecosystem is also more mature and enterprise-ready because it is older, however JSON has gained so much popularity in the recent decade that it is expected that the JSON ecosystem will catch up.

Chapter 6

Wide column stores

Now that we have looked into some textual data formats (syntax) like CSV, JSON, XML, how do we store these CSV, JSON and XML files? A first obvious solution is: on the local disk, or on an object storage service (S3, Azure Blob Storage), or on a distributed file system (HDFS).

The problem with HDFS is its latency: HDFS works well with very large files (at least hundreds of MBs so that blocks even start becoming useful), but will have performance issues if accessing millions of small XML or JSON files.

Wide column stores were invented to provide more control over performance and in particular, in order to achieve high-throughput *and* low latency for objects ranging from a few bytes to about 10 MB, which are too big and numerous to be efficiently stored as so-called clobs (character large objects) or blobs (binary large objects) in a relational database system, but also too small and numerous to be efficiently accessed in a distributed file system.

6.1 A sweet spot between object storage and relational database systems

The astute reader will argue that a large number of JSON or XML files would be handled quite well with an object storage service. But a wide column store has additional benefits:

- a wide column store will be more tightly integrated (because on the same machines) with the parallel data processing systems, which we will introduce in subsequent chapters;
- wide column stores have a richer logical model than the simple key-value model behind object storage;

- wide column stores also handle very small values (bytes and kBs) well thanks to batch processing.

Note that a wide column store is not a relational database management system:

- it does not have any data model for values, which are just arrays of bytes;
- since it efficiently handles values up to 10 MB, the values can be nested data in various formats, which breaks the first normal form;
- tables do not have a schema;
- there is no language like SQL, instead the API is on a lower level and more akin to that of a key-value store;
- tables can be very sparse, allowing for billions of rows and millions of columns at the same time; this is another reason why data stored in HBase is denormalized.

6.2 History

The good old relational database management systems (RDBMS) from the 1970s were built as monolithic engines that run on a single machine. They can only hold as much data as fits on one machine, and are only as fast as the CPU on that machine.

How can we handle more data or process it faster with such a system? This was a problem first encountered by search engines, which need to cache and index the Web (HTML pages, etc). As we previously saw, we could scale up by buying a bigger machine, with a more powerful CPU, with larger or more drives, and with more RAM. But if you remember, we also saw that scaling up has its limits because the price will not grow linearly, and what is feasible in a give year will also hit a hard limit.

Another approach is to scale out to several machines. In fact, this was attempted by several companies (some of the tech giants) in the early 2000s as the quantity of data they had to manage was challenging their infrastructure. The early systems attempting to scale out an RDBMS were rudimentary: several machines would be purchased, set up and connected with each other via the network and the same RDBMS software would be installed on each machine.

Then, the data would be spread over these different systems and possibly stored redundantly. In fact, the underlying idea is similar to how we partitioned file into blocks in HDFS, how the blocks are

physically stored as several replica, and how the replicas are spread all over the cluster of machines.

However, the logic for partitioning the data across machines and for handling the replication had to be done with additional software, written in programming languages such as Java or C++, connecting as clients to the RDBMS instances. This was not only very difficult to set up, it was also very costly in terms of maintenance.

Furthermore, additional logic to query the data on the entire cluster was also needed: a program, outside the RDBMS, needed to figure out where to read the data from or where to write it to, how to reassemble it for the querying user, etc.

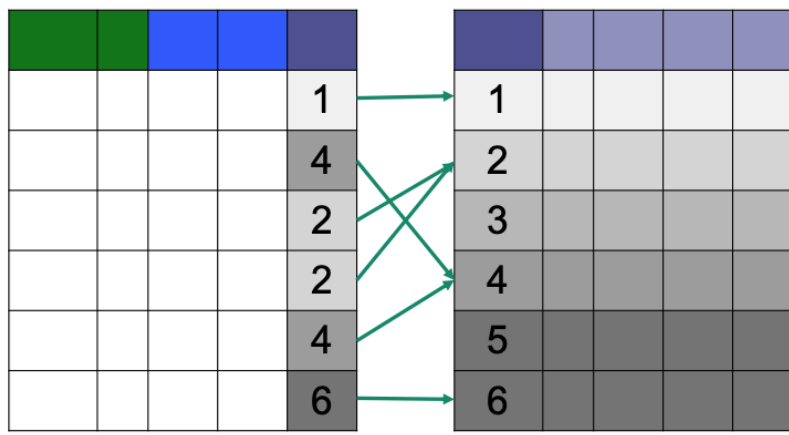
In fact, the engineers realized that they were really building an entirely new database management system, with an entirely new architecture designed for a cluster rather than for a single machine and optimized for clobs and blobs. The company who came up with the first fully integrated product was Google, and this product was called BigTable. An open-source equivalent then followed as part of the Hadoop ecosystem, called HBase.

6.3 Logical data model

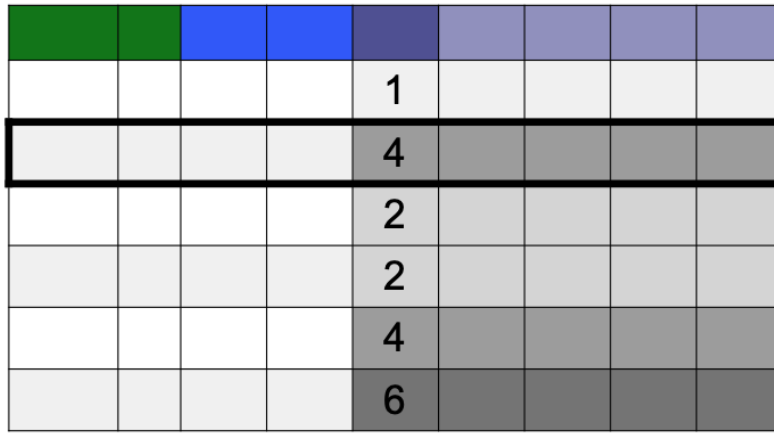
6.3.1 Rationale

The data model of HBase is based on the realization that joins are expensive, and that they should be avoided or minimized on a cluster architecture. Joins can be avoided if they are pre-computed, that is, instead of storing the data as separate tables, we store, and work on, the joined table.

Visually, what would look like this in a traditional RDBMS:



is instead stored like so in HBase:



There is a direct consequence of this change: the number of columns in a traditional RDBMS is limited, typically to somewhere around 256, or maybe in the low four digits if the columns have simple and compact types. But in a denormalized table, the number of columns can easily skyrocket, with many cells being in fact empty. The table is thus very wide and sparse, which is a completely different use case than what RDBMS software was designed for.

The second design principle underlying HBase is that it is efficient to store together what is accessed together. In the big picture, this is a flavor of batch processing, one of the overarching principles in Big Data. Batch processing reduces the impact of latency – remember that we saw in Chapter 1 that latency barely improved in the past few decades in comparison to capacity and throughput – by involving fewer, larger requests instead of more, smaller requests.

These two design principles are tied to a specific usage pattern of the database management system: when the emphasis is reading and processing data in large amounts. This is, at first sight, in direct conflict with efficiently writing data. First, writing denormalized data is more cumbersome: one needs to deal with insertion, update and deletion anomalies. Second, writing data under the constraint of storing together what is accessed together is a challenging endeavor, because one cannot just "insert" new data at a specific physical location without moving the existing data around.

As we will see in this chapter, HBase provides an impressive solution for handling writes quite efficiently, while preserving a batch-processing paradigm. Moreover, the underlying idea also impacted the handling

of intermediate data by MapReduce and Apache Spark, which we will cover in later chapters.

6.3.2 Tables and row IDs

From an abstract perspective, HBase can be seen as an enhanced key-value store, in the sense that:

- a key is compound and involves a row, a column and a version;
- keys are sortable;
- values can be larger (clobs, blobs), up to around 10 MB.

This is unlike traditional key-value stores, which have a flat key-value model, which (in the case of distributed hash tables) do not sort keys, which thus do not store "close" keys together, and which usually support smaller values.

On the logical level, the data is organized in a tabular fashion: as a collection of rows. Each row is identified with a row ID. Row IDs can be compared, and the rows are logically sorted by row ID:

Row ID	A	B	1	2	I
000					
002					
0A1					
1E0					
22A					
4A2					

A row ID is logically an array of bytes, although there is a library to easily create row ID bytes from specific primitive values (byte, short, int, long, string, etc).

6.3.3 Column families

The other attributes, called columns, are split into so-called column families. This is a concept that does not exist in relational databases and that allows scaling the number of columns. Very intuitively, one can think of column families as the tables that one would have if the data were actually normalized and the joins had not been pre-computed.

On the picture above, we separated the columns into column families, using a different color and alphabet for each family. The name of a column family is a string, just like the name of a table. Often, we also use the terminology "column family" to refer to the name of the column family, i.e., we identify the column family with its name.

6.3.4 Column qualifiers

Columns in HBase have a name (in addition to the column family) called column qualifier, however unlike traditional RDBMS, they do not have a particular type. In fact, as far as HBase is concerned, all values are binary (arrays of bytes) and what the user does with it (string, integer, large objects, XML, JSON, HTML etc) is really up to them. There are many different frameworks that can be used in complement of HBase to add a type system (Avro, etc) and it is, in fact, very common to store large blobs of data in the cells. We will cover the paradigm for this in Chapter 7.

In fact, it goes further than that. Not only are there no column types: even the column qualifiers are not specified as part of the schema of an HBase table: columns are created on the fly when data is inserted, and the rows need not have data in the same columns, which natively allows for sparsity. Column qualifiers are arrays of bytes (rather than strings), and as for row IDs, there is a library to easily create column qualifiers from primitive values.

Thus, on the logical level, columns come and go as the table lives its life:

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

Unlike the values which can be large arrays of bytes (blobs), it is important to keep column families and column qualifiers short, because as we will see, they are repeated a gigantic number of times on the physical layer.

6.3.5 Versioning

HBase generally supports versioning, in the sense that it keeps track of the past versions of the data. As we will see, this is implemented by associating any value with a timestamp, also called version, at which it was created (or deleted). Users can also override timestamps with a value of their choice to have more control about versions.

6.3.6 A multidimensional key-value store

As we previous explained, one way to look at an HBase table is that it is an enhanced key-value store where the key is four-dimensional. Indeed, in HBase, the key identifying every cell consists of:

- the row ID
- the column family
- the column qualifier
- the version

HBase is able to efficiently look up any cell given its key.

6.4 Logical queries

Having realized that an HBase table is nothing but a four-dimensional key-value store, it follows logically that the HBase API also resembles that of a key-value store: HBase supports four kinds of low-level queries: get, put, scan and delete. Unlike a traditional key-value store, HBase also supports querying ranges of row IDs and ranges of timestamps.

In comparison to a full-fledged RDBMS, this is quite limited and, as for data types, support for higher-level queries (such as SQL) is brought by additional frameworks that come as a complement of, and atop HBase (Apache Hive, Apache Phoenix, etc).

6.4.1 Get

With a get command, it is possible to retrieve a row specifying a table and a row ID.

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

Optionally, it is also possible to only request some but not all of the columns, or a specific version or within a time range (interval of versions).

6.4.2 Put

With a put command, it is possible to put a new value in a cell by specifying a table, column family and column qualifier.

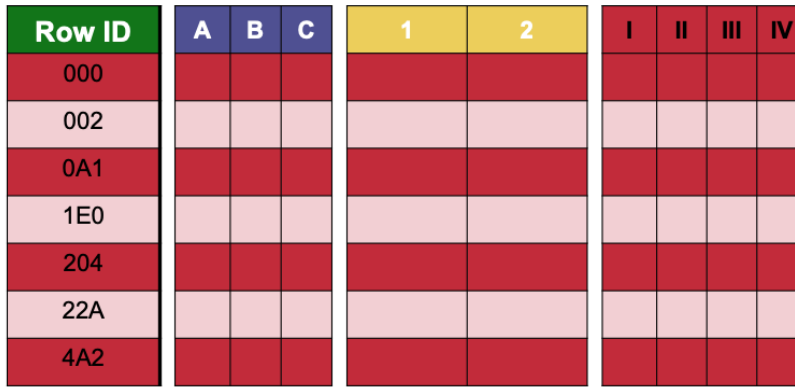
Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
204									
22A									
4A2									

It is also possible to optionally specify the version (as opposed to the current time).

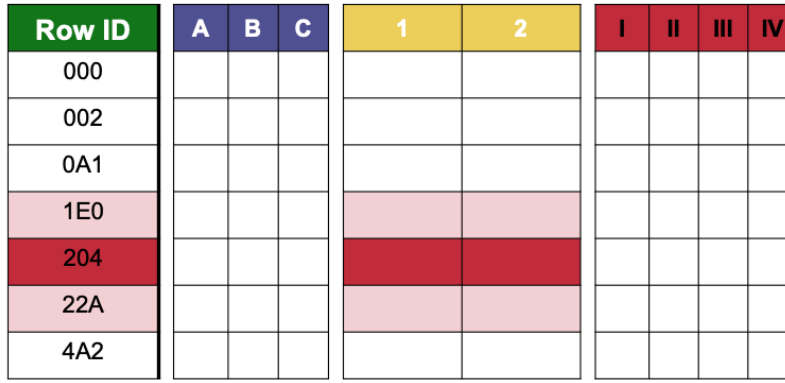
HBase offers a locking mechanism at the row level, meaning that different rows can be modified concurrently, but the cells in the same row cannot: only one user at a time can modify any given row.

6.4.3 Scan

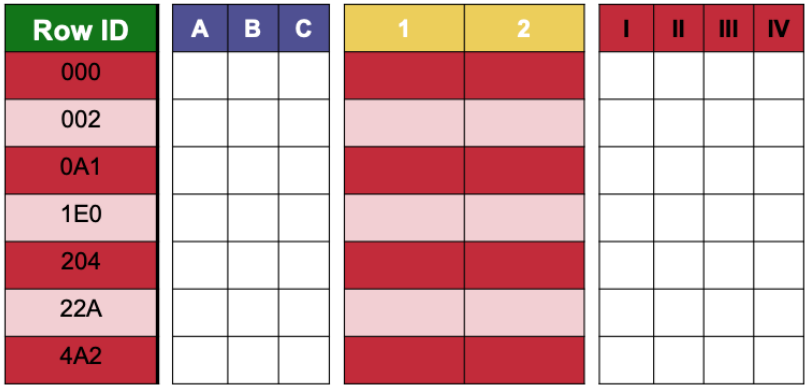
With a scan command, it is possible to query a whole table or part of a table, as opposed to a single row.



It is possible to restrict the scan to specific columns families or even columns.



It is possible to restrict the scan to an interval of rows.

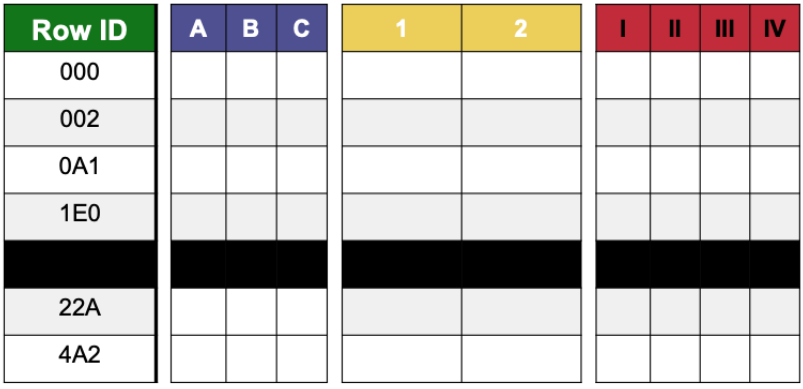


It is possible to run the scan at a specific version, or on a time range.

Scans are fundamental for obtaining high throughput in parallel processing.

6.4.4 Delete

With a delete command, it is possible to delete a specific value with a table, row ID, column family and qualifier. Optionally, it is also possible to delete the value with a specific version, or all values with a version less or equal to a specific version.

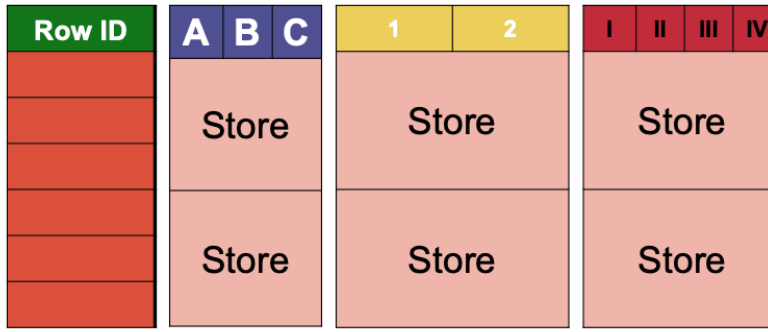


6.5 Physical architecture

6.5.1 Partitioning

A table in HBase is physically partitioned in two ways: on the rows and on the columns.

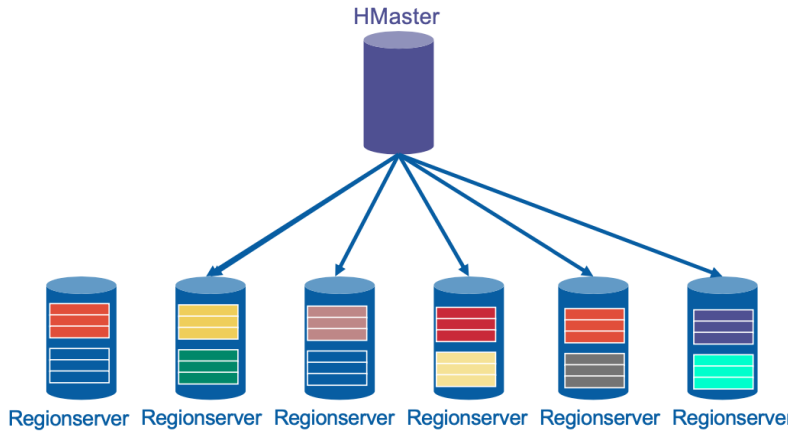
The rows are split in consecutive regions. Each region is identified by a lower and an upper row key, the lower row key being included and the upper row key excluded.



A partition is called a store and corresponds to the intersection of a region and of a column family.

6.5.2 Network topology

HBase has exactly the same centralized architecture as HDFS

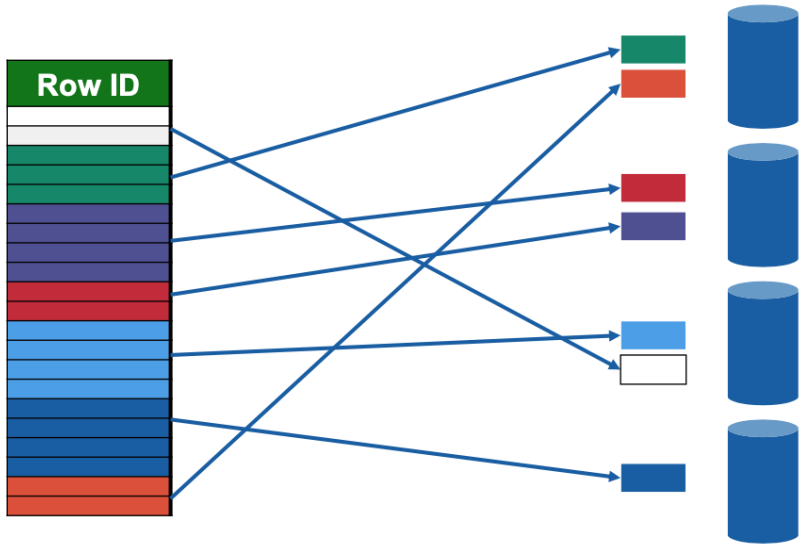


The HMaster and the RegionServers should be understood as processes running on the nodes, rather than the nodes themselves, even though it is common to use "HMaster" to designate the node on which the HMaster process runs, and "RegionServer" to designate a node on which a RegionServer process runs.

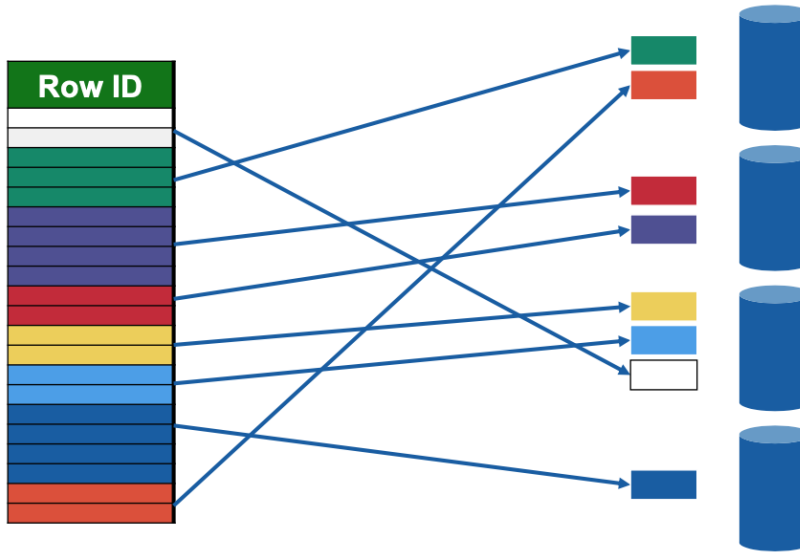
It is common for the HMaster process to run on the same node as the NameNode process. Likewise, it is common for the RegionServer processes to run on those same nodes on which the DataNode processes run.

The HMaster assigns responsibility of each region to one of the RegionServers. This does mean that, for a given region (remember: interval of row IDs), all column families – each one within this region being a store – are handled by the same RegionServer.

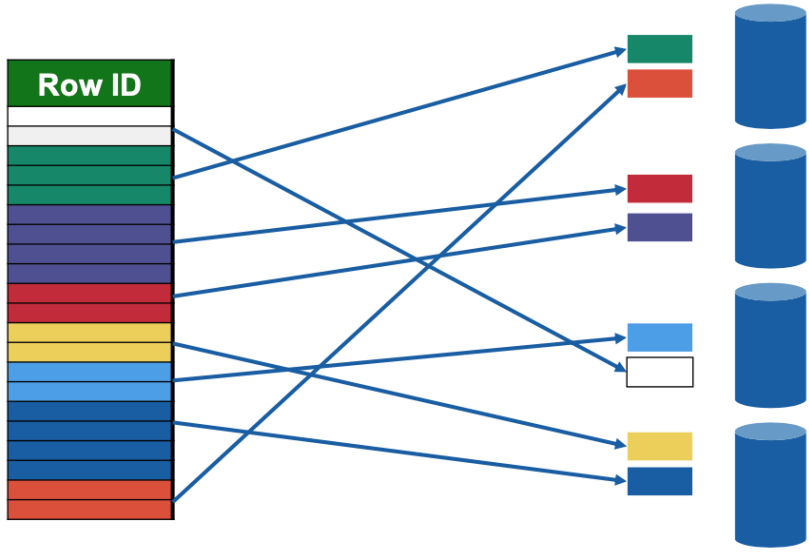
There is no need to attribute the responsibility of a region to more than one RegionServer at a time because, as we will see soon, fault tolerance is already handled on the storage level by HDFS.



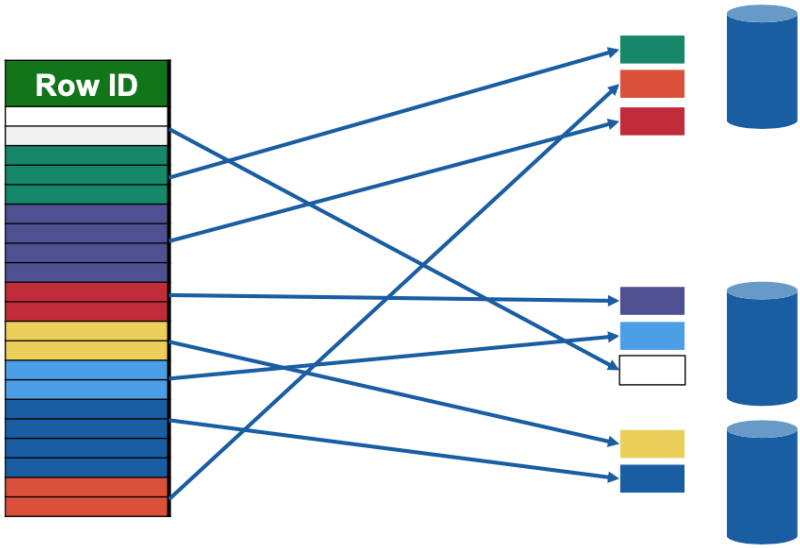
If a region grows too big, for example because of many writes in the same row ID interval, then the region will be automatically split by the responsible RegionServer. Note, however, that concentrated writes ("hot spots") might be due to a poor choice of row IDs for the use case at hand. There are solution to this such as salting or using hashes in row ID prefixes.



If a RegionServer has too many regions compared to other RegionServers, then the HMaster can reassign regions to other RegionServers.

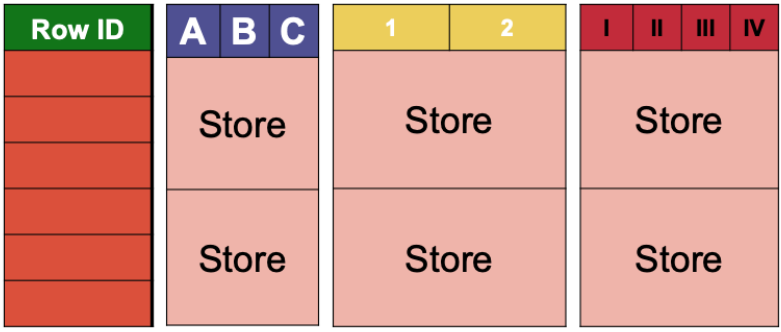


Likewise, if a RegionServer fails, then the HMaster can reassign all its regions to other RegionServers.

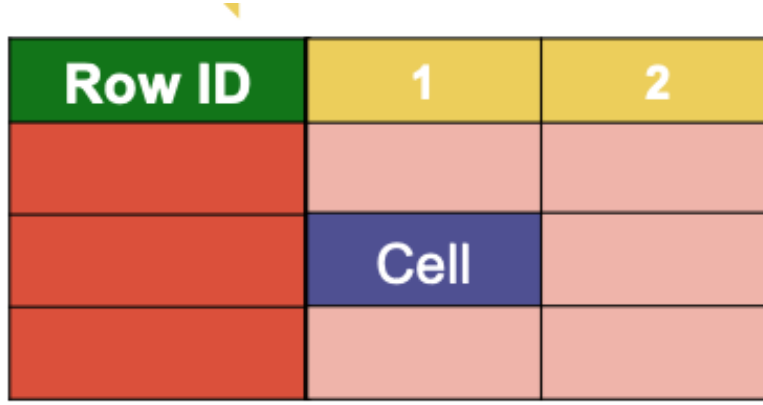


6.5.3 Physical storage

Let us now dive into the actual physical storage. As we saw, the data is partitioned in stores, so we need to look at how each store is physically stored and persisted.



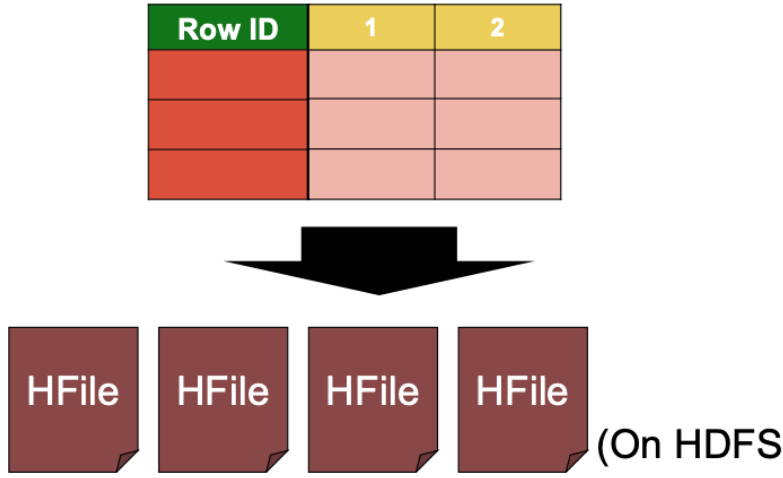
The store is, physically, nothing less than a set of cells:



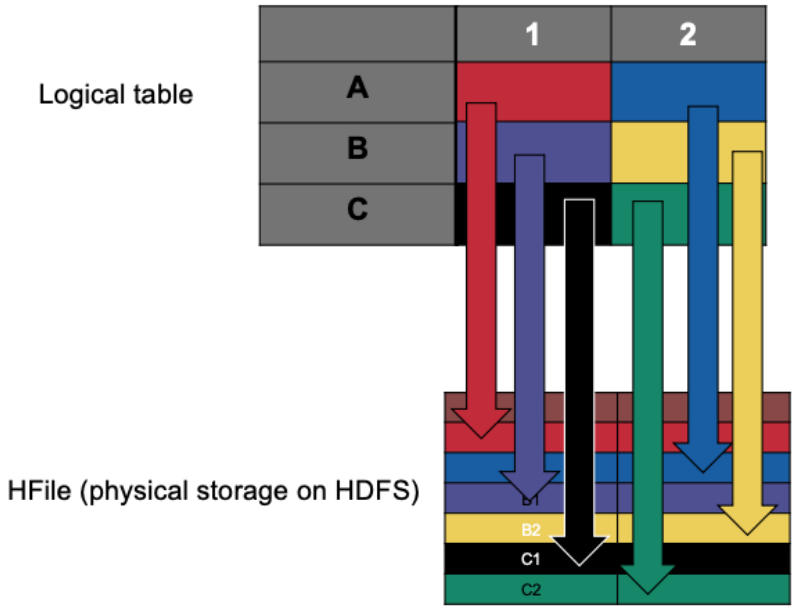
Each cell is identified by a row ID (within the region handled by the store), a column family (the one handled by the store), a column qualifier (arbitrary) and a version (arbitrary). The version is often implicit, such as in the picture above where several version of the "same" cell can co-exist, but it is an important component in the identification of a cell. This tuple of will be referred to as the key of the cell.

Each cell is thus handled physically as a key-value pair where the key is a (row ID, column family, column qualifier, version) tuple and the value is its content. On the physical level, a cell is often referred to in CamelCase as a KeyValue to disambiguate from other contexts in which key-values might appear within HBase.

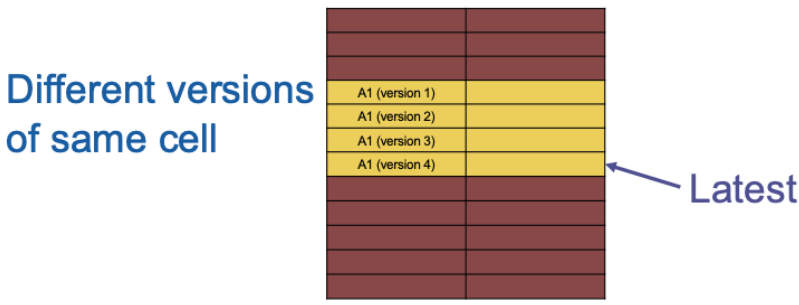
All the cells within a store are eventually persisted on HDFS, in files that we will call HFiles.



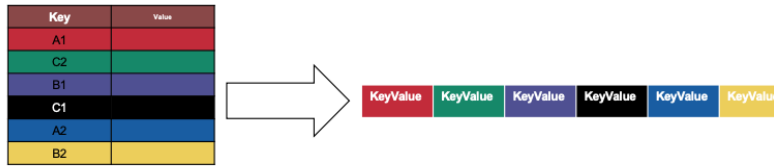
An HFile is, in fact, nothing else than a (boring) flat list of KeyValues, one per cell. What is important is that, in an HFile, all these KeyValues are sorted by key in increasing order, meaning, first sorted by row ID, then by column family (trivially unique for a given store), then by column qualifier, then by version.



This means that all versions of a given cell that are in the same HFile are located together, and one of the (within this HFile) is the latest:

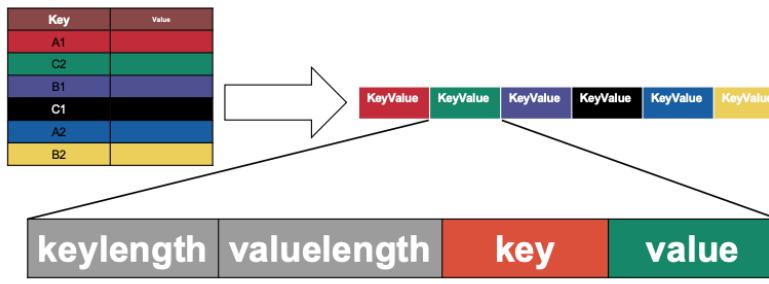


Of course, on the disk, a file is a sequence of 0s and 1s with no tabular structure, so that what in fact happens is that the KeyValues are stored sequentially, like so:



Now if we zoom in at the bit level, a KeyValue consists in four parts:

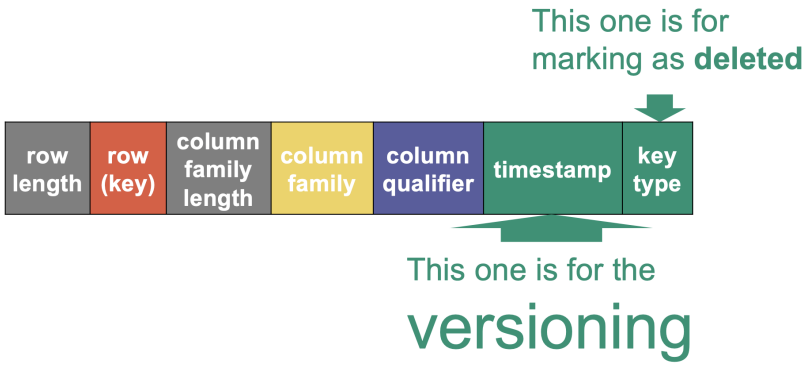
- The length of the keys in bits (this length is encoded on a constant, known number of bits)
- The actual key (of variable length)
- The length of the value in bits (this length is encoded on a constant, known number of bits)
- The actual value (of variable length)



Why do we not just store the key and the value? This is because their length can vary. If we do not know their length, then it is impossible to know when they stop just looking at the bits. Thus, the trick is to start with saving the length of the keys, this length being itself always stored as 32 bits. The engine can thus look at the next 32 bits, decodes the length n, then look at the next n bytes (remember, 1 byte is 8 bits) and stops. We have the key. Then The engine looks at the next 32 bits, decode the length m, then look at the next m bytes and stop. We have the value. And then proceed to the next KeyValue.

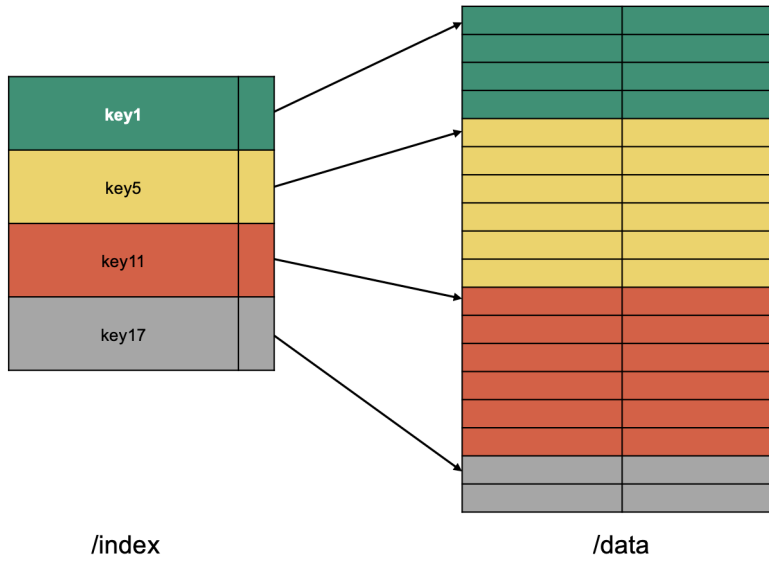
Zooming in, the key is itself made of the row ID, the column family, the column qualifier and the timestamp. We need also a row ID length and a column family length (similar to the key length and the value length). The timestamp has a fixed length (64 bits) and does not need additional input on its length. Finally the last bit (named "key type" for some reason) is a deletion flag that indicates that the cell with this version was deleted.

Why does the column qualifier not have an additional column qualifier length? This is because we know the value length, so the column qualifier length would be superfluous as it can be deducted with simple arithmetics.



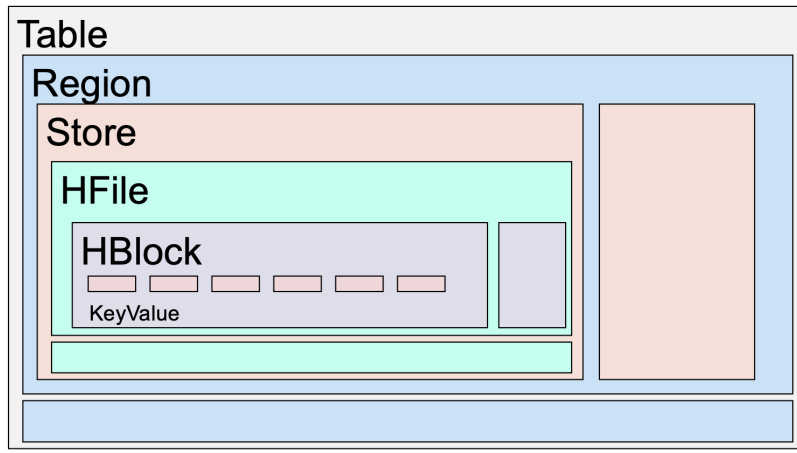
The next thing to know is that KeyValues, within an HFile, are organized in blocks. But to not confuse them with HDFS blocks, we will call them HBlocks. HBlocks have a size of 64 kB, but this size is variable: if the last KeyValue goes beyond this boundary, then the HBlock is simply longer and stops whenever the last KeyValue stops. This is in particular the case for large values exceeding 64 kB, which will be "their own HBlock."

The HFile then additionally contains an index of all blocks with their key boundaries.



This separate index is loaded in memory prior to reading anything from the HFile. It can then be kept in memory for subsequent reads. Thanks to the index, it is possible to efficiently find out in which HBlock the KeyValues with a specific key (or within a specific key range) are to be read. We will study such indices in more details in Chapter **??**.

The following is a summary of the entire physical storage hierarchy of KeyValues on HDFS:

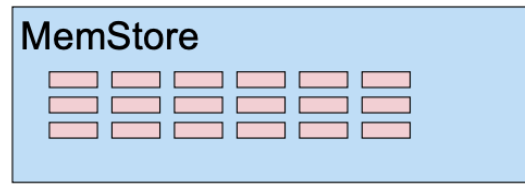


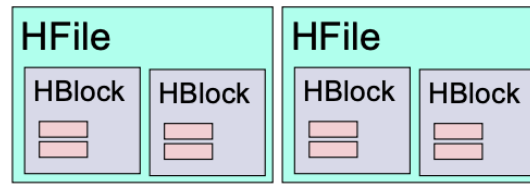
At this point, the astute reader may be wondering: but wait, if all the KeyValues within an HFile are sorted, and it is not possible to modify files on HDFS randomly, how is it even possible to insert any new KeyValues when the HBase users puts new cells into the tables?

6.5.4 Log-structured merge trees

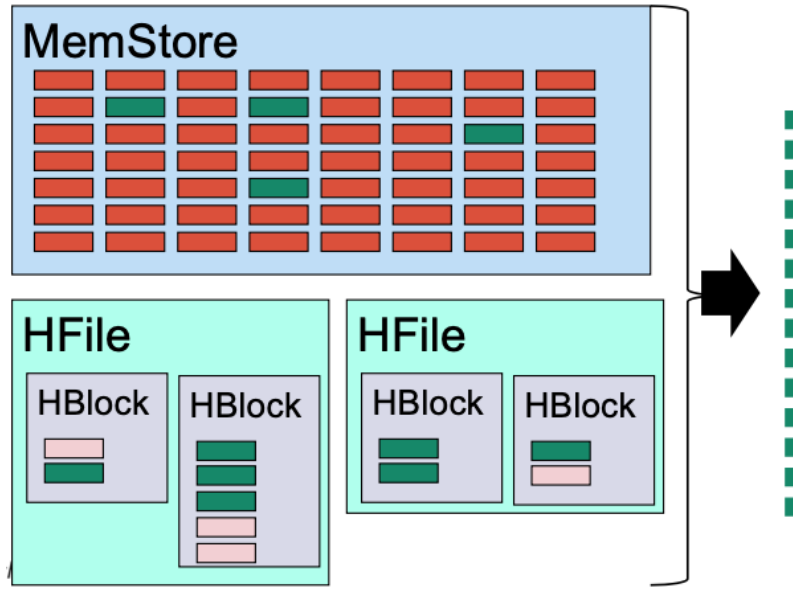
Before we dive into writing and persisting new data, it is important to understand where the data is read from.

Generally, "old" data is persisted to HDFS in HFiles, while "fresh" data is still in memory on the RegionServer node, and has not been persisted to HDFS yet.

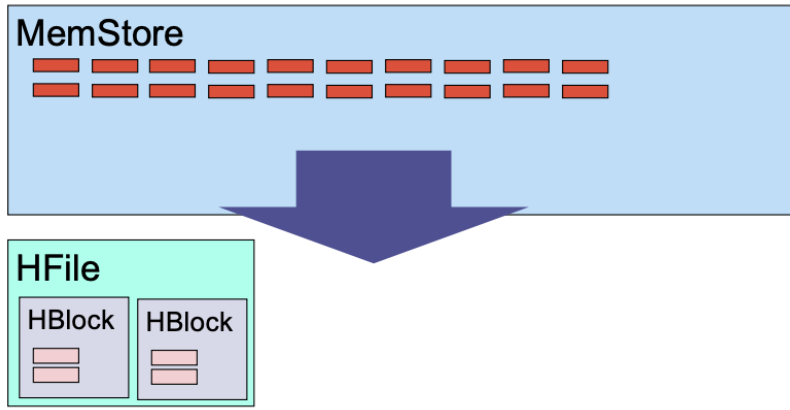




Thus, when accessing data, HBase needs to generally looks everywhere for cells to potentially return: in every HFile, and in memory.

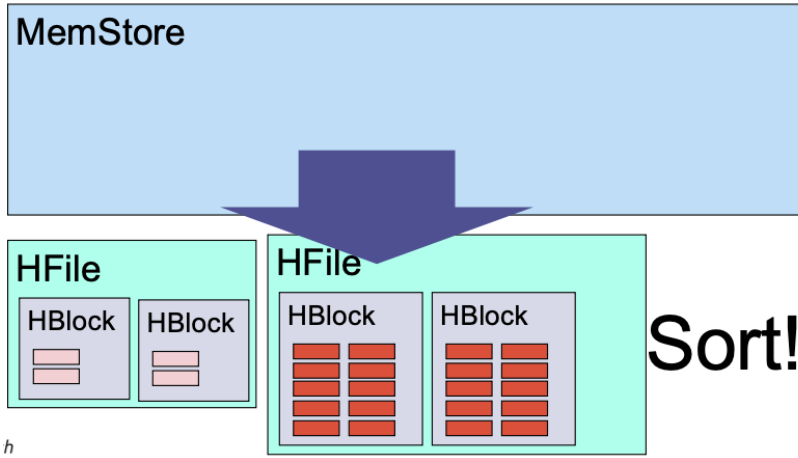


As long as there is room in memory, freshly created cells are added in memory. At some point, the memory becomes full (or some other limits are reached). When this happens, all the cells need to be flushed to a brand new HFile.



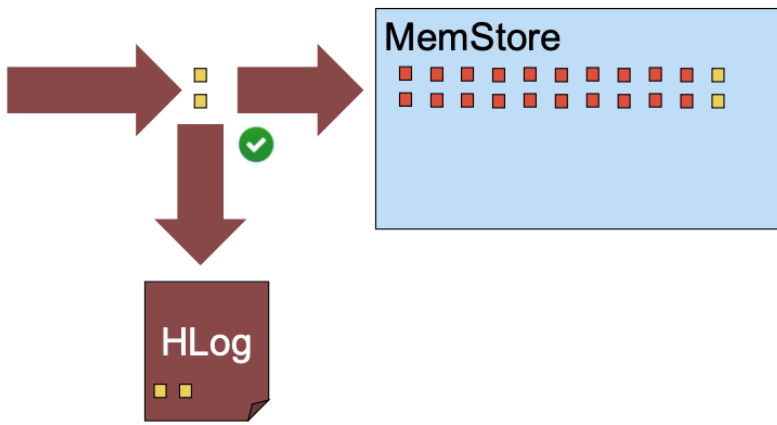
Upon flushing, all cells are written sequentially to a new HFile in ascending key order, HBlock by HBlock, concurrently building the index structure. In fact, sorting is not done in the last minute when flushing. Rather, what happens is that when cells are added to memory, they

are added inside a data structure that maintains them in sorted order (such as tree maps) and then flushing is a linear traversal of the tree.

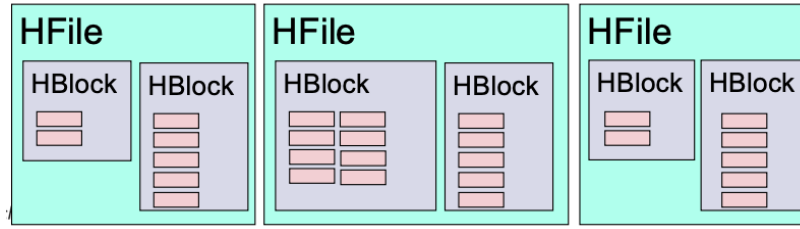


What happens if the machine crashes and we lose everything in memory? We have a so-called write-ahead-log for this. Before any fresh cells are written to memory, they are written in sequential order (append) to an HDFS file called the HLog. There is one per store. A full write-head-log also triggers a flush of all cells in memory to a new HFile.

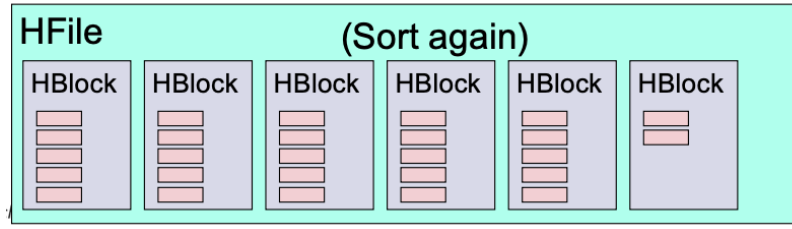
If there is a problem and the memory is lost, the HLog can be retrieved from HDFS and "played back" in order to repopulate the memory and recreate the sorting tree structure.



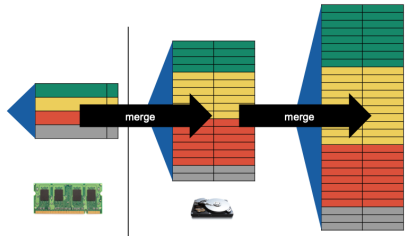
After many flushes, the number of HFiles to read from grows and becomes impracticable. For this reason, there is an additional process called compaction that takes several HFiles:



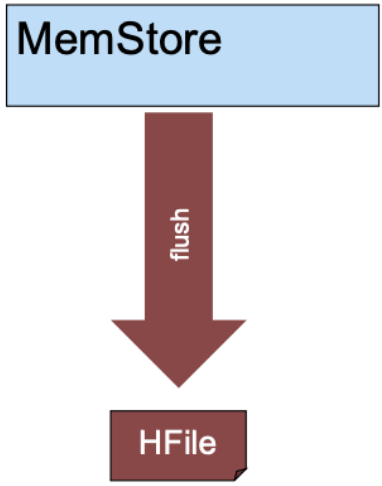
And outputs a single, merged HFile. Since the cells within each HFile are already sorted, this can be done in linear time, as this is essentially the merge part of the merge-sort algorithm.



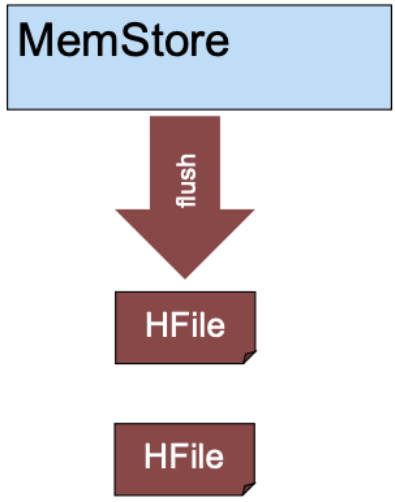
With flushing and compaction, we are starting to see some cycle of persistence, as illustrated below. On a first level, the cells in memory, on a second level, the cells that have been flushed, on the third level, the cells that have been flushed and compacted once, etc.



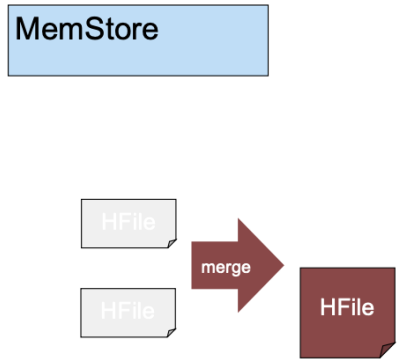
Compaction is not done arbitrarily but follows a regular, logarithmic pattern. Let us go through it. In a fresh HBase store, the memory becomes full at some point and a first HFile is output in a flush.



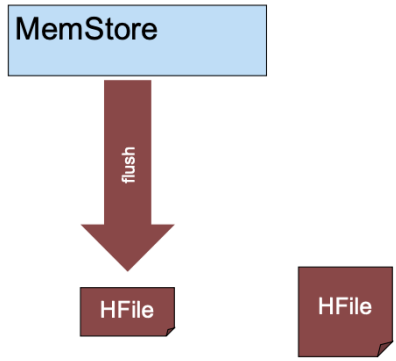
Then the memory, which was emptied, becomes full again and a second HFile is output in a flush.



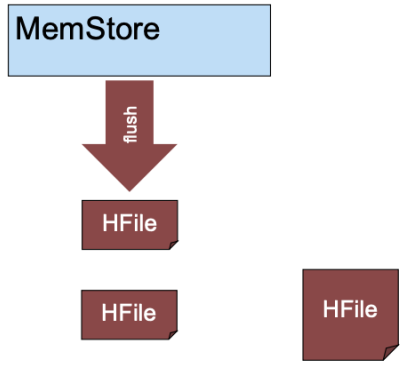
This results in two HFiles of "standard size" that are immediately compacted to one HFile, twice as large.



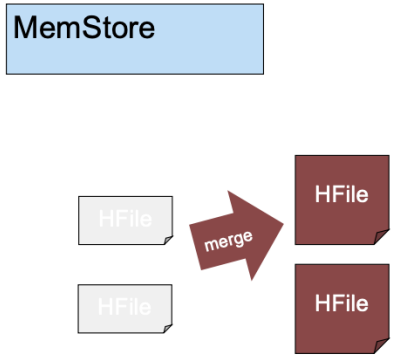
Then the memory, which was emptied, becomes full again and a new "standard-size" HFile is output in a flush.



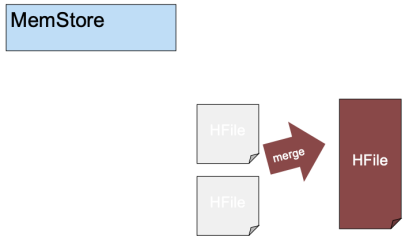
Then the memory, which was emptied, becomes full again and a second "standard-size" HFile is output in a flush.



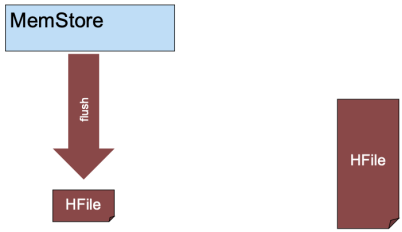
This results in two HFiles of "standard size" that are immediately compacted to one HFile, twice as large.



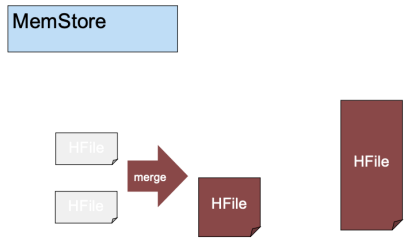
This results in two HFiles of "double size" that are immediately compacted to one HFile, four times as large as the standard size:



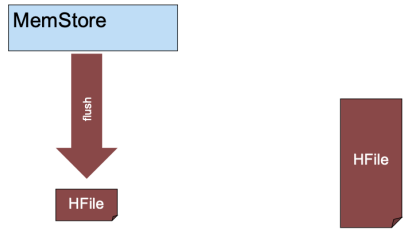
Then the memory, which was emptied, becomes full again and a new "standard-size" HFile is output in a flush.



By now, the process should be clear: when the memory is flushed again, an standard-size HFile is written and the two standard-size HFiles are immediately compacted to a double-size HFile.



When the memory is flushed again, an standard-size HFile is written, and so on, and so on.

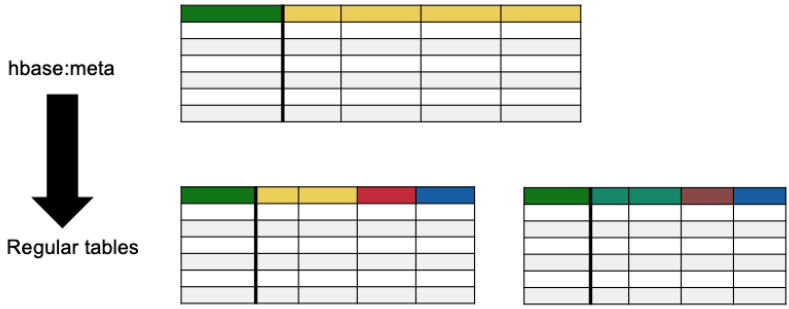


If you paid attention, you will have noticed that the pattern of the HFiles, looked at in a mirror, is simply counting in base 2: 1, 10, 11, 100, 101, 110, 111, and so on. In fact, this number in base two times the size of a standard HFile gives you the total persisted size on HDFS.

6.6 Additional design aspects

6.6.1 Bootstrapping lookups

In order to know which RegionServer a client should communicate with to receive cells corresponding to a specific region, there is a main, big lookup table that lists all regions of all tables together with the coordinates of the RegionServer in charge of this region as well as additional metadata (e.g. to support splitting regions, etc).



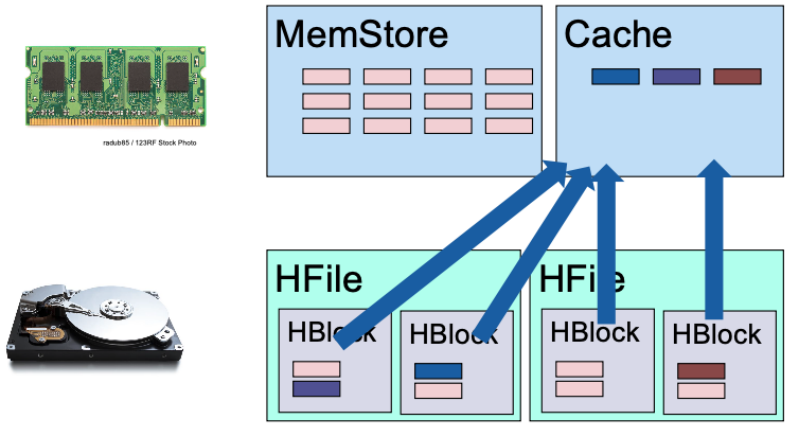
This big table is, in fact, also an HBase table, but it is special because this one fits on just one machine, known to everybody. Thus, the clients use this so-called meta table to know which RegionServers to communicate with.

There also exists an alternate design (commonly found in early versions of BigTable or HBase) with two levels of meta tables in order to scale up the number of tables.

To create, delete or update tables, clients communicate with the HMaster.

6.6.2 Caching

In order to improve latency, cells that are normally persisted to HFiles (and thus no longer in memory) can be cached in a separate memory region, with the idea of keeping in the cache those cells that are frequently accessed. We will not go into much details about the cache here and refer to computer architecture textbooks.

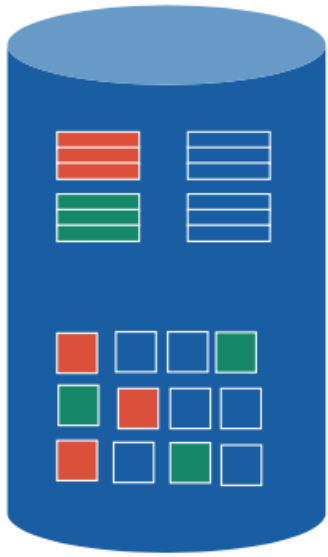


6.6.3 Bloom filters

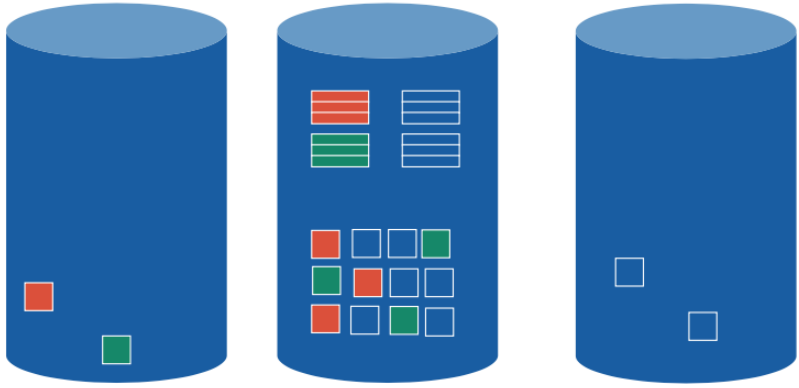
HBase has a mechanism to avoid looking for cells in *every* HFile. This mechanism is called a Bloom filter. It is basically a black box that can tell with absolute certainty that a certain key does not belong to an HFile, while it only predicts with good probability (albeit not certain) that it does belong to it. A Bloom filter is implemented using a multiple hashing mechanism that sets Boolean flags in an array, which is very efficient. By maintaining Bloom filters for each HFile (or even each column), HBase can known with certainty that some HFiles need not be read when looking up certain keys.

6.6.4 Data locality and short-circuiting

It is informative to think about the interaction between HBase and HDFS. In particular, recollect when we said that HDFS outputs the first replica of every block on the same (DataNode) machine as the client. Who is the client here? The RegionServer, which does co-habit with a DataNode. Now the pieces of the puzzle should start assembling in your mind: this means that when a RegionServer flushes cells to a new HFile, a replica of each (HDFS) block of the HFile is written, by the DataNode process living on the same machine as the RegionServer process, to the local disk. This makes accessing the cells in future reads by the RegionServer extremely efficient, because the RegionServer can read the data locally without communicating with the NameNode: this is known as short-circuiting in HDFS.



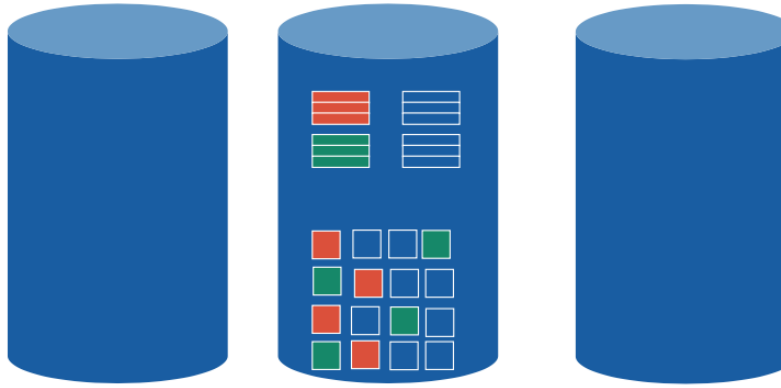
However, as time flies and the HDFS cluster lives its own life, some replicas might be moved to other DataNodes when rebalancing, making short-circuiting not (always) possible[1].



This, however, is not a problem, because with the log-structured merge tree mechanism, compactions happen regularly. And with every

[1] Trick to remember: quantum physicists might see here something akin to a wave function spreading to a broader support over time!

compaction, the replicas of the brand new HFile are again written on the local disk[2].



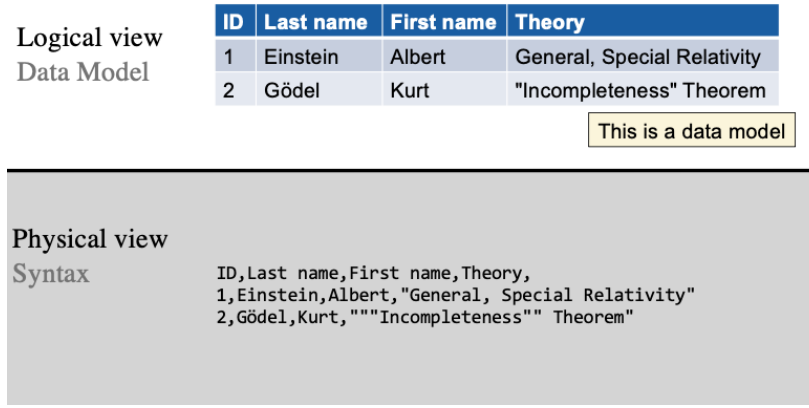
[2]Trick to remember: quantum physicists might see here something akin to a quantum measurement with wave function collapse!

Chapter 7

Data models and validation

Even though the data is physically stored as bits – or as text directly encoded to bits in the case of XML and JSON – it would not be appropriate to directly manipulate the data at the bit or text level. This is, in fact, in the spirit of data independence to abstract away. Doing so is called data modelling.

A data model is an abstract view over the data that hides the way it is stored physically. For example, a CSV file should be abstracted logically as a table. This is because CSV enforces at least relational integrity as well as atomic integrity. As for domain integrity, this can be considered implicit since an entire column can be interpreted as a string in the case of incompatible literals.



So how do we do the same for JSON and XML?

7.1 The JSON Information Set

Obviously, a model based on tables is not appropriate for JSON. This is because, unlike CSV, JSON enforce neither relational integrity, nor atomic integrity, nor domain integrity. In fact, we will see that the appropriate abstraction for any JSON document is a tree.

The nodes of that tree, which are JSON logical values, are naturally of six possible kinds: the six syntactic building blocks of JSON.

These are the four leaves corresponding to atomic values:

- Strings
- Numbers
- Booleans
- Nulls

As well as two intermediate nodes (possibly leaves if empty):

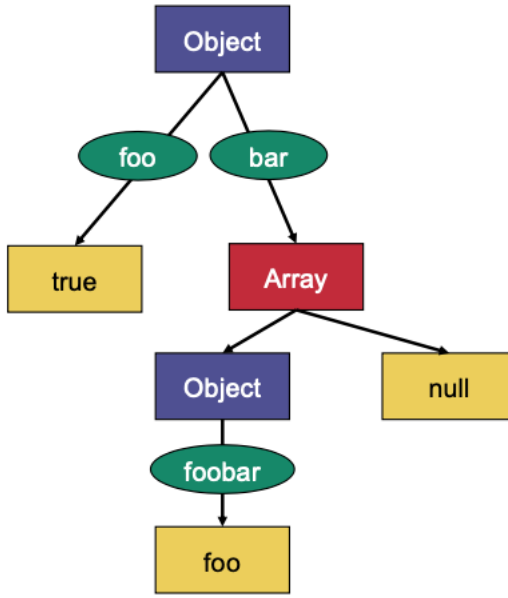
- Objects (String-to-value map)
- Arrays (List of values)

Formally, and not only for JSON but for all tree-based models, these nodes are generally called *information items* and form the logical building blocks of the model, called *information set*.

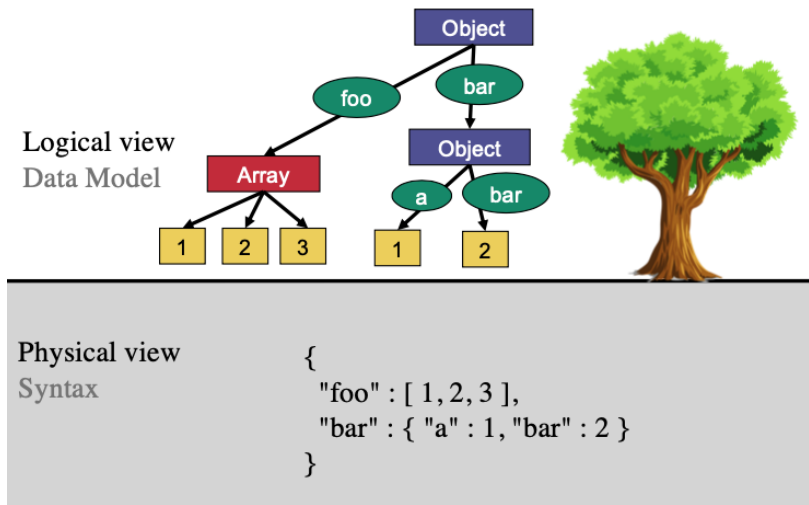
Let us take the following example.

```
{
  "foo" : true,
  "bar" : [
    {
      "foobar" : "foo"
    },
    null
  ]
}
```

It is possible to draw this document as a logical tree, where each information item (node) corresponds to each one of the values present in the document: two objects, one array, and three atomics. Note that the information items are the rectangles; the ovals are not information items but labels on the edges connecting the information items. The ovals correspond to object keys.



It is possible to do so for any JSON document. Thus, we have now obtained a similar logical/physical mapping to what we previously did with CSV and tables, except that this is now with JSON and trees:

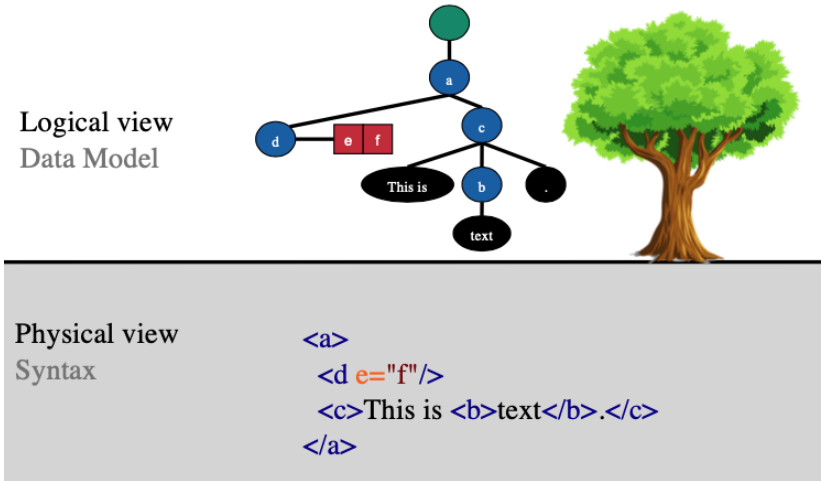


When a JSON document is being parsed by a JSON library, this tree is built in memory, the edges being pointers, and further processing will be done on the tree and not on the original syntax.

Conversely, it is possible to take a tree and output it back to JSON syntax. This is called *serialization*.

7.2 The XML Information Set

It is possible to do the same logical abstraction, also based on trees, with XML, where information items correspond to elements, attributes, text, etc:



A fundamental difference between JSON trees and XML trees is that for JSON, the labels (object keys) are on the *edges* connecting an object information item to each one of its children information items. In XML, the labels (these would be element and attribute names) are on the *nodes* (information items) directly. Another way to say it is that a JSON information item *does not know* with which key it is associated in an object (if at all), while an XML element or attribute information item *knows* its name.

Let us dive more into details. In XML, there are many more information items:

- Document information items
- Element information items
- Attribute information items

- Character information items
- Comment information items
- Processing instruction information items
- Namespace information items
- Unexpanded entity reference information items
- DTD information items
- Unparsed entity information items
- Notation information items

For the purpose of this course, though, we will only go into the most important ones from a data perspective: documents, elements, attributes, and characters. We will leave comments and namespaces aside to keep things simple, even though we saw what they look like syntactically, and will also skip all other information items, for which we have not studied the syntax.

Let us take this example:

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

which we can also highlight with colors to ease the read (any editor will do this with a well-formed XML document):

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

Formally, the XML Information Set is defined in a standard of the World Wide Web consortium (W3C). Each kind of information item has specific properties, and some of these properties link it to other information items, building the tree.

Let us go through the information items for the above document and list some of its properties.

7.2.1 Document information item

The document information item is just the root of an XML tree. It does not correspond to anything syntactically or, if at all, it would correspond to the text and doctype declarations.

The documentation information has two important properties:

- [children] Element information item *metadata*
- [version] 1.0

7.2.2 Element information items

There is one element information item for each element. Here we have three.

The element information item metadata has four important properties:

- [local name] metadata
- [children] Element information item title, element information item publisher
- [attributes] (empty)
- [parent] Document information item

The element information item title has four important properties:

- [local name] title
- [children] Character information items (Systems Group)
- [attributes] Attribute information item language, Attribute information item year
- [parent] Element information item metadata

The element information item publisher has four important properties:

- [local name] publisher

- [children] Character information items (ETH Zurich)
- [attributes] (empty)
- [parent] Element information item metadata

7.2.3 Attributes information items

There is one attribute information item for each attribute. Here we have two.

The attribute information item language has three important properties:

- [local name] language
- [normalized value] en
- [parent] Element information item title

The attribute information item year has four important properties:

- [local name] year
- [normalized value] 2019
- [parent] Element information item title

7.2.4 Character information items

There are as many character information items as characters in text (between tags). For example, for the S in Systems Group:

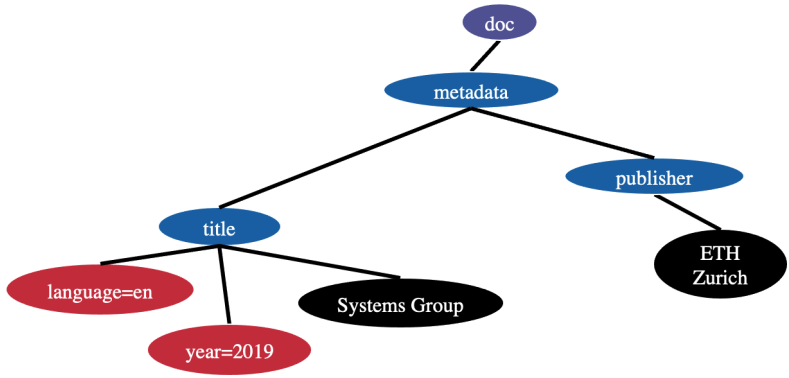
- [character code] the unicode code point for the letter S
- [parent] Element information item title

It is sometimes simpler to group them into a single (non standard) "text information item":

- [characters] S y s t e m s G r o u p
- [parent] Element information item title

7.2.5 The entire tree

All information items built previously can finally be assembled and drawn as a tree. The edges, corresponding to children and parent properties, will correspond to pointers in memory when the tree is built by the XML library:



When an XML document is being parsed by a XML library, this tree is built in memory, the edges being pointers, and further processing will be done on the tree and not on the original syntax.

Conversely, it is possible to take a tree and output it back to XML syntax. This is called *serialization*.

7.3 Validation

Once documents, JSON or XML, have been parsed and logically abstracted as a tree in memory, the natural next step is to check for further structural constraints.

For example, you could want to check whether your JSON documents all associate key "name" with a string, or if they all associate "years" with an array of positive integers. Or you could want to check whether your XML documents all have root elements called "persons," and whether the root element in each document has only children elements called "person," all with an attribute "first" and an attribute "last."

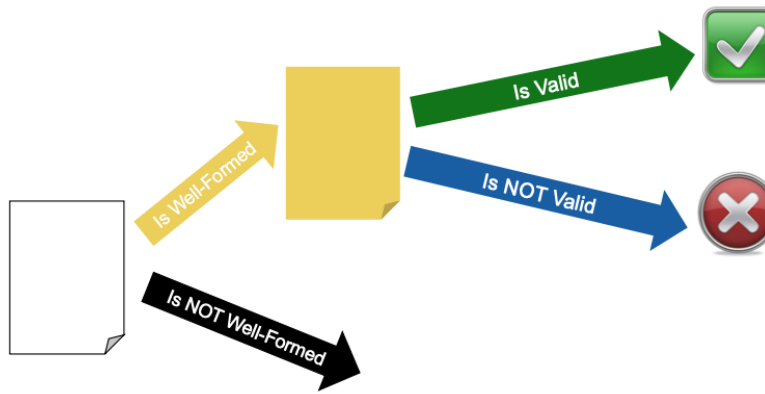
This might remind the reader of schemas in a relational database, but with a major difference: in a relational database, the schema of a table is defined before any data is populated into the table. Thus, the data in the table is guaranteed, at all times, to fulfil all the constraints of the schema. The exact term is that the data is guaranteed to be

valid against the schema, because the schema was enforce at write time (schema on write).

But in the case of a collection of JSON and XML documents, this is the other way round. A collection of JSON and XML documents out there can exist without any schema and contain arbitrary structures. Validation happens "ex post," that is, only after reading the data (schema on read).

Thus, it means that JSON and XML documents undergo two steps:

- a well-formedness check: attempt to parse the document and construct a tree representation in memory
- (if the first step succeeded) a validation check given a specific schema



Thus, a text document can be either not well-formed, or well-formed and invalid against a specific schema, or well-formed and valid against a specific schema.

Note that, unlike well-formedness, validation is schema dependent: a given well-formed document can be valid against schema A and invalid against schema B.

Validation is often performed on much more than a document at a time: an entire collection. Thus, we distinguish between heterogeneous collections, whose documents are not valid against any particular schema, and homogeneous collections, whose documents are all valid against a specific schema.

No schema
(heterogeneous)

With schema
(homogeneous)

To give an intuitive feeling, this is what a not-validated, "messy" document looks like. Notice in particular how values in the same array are of different types.

```
{
  "a" : 1,
  "b" : [
    "foo",
    true,
    null,
    {
      "foo" : "bar"
    }
  ],
  "c" : {
    "d" : { "foo" : null },
    "e" : [ 1, 2, [ 3, 4 ] ],
    "f" : 3.14
  }
}
```

This, on the other hand, is a document that has more structure and could easily be validated against an appropriate schema: for example, the array associated with key "c" only has object elements, which all have a "foo" key associated with a string and an optional "bar" key associated with an array of integers.

```
{
  "a" : 1,
  "b" : true,
  "c" : [
    { "foo":"bar1", "bar":[1,2 ] },
    { "foo":"bar2", "bar":[ 3,4,5 ]},
    { "foo" : "bar3" }
  ]
}
```

7.4 Item types

A fundamental aspect of validation is the type system. A well-designed type system, in turn, allows for storing the data in much more efficient, binary formats tailored to the model.

There are many different technologies and type systems for arborescent, denormalized data, well beyond only JSON and XML. But there is good news: all these type systems are very similar to each other and have a lot in common. Therefore, in this section, we will present an overview of an agnostic type system that is very representative of common practice. After reading this, learning a new validation or storage format technology will then amount to reading the documentation and immediately recognizing familiar patterns, to then focus on the small deviations and peculiarities of each specific technology.

The first aspect that almost all, if not all type systems, have in common, is the distinction between atomic types and structured types. In fact, this distinction is so universal that even object-oriented languages like Java and Python make such a distinction.

The distinction should also not really come as a surprise to the reader, because we have seen it several times before: in Chapter 5 and in this one, when we looked at JSON and XML.

7.4.1 Atomic types

Atomic types correspond to the leaf of a tree data model: these are types that do not contain any further nestedness.

The kinds of atomic types available are also relatively standard and common to most technologies.

Also, all atomic types have in common that they have a logical value space and a lexical value space. The logical value space corresponds to the actual logical abstraction of the type (e.g., a mathematical set of integers), while the lexical value space corresponds to the representation of logical values in a textual format (such as the decimal, or binary, or hexadecimal representation of an integer).

An atomic type also has a (not necessarily injective) mapping from its lexical value space to its logical value space (e.g., mapping the hexadecimal literal x10 to the mathematical integer sixteen), and often, a canonical mapping from the logical value space to the lexical value space (e.g., mapping the mathematical integer sixteen to its canonical decimal representation 16).

Atomic types can be in a subtype relationship: a type is a subtype of another type if its logical value space is a subset of the latter. Normally it means that the same holds for the lexical value spaces and the related mappings should be consistent with each other. The subtype relationship over types organizes the types as a hierarchy, called the type hierarchy. The type hierarchy gives a good visual to get a quick overview of all available types in a specific technology.

Let us take a tour.

Strings

Strings are simply finite sequences of (usually printable) characters. Formally, strings form a monoid under concatenation, where the neutral element is the empty string.

These are three examples:

```
foo
Zurich
Ilsebill salzte nach.
```

Often, the lexical representation of a string is double-quoted, sometimes also single-quoted.

```
"foo"
"Zurich"
'Ilsebill salzte nach.'
```

The difference between the lexical representation and the logical value of a string becomes immediately apparent when escaping is used. For example, the lexical representation

```
"\\\""
```

corresponds to the (logical) string \".

In "pure computer science" textbooks, strings are often presented as structured values rather than as atomic values because of their complexity on the physical layer. However, for us data scientists, strings are atomic values.

Numbers: integers

Integers correspond to finite cardinalities (counting) as well as their negative counterparts. These are decimal numbers without anything after the decimal period, or fractions with a denominator of 1.

In older programming languages, support for integers used to be bounded. This is why classical types, still in use today, correspond to 8-bit (often called byte), 16-bit (often called short), 32-bit (often called int) and 64-bit integers (often called long). This means that, expressed in base 2, they use 8, 16, 32 or 64 bits (binary digits).

However, in modern databases, it is customary to support unbounded integers. In XML, the corresponding type is simply called integer, as opposed to int. Engines can optimize computations for small integers, but might become less efficient with very large integers.

The other, restricted integer types are called *subtypes* of the integer type, because their logical value space is a subset of the set of all integers.

Other commonly found integer subtypes include positive integers, nonpositive integers (also called unsigned integers), negative integers, non-negative integers.

The lexical representation of integers is usually done is base 10, in the familiar decimal system, even though base 2, 8 or 16 can be found, too. Leading 0s are optional, but when an logical integer value is canonically serialized, it is done without a leading 0.

Note that the exact names of the types can vary! Some systems might use integer for 32-bit integers or int for the entire value space. You might need to read the documentation of the specific technology you are using if in doubt.

Numbers: decimals

Decimals correspond to real numbers that can be written as a finite sequence of digits in base 10, with an optional decimal period. Equivalently, these are fractions that can be expressed with a power of 10 in the denominator.

Many modern databases or storage formats support the entire logical decimal value space with no restriction on how large, small or precise a decimal number is.

The lexical representation of integers is usually done is base 10, it is not common to use other bases for decimals.

Numbers: floating-point

Support for the entire decimal value space can be costly in performance. In order to address this issue, a floating-point standard (IEEE 754) was invented and is still very popular today. These are the types known

as float and double in many programming languages. They can be processed natively by processors.

Floating-point numbers are limited both in precision and magnitude (both upper and lower) in order to fit on 32 bits (float) or 64 bits (double). Floats have about 7 digits of precision and their absolute value can be between roughly $10^{-}37$ and $10^{3}7$, while doubles have 15 digits of precision and their absolute value can be between roughly $10^{-}307$ and $10^{3}08$

Double and float types also cover additional special values: NaN (not a number), positive and negative infinity, and negative zero (in addition to the "positive" 0).

Note that the exact names of the types can vary! Some systems might use float for double, or decimal for double, etc. You might need to read the documentation of the specific technology you are using if in doubt.

The lexical representation of floats and doubles often use the scientific notation:

```
-12.34E-56
```

And the lexical values corresponding to the special logical values look like so:

```
NaN
INF
-INF
-0
```

Booleans

The logical value space for the Boolean type is made of two values: true and false as in NoSQL queries, two-valued logic is typically assumed.

The corresponding lexical values are typically:

```
true
false
```

If an unknown value is needed in the spirit of three-valued logic, the null can be used (see below) or it can be left absent.

Dates and times

Dates and times are a very important component of databases because they are heavily needed by users, albeit often neglected or forgotten by some formats.

Dates are commonly using the Gregorian calendar (with some technologies possibly supporting more) with a year (BC or AD), a month and a day of the month.

Times are expressed in the hexagesimal (60) basis with hours, minutes, seconds, where the seconds commonly go all the way to microseconds (six digits after the decimal period).

Datetimes are expressed with a year, a month, a day of the month, hours, minutes and (decimal) seconds.

Some technologies offer types that support time zones (UTC, CEST, PDT...) or the explicit absence of any time zone for dates, times and datetimes.

The timestamp type corresponds to a datetime with a timezone, but treating datetimes as equivalent if they express the same point in time (formally, it means that it is a timezoned datetime quotiented with an equivalence relation). Timestamp values are typically stored as longs (64-bit integers) expressing the number of milliseconds elapsed since January 1, 1970 by convention.

The lexical values can also vary, although many technologies follow the ISO 8601 standard, where lexical values look like so (with many parts optional):

```
2022-08-07T14:18:00.123456+02:00
2022-08-07
14:18:00.123456
14:18:00.123456Z
```

The names of date, time, datetime and timestamp types vary largely between technologies and formats. For the purpose of this lecture, we will focus on the standardized XML Schema types, which are also the same as in JSound and the JSONiq language that we will study later. The types are called date, time, dateTime and dateTimeStamp. XML Schema additional supports values made of just a year (gYear), just a month (gMonth), just a day of the month (gDay), or a year and month (gMonthYear), or a month and day of the month (gMonthDay). XML Schema, JSound and JSONiq follow the ISO 8601 standard.

Durations

Durations can be of many different kinds, generally a combination of years, months, days, hours, minutes and (possibly with decimals) seconds.

What is important to understand is that there is a "wall" between months and days: what is the duration "1 month and 1 day?" It could be 29, 30, 31, or 32 days and should thus be avoided. Thus, most durations, for the sake of being unambiguous, are either involving years

and/or months, or are involving days and/or hours and/or minutes and/or seconds.

The lexical representation can vary, but there is a standard defined by ISO 8601 as well, starting with a P and prefixing sub-day parts with a T.

For example 2 years and 3 months:

```
P1Y3M
```

4 days, 3 hours, 2 minutes and 1.123456 seconds:

```
P1DT3H2M1.123456S
```

3 hours, 2 minutes and 1.123456 seconds:

```
PT3H2M1.123456S
```

```
2022-08-07T14:18:00.123456+02:00
2022-08-07
14:18:00.123456
14:18:00.123456Z
```

XML Schema, JSound and JSONiq, used in this course, follow the ISO 8601 standard.

Binary data

Binary data is, logically, simply a sequence of bytes.

There are two main lexical representations used in data: hexadecimal and base64.

Hexadecimal expresses the data with two hexadecimal digits per byte, like so:

```
0123456789ABCDEF
```

which would correspond to the bit sequence:

```
0000000100100011010001010110011111000100110101011110011011110111 1
```

Base 64, formally, does the same but in the base 64, which "wastes" less lexical space in the text. It does so by encoding the bits six by six, encoding each sequence of six bits with one base-64 digit. Equivalently, it means that each group of three bytes is encoded with four base-64 digits. The base-64 digits are, by convention, decimal digits, all uppercase latin alphabet letters, all lowercase latin alphabet letters, as well as + and /. = is used for padding if the length of the base-64 string is not a multiple of four.

This is the base-64 lexical representation of the same binary data as above, which is textually more compact than the hexadecimal version:

```
ASNFZ4mrze8=
```

In XML Schema, JSound and JSONiq, this is covered with two types hexBinary and base64Binary.

Note that the string type could also be considered to provide an additional lexical representation of a binary type (e.g., with the UTF-8 encoding), although this can cause issues with non-printable characters.

Null

Many technologies and formats also provide support for null values, although how this is done largely varies.

Some technologies allow null to appear as a (valid) value for *any* type. A schema can either allow, or disallow the null value. Often, the terminology used is that a type can be nullable (or nillable) or not.

Other technologies consider that there is a singleton-valued null type, containing only the null value with the lexical representation

```
null
```

Allowing nulls is done by taking the union of the desired type with the null type.

Yet other technologies consider null when it appears as a value in an object to be semantically equivalent by the absence of a value and then, allowing or disallowing null is achieved by (e.g. in JSON) making the field required or optional.

It is important to understand that the latter technologies are unable to distinguish between the following two JSON objects, so that information in the input dataset is lost upon validating:

```
{ }
{ "foo" : null }
```

This can be problematic with datasets where this distinction is semantically relevant.

XML also supports null values, but calls them “nil” and does so with a special attribute and no content rather than with a lexical representation

```
<foo
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:nil="true"/>
```

7.4.2 Structured types

Most technologies and formats offer four kinds of structured types (or a subset thereof).

Lists

Lists correspond to JSON arrays and are ordered sequences of (atomic or structured) values.

With a schema, it is possible to defines types that are, for example, lists or integers, or lists of strings, or lists of lists of doubles, etc.

Records

Records, or structs, correspond to JSON objects and are maps from strings to values.

With a schema, it is possible to restrict the type accepted for every key, e.g., name must be a string, birthday must be a date, etc.

Maps

Maps (not be confused with records, which are similar) are maps from any atomic value to any value, i.e., generalize objects to keys that are not necessarily strings (e.g., numbers, dates, etc).

With a schema, it is possible to restrict the type of the keys, as well as the type of the values. However, unlike records, the type of the values must be the same for all keys. For example, a map from dates to numbers.

Sets

Sets are rarer and supported by rather few technologies and formats, but they exist.

Sets are like lists, but without any specific ordering, and without duplicate values.

XML elements and attributes

XML Schema stands apart from most other technologies and formats, in that it does not offer specific support for records and maps; it offers some limited support for lists, but considers them to be simple types, which are "inbetween" atomic types and structured types. In XML Schema, structure is obtained, instead, with elements and attributes, and the machinery for elements and attributes is highly specific to XML. Elements and attributes can in particular easily emulate records and lists, but are more powerful (i.e., elements can be repeated, and intermixing text with elements is allowed).

Most formats besides XML are "JSON-like" from a modelling perspective and structure their data at the very minimum with lists and records.

Type names

For convenience, we provide below a summary of many types over various technologies and languages and how they correspond to each other. The most important thing to see here is that on the high level, atomic types are almost always the same everywhere, even though the names can vary.

JSound/JSONiq	JSON Schema	XML Schema	SparkSQL	PostgreSQL	Python
decimal	number	xs:decimal	DecimalType	numeric(., .)	-
integer	integer	xs:integer	-	numeric(.)	int
byte	integer +range	xs:byte	ByteType	-	-
short	integer +range	xs:short	ShortType	smallint	-
int	integer +range	xs:int	IntegerType	int	-
long	integer +range	xs:long	LongType	bigint	-
double	number	xs:double	DoubleType	double precision	float
float	-	xs:float	FloatType	real	-
string	string	xs:string	StringType	text	str
boolean	boolean	xs:boolean	BooleanType	boolean	bool
hexBinary	string +pattern	xs:hexBinary	BinaryType	bytea	bytes/bytearray
base64Binary	-	xs:base64Binary	BinaryType	-	bytes/bytearray
date	string +format	xs:date	DateType	date	datetime.date
dateTimeStamp	string +format	xs:dateTimeStamp	TimestampType	timestamp	- (int)
object	object	-	StructType	(composite types)	dict
array	array	-	ArrayType	[]	list

71

7.5 Sequence types

7.5.1 Cardinality

In the context of data querying but also of nested lists and arrays, items (single values) rarely appear alone. They often appear as a sequence of many values. Thus, many type system give options regarding the number of occurrences of items in a sequence.

There are four main occurrence indicators:

- just once (often implicit): exactly one item
- optional: zero or one item. Often represented with a question mark (?).
- any occurence: zero, one or more items. Often represented with a Kleene star (*).
- at least once: one or more items. Often represented with a Kleene star (+).

The symbols commonly used (?, *, +) correspond to those used in regular expressions[1].

There is a variety of other ways to specify such indicators: some technologies use keywords such as "repeated," "required," "optional," etc. Finally, some technologies even allow specifying a precise interval (e.g., between 2 and 5 items).

7.5.2 Collections vs. nested lists

There are different kinds of sequences of items. It is common to distinguish between collections of items, and lists (or arrays) of items.

A collection of items is on the outer level, and can be massively large (billions, trillions of items). A collection of items corresponds to a relational table, i.e., a relational table can be seen as a collection of flat object items in the context of this chapter.

A list (or array) of items, however, usually refers to a nested structure, for example an array nested inside a document or object. Such lists of items are usually restricted in size for reasons of performance and scalability. Many technologies do not allow items to exceed a two-digit number of Megabytes (e.g., 10 MB, 16 MB, etc), or if they do allow them, might become slow and inefficient with too large items. Thus, an array of items will usually not exceed a few thousand items, a million at best.

It is thus important to keep this subtle difference in mind, in particular, do not confuse a collection of integers with a collection that contains a single array of integers.

[1] for non-computer scientists, regular expressions are formulas that give pattern for accepting strings, for example for an integer: $-?[1-9][0-9]*$

7.6 JSON validation

Let us now look at how to use what we have learned so far to validate JSON documents. First, it is important to remember that we can only attempt to validate well-formed JSON documents. If a document cannot be parsed as JSON, and thus, cannot be represented as a tree in memory, then validation makes no sense on it.

7.6.1 Validating flat objects

JSound is a schema language that was designed to be simple for 80% of the cases, making it particularly suitable in a teaching environment. It is independent of any programming language.

The compact syntax of JSound, which we focus on here, is very close to how the original documents look like. Let us take an example:

```
{
  "name" : "Einstein",
  "first" : "Albert",
  "age" : 142
}
```

Let us say we want to validate the above document, in the sense that "name" should be a string, and "first" should be a string. The corresponding JSound schema looks like so:

```
{
  "name" : "string",
  "first" : "string",
  "age" : "integer"
}
```

"string" can be replaced with any other named type, in particular taken from the table shown in the former section. Let us list them here:

- Strings: string, anyURI (for strings containing a URI);
- Numbers: decimal, integer, float, double, long, int, short, byte, negativeInteger, positiveInteger, nonNegativeInteger, nonPositiveInteger, unsignedByte, unsighedShort, unsignedInt, unsignedLong;
- Dates and times: date, time, dateTime, gYearMonth, gYear, gMonth, gDay, gMonthDay, dateTimeStamp;
- Time intervals: duration, yearMonthDuration, dayTimeDuration;
- Binary types: hexBinary, base64Binary

- Booleans: boolean
- Nulls: null

decimal, integer, byte, short, int, long, double, float, string, boolean, hexBinary, base64Binary, date, time, dateTime, dateTimeStamp, etc.

This is a standardized list of types defined by the W3C[2].

JSON Schema is another technology for validating JSON documents. A JSON Schema against which the same document as above is valid would be:

```
{
  "type" : "object",
  "properties" : {
    "name" : "string",
    "first" : "string",
    "age" : "number"
  }
}
```

The available JSON Schema types are string, number, integer, boolean, null, array and object. This closely matches the original JSON syntax with the only exception that numbers have this additional integer subtype. The type system of JSON Schema is thus less rich than that of JSound, but extra checks can be done with so-called formats, which include date, time, duration, email, and so on including generic regular expressions. Like JSound, JSON Schema also allow restricting the length of a string, constraining numbers to intervals, etc.

7.6.2 Requiring the presence of a key

By default, the presence of a key is optional, so that each one of the following objects is also valid against the previous schema:

```
{ "name" : "Einstein" }
{ "first" : "Albert" }
{ "age" : 142 }
{ "name" : "Einstein", "age" : 142 }
{ }
```

It is possible to require the presence of a key by adding an exclamation mark, like so.

[2]In the context of XML schema, however this list is universal enough to not be tied with XML.

```
{
"!name" : "string",
"!first" : "string",
"age" : "integer"
}
```

which would make the following two documents valid:

```
{ "name" : "Einstein", "first" : "Einstein", "age" : 142 }
{ "name" : "Einstein", "first" : "Einstein" }
```

This is the equivalent JSON Schema, which uses a "required" property associated with the list of required keys to express the same:

```
{
  "type" : "object",
  "required" : [ "name", "first" ]
  "properties" : {
    "name" : "string",
    "first" : "string",
    "age" : "number"
  }
}
```

7.6.3 Open and closed object types

In the JSound compact syntax, extra keys are forbidden, i.e., this document is valid against neither of the previous two schemas:

```
{
  "name" : "Einstein",
  "first" : "Albert",
  "profession" : "physicist"
}
```

The schema is said to be closed. There are ways to define JSound schemas to allow arbitrary additional keys (open schemas), with a more verbose syntax.

Unlike JSound, in JSON Schema, extra properties are allowed by default, i.e., this document is also valid against previous schemas:

```
{
  "name" : "Einstein",
  "first" : "Albert",
  "profession" : "physicist"
}
```

JSON Schema then allows to forbid extra properties with the "additionalProperties" property, like so:

```
{
  "type" : "object",
  "required" : [ "name", "first" ]
  "properties" : {
    "name" : "string",
    "first" : "string",
    "age" : "number"
  },
  "additionalProperties" : false
}
```

7.6.4 Nested structures

What about nested structures? Let us consider this document, which contains a nested array of integers.

```
{
  "numbers" : [ 1, 2, 6, 2, 7, 1, 57, 4 ]
}
```

This document is valid against the following schema (where of course, "integer" can be replaced with any other type):

```
{ "numbers" : [ "integer" ] }
```

This also works with multiple dimensions

```
{ "matrix" : [ [ "decimal" ] ] }
```

for validating matrices:

```
{ "matrix" : [ [ 0, 1 ], [ 1, 0 ] ] }
```

With the JSound compact syntax, object and array types can nest arbitrarily:

```
{
  "datapoints" : [
    {
      "features" : [ "double" ],
      "label" : "integer"
    }
  ]
}
```

The following document is valid against the above schema:

```
{
  "datapoints" : [
    {
      "features" : [ 1.2, 3.4, 5.6 ],
      "label" : 0
    },
    {
      "features" : [ 9.3, 2.6, 2.4 ],
      "label" : 1
    },
    {
      "features" : [ 1.1, 4.3, 6.5 ],
      "label" : 0
    }
  ]
}
```

The same document can also be validated against a more complex JSON Schema with nested arrays and objects, like so:

```
{
  "type" : "object",
  "properties" : {
    "datapoints" : {
      "type" : "array",
      "items" : {
        "type" : "object",
        "properties" : {
          "features" : {
            "type" : "array",
            "items" : {
              "type" : "number"
            }
          },
          "label" : {
            "type" : "integer"
          }
        }
      }
    }
  }
}
```

As we will see shortly, the "shape" of a collection of documents captured with a compact JSound schema is of particular relevance in the context of efficient data processing, and a collection of valid JSON documents with such a shape is known as a data frame.

Every schema can be given a name, turning into a type. When a document is valid against a schema, it is typical to also *annotate* the document, which means that its tree representation in memory contains additional type information and values are stored natively in their type, enabling efficient processing and space efficiency.

JSound allows for the definition not only of arbitrary array and object types as shown above, but also of additional atomic types, by imposing some constraint on existing types (e.g., airport codes by restricting the length of a string to 3 and requiring all three characters to be uppercase letters, shoe sizes with intervals, etc). These are called user-defined types.

7.6.5 Primary key constraints, allowing for null, default values

There are a few more features available in the compact JSound syntax (not in JSON Schema) with the special characters @, ? and =:

```
{
  "datapoints" : [
    {
      "@id" : "int",
      "features" : [ "double" ],
      "label?" : "integer",
      "set" : "string=training"
    }
  ]
}
```

The question mark (?) allows for null values (which are not the same as absent values). Technically, it creates a union of the specified type with the null type.

The arobase (@) indicates that one or more fields are primary keys for a list of objects that are members of the same array. In this case, it means all id fields must be different for the datapoints array of each document.

The equal sign (=) is used to indicate a default value that is automatically populated if the value is absent. In this case, if the field "set" is missing, then upon annotating the document after its validation, it will be added with a value "training".

The following document is valid against the above schema. Not that some values are quoted, which does not matter for validation:

validation only checks whether lexical values are part of the type's lexical space.

```
{
  "datapoints" : [
    {
      "id" : "10",
      "features" : [ 1.2, 3.4, 5.6 ],
      "label" : null,
      "set" : "training"
    },
    {
      "id" : 11,
      "features" : [ "9.3", 2.6, 2.4 ],
      "label" : 1
    },
    {
      "id" : 12,
      "features" : [ 1.1, 4.3, 6.5 ],
      "label" : "0",
      "set" : "test"
    }
  ]
}
```

And, after annotating it, it will look like so (except it will be represented efficiently in memory, and no longer in actual JSON syntax).

```
{
  "datapoints" : [
    {
      "id" : 10,
      "features" : [ 1.2, 3.4, 5.6 ],
      "label" : null,
      "set" : "training"
    },
    {
      "id" : 11,
      "features" : [ 9.3, 2.6, 2.4 ],
      "label" : 1,
      "set" : "training"
    },
    {
      "id" : 12,
      "features" : [ 1.1, 4.3, 6.5 ],
      "label" : 0,
```

```
      "set" : "test"
    }
  ]
}
```

7.6.6 Accepting any values

Accepting any values in JSound can be done with the type "item", which contains all possible values, like so:

```
{
  "!name" : "item",
  "!first" : "item",
  "age" : "number"
}
```

In JSON Schema, in order to declare a field to accept any values, you can use either true or an empty object in lieu of the type, like so:

```
{
  "type" : "object",
  "required" : [ "name", "first" ]
  "properties" : {
    "name" : {},
    "first" : true,
    "age" : "number"
  },
  "additionalProperties" : false
}
```

The following document validates successfully against the above JSound and JSON Schemas:

```
{
  "name" : [ "Ein", "st", "ein" ],
  "first" : "Albert",
}
```

JSON Schema additionally allows to use false to forbid a field:

```
{
  "type" : "object",
  "properties" : {
    "name" : "string",
    "first" : false,
  }
}
```

Making this document invalid:

```
{
  "name" : "Einstein",
  "first" : "Albert",
}
```

7.6.7 Type unions

In JSON Schema, it is also possible to combine validation checks with Boolean combinations.

First, disjunction (logical or) is done with

```
{
  "anyOf" : [
    { "type" : "string" },
    { "type" : "array" }
  ]
}
```

JSound schema allows defining unions of types with the vertical bar inside type strings, like so:

```
"string|array"
```

7.6.8 Type conjunction, exclusive or, negation

In JSON Schema only (not in JSound), it is also possible to do a conjunction (logical and) with

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

as well as exclusive or (xor):

```
{
  "oneOf" : [
    { "type" : "number", "minimum" : 2  },
    { "type" : "number", "multipleOf" : 2 }
  ]
}
```

as well as negation:

```
{
  "not" : { "type" : "array" }
}
```

7.7 XML validation

XML validation is also supported by several technologies. We will briefly show how one of them works, XML Schema.

Similar to how a JSound schema or a JSON Schema is a JSON document, an XML Schema is also an XML document.

7.7.1 Simple types

This is an example (well-formed) XML document:

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

And this is an XML Schema against which the above XML document is valid:

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

So what is going on here? First, you notice that all elements in an XML Schema are in a namespace, the XML Schema namespace. We explained the namespace machinery in Chapter 5, so that the xmlns:xs construct should be familiar to you. The namespace is prescribed by the XML Schema standard and must be this one. It is recommended to stick to the prefix xs, or xsd, which is also quite popular. We do not recommend declaring the XML Schema namespace as a default namespace, because it can create confusion in several respects.

Next, you will notice that the top element in an XML Schema document is the xs:schema element, and inside there is an element declaration done with the xs:element element. It has two attribute: one defines the name of the element to validate (foo) and the other one specifies its type (xs:string). The list of predefined atomic types is the same as in JSound, except that in XML Schema, all these predefined types live in the XML Schema namespace and thus bear the prefix xs as well. In fact, formally, this list of predefined types is standardized by XML Schema (along with more XML-specific types that are less known).

Let us try to change the type. integer (prefixed with xs!) needs no introduction...

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

This document is then valid:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>142857</foo>
```

Note that extra whitespaces, newlines and indentation are fine:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  142857
</foo>
```

7.7.2 Builtin types

Let us list again the most important predefined types here, almost the same as JSound, but with the xs prefix:

- Strings: xs:string, xs:anyURI (for strings containing a URI);
- Numbers: xs:decimal, xs:integer, xs:float, xs:double, xs:long, xs:int, xs:short, xs:byte, xs:negativeInteger, xs:positiveInteger, xs:nonNegativeInteger, xs:nonPositiveInteger, xs:unsignedByte, xs:unsignedShort, xs:unsignedInt, xs:unsignedLong;
- Dates and times: xs:date, xs:time, xs:dateTime, xs:gYearMonth, xs:gYear, xs:gMonth, xs:gDay, xs:gMonthDay, xs:dateTimeStamp;
- Time intervals: xs:duration, xs:yearMonthDuration, xs:dayTimeDuration;
- Binary types: xs:hexBinary, xs:base64Binary
- Booleans: xs:boolean
- Nulls: does not exist as a type in XML Schema (JSON specific).

XML Schema allows you to define user-defined atomic types, for example restricting the length of a string to 3 for airport codes, and then use it with an element (there is no prefix because, to keep things simple for teaching, we are not working with any namespaces for our own declared elements and types):

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="airportCode">
    <xs:restriction base="xs:string">
      <xs:length value="3"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="foo" type="airportCode"/>
</xs:schema>
```

With this valid document:

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

7.7.3 Complex types

It is also possible to constrain structures and the element/attribute/-text hierarchy with complex types applying to element nodes. There are four main kinds of complex types:

- complex content: there can be nested elements, but there can be no text nodes as direct children.
- simple content: there are no nested elements: just text, but attributes are also possible.
- empty content: there are neither nested elements nor text, but attributes are also possible.
- mixed content: there can be nested elements and it can be intermixed with text as well.

This is an example of complex content:

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

which is valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="complex">
    <xs:sequence>
      <xs:element
          name="twotofour"
          type="xs:string"
          minOccurs="2"
          maxOccurs="4"/>
      <xs:element
          name="zeroorone"
          type="xs:boolean"
          minOccurs="0"
          maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="foo" type="complex"/>
</xs:schema>
```

Note how children elements can be repeated, and the number of occurrences can be constrained to some interval with minOccurs and maxOccurs attributes in the schema. Of course, this all works recursively, i.e., the nested elements can also have complex types with complex content and so on (even though in this example they have simple types).

This is an example of simple content:

```
<foo country="Switzerland">2014-12-02</foo>-
```

which is valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="dateCountry">
    <xs:simpleContent>
      <xs:extension base="xs:date">
        <xs:attribute name="country" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Note how a complex type with simple content is defined as the extension of a simple type, adding one or more attributes to it. If there are no attributes, of course, there is no need to bother with a complex type: a simple type does the trick as shown before.

This is an example of empty content:

```
<foo/>
```

which is valid against this schema:

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

As you can see, empty content is "boring", in that it is defined just like complex content, but with no nested elements at all (attributes, though, can absolutely be added).

Finally, this is an example of mixed content:

```
<foo>Some text and some <b>bold</b> text.</foo>
```

which is valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="mixedContent" mixed="true">
    <xs:sequence>
      <xs:element
          name="b"
          type="xs:string"
          minOccurs="0"
          maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Mixed content is also "boring" to define: all it takes is a structure just like complex content, plus an extra attribute in the xs:complexType declaration called "mixed" and set to true.

7.7.4 Attribute declarations

Finally, all types of content can additionally contain attributes. Attributes always have a simple type. An example with empty content involving one attribute:

```
<foo country="Switzerland"/>
```

which is valid against:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType-name="withAttribute">
    <xs:sequence/>
    <xs:attribute name="country"
        type="xs:string"
        default="Switzerland"/>
   </xs:complexType>
 </xs:schema>
```

The default attribute of the attribute declaration will automatically add an attribute with the corresponding name and specified value in memory in case it was missing in the original instance. This works just like in JSound.

7.7.5 Anonymous types

Finally, it is not mandatory to give a name to all types. It is possible, instead of the type attribute of an element or attribute declaration, to instead nest a type declaration with no name attribute:

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

Be careful: if there is neither a type attribute nor a nested type declaration, then anything is allowed!

7.7.6 Miscellaneous

XML Schema has many more features than described here: primary keys, constraints, support for namespaces, etc.

Finally, XML Schema documents are themselves XML documents, and can thus be validated against a "schema or schemas", itself written as an XML Schema. This schema has the wonderful property of being valid against itself, which will delight aficionados of Douglas Hofstadter's Pulitzer-prize-winning Gödel, Escher, Bach book.

7.8 Data frames

7.8.1 Heterogeneous, nested datasets

Now that we have defined data models for JSON and XML, and added a mechanism (schema validation) to enforce additional constraint on collections of JSON objects or of XML documents, we can take a step back and look at the bigger picture.

First, it should now be clear to you that, in both XML and JSON, datasets are simply collections of trees. If we now consider the particular case of JSON, which is growing in popularity, we can express a collection of JSON trees formally as a list of maps.

Now, we have already seen lists of maps before: relational tables are lists of maps, too, at least in the list semantics, as opposed to the set or bag semantics of the relational algebra. But in the context of relational tables, we called these maps tuples, or rows. The reason is that relational tables are not *any* lists of maps: they have constraints, as we explained in Chapter 2. These constraints include atomic integrity, relational integrity, and domain integrity.

Casually explained, atomic integrity means that relational tables are *flat*.

ID	Name	Living
1	Einstein	false
2	Penrose	true
3	Turing	false
4	Dean	true

A counterexample of flatness (or of atomic integrity) is the presence of nested tables:

ID	Profession	People
1	Physicist	Name: Einstein, false; Penrose, true (Living)
2	Computer Scientist	Name: Turing, false; Dean, true (Living)

Casually explained, relational integrity and domain integrity means that relational tables are *homogeneous*.

ID	Name	Living
1	Einstein	false
2	Penrose	true
3	Turing	false
4	Dean	true

A counterexample of homogeneity (or of relational and domain integrity) is the presence of missing or additional columns, and of values with different types in the same columns:

ID	Name	Living	Profession
1	Einstein	false	Physicist
2	Penrose	true	CS
3	Turing		
4	Dean	true	

Finally, we can even provide a counterexample of a dataset that is neither flat, nor homogeneous: it is both nested and heterogeneous.

ID	Profession	People	Comment
1	Physicist	Name: Einstein, Living: false Penrose	
2	Computer Scientist	Name: Turing, Living: Name: Dean, Living: true	They rock

The beauty of the JSON data model is that, like the relational model and the CSV syntax, it supports nested, heterogeneous datasets, while also supporting *as a particular case* flat, homogeneous datasets like so:

```
{"ID":1, "Name": "Einstein", "Living" : false}
{"ID":2, "Name": "Penrose", "Living" : true}
{"ID":3, "Name": "Turing", "Living" : false}
{"ID":4, "Name": "Dean", "Living" : true}
```

This is now an example of nested (but homogeneous) collection in JSON syntax, corresponding to the nested visual above:

```
{
  "ID":1,
  "Profession": "Physicist",
  "People": [
    {"Name": "Einstein", "Living" : false},
    {"Name": "Penrose", "Living" : true}
  ]
}
{
  "ID":2,
  "Profession": "Computer Scientist",
  "People": [
    {"Name": "Turing", "Living" : false},
    {"Name": "Dean", "Living" : true}
  ]
}
```

This is now an example of heterogeneous (but flat) collection in JSON syntax, corresponding to the heterogeneous visual above:

```
{
  "ID":1,
  "Name": "Einstein",
  "Living" : false,
  "Profession" : "Physicist"
}
{
  "ID":2,
  "Name": "Penrose",
  "Living" : true,
  "Profession" : "CS"
}
{
  "ID":3,
  "Name": "Turing"
}
{
  "ID":4,
  "Name": "Dean",
  "Living" : true
}
```

And finally, this is an example of heterogeneous and nested collection in JSON syntax, corresponding to the heterogeneous and nested visual above:

```
{
  "ID":1,
  "Profession": "Physicist",
  "People": [
    {"Name": "Einstein", "Living" : false},
    "Penrose"
  ]
}
{
  "ID":2,
  "Profession": "Computer Scientist",
  "People": [
    {"Name": "Turing"},
    {"Name": "Dean", "Living" : true}
  ],
  "Comment": "They rock"
}
```

For completeness, this shows how the same can be expressed in XML. However, XML is more powerful as we explained before, so that

in the remainder of the course, we will tend to focus much more on JSON.

```
<professions>
  <profession id="1">
    <name>physicist</name>
    <persons>
      <person>
        <name>Einstein</name>
        <living>false</living>
      </person>
      Penrose
    </persons>
  </profession>
  <profession id="2">
  <name>Computer Scientist</name>
  <persons>
    <person>
      <name>Turing</name>
    </person>
    <person>
      <name>Dean</name>
      <living>true</living>
    </person>
  </persons>
  <comment>They rock</comment>
  </profession>
<professions>
```

7.8.2 Dataframe visuals

There is a particular subclass of semi-structured datasets that are very interesting: valid datasets, which are collections of JSON objects valid against a common schema, with some requirements on the considered schemas.

The datasets belonging to this particular subclass are called data frames, or dataframes[3]

Specifically, for the dataset to qualify as a data frame, firstly, we forbid schemas that allow for open object types, that is, schemas must disallow any additional attributes, and, secondly, we forbid schemas

[3]Unlike German, compound words in English are normally space-separated or dash-separated. However, when a compound word is used very often, at some point, people start pasting them together in the German style: doormat, today, keyboard, etc. In the case of dataframes though, this might also be due to the CamelCase used in many libraries: DataFrames.

that allow for object or array values to be too permissive and allow *any* values, that is, we ask that schemas require specific types such as integers, strings, dates, objects representing a person, arrays of binaries, etc. We, however, include schemas that allow for null values and/or absent values.

Under the above conditions, we call the collection of objects a data frame. It should be immediate to the reader that relational tables are data frames, while data frames are not necessarily relational tables: data frames can be (and are often) nested, but they are still relatively homogeneous to some extent. Relatively, because schemas can still allow for a value to be missing.

Data frames have the nice property that they can be drawn visually in structures that look like generalized relational tables and that look a bit nicer and more structured that the previous visuals with nested tables. Further, JSound compact schemas provides a natural syntax for constraining data frames, because object types in this syntax are always closed, and it allows for requiring or not values, and for including or not null values. Thus, we can now give a few examples of JSound schemas and their corresponding visuals. Let us start with the example of a "flat" JSound schema:

```
{
  "ID" : "integer",
  "Name" : "string",
  "Living" : "boolean"
}
```

This schema could also be described as a SQL CREATE TABLE statement just as well. In fact, for this use case, a relational database might make more sense than a datalake altogether. Concretely, this means the data can be drawn as a table, like so:

ID	Name	Living
1	Einstein	false
2	Penrose	true
3	Turing	false
4	Dean	true

But things get interesting if we denormalize and define one of the fields to be an array of strings, for example, like so:

```
{
  "ID" : "integer",
  "Name" : [ "string" ],
  "Living" : "boolean"
}
```

Then, the Data frame visual becomes:

ID	Name	Living
1	Albert	false
	Einstein	
2	Penrose	true
3	Alan	false
	Turing	
4	Dean	true

Thus, Data frames are a generalization of (normalized) relational tables allowing for (organized and structured) nestedness.

Data frames also can have nested objects, as described by the following schema:

```
{
  "ID" : "integer",
  "Name" : {
    "First" : "string",
    "Last" : "string"
  },
  "Living" : "boolean"
}
```

A valid instance can then be drawn like so:

ID	Name		Living
	First	Last	
1	Albert	Einstein	false
2	Roger	Penrose	true
3	Alan	Turing	false
4	Jeff	Dean	true

Finally, a very common use case modelling "tables in tables" involves objects nested in arrays, like so:

```
{
  "ID" : "integer",
  "Who" : [
    {
      "Name" : "string",
      "Type" : "string"
    }
  ],
  "Living" : "boolean"
}
```

Leading to the following visual:

ID	Who		Living
	Name	Type	
1	Albert	first	false
	Einstein	last	
2	Penrose	last	true
3	Alan	first	false
	Turing	last	
4	Dean	last	true

Note that the former visual could also match the following different, but less natural, schema:

```
{
  "ID" : "integer",
  "Who" : {
      "Name" : [ "string" ],
      "Type" : [ "string" ]
  },
  "Living" : "boolean"
}
```

This schema structure is, however, much less common even though it can be used by savvy users to optimize the layout under specific circumstances (an example where this can be seen is in high-energy physics datasets found at CERN). Beware in particular that this alternate schema structure does not enforce that there must the same number of items in each array.

7.9 Data formats

Having now seen models for XML and in particular JSON, and having discovered data frames, it should be come clear to the reader that the data structures in memory have nothing to do with the original syntax any more.

In fact, if the data is structured as a (valid) data frame, then there are many, many different formats that it can be stored in, and in a way that is much more efficient than JSON. These formats are highly optimized and typically stored in binary form, for example Parquet, Avro, Root, Google's protocol buffers, etc. If you see a JSON dataset that you are able to validate against a (e.g., JSound) schema that is "data-frame friendly", i.e., such that the data can be visualized as shown in the previous section, then you are highly encouraged to immediately build this schema, and convert the dataset to, say, Parquet. This gives you two immediate advantages:

- space efficiency: the file will be considerably smaller, meaning you can fit much more data even on your local laptop and it is also faster to transfer to and from the cloud to share with others.

- performance efficiency: the smaller binary file will be much faster to read from disk when you write queries. In fact, many optimizers are able to skip entire sections of the data based on the query (projecting away a column, etc), making it even faster than it already was.

Also, at the risk of repeating myself, if the schema does not involve any nested structures and contains only closed object types, and the data is less than a Terabyte, then a relational database with SQL queries is very often the best way to go.

Why is it possible to store the data more efficiently when it is valid and data-frame-friendly? One important reason is that the schema can be stored as a header in the binary format, and the data can be stored without repeating the fields in every record (as is done in textual JSON). Furthermore, there are plenty of techniques that allow compressing homogeneous sequences of values and, because of constant size, perform direct lookups (e.g., directly return the field "Name" of the 100th record) without having to scan through all the dataset. In fact, Parquet is called that way because the data is stored in a columnar fashion, grouping all the values (across objects) together when they are associated with the same key, which might remind you of the parquet floor of your living room.

Generally, data formats can be classified along three dimensions:

- whether they require validity against a data frame compatible schema (Parquet, protocol buffers, etc.) or not (JSON, XML, YAML, etc.).
- whether they allow for nestedness (CSV) or not (Parquet, etc.).
- whether they are textual (CSV, XML, JSON, etc.) or binary (Parquet, etc.), even though typically, formats that require a schema will be binary because of the "performance free lunch."

We finish this chapter with good news: we will not give more details about the formats. This is because if you look the documentation (take Parquet as an exercise for example), you will immediately notice that the modelling will be very close to what we studied in this chapter: you only need to learn the terminology specific to the format (e.g., int64 instead of long, UTF8 instead of string, LIST instead of array, etc.). For Swiss readers, going from JSound+JSON to Parquet or Avro comes down to speaking Zurich German and going to Basel or to Bern and adapting to the different dialect: similar grammar, with some different words. XML then comes down to going to Wallis, a trip really worth it because it has stunning landscapes, too.

Chapter 8

Massive Parallel Processing

Now that we know how and where to store datasets; and we know how to model nested, heterogeneous datasets; and we know how to validate them; and we know how to build data frames (when applicable); and we know how they can be stored in various optimized binary formats, it is time to finally move on to the truly awesome part of Big Data: actually processing gigantic amounts of data.

8.1 Counting cats

As I would like for this textbook to be used to also teach in other universities, it needs to involve cats. This is where Erwin, who lived in Zurich, comes into play. Erwin has hundreds of cats of ten various kinds, specifically he has:

- Persian cats
- Siamese cats
- Bengal cats
- Scottish folds
- American shorthairs
- Maine coons
- British shorthairs
- Sphynx cats
- Ragdolls
- Norwegian forest cats

However, Erwin lost track of counts, and would like to know how many cats of each kind he has with him. He could, of course, count them one by one, but Erwin thinks there has to be a better way.

Fortunately, his 22 grandchildren are visiting him this week-end, and Erwin has a plan.

First, he will distribute the cats across all 17 rooms of his large mansion. Then, he will send 17 of his grandchildren to each room, one per room, and ask them to count the cats, by kind, of the room they are assigned to, and to then write the counts of each kind on a piece of paper. This is an example for one room:

Persian cats	12
Siamese cats	23
Bengal cats	14
Scottish folds	23
American shorthairs	36
Maine coons	3
British shorthairs	5
Sphynx cats	54
Ragdolls	2
Norwegian forest cats	63

Meanwhile, the other 5 grandchildren have been each assigned several kinds of cats. Each kind of cat will be processed by one, and exactly one, of these 5 grandchildren. Mathias, who is the oldest grandchild, will process 3 kinds of cats. Leane, Anna and Elliot will process 2 kinds each, and Cleo, the youngest one, will process 1 kind of cats, making it 10 in total.

Persian cats	Mathias
Siamese cats	Anna
Bengal cats	Cleo
Scottish folds	Mathias
American shorthairs	Leane
Maine coons	Mathias
British shorthairs	Elliot
Sphynx cats	Leane
Ragdolls	Anna
Norwegian forest cats	Elliot

By the time this is done, the 17 grandchildren are coming back with their sheets of paper. Erwin gives them scissors and asks them to divide these sheets of paper into stripes: each stripe with a kind of cat and its count. Then, the group of 17 and the group of 5 start walking in all directions, where each stripe of paper is handed over by the grandchild

(from the group of 17) who created it, to the one supposed to process it (from the group of 5).

Needless to say, the mansion is quite noisy at that time and it takes quite a while until all stripes of paper are finally in the hands of Mathias, Leane, Anna, Elliot, and Cleo. Cleo received 16 stripes with Bengal cats counts; indeed, in one of the 17 rooms, there was no Bengal cats at all. Mathias received 17 stripes of Persian cat counts, 15 stripes of Scottish fold counts, and 14 stripes of Maine coons, and so on.

Now, Erwin asks the 5 grandchildren to add the counts for each kind of cat. Cleo has one big addition to do with all the numbers on her stripes:

Bengal cats	12
Bengal cats	13
Bengal cats	23
Bengal cats	3
Bengal cats	1
Bengal cats	6
Bengal cats	13
Bengal cats	6
Bengal cats	14
Bengal cats	9
Bengal cats	7
Bengal cats	7
Bengal cats	9
Bengal cats	11
Bengal cats	12
Bengal cats	11

Cleo proudly creates a new stripe with her grand total:

Bengal cats	157

Mathias, Anna, Elliot and Leane do the same and give back their newly created stripes to Erwin, who now has the overview of his cat counts:

Persian cats	412
Siamese cats	233
Bengal cats	157
Scottish folds	233
American shorthairs	36
Maine coons	351
British shorthairs	153
Sphynx cats	236
Ragdolls	139
Norwegian forest cats	176

What did just happen? This is simply an instance of a MapReduce computation. The first phase, with the 17 separate rooms, is called the Map phase. Then came the Shuffle phase, when the stripes were handed over. And finally came the Reduce phase, with the computation of the final grand totals.

8.2 Patterns of large-scale query processing

We saw that the data we want to query can take many forms. First, it can be billions of lines of text (this is from Sherlock Holmes, which came into public domain a few years ago, which in turns explains why there are so many movies and series about Sherlock Holmes these days):

```
In the year 1878 I took my degree of Doctor of Medicine of
the University of London, and proceeded to Netley to go
through the course prescribed for surgeons in the army.
Having completed my studies there, I was duly attached
to the Fifth Northumberland Fusiliers as Assistant Surgeon.
The regiment was stationed in India at the time, and before
I could join it, the second Afghan war had broken out. On
landing at Bombay, I learned that my corps had advanced
through the passes, and was already deep in the enemy's
country. I followed, however, with many other officers who
were in the same situation as myself, and succeeded in
reaching Candahar in safety, where I found my regiment,
and at once entered upon my new duties.
```

It can also be plenty of CSV lines (these will likely be familiar to residents of Zurich):

```
Year,Date,Duration,Guest
2022,2019-04-25,00:37:59,UR
2021,2019-04-19,00:12:57,UR
2020,NULL,NULL,NULL
2019,2019-04-08,00:17:44,Strassburg
2018,2018-04-16,00:20:31,BS
2017,2017-04-24,00:09:56,GL
2016,2016-04-18,00:43:34,LU
...
```

Or maybe some use-case-specific textual format (common for weather data, for example):

```
20222019-04-2500:37:59UR
20212019-04-1900:12:57UR
20200000-00-0000:00:0000
```

```
20192019-04-0800:17:44FR
20182018-04-1600:20:31BS
20172017-04-2400:09:56GL
20162016-04-1800:43:34LU
...
```

Such a format can also be made of a key-value pattern, here, with key and value separated by a space character:

```
2022 00:37:59.000000
2021 00:12:57.000000
2020 00:00:00.000000
2019 00:17:44.000000
2018 00:20:31.000000
2017 00:09:56.000000
2016 00:43:34.000000
...
```

A popular format that is more machine readable is the JSON Lines format, with one JSON object per line (to fit it on this page, we had to write them on two lines, but in the real file, it would be a single line). Note that it can be heterogeneous:

```
{ "year": 2022, "date": "2022-04-25",
    "duration": "00:37:59", "canton": "UR" }
{ "year": 2021, "date": "2021-04-19",
    "duration": "00:12:57", "canton": "UR" }
{ "year": 2020, "duration": null }
{ "year": 2019, "date": "2019-04-08",
    "duration": "00:17:44", "city": "Strasbourg" }
{ "year": 2018, "date": "2018-04-16",
    "duration": "00:20:31", "canton": "BS" }
{ "year": 2017, "date": "2017-04-24",
    "duration": "00:09:56", "canton": "GL" }
{ "year": 2016, "date": "2016-04-18",
    "duration": "00:43:34", "canton": "LU" }
```

Some other formats (e.g., Parquet, ...) can be binary, as we saw in Chapter 7. Personally, I am not able to decode it directly, but luckily computers can:

```
01010010101010101010010101101010010101010101
00101010101010101101001111010100010110100101O
01010010101010101010010101101010010101010101
00101010101010101101001111010100010110100101O
01010010101010101010010101101010010101010101
```

```
00101010101010101101001111010100010110100101 0
01010010101010101010010101101010010101010101
00101010101010101101001111010100010110100101 0
01010010101010101010010101101010010101010101
00101010101010101101001111010100010110100101 0
01010010101010101010010101101010010101010101
00101010101010101101001111010100010110100101 0
01010010101010101010010101101010010101010101
00101010101010101101001111010100010110100101 0
```

We also encountered HFiles in Chapter 6, which are lists of key-value pairs. In fact, Hadoop has another such kind of key-value-based format called Sequence File, which is simply a list of key-values, but not necessarily sorted by key (although ordered) and with keys not necessarily unique. This is because, as we will see shortly, this format is used for intermediate data generated with MapReduce.

Sequence Files also have a flavour in which values are compressed, and another flavour in which their blocks (akin to what we called HBlocks in HFiles) are compressed.

How do we store Petabytes of data on online cloud storage, such as S3 or Azure blob storage, where the maximum size of a file is limited? Simply by spreading the data over many files. It is very common to have datasets lying in a directory spread over 100 or 1,000 files. Often, these files are named incrementally: part-0001, part-0002, etc. These files are often also called "shards" of the dataset.

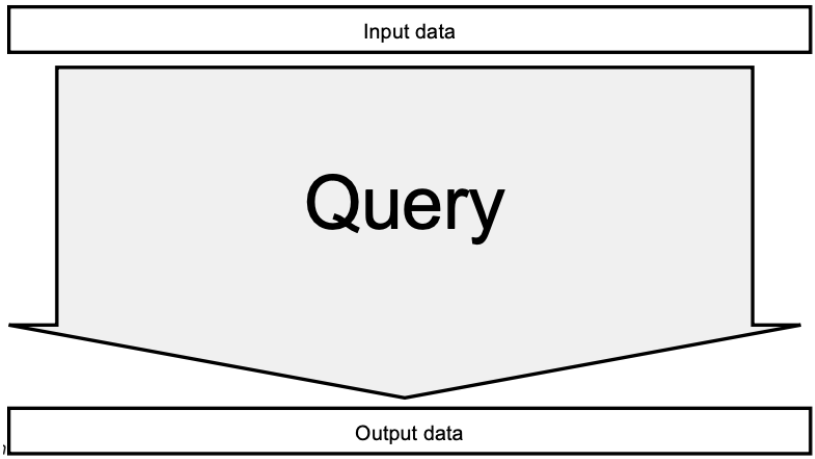
What about HDFS? Technically, HDFS would make it possible to have a gigantic, single file, automatically partitioned into blocks. However, also for HDFS, it is common to have a pattern with a directory containing many files named incrementally. The size of these files is typically higher than that of a block, for example 10 GB files.

Note that the size of the files do not constrain parallelism: with HDFS, even with 10 GB files, the 128 MB blocks within the same file can still be processed in parallel. S3 is also compatible with infra-file parallelism. There are several practical motivations for the many-files pattern even in HDFS:

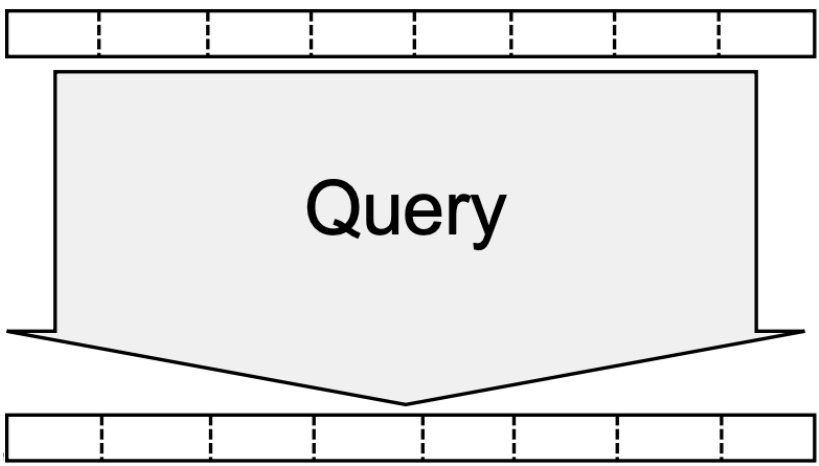
- As we will see, it is much more convenient to output several shards from a framework like MapReduce than it is to create a single, big final file (which would require a way to concatenate it properly).
- It is considerably easier to download or upload datasets that are stored as many files. This is because network issues happen once in a while, and you can simply retry with only the files that failed. With a single 10 PB file, the time it takes is so long that it is extremely likely that a network issue will force you to start it all over and over.

Of course, the data does not need to be stored directly as files on cloud storage or a distributed file system: it could be stored as a relational table in a relational database, or in a wide column store like HBase. In fact, it is quite common to store XML, HTML or JSON data in HBase cells. MapReduce can also process data on these higher-level stores.

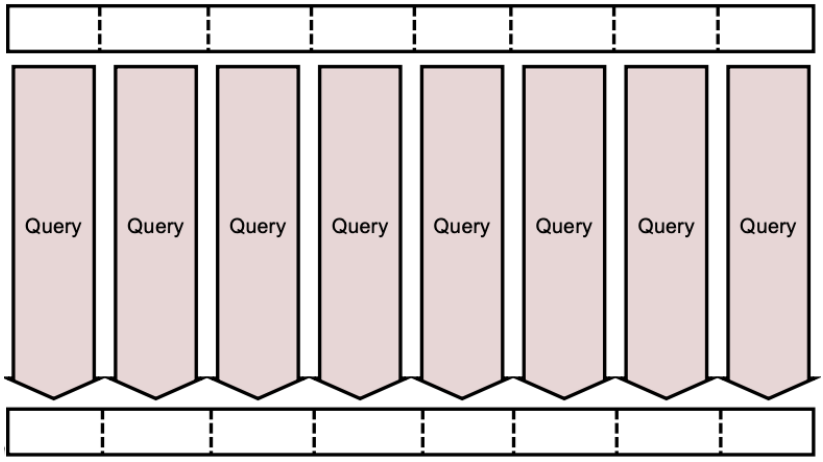
Now that we have the data, how does querying it look like? On the very high-level, it converts some input to some output like so:



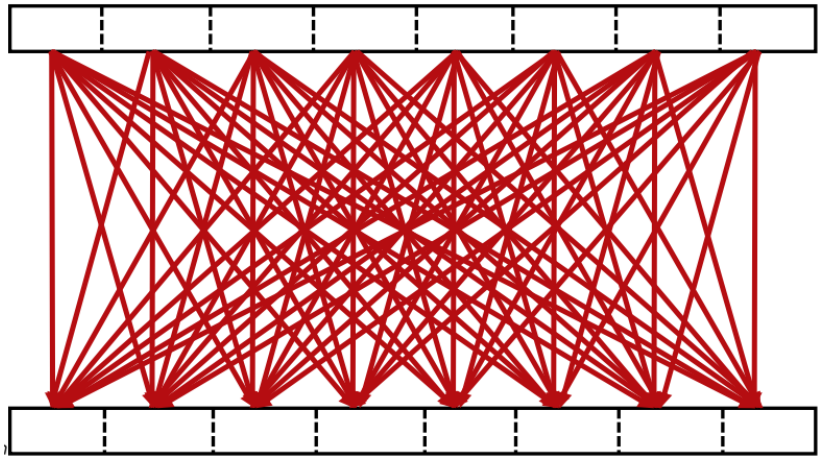
However, because the underlying storage supports parallelism (via shards, blocks, regions, etc), the input as well as the output are partitioned:



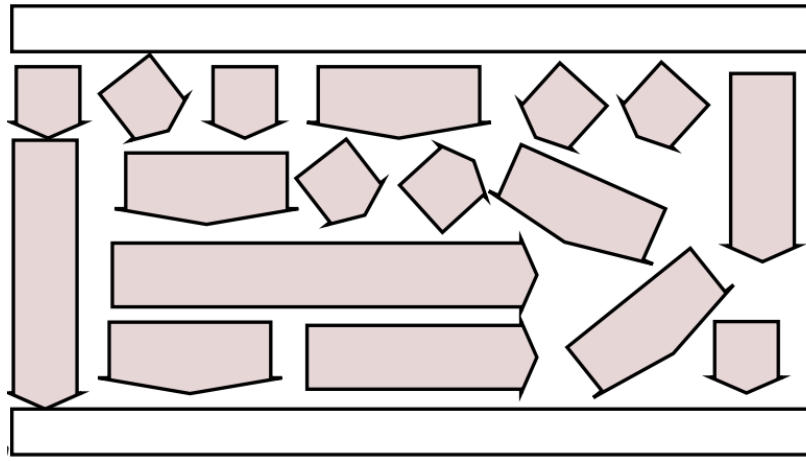
So far, that does not really help us. But what if we are lucky and the query can, in fact, be reexpressed equivalently to simply map every input partition to an output partition, like so?



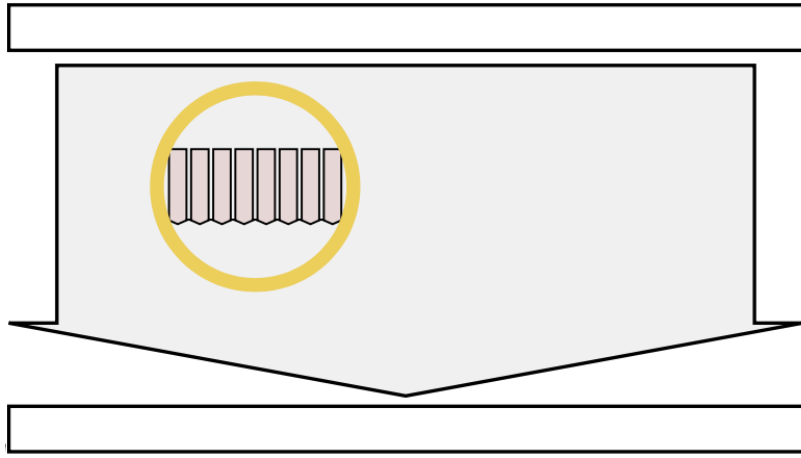
Sadly, in reality, this is what you might find instead something that looks more like spaghetti:



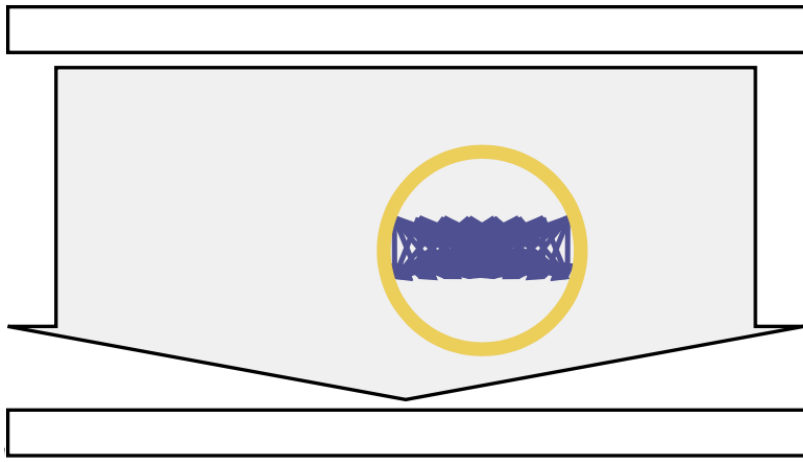
Fortunately, what happens often is somewhere in the middle, with some data flow patterns:



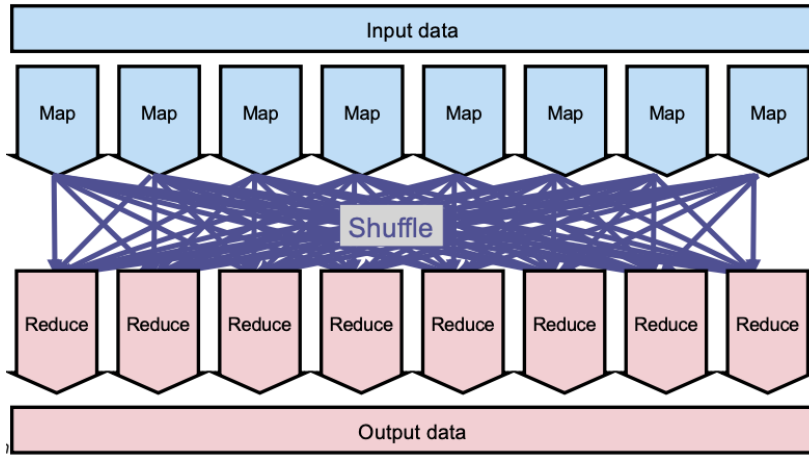
Some places have data flowing in parallel (map-like):



While some others are more spaghetti-y (shuffle-like):



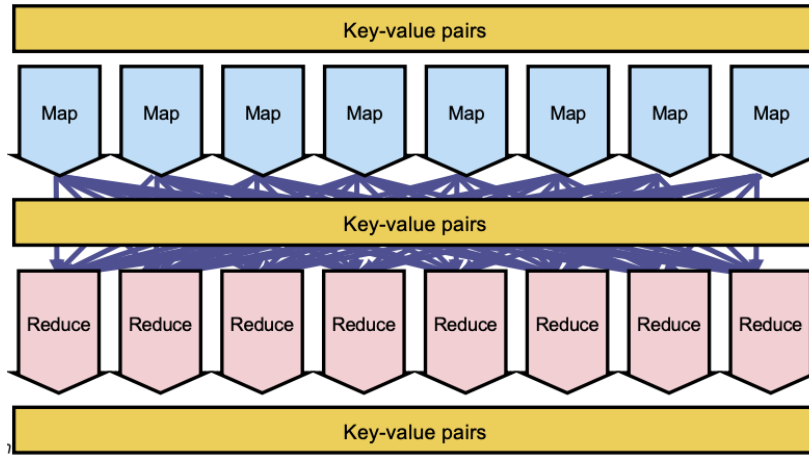
This is the motivation behind the standard MapReduce pattern: a map-like phase on the entire input, then a shuffle phase on the intermediate data, then another map-like phase (called reduce) producing the entire output:



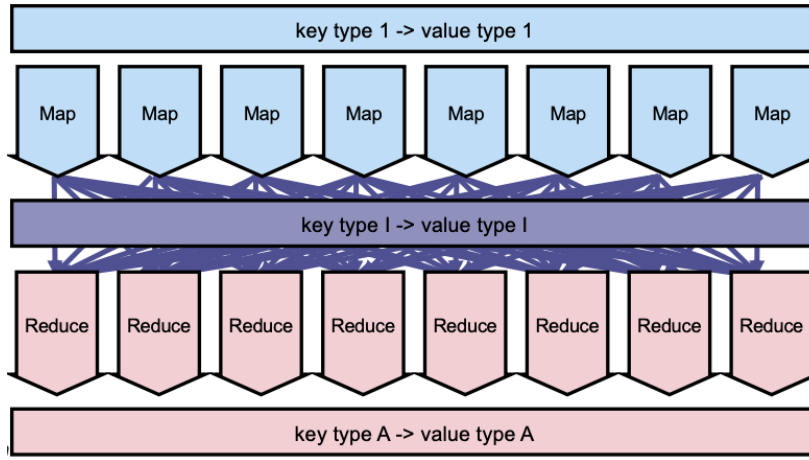
This seems restrictive, but in reality, it is extremely powerful and generic to accommodate for many use cases, in particular if you chain MapReduce jobs with each other.

8.3 MapReduce model

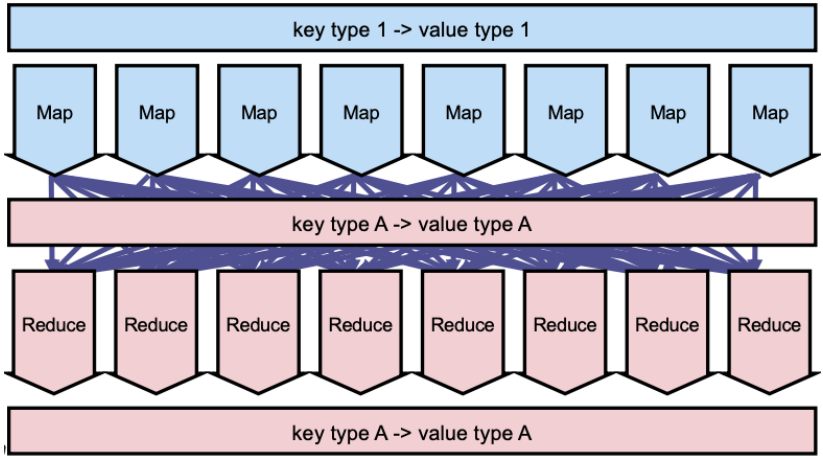
In MapReduce, the input data, intermediate data, and output data are all made of a large collection of key-value pairs (with the keys not necessarily unique, and not necessarily sorted by key):



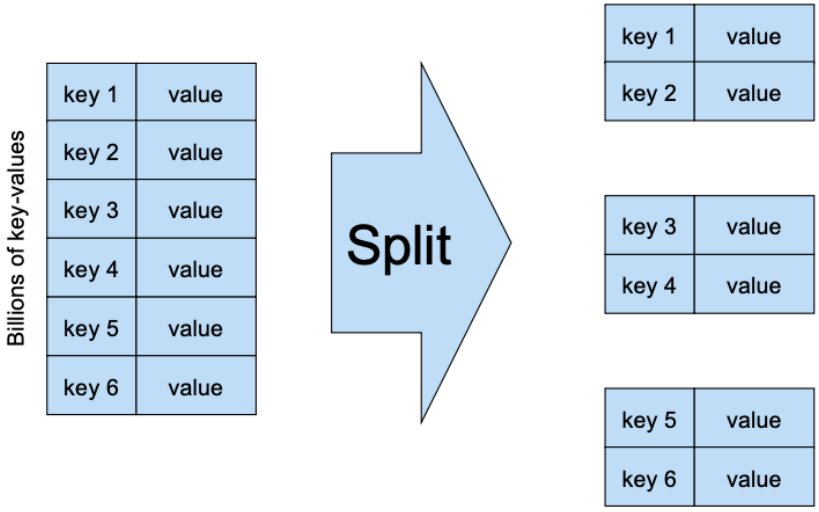
The types of the keys and values are known at compile-time (statically), and they do not need to be the same across all three collections:



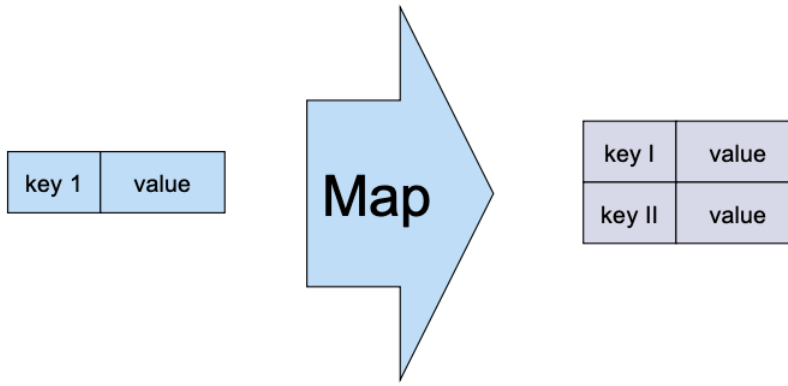
In practice, however, it is quite common that the type of the intermediate key-value pairs is the same as that of the output key-value pairs:



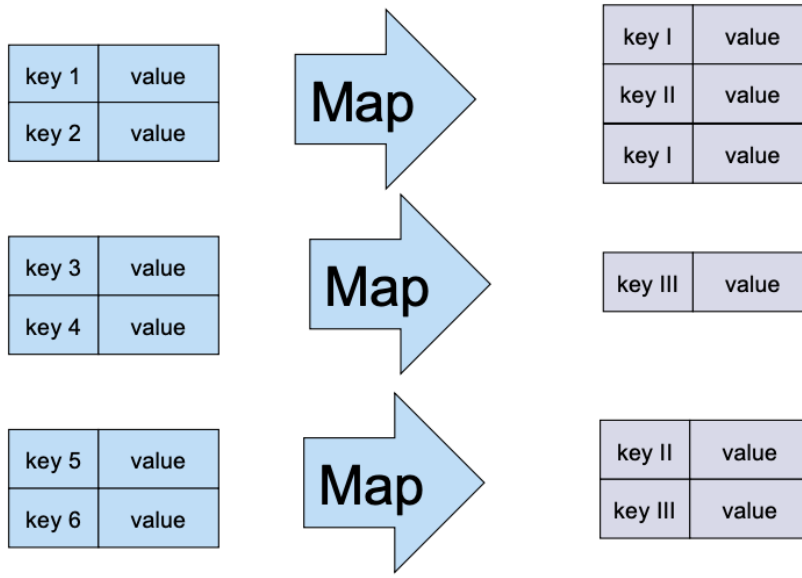
Let us now walk through a MapReduce query, but on the logical level for now. Everything starts with partitioning the input. MapReduce calls the partitions "splits":



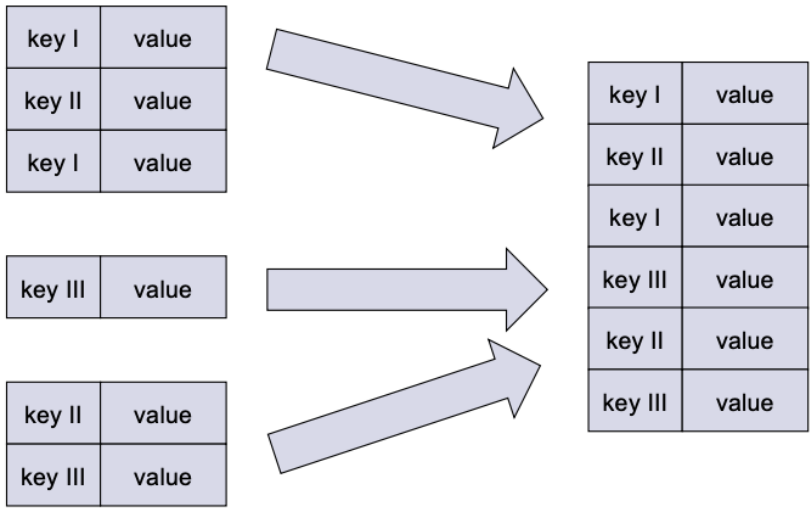
Each key-value will be fed into a map function:



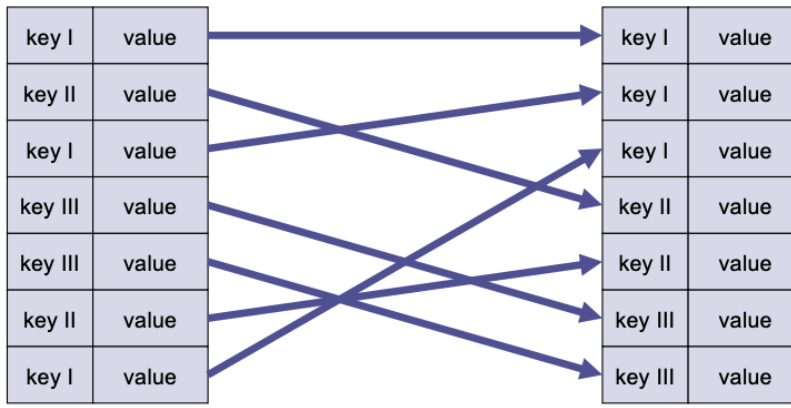
However, as you can imagine, it would be extremely inefficient to do it pair by pair, thus, the map function is called in batches, on each split:



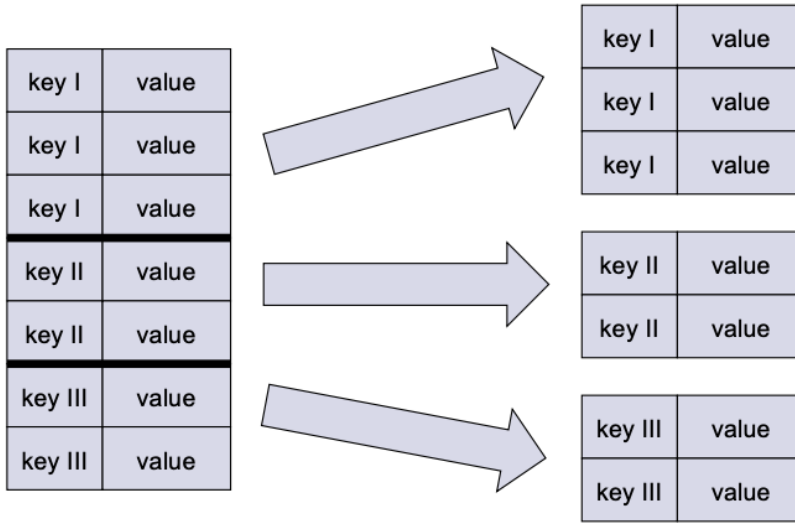
The intermediate pairs can be seen, logically, as a single big collection:



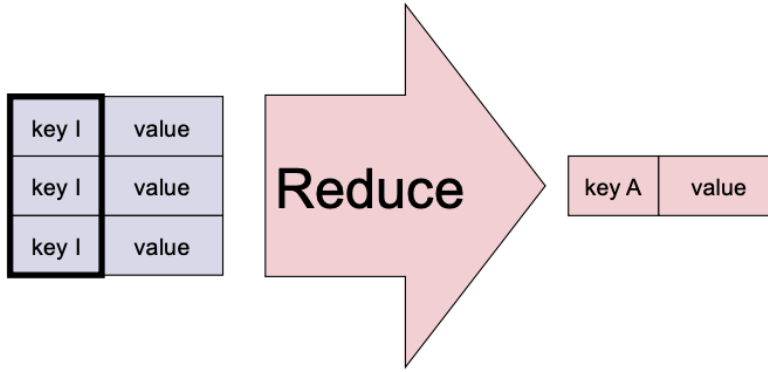
This collection can be thought of as logically sorted:



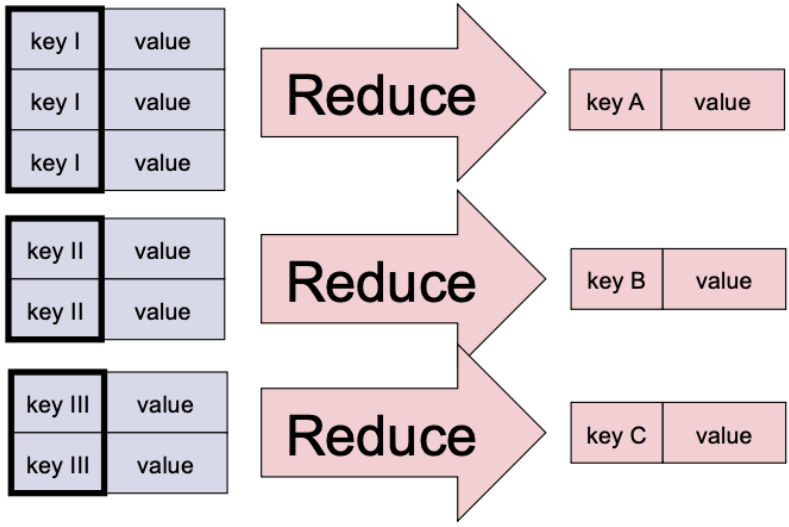
And then partitioned again, making sure the pairs with the same key are always in the same partition (but a partition can have pairs with several keys):



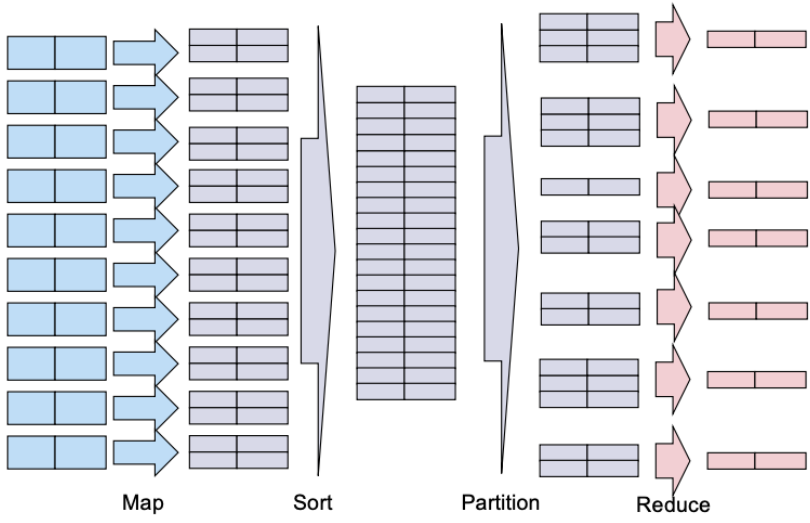
The reduce function must then be called, for every unique key, on all the pairs with that key, outputting zero, one or more output pairs (here, we just drew one, which is the most common case):



Just like the map function, the reduce function is called in batches, on each intermediate partition (multiple calls, one per unique key):



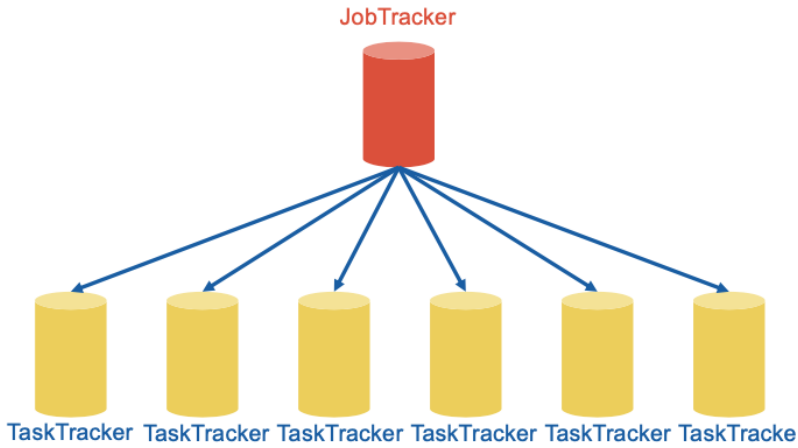
If we recap, this is how the entire process looks like on the high (and logical) level:



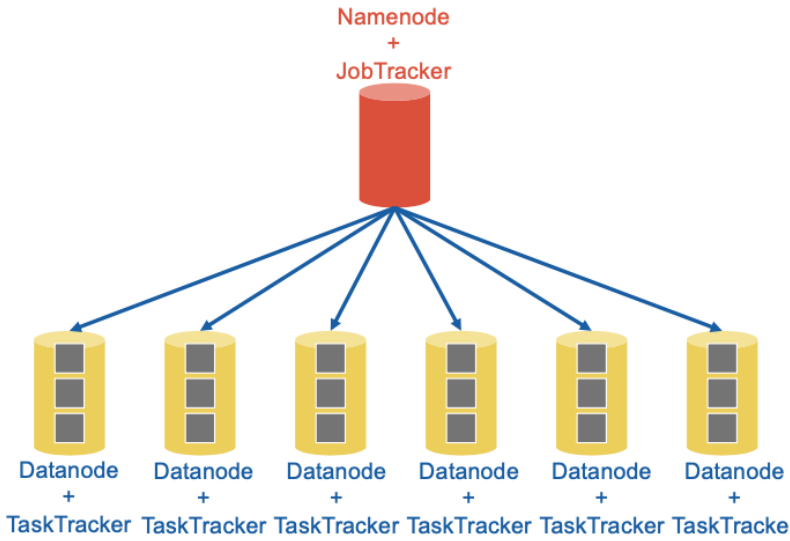
8.4 MapReduce architecture

MapReduce can read its input from many places as we saw: a cloud storage service, HDFS, a wide column store, etc. The data can be Petabyte-sized, with 1,000 of machines involved.

On a cluster, the architecture is centralized, just like for HDFS and HBase. In the original version of MapReduce, the main node is called JobTracker, and the worker nodes are called TaskTrackers.



In fact, the JobTracker typically runs on the same machine as the NameNode (and HMaster) and the TaskTrackers on the same machines as the DataNodes (and RegionServers). This is called "bring the query to the data." If using HDFS, then most of the time, for the map phase, things will be orchestrated in such a way that there is a replica of the block corresponding to the split on the same machine (since it is also a DataNode...), meaning that it is a local read and not a network connection.



As the map phase progresses, there is a risk that the memory becomes full. But we have since this before with HBase: the intermediate pairs on that machine are then sorted by key and flushed to the disk to a Sequence File. And as more flushes happen, these Sequence Files can be compacted to less of them, very similarly to HBase's Log-Structured Merge Trees.

When the map phase is over, each TaskTracker runs an HTTP server listening for connections, so that they can connect to each other and ship the intermediate data over to create the intermediate partitions ensuring that the same keys are on the same machines (we will look at this again with more precise terminology later in this chapter). This is the phase called shuffling. Then, the reduce phase can start.

Note that shuffling can start before the map phase is over, but the reduce phase can only start after the map phase is over (why?).

When the reduce phase is completed, each output partition will be output to a shard (as we saw, a file named incrementally) in the output destination (HDFS, S3, etc) and in the desired format.

8.5 MapReduce input and output formats

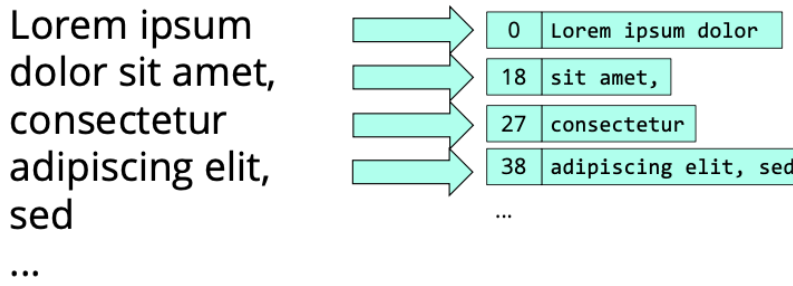
Let us know get back to the input received by and the output produced by MapReduce. We looked at various examples of textual and binary formats in Section 8.2, and saw in particular that MapReduce can read its input from files lying in a data lake as well as directly from a

database system such as HBase or a relational database management system.

As the reader will have noticed, MapReduce only reads and writes lists of key-value pairs, where keys may be duplicates and need not appear in order. However, the inputs we considered are not key-value pairs. So we need an additional mechanism that allows MapReduce to interpret this input as key-value pairs.

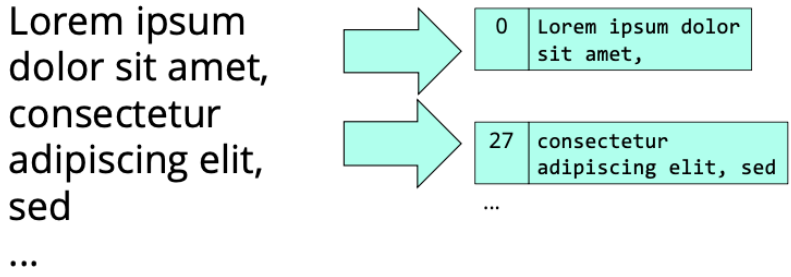
For tables, whereas relational or in a wide column stores, this is relatively easy: indeed, tables have primary keys, consisting of either a single column or multiple columns. Thus, each tuple can be interpreted as a key-value pair, where the key is the (sub)tuple containing all the values associated with columns that are part of the primary key, while the value is the (sub)tuple containing the values associated with all other columns.

Let us thus focus on files. How do we read a (possibly huge) text file as a list of key-value pairs? The most natural way to do so is to turn each line of text in a key value pair[1]: the value is the string corresponding to the entire line, while the value is an integer that expresses the position (as a number of characters), or offset, at which the line starts in the current file being read, like so:

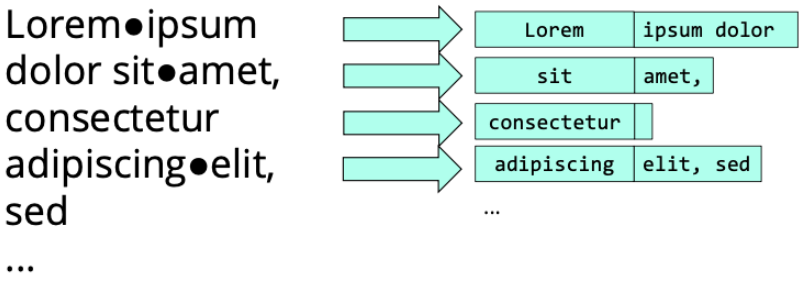


A small variation consists in reading N lines at a time, mapping them to a single key-value:

[1]This means we take \n, or \r \n as a special separator, depending on the operating system.



Another variation consists of treating a character (picked by the user) specially, as the separator between the key and the value:



Since MapReduce is not aware of JSON or any syntax, a JSON Line file will be read as text as explained above, and it is the responsibility of the user to parse this text to JSON. This is, of course, not very convenient, as it means MapReduce pushes the burden of doing so to the user, but we will see in subsequent chapters that there are additional layers that will come on top of the technology stack, which free the user from having to deal with these details.

Sequence Files are easier to handle: since they already contain lists of key-value pairs in a format native to MapReduce, their interpretation is straightforward. In fact, intermediate data spilled to disk will be written and read back in this format.

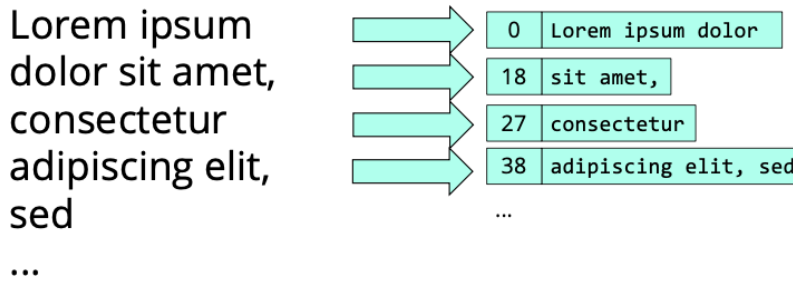
8.6 A few examples

Let us use a concrete example and count the number of occurences of each word in this document

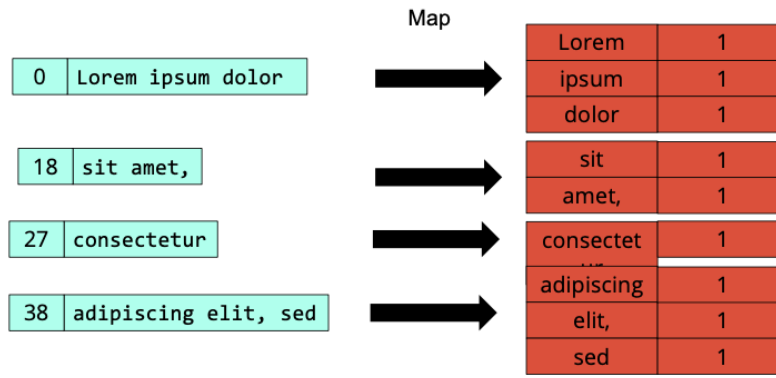
```
Lorem ipsum
dolor sit amet,
```

```
consectetur adipiscing elit,
sed
...
```

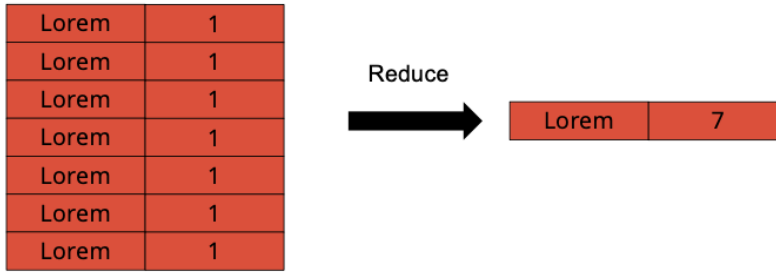
We already saw how this is interpreted as key-value pairs:



We can count the words within each line similar to our motivation example with the cats, by mapping each line key-value to several key-values, one per word and with a count of 1. This gives us our map function:

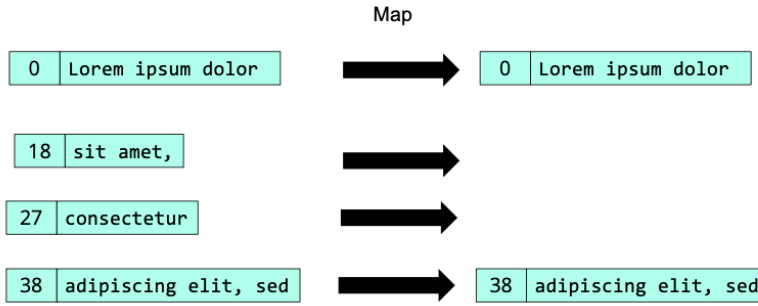


The reduce function is then obtained by summing the values with the same key, and keeping the same key:

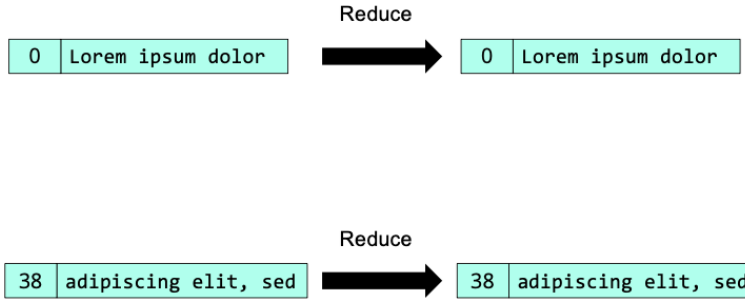


The output will then consist of a list of unique key-values, with one key for each word, and the number of its occurrences as the associated value.

How about, say, filtering the lines containing a specific word? This is something easily done by having a map function that outputs a subset of its input, based on some predicate provided by the user:



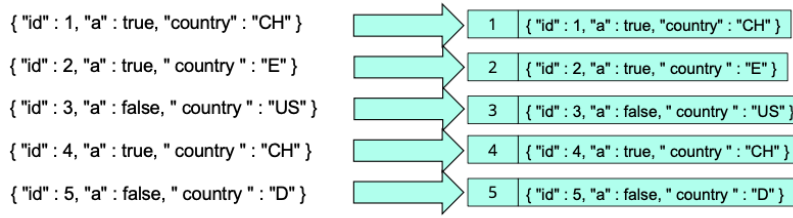
Here we notice that the output of the map phase already gives us the desired result; we still need to provide a reduce function, which is taken trivially as the identity function. This is not unusual (and there are also examples where the map function is trivial, and the reduce function is doing the actual processing):



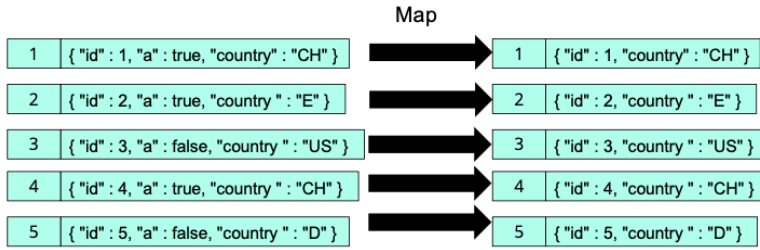
In fact, what we have just implemented in MapReduce is nothing else than a selection operator from the relational algebra. So now, you know how to implement the following SQL query on text, seen as a one-column table, on top of MapReduce, and in a way that scales to billions of lines!

```
SELECT text
FROM input
WHERE text LIKE '\%foobar\%'
```

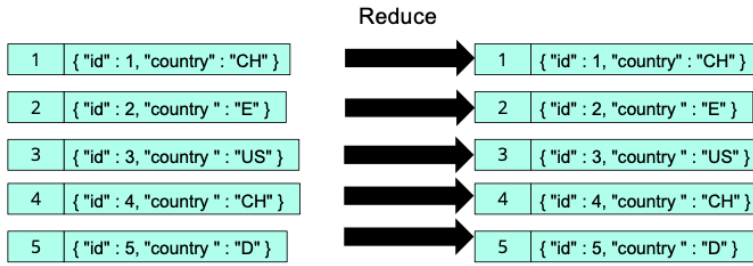
What about projection on some input in the JSON Lines format? Well, MapReduce doesn't know anything about attributes. So it is up to the user to parse, in their code, each line to a JSON object (e.g., if using Python, to a dict), like so:



Then, the map function can project this object to an object with less attributes:



And, like previously, we use a trivial identity function for the reduce function:



MapReduce will then output the results as one or several output files in the JSON Lines format. What we have just implemented on billions of records is the following equivalent SQL query:

```
SELECT id, country
FROM input
```

As an exercise, try to figure out how to implement a GROUP BY clause and an ORDER BY clause. What about the HAVING clause?

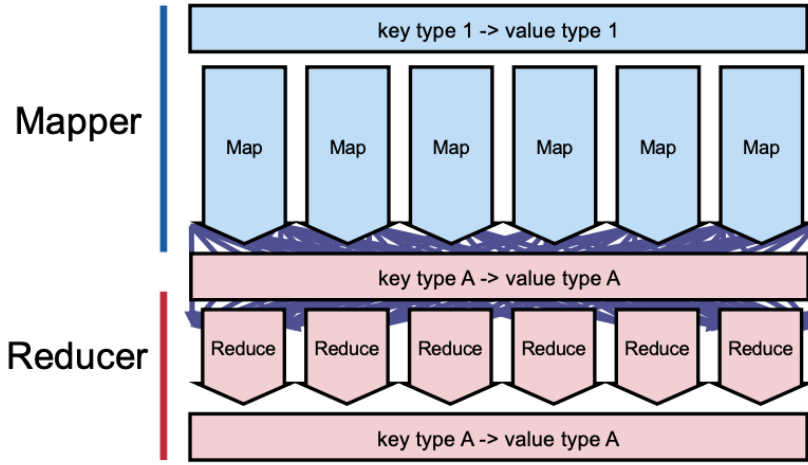
Do take a few seconds to realize what we have achieved here: previously, in the 2000s, the execution of such queries was only possible on inputs that fit on a single machine, e.g., in a PostgreSQL installation. This means maybe millions of records, perhaps one billion with today's machines, but nothing beyond this. But now, we know how to handle trillions of them on a cluster.

Naturally, executing these queries directly in MapReduce is very cumbersome because of the low-level code we need to write, and understanding how this is done has a pedagogical purpose more than a practical purpose. If I did my job correctly as a teacher, most of my

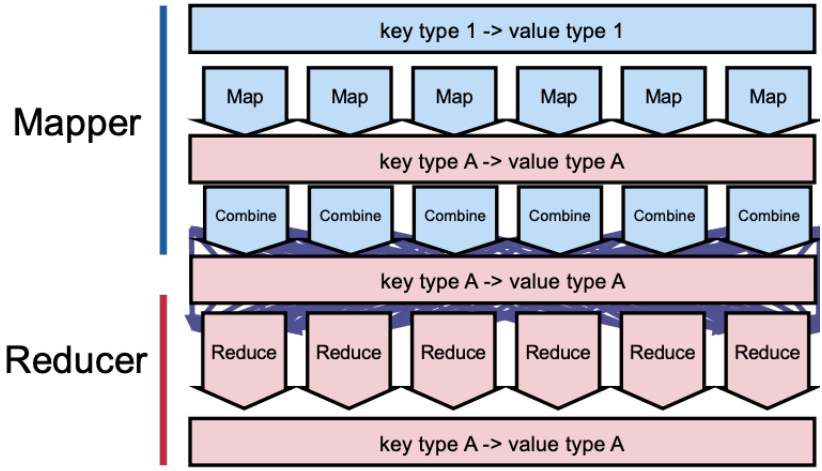
students will never use MapReduce directly (except Data Engineers maybe, implementing this inside a larger system). But in real life, this is nothing to worry about, because all of this complexity will be again hidden behind SQL and similar languages, following the data independence paradigm.

8.7 Combine functions and optimization

Let us look again at the MapReduce phases:



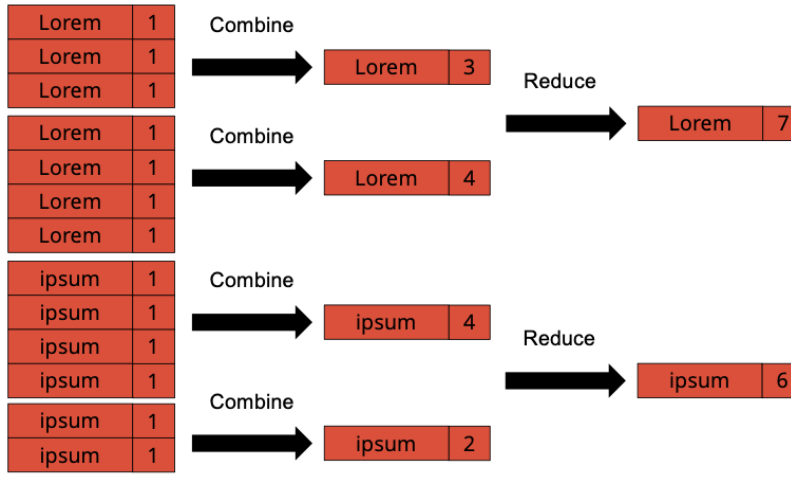
Is there any way we can optimize things and run even faster? In fact, there is. In our counting example, we created an intermediate key-value for each occurrence of a word with a value set to 1. But what if a word appears 5 times on the same line? In this case, we can replace the corresponding key-value pairs with just one pair, with the value 5. Doing so is called combining and happens during the map phase:



Thus, in addition to the map function and the reduce function, the user can supply a combine function. This combine function can then be called by the system during the map phase as many times as it sees fit to "compress" the intermediate key-value pairs. Strategically, the combine function is likely to be called at every flush of key-value pairs to a Sequence File on disk, and at every compaction of several Sequence Files into one.

However, there is no guarantee that the combine function will be called at all, and there is also no guarantee on how many times it will be called. Thus, if the user provides a combine function, it is important that they think carefully about a combine function that does not affect the correctness of the output data. In fact, in most of the cases, the combine function will be identical to the reduce function, which is generally possibly if the intermediate key-value pairs have the same type as the output key-value pairs, and the reduce function is both associative and commutative. This is the case for summing values as well as for taking the maximum or the minimum, but not for an unweighted average (why?). As a reminder, associativity means that $(a+b)+c = a+(b+c)$ and commutativity means that $a+b = b+a$.

In our example, the reduce function fulfils these criteria and can then be used as a combine function as well:



8.8 MapReduce programming API

Let us now move on to the concrete use of MapReduce in a computer program.

In Java, the user needs to define a so-called Mapper class that contains the map function, and a Reducer class that contains the reduce function.

A map function takes in particular a key and a value. Note that it outputs key-value pairs via the call of the write method on the context, rather than with a return statement. That way, it can output zero, one or more key-values. A Mapper class looks like so:

```
import org.apache.hadoop.mapreduce.Mapper;
public class MyOwnMapper extends Mapper<K1, V1, K2, V2>{
  public void map(K1 key, V1 value, Context context)
      throws IOException, InterruptedException
  {
    ...
    K2 new-key = ...
    V2 new-value = ... context.write(new-key, new-value); ...
  }
}
```

A reduce function takes in particular a key and a list of values. Note that it outputs key-value pairs via the call of the write method on the context, rather than with a return statement. That way, it can output zero, one or more key-values. A Reducer class looks like so:

```
import org.apache.hadoop.mapreduce.Reducer;
public class MyOwnReducer extends Reducer<K2, V2, K3, V3>{
public void reduce (K2 key, Iterable<V2> values, Context context)
    throws IOException, InterruptedException
  {
    ...
    K3 new-key = ...
    V3 new-value = ... context.write(new-key, new-value); ...
  }
}
```

Finally, a MapReduce job can be created and invoked by supplying a Mapper and Reducer instance to the job, like so:

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

    job.setMapperClass(MyOwnMapper.class);
    job.setReducerClass(MyOwnReducer.class);

    FileInputFormat.addInputPath(job, ...);
    FileOutputFormat.setOutputPath(job, ...);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

A combine function can also be supplied with the setCombinerClass method (passing, for example, the same Reducer instance.

It is also possible to use Python rather than Java, via the so-called Streaming API. The Streaming API is the general way to invoke MapReduce jobs in other languages than Java (or than JVM languages, like Scala). This is done by creating two files, say, mapper.py and reducer.py. These two files take input (or intermediate) key-value pairs from standard input, and write intermediate (or output) key-value pairs to standard output. They are then invoked on the command line:

```
$ hadoop jar hadoop-streaming*.jar \
-files mapper.py,reducer.py \
-mapper mapper.py \
-reducer reducer.py \
-input input
-output output
```

Without going too much in details, the input formats and output formats can also be specified in all programming languages. In Java, this is in the form of picking Java classes inheriting from InputFormat (DBInputFormat for a relational database, TableInputFormat for HBase, KeyValueTextInputFormat, SequenceFileInputFormat, TextInputFormat, FixedLengthInputFormat, NLineInputFormat...) or OutputFormat (DBOutputFormat, TableOutputFormat, SequenceFileOutputFormat, TextOutputFormat, MapFileOutputFormat...).

8.9 Using correct terminology

Let us now have a word of warning: the terminology "Mapper" and "Reducer" should only be used in the context of naming classes and files, but never when describing the MapReduce architecture. Even less so with "Combiner."

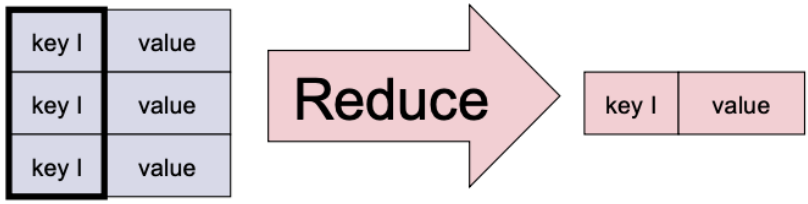
This is because this terminology is very imprecise and makes it very difficult to comprehend the MapReduce architecture. Sadly, many resources in books and on the Web do refer to mappers, reducers and combiners. We hope the reader will find what follows enlightening and helpful to truly understand what is going on in a MapReduce cluster.

Rather than "mapper", we encourage the reader to use "map function", "map task", "map slot" or "map phase" depending on what is meant. Rather than "reducer", we encourage the reader to use "reduce function", "reduce task", "reduce slot" or "reduce phase" depending on what is meant. Rather than "combiner", we encourage the reader to use "combine function" – there is no such thing as a combine task, a combine slot or a combine phase.

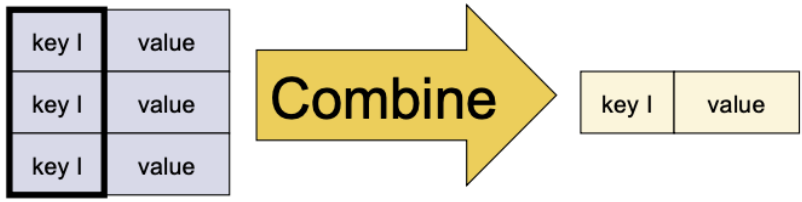
Let us start with functions.

A map function is a mathematical, or programmed, function that takes one input key-value pair and returns zero, one or more intermediate key-value pairs.

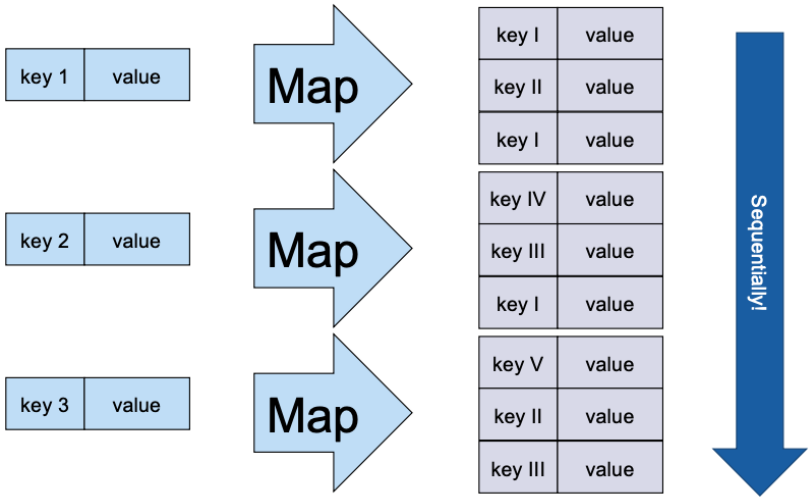
A reduce function is a mathematical, or programmed, function that takes one or more intermediate key-value pairs and returns zero, one or more output key-value pairs.



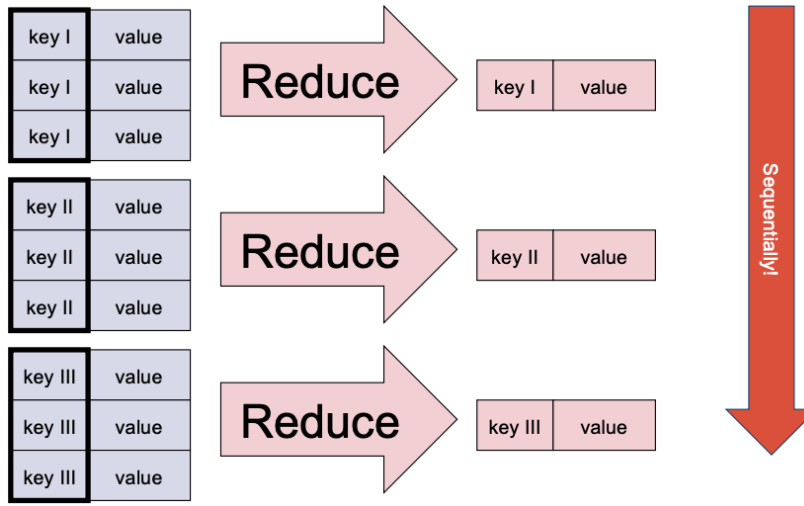
A combine function is a mathematical, or programmed, function that takes one or more intermediate key-value pairs and returns zero, one or more intermediate key-value pairs.



Then, a map task is an *assignment* (or "homework", or "TODO") that consists in a (sequential) series of calls of the map function on a subset of the input. There is one map task for every input split, so that there are many map tasks as partitions of the input.

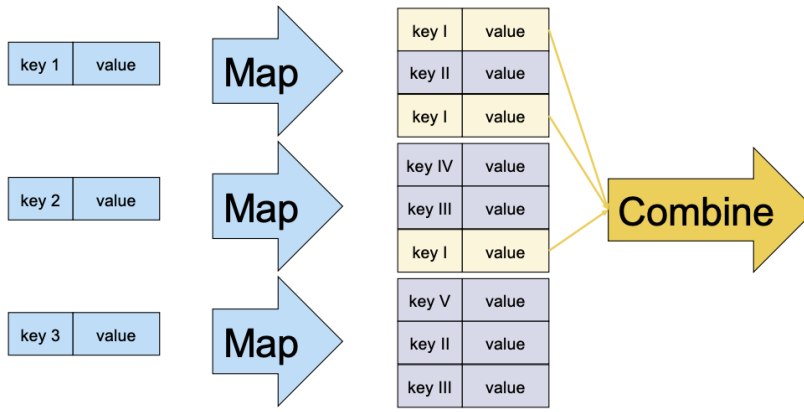


A reduce task is an assignment that consists in a (sequential) series of calls of the reduce function on a subset of the intermediate input. There are as many reduce tasks as partitions of the list of intermediate key-value pairs.

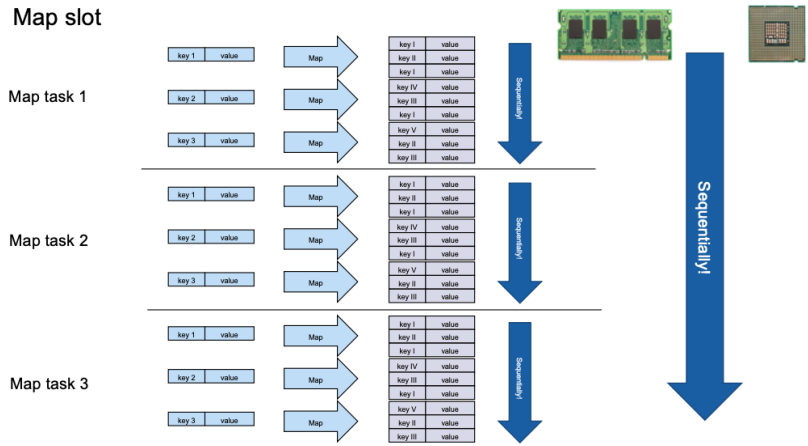


We insist that the calls within a task are sequential, meaning that there is no parallelism at all within a task. You can think of it as a for loop calling the function repeatedly, with the size of the for loop being, in a typical setting, between 1,000 and 1,000,000 calls.

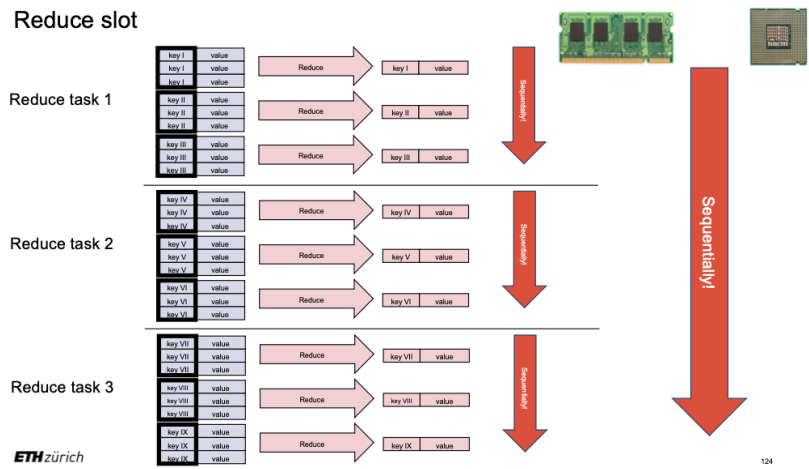
There is no such thing as a combine task. Calls of the combine function are not planned as a task, but is called ad-hoc during flushing and compaction.



The map tasks are processed thanks to compute and memory resources (CPU and RAM). These resources are called map slots. One map slot corresponds to one CPU core and some allocated memory. The number of map slots is limited by the number of available cores. Each map slot then processes one map task at a time, sequentially. This means that the same map slot can process several map tasks.

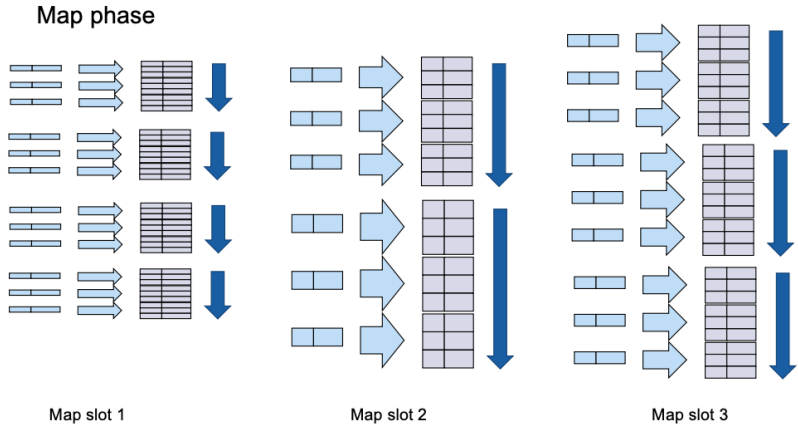


The resources used to process reduce tasks are called reduce slots. Again, one reduce slot corresponds to one CPU core and some allocated memory. The number of reduce slots is limited by the number of available cores. Each reduce slot then processes one reduce task at a time, sequentially. This means that the same reduce slot can process multiple reduce tasks.



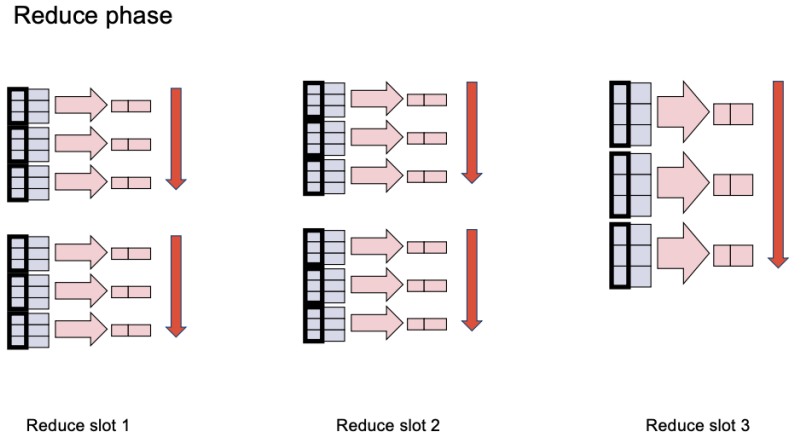
So, there is no parallelism either within one map slot, or one reduce slot. In fact, parallelism happens across several slots. In a typical MapReduce job, there will be more tasks than slots. Initially, each slot will receive one task, and the other tasks are kept pending. Every time a slot is done processing a task, it receives a new task from the pending list, and so on, until no task is left: then, some slots will remain idle until all tasks have been processed[2]. If a task fails, it can be reassigned to another slot.

The map phase thus consists of several map slots processing map tasks in parallel:

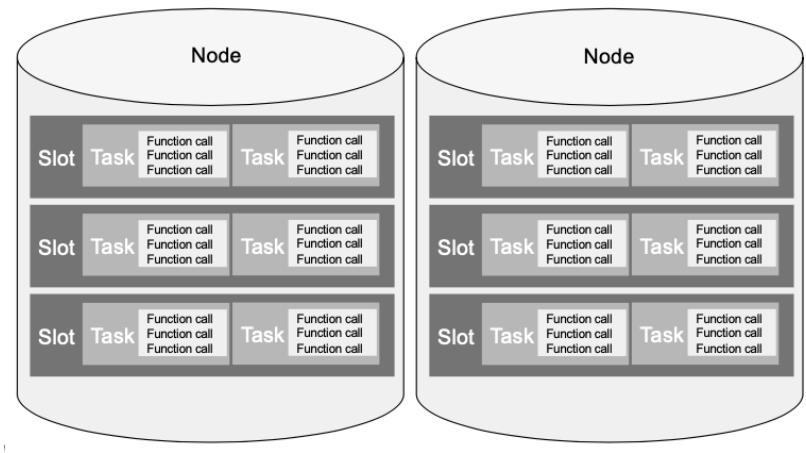


And the reduce phase consists of several reduce slots processing reduce tasks in parallel:

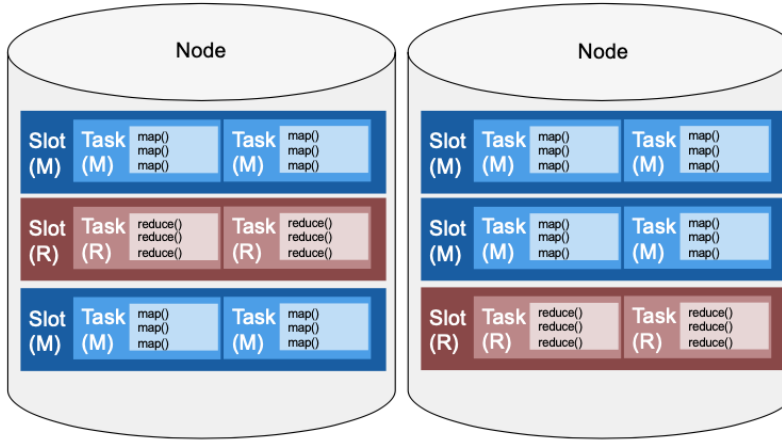
[2]For those who have already volunteered to count votes after an election or votation, this will feel familiar, as votes are also processed in batches of, say, 100 ballots, by multiple groups across several tables. Next time you do, tell them they are actually using the MapReduce framework!



This is a summary of how functions, tasks, slots and phases fit together and within cluster nodes:



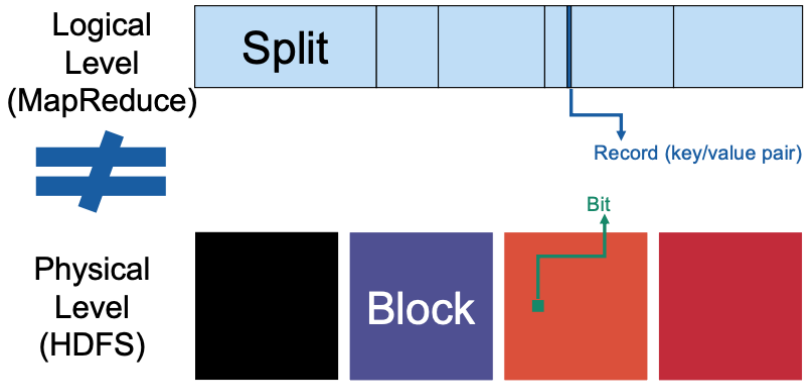
In the very first version of MapReduce (with a JobTracker and TaskTrackers), map slots and reduce slots are all pre-allocated from the very beginning, which blocks parts of the cluster remaining idle in both phases. We will see in the next Chapter that this can be improved.



8.10 Impedance mismatch: blocks vs. splits

We finish this chapter with a more in-depth discussion of how MapReduce and HDFS interact. Now that we have the necessary terminology to express it, we can say that the "bring the query to the data" paradigm means that the data belonging to the split that is the input of a map task resides on the same machine as the map slot processing this map task. How can it be? This is because the DataNode process of HDFS and the TaskTracker process of MapReduce are on the same machine. Thus, getting a replica of the block containing the data necessary to the processing of the task is as simple as a local read. This is called short-circuiting, the same name we gave to the local read of HFiles in HBase as well.

But there is something important to consider: HDFS blocks have a size of (at most) 128 MB. In every file, all blocks but the last one have a size of exactly 128 MB. Splits, however, only contain full records: a key-value pair will only belong to one split (and thus be processed by one map task). This means that, while most key-value pairs will be in the same block, the first and/or last key-value pair in a split will be spread across two blocks. This means that, while most of the data is obtained locally, getting the first and/or last record in full will require a remote read over the HDFS protocol. This, in turn, is also the reason why the HDFS API gives the ability to only read a block partially.


Logical
Level
(MapReduce)
Split
Record (key/value pair)
Physical
Level
(HDFS)
Block
Bit

List of Figures