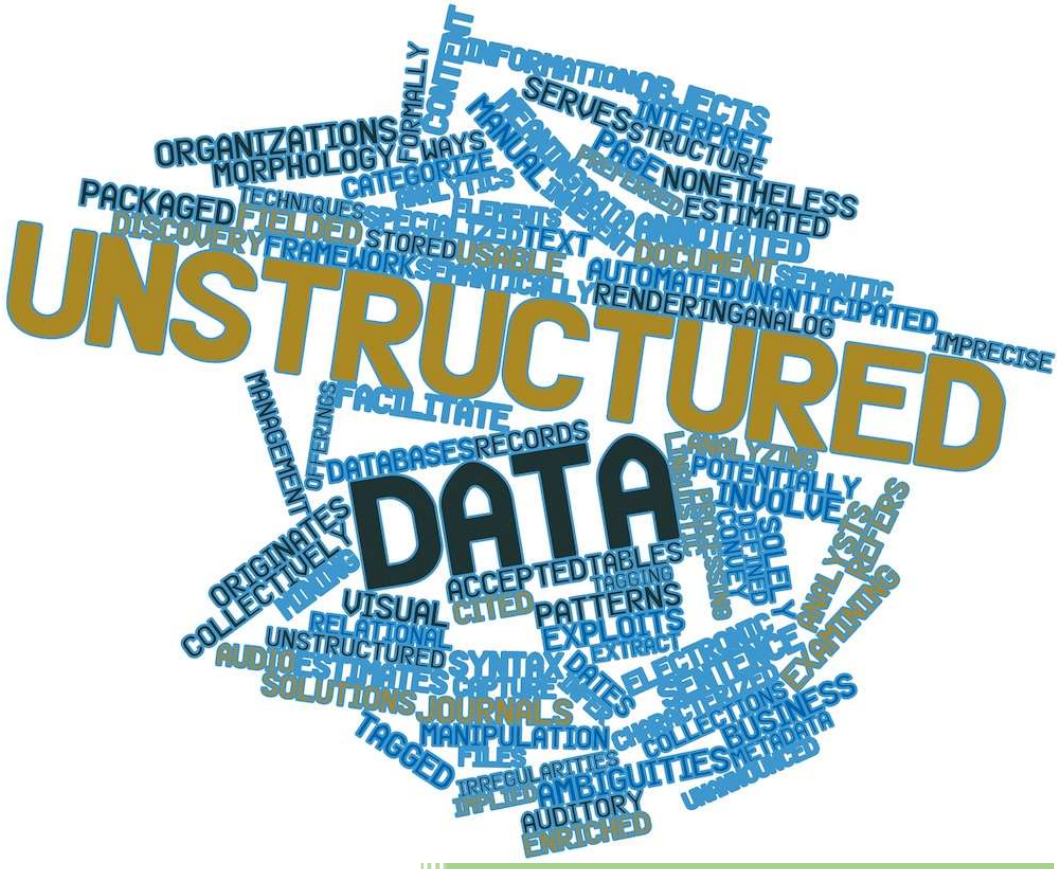
2023-2024

Systems and Methods for Big and Unstructured Data



Luca Gerin

Politecnico di Milano

2023-2024

Disclaimer

This document is the product of the course slides and notes taken during the "SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA" course at *Politecnico di Milano* held in year 2023-2024.

The consequences are the following:

- The content might be incomplete, unprecise, or even wrong in some parts.
- The majority of the images may come from the slides of the course or be taken from the internet, and are not created by the author except for few cases.

Contents

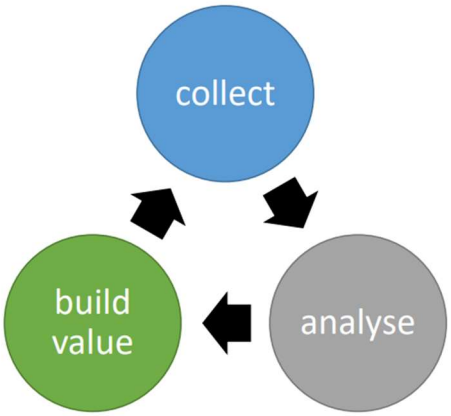
1. Big Data

Big data primarily refers to data sets that are too large or complex to be dealt with by traditional data-processing application software.

Big data analysis challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy, and data source.

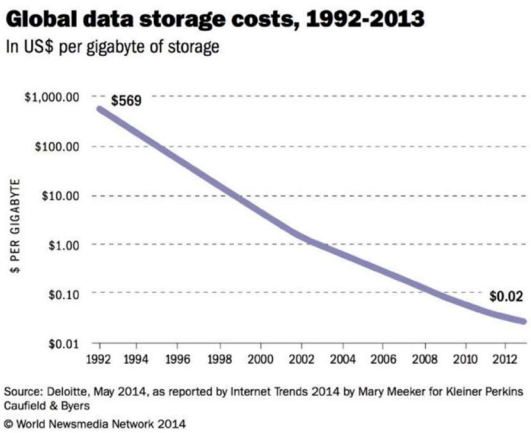
The Big Data market includes providers that deal in Infrastructure to handle them, Analytics services, Applications, Cross-infrastructure/Analytics, Open-source providers, Data sources and APIs. There are many ingredients, each one with a different perspective on the data. Big Data is also the enabler of several AI technologies.

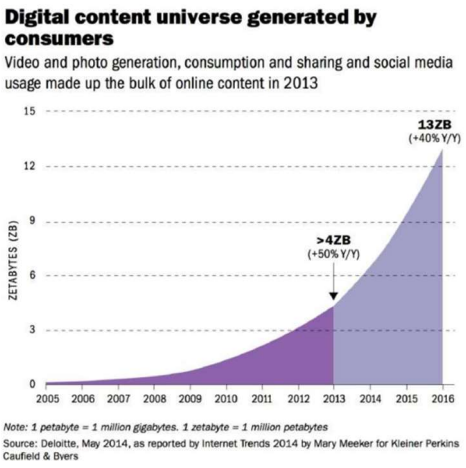
Data are processed in a cycle that consists in collecting the data (Data Ingestion), analyzing them for descriptive, predictive, and prescriptive purposes (Data Analysis), and build value from them (Value Generation).



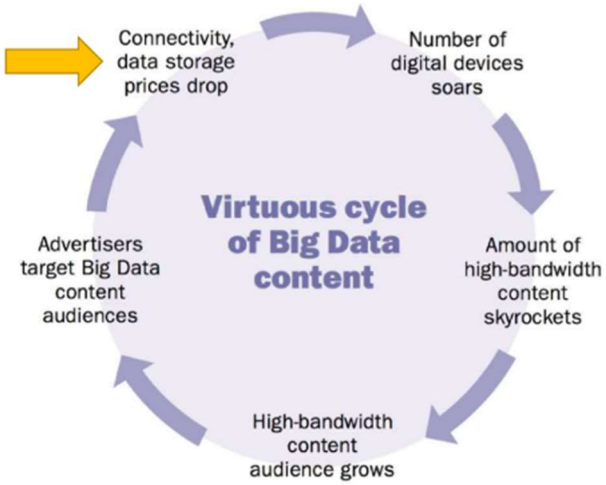
The value built by big data lies in: increasing economic transactions, creating brand awareness, making cities a better place, providing cultural value, collecting, and building knowledge.

The recent development of technology and the availability of a big amount of data are the two factors determining why Big Data have gained such importance only nowadays. We must note that memory (hard drives) is becoming cheaper, and all people have become data producers, leading to data getting cumulated everywhere.

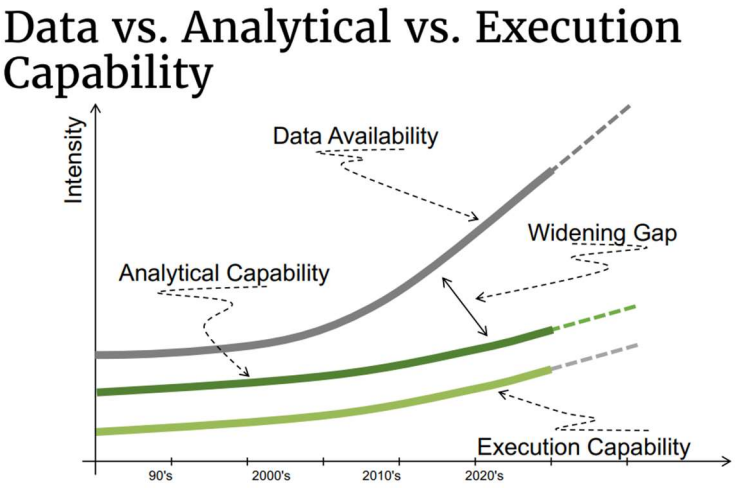




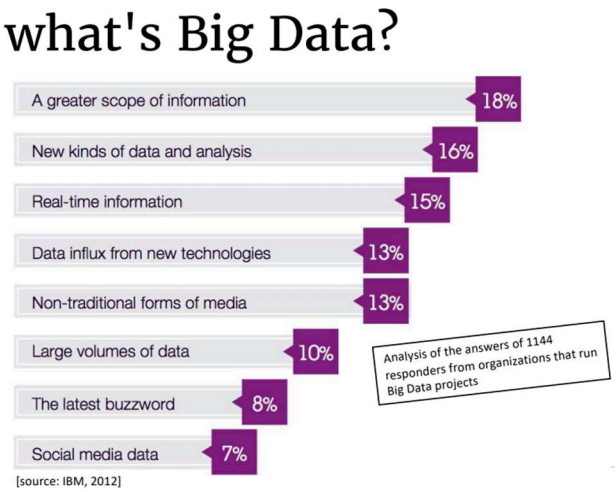
It's worth noticing, though, that the rate at which digital content is generated by users is now slowing down, but still the curve is going up.



However, the present technology seems not to be able to keep up with the growing amount of data available. So, data is not fully exploited and the gap between capabilities and data availability seems destined to grow.

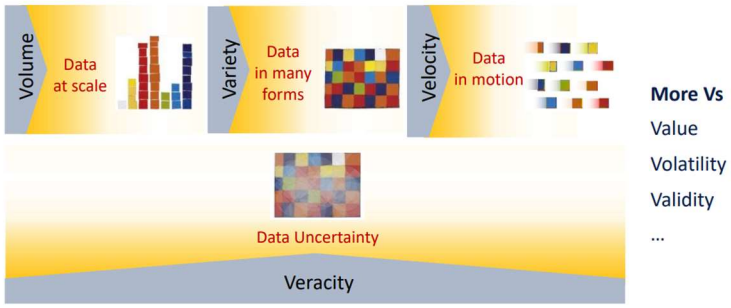


Going back to the definition of Big Data, we see that it is strictly correlated to the value it generates.

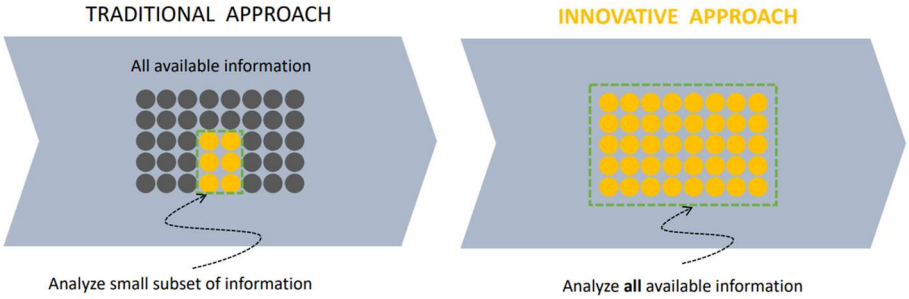


Big Data can be described with the **4 Vs**:

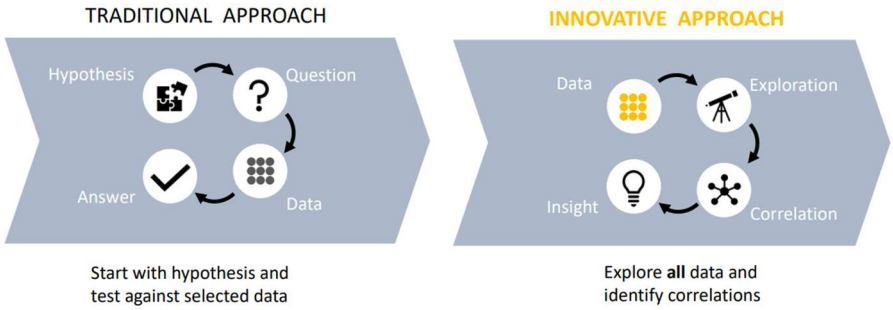
- Volume: data that ranges in size from Terabytes to Petabytes (in Italy a company with tens of Terabytes of analytical data is considered pretty big). “Data is Big”.
- Variety: data that can assume many different forms, from structured (like relational tables), to semi-structured (like JSON and XML) and even completely unstructured (text and multimedia files). “Data is heterogeneous”.
- Velocity: information that flows continually creating streams, that often needs to be analyzed in a time frame of a few seconds. “Data is in motion and comes in very quickly”.
- Veracity: big data can often be imprecise and unpredictable, and this unreliability and uncertainty needs to be addressed and managed. There is high risk of having polluted and uncertain data.



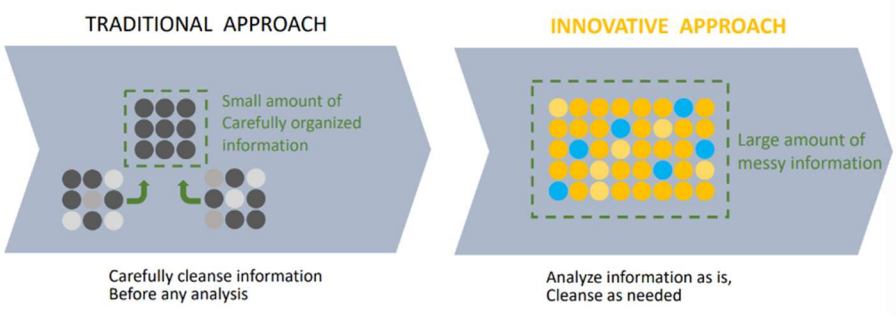
Traditionally, when a lot of information was available, as analyzing everything was unfeasible, analysis was performed on a sample of the whole information and, from this subset, characteristics about the whole population were inferred. The innovative approach enabled by the technology of big data is that now it’s possible to analyze all the available information, leveraging this way more of the data being captured and avoiding to introduce more uncertainty.



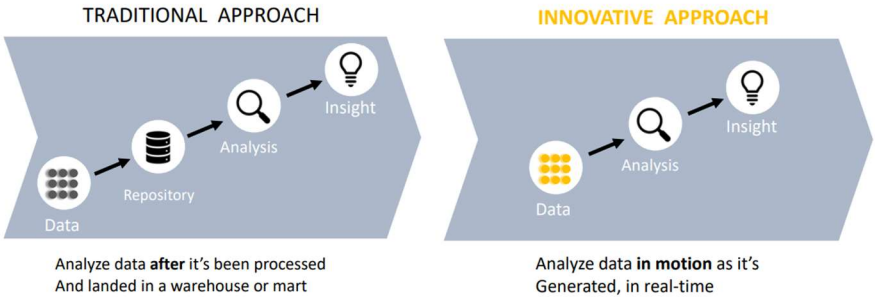
Another shift in paradigm happened in how consequences from data are studied. The traditional approach is the scientific one, for which an initial hypothesis is tested against selected data to prove it. With big data available, we can work in the opposite way, without needing to formulate a hypothesis before analyzing the data, but all data are explored, and correlations are identified.



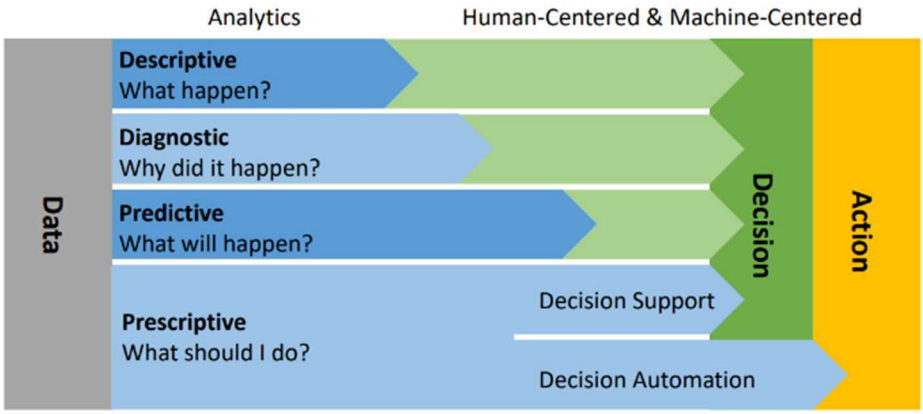
Additionally, there is no need to select data among the available before performing an analysis, but information is generally just analyzed as is, and cleansed as needed. The effort required to leverage data Is reduced.



Data can be leveraged as it is captured. The traditional approach consists in a linear process where data analysis is the last step, while now the data can be analyzed as it comes in.



Data is transformed in information, and the same information that is output of the process has a feedback value for future analysis.



It's worth noticing that Big Data also implies some problems:

- Data access
- Noise and trustworthiness
- Algorithmic (and many other types of) bias
- Cost of maintenance / problem solving

2. Data-Driven Decisions

In many organizations decisions are made by "questionable" methodologies such as Highest Paid Person Opinion (HiPPO) or Flipism (all decisions are made by flipping a coin). These approaches are deprecated in the digital era, where organizations can be data-driven.

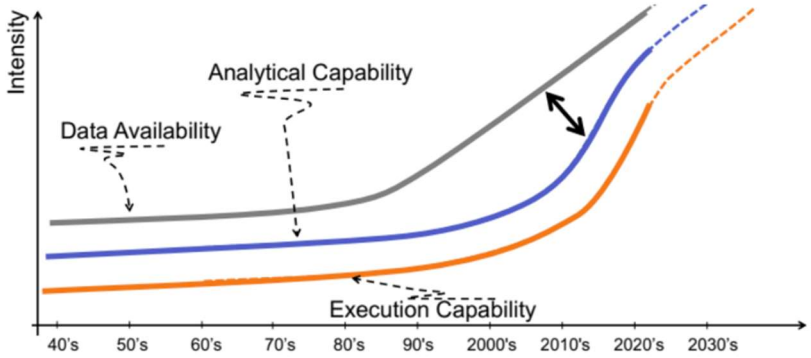
A **data-driven** organization is an organization that uses data insights to guide its decision-making processes. Data-driven companies rely upon data-driven processes, data analytics tools, and data-driven culture to generate business insights that help inform their decisions.

Data-driven organizations:

- Perform better: the data shows where they can streamline their processes.
- Are operationally more predictable: data insights fuel current and future decision making.
- Are more profitable: constant improvements and better predictions help to outsmart the competition and improve innovation.

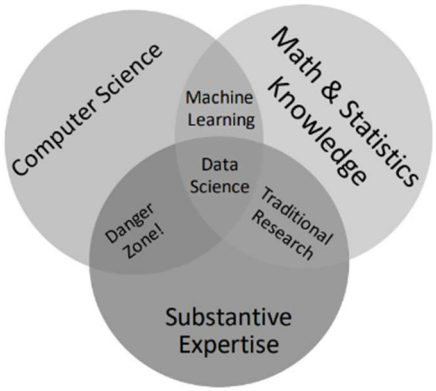
These companies need reliable sources of data from which to extract valuable information to use to perform decision making. This data can come from the IT systems of the companies themselves, can be taken from the internet or can be bought. For decisions to be more effective and profitable, data needs to be a lot, and, in many cases, it needs to be organized and stored in a good way.

The available data is an upper bound for the analytical capability of a company, which in turn is an upper bound to the execution capability.
Thanks to Big Data technologies, data science and engineering, the gap between data availability and the capabilities of companies is getting less and less wide.



Data science is the science of:

- Discovering what we don't know from data.
- Obtaining predictive, actionable insight from data.
- Creating Data Products that have business impact now.
- Communicating relevant business stories from data.
- Building confidence in decisions that drive business value.



3. Recap on ER and data architecture

E-R and Relational Data Models

The levels of abstraction for the design of a database are the following, to be used with a top-down approach:

- Conceptual database design (Representation): constructing an information model, independent from all physical consideration for an enterprise.
- Logical database design (Schema): building an organization database based on a specific data model.
- Physical database design (Optimization): implementing a database using specific data storage structure(s) and access methods.

The **ER Model** (Entity-Relationship Model) was invented in 1976 to represent information and is used to perform the conceptual design of a database.

The **relational model** was invented before, in 1970, based on the mathematical notion of relations, that are represented by means of tables. (A relation is mathematically a subset of the cartesian product of some sets).

Conceptual models are converted to relational models by converting entities into tables and relationships into other tables or into foreign keys.

Normalization is a two-step process that puts data into tabular form by removing repeating groups and then removes duplicated data from the relational tables. Normalization theory is based on the concepts of normal forms. The goal of normalization is to create a set of relational tables that are free of redundant data and that can be consistently and correctly modified or updated. This means that all tables in a relational database should be in the third normal form (3NF).

A relational table is in 3NF if and only if all non-key columns are mutually independent and fully dependent upon the primary key.

Data architecture

When we talk about databases, we always have a **schema**, that is the structure of a database described in a formal language, composed by tables representing entities and relationships. The schema defines in a unique way the structure of the object, defines the data types, cardinality, and guarantees uniformity and coherency across the database.

A **transaction** is an elementary unit of work performed by an application, encapsulated between a *begin transaction* and a *end transaction* command, and within which either a *commit* or a *rollback* happens. A **Transactional System (OLTP)** is a system capable of providing the definition and execution of transactions on behalf of multiple, concurrent applications.

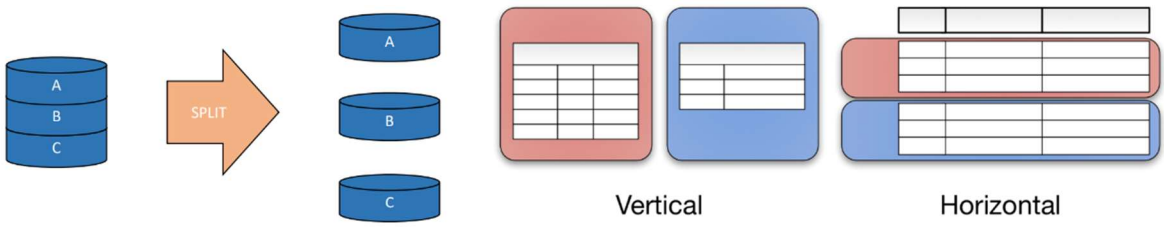
Well-formed transactions enforce ACID properties: Atomicity, Consistency, Isolation, Durability.

- **begin transaction**
- code for data manipulation (reads and writes)
- **commit work – rollback work**
- no data manipulation
- **end transaction**

Data partitioning is performed to obtain more scalability and a distributed system. Partitioning splits the data in the database and partitions pieces of it to different storage nodes.

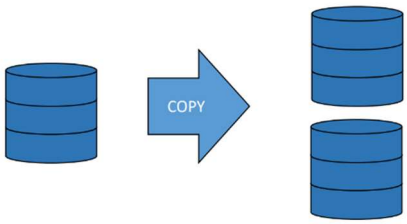
Databases can be sharded horizontally (by rows) or vertically (by columns). When horizontal partitioning happens, we talk about *sharding*.

The advantages of partitioning are that it enables fast data writing and reading and requires lower memory overhead, but exposes to potential data losses.

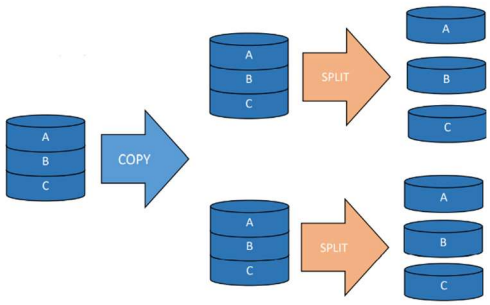


Replication is instead performed to obtain fault-tolerance and backups for data. The database is copied across all nodes available in the distributed system.

Replication favors fast data reading and brings high data reliability, but requires high network and memory overheads.



Partitioning and replication can be combined in different ways, to obtain a trade-off between the desired properties.



Database **scalability** is the ability of a database to improve its availability and behavior when the business demands more resources.

The **elasticity** of a data store is its capability to be enlarged when needed or to release resources when they are not needed and relates to the flexibility of its data model and clustering capabilities. The greater the number of data model changes that can be tolerated, and the more easily the clustering can be managed, the more elastic the data store is.

Data ingestion is the process of importing, transferring, and loading data for storage and later use.

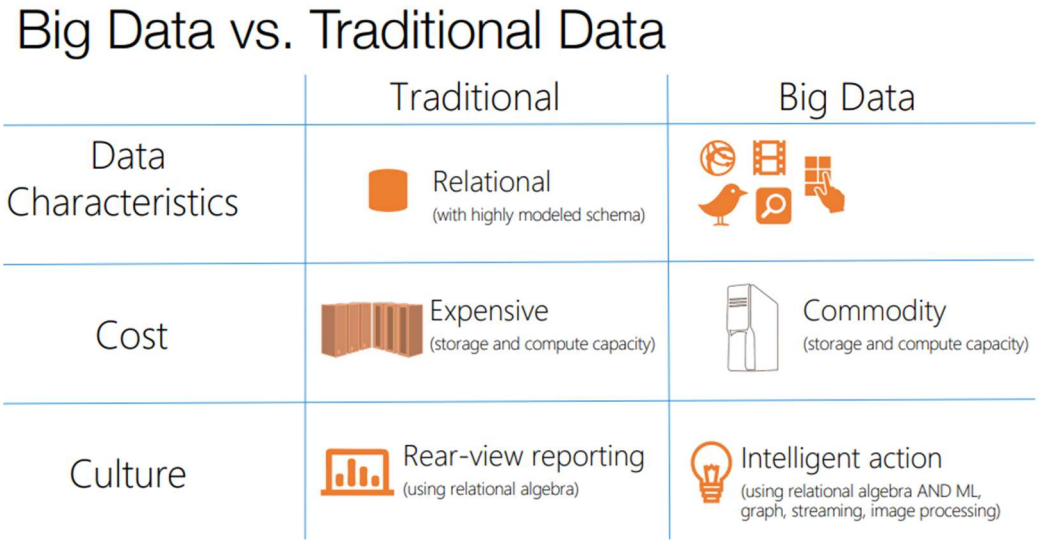
Data wrangling is he process of cleansing "raw" data and transforming raw it into data that can be analyzed to generate valid actionable insights.

4. Towards NoSQL

The Big Data world has a lot of differences from the traditional data managing approach. First, we need to move from the relational world to new kinds of unstructured data storage options.

Also, there is a shift in the hardware technologies, going from the typical vertically scalable, on-premises mainframes to cloud based, horizontally scalable commodity machines.

Lastly, there is a change in data processing culture, from a rear-view reporting approach based on relational algebra to a smarter model of continuous analytics with streaming and machine learning techniques.

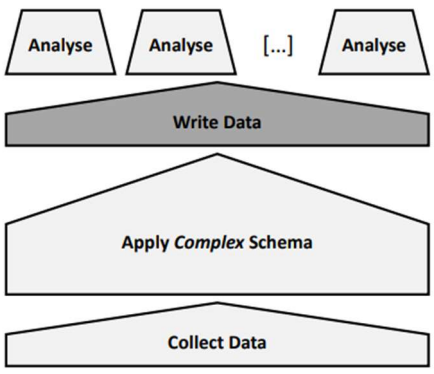


Flexibility and Big Data

To make a step towards Big Data architectures, systems need more and more **flexibility**.

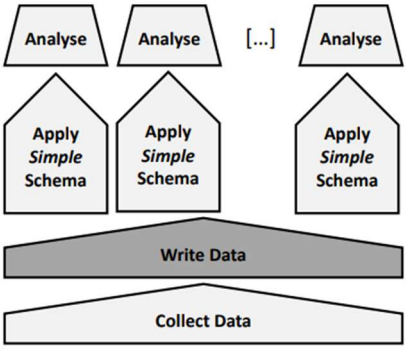
To achieve such flexibility, the starting point is to remove the schema in order not to have an imposed structure that may be too strict.

Traditionally, the schema is imposed *On Write*, meaning that the data, after being collected, is transformed to fit the schema of the database where it needs to go and then inserted in it. This process means that some information is lost at writing time, and so some analysis can no longer be performed.

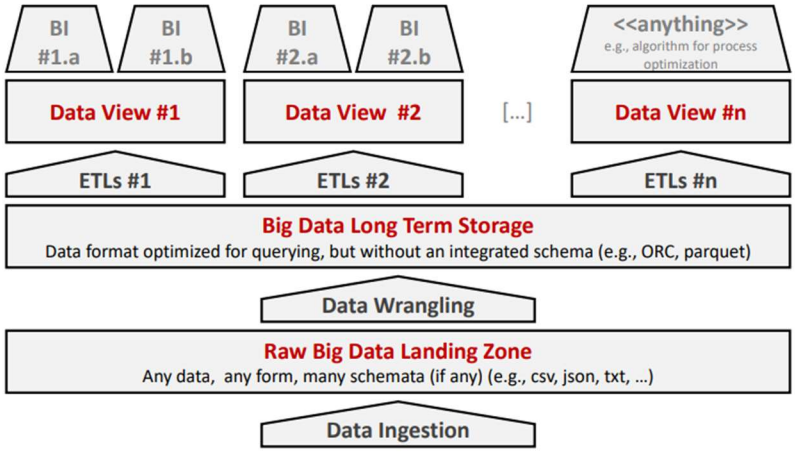


In the so-called" **schema-less**" approach, data doesn't need to strictly respect a predefined structure to be stored in the database. With the schema *On Read* approach, data is first loaded in the database as is and all data is kept. When analysis need to be performed, a minimal schema with a structure fit for that specific analysis is temporarily applied only to the needed data.

This way, data compliance to the schema is delayed as late as possible and in a later moment with respect to when it's collected and stored, so new analyses can be introduced in any point in time.



The logical architecture of big data is the following:

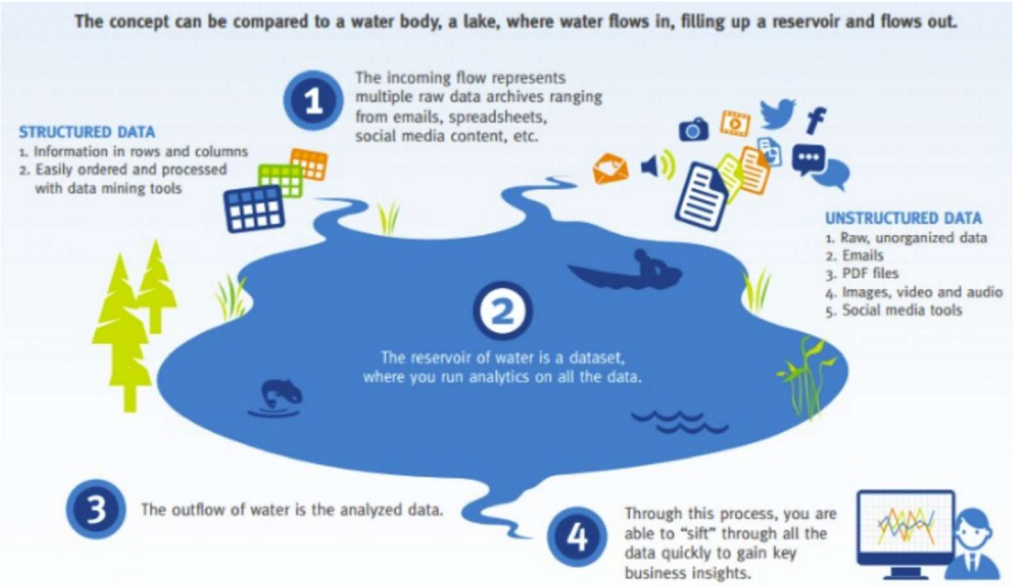


The advantages of the *Schema-On-Read* approach with respect to the *Schema-On-Write* approach are:

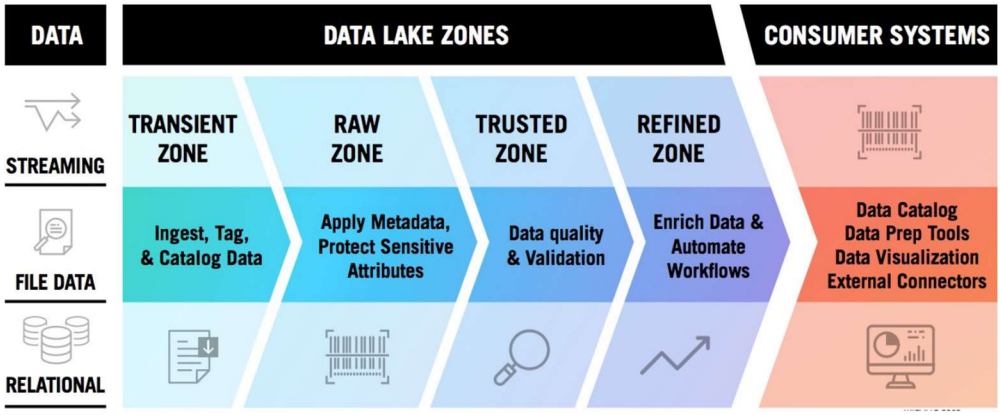
- Since we don't have to check the correctness of the schema the database writes are much faster.
- We have a smaller IO throughput since when reading data, we can only fetch the properties we need for that task.
- The same data can assume a different schema according to the needs of the analytical jobs that is reading it, optimizing performance.

The extremization of the *Schema-On-Read* approach at a whole company level is the concept of **Data Lake**, in which all the company's data is in the same place. A data lake is a single system or repository of data stored in its natural/raw format and data transformed for tasks such as reporting, visualization, advanced analytics, and machine learning. A Data Lake has incoming flows of both structured and unstructured data and outgoing flows consumed by the analytical jobs. This way organizations have only one centralized data source for all its analytical needs, as opposed to silos-

based approaches.



However, building such a huge and centralized data storage system has many challenges and it's crucial to organize and index well all the incoming data, mostly using metadata, or the result could be to end up with a huge *Data Swamp* where it's difficult to find the desired information. Thus, in a properly functioning Data Lake the incoming raw data gets incrementally refined and enriched to improve accessibility, by adding metadata and indexing tags and by performing quality checks.



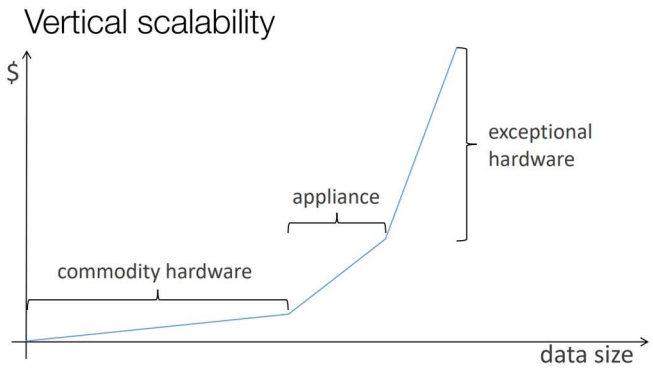
Another issue with Big Data is the Impedance Mismatch between object oriented and relational models that need to coexist and cooperate in systems. The problem is the one of the **Object-Relational Mapping** and solutions to it exist.

Scalability for Big Data

Systems working with Big Data need to be able to grow in size with the increase of the amount of data.

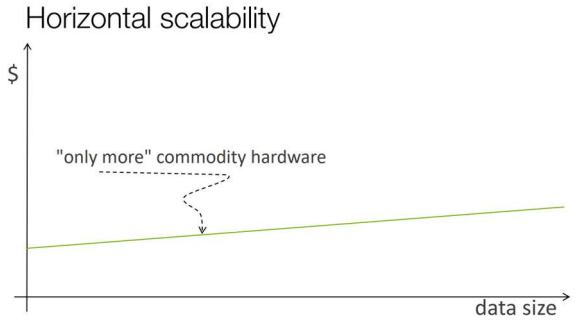
Traditional SQL system scale vertically, meaning that when the machine on which the system runs no longer performs as required, the solution is to buy a better performing (and more expensive) machine.

This approach can go on only so far, as eventually buying a better machine will become too expensive or the technical and physical limits of the current technology will be reached. Vertical scalability is not always convenient after a certain point is reached and needs the investment to be carried out all at the same moment.

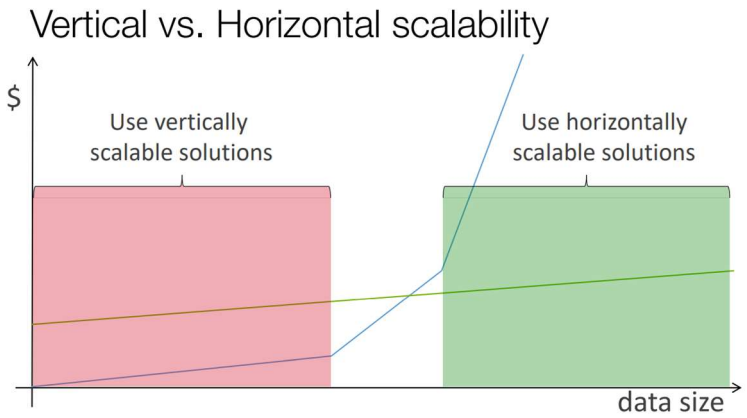


Instead, Big Data solutions scale horizontally, meaning that when the machines, where the big data solution runs, no longer perform as required, the solution is to add another machine of the same quality to work in parallel to the already existing ones. This implies the need for an overhead given by the software database technology and network to support the expanding architecture.

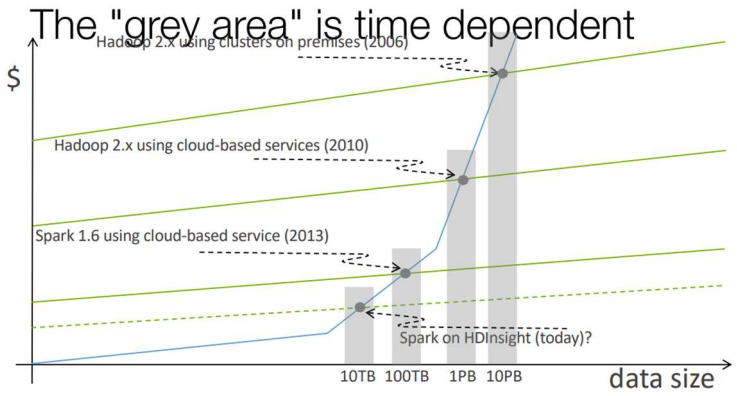
The advantages of horizontal scalability are that the investment in machines can be spread according to needs, but also it enables elasticity if for example the hardware is rent and released when not needed anymore. However, a big investment is needed at the beginning in the infrastructure to support the architecture.



There is a critical point that delimits the situations in which vertically or horizontally scalable solutions are more convenient. Usually, when data is small in size vertically scalable solutions are preferrable. Then, when data begins to be a lot, horizontally scalable solutions become the optimal.



The decision point depends on how much data the company forecasts to have in the future, and it's important to note that the position of this decision point may change significantly in time with the change of prices and of the market.



Transactional properties in NoSQL

Transactions in relational systems are elementary units of work that need to enforce the **ACID properties:**

- **Atomicity**: either the entire transaction takes place at once or doesn't happen at all. A transaction is an atomic transformation from the initial state to the final state.
- **Consistency**: integrity constraints must be maintained and satisfied by transactions, so that the database is consistent before and after the transaction.
- **Isolation**: multiple transactions can occur concurrently, independently and without interference, without leading to the inconsistency of the database state.
- **Durability**: once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist.

These properties are fundamental in traditional SQL based OLTP applications.

However, enforcing ACID properties means that there are algorithms running to enforce some rules, and therefore the system complexity raises, and the system performance is reduced.

Giving up complete ACID transaction guarantees could mean more flexibility, and is what Big Data system do.

The **CAP theorem** states that, for a distributed system (Big Data systems are all distributed and use horizontal scalability), it is impossible to simultaneously provide all three of the following guarantees:

- **Consistency**: all nodes see the same data at the same time → data is coherent.
- **Availability**: node failures do not prevent other survivors from continuing to operate (a guarantee that every request receives a response about whether it succeeded or failed).
- **Partition tolerance**: the system continues to operate despite arbitrary partitioning due to network failures (e.g., message loss)

A distributed system can satisfy any *two of these three* guarantees at the same time but not all three.

This means that system designers need to choose two out of three of these properties, respectively about data, the system and the network, to enforce in their distributed systems for Big Data.

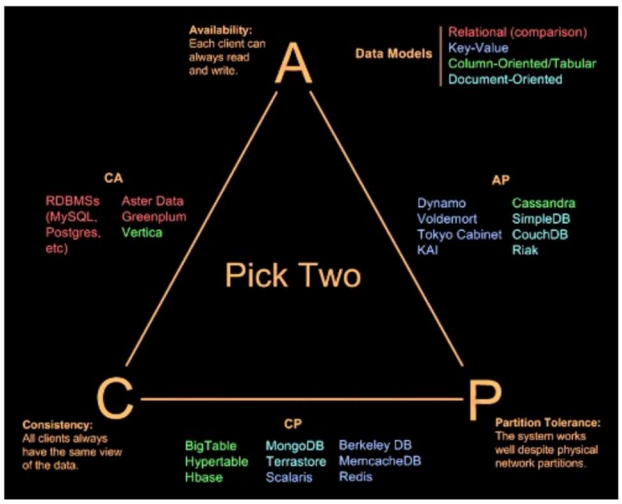
Usually, the partition tolerance (P) is always chosen as first property to enforce, because failures will always occur to a networked system. Not choosing partition tolerance means choosing Consistency and Availability, so having a CA system that is a traditional SQL system if it is all in the same place (if it's not distributed).

The usual choice is hence between:

- AP (Availability + Partition tolerance): a partitioned node returns a correct value if in a correct state, a timeout error, or an error otherwise.
- CP (Consistency + Partition tolerance): a partitioned node returns the most recent version of the data, which could be stale.

Historically, Consistency (C) was always chosen instead of availability, but making this choice means sacrificing performance, because consistency needs time and resources to be checked.

Many systems nowadays chose Availability (A), meaning that the application can get wrong answers, but there can just be a delay and the application can retry later for a correct one. Availability is considered in this case more important than consistency. Examples of systems that give up Consistency for Availability are edge computing, streaming services, and games, download services, search engines, social networks, web applications, e-commerce, sometimes even operational data. The rationale is that it is ok to use stale data or to give approximate answers: a streaming or a game may lag, an e-commerce can sometimes give the wrong number of available products.



Giving up ACID properties or Consistency does not mean of course to give up to every property at all, but some properties can be preserved. These properties take the name of **BASE properties** (Basically Available, Soft-State, Eventually Consistent):

- Basic Availability: fulfill request, even in partial consistency.
- Soft State: abandon the consistency requirements of the ACID model pretty much completely.
- Eventual Consistency: at some point in the future, data will converge to a consistent state; delayed consistency, as opposed to immediate consistency of the ACID properties.

These features are voluntarily vague and generic since they can be finetuned to the needs of the application as stated by the CAP theorem.
The disadvantages are that given BASE's loose consistency, developers need to be more knowledgeable and rigorous about consistent data if they choose a BASE store for their application. Planning around ACID properties is easier and fits better the case in which data reliability and consistency are essential.

The NOSQL World

NoSQL has different interpretations:

- Systems not using the SQL language.
- Systems that are non-relational.
- N.O.SQL: Not Only SQL.

The two major areas of NoSQL solutions are:

- **Key/Value** or "the big hash table"
  Amazon S3 (Dynamo), Voldemort, Scalaris, MemcacheDB, Azure Table Storage, Redis, Riak
- **Schema-less**
  Cassandra (column-based), CouchDB (document-based), Neo4J (graph-based), HBase (column-based)

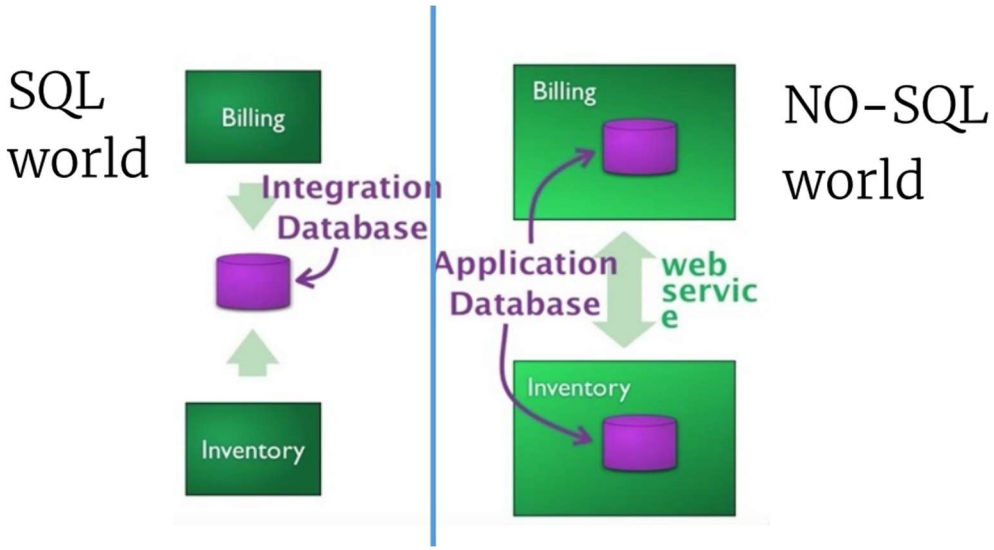
And the different types of NoSQL are:

- **Key-Value Store**: a key that refers to a payload (actual content / data)
  (MemcacheDB, Azure Table Storage, Redis)
- **Column Store**: column data is saved together, as opposed to row data. These systems are useful for data analytics.
  (Hadoop, Cassandra, Hypertable)
- **Document / XML / Object Store**: key (and possibly other indexes) point at a serialized object, and the DB can operate against values in document.
  (MongoDB, CouchDB, RavenDB)
- **Graph Store**: nodes are stored independently, and the relationship between nodes (edges) are stored with data.
  (Neo4j)

It's interesting that the companies that develop NoSQL solutions were not database producers, but they had needs that were not addressed by the market, where traditional systems had the monopoly, and decided to address these needs themselves. Companies like Google, LinkedIn, Facebook, Twitter, Netflix, Amazon also often collaborate to find better solutions and make big part of the technologies open source.

These NoSQL databases are not necessarily well structured and accept BASE properties as enough.

With these new systems, there is a change of paradigm in the way different parts of the companies collaborate. The NoSQL world has architectures that are service based (or API based or S.O.A.).



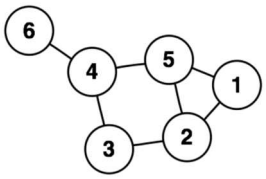
Document Database	Graph Databases
Couchbase, MarkLogic, mongoDB	Neo4j, InfiniteGraph The Distributed Graph Database
Key-value Databases	**Wide Column Stores**
redis, amazon DynamoDB, riak, AEROSPIKE	accumulo, HYPERTABLE, Cassandra, APACHE HBASE, Amazon SimpleDB

5. Graph Databases – Neo4J

Graph theory

Graph theory is a tool to solve different problems, the most famous being "the traveling salesman" problem or the "four color problem". Other applications are: optimization of the cost of wiring electronic components, finding the shortest route between two cities or the distance between all pairs of cities in a road atlas, matching and resource allocation, task scheduling and visibility coverage, also on social networks that are represented as graphs.

A **graph** is a set of nodes (or vertices or points) joined by a set of edges (or lines). Additionally, a function f is defined to map each edge to an unordered pair of vertices.



V:={1,2,3,4,5,6}
E:={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

Graph analytics technologies are used to find properties of graphs useful to study the real-world scenarios represented by the graphs themselves.

Simple graphs are graphs without multiple edges or self-loops.

A path is a sequence of vertices such that there is an edge from each vertex to its successor. A path is *simple* if each vertex is distinct.

If there is a path from a node to another, then the second node is *reachable* from the first via that path.

A path from a vertex to itself is called a cycle. A graph is called *cyclic* if it contains a cycle, otherwise it is called *acyclic*.

A connected graph is a graph in which every node is reachable from every other node, i.e. any two nodes are connected by a path. A directed graph is strongly connected if there is a directed path from any node to any other node.

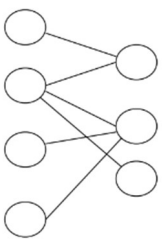
A graph is sparse if $|E| \approx |V|$.

A graph is dense if $|E| \approx |V|$^2.

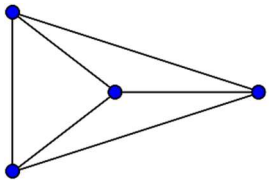
A weighted graph is a graph for which each edge has an associated weight.

A directed graph is a graph in which edges have directions.

A set of nodes can be partitioned into two sets such that no node belongs to both and the only edges are the ones connecting a node of one set with a node of the other. A graph with such structure is called a bipartite graph.

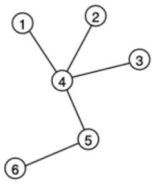


A graph is a complete graph (denoted with K_n) if every pair of vertices are adjacent. In a complete graph with n vertices, the total number of edges is $n(n-1)$.



A planar graph is a graph that can be drawn on a plane such that no two edges intersect. K_4 is the largest complete graph that is planar.

A tree is a connected acyclic graph, in which two nodes have exactly one path between them.



An Hypergraph is the generalization of the concept of graph, for which edges can connect any number of vertices. Hyperedges are then arbitrary sets of nodes that are linked.

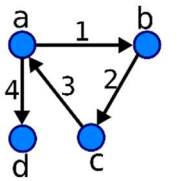
The degree of a node is the number of edges connected to that node. When dealing with directed graphs, the distinction between In-degree, the number of edges entering into a node, and Out-degree, the number of edges leaving a node, can be done. In this case, the degree of a node is the sum of the In-degree with the Out-degree.

A subgraph is a graph whose vertex and edges sets are subsets of those of another graph. A supergraph of a graph G is a graph that contains G as a subgraph.

A spanning subgraph is a subgraph that has the same vertex set of its supergraph. We say that the subgraph "spans" the supergraph.

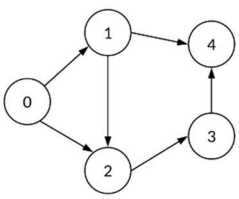
In computer science, a graph is an **abstract data type** (ADT), consisting of a set of nodes and a set of edges as some way to establish the relationship between nodes.

One way to represent a graph is to use an incidence matrix, a matrix with as dimensions the number of edges and the number of vertexes, whose elements of the form [edge, vertex] contain the information about that edge.



	1	2	3	4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1

Another way is to use the adjacency matrix, a matrix of dimensions $\#vertices * \#vertices$ containing Boolean values indicating whether the two nodes are connected or weights indicating the weight of the connection between two nodes.



Adjacency Matrix

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

Using lists, one can represent a graph using an edge list, that is a list of pairs of vertices, representing edges between the two, or adjacency lists.

Different algorithms to study graph properties do exists, they are about:

- Shortest Path
  - Single Source
  - All pairs (Ex. Floyd Warshall)
- Network Flow
- Matching
  - Bipartite
  - Weighted
- Topological Ordering
- Strongly Connected
- Biconnected Component / Articulation Point
- Bridge
- Graph Coloring
- Euler Tour
- Hamiltonian Tour
- Clique
- Isomorphism
- Edge Cover
- Vertex Cover
- Visibility

Graph databases

Relational databases, whose functioning is based on tables, are not good in managing relationships! Graph databases aim to resolve this problem, focusing in particular on efficiently managing relationships.

Graph databases are databases that use graph structures with nodes, edges and properties to store data.

They provide index-free adjacency, meaning that every node is a pointer to its adjacent element. Edges hold most of the important information and connect nodes to other nodes or to properties.

When the aim is to analyze relationships, graph databases are the best fit because of their data structure. Graph databases are very fast for associative data sets (e.g., social networks) and additionally they map more directly to object-oriented applications.

Relational database representation	Graph representation

- Sailor(sid:integer, sname:char(10), rating: integer, age:real)
- Boat(bid:integer, bname:char(10), color:char(10))
- Reserve(sid:integer, bid:integer, day:date)

Sailor

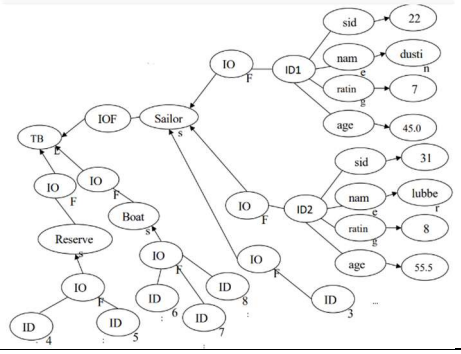
sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Reserve

sid	bid	day
22	101	10/10/96
58	103	11/12/96

Boat

bid	bname	color
101	Interlake	red
102	Clipper	green
103	Marine	red

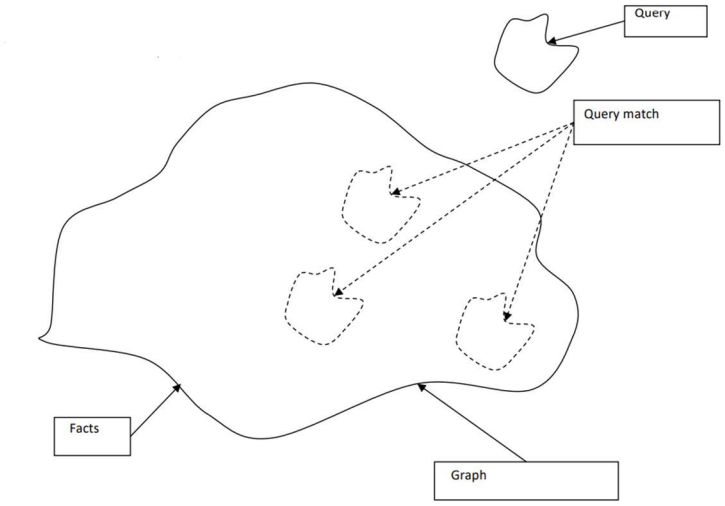


So, it's very easy to store relationships and connections between objects, together with relationships and entities attributes.

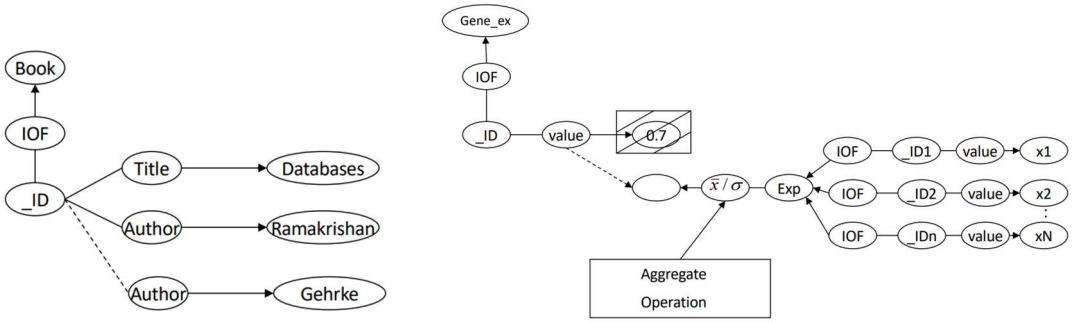
(:Sailor) –[:reserves]-> (:Boat)

The representation is in some way similar to the E-R model but has a different interpretation. In fact, in graph databases graph represent directly instances, not the structure of data.

To perform queries on graph databases, SQL cannot work because there is a different logic underneath. The approach used is that of *pattern matching*: the query describes a pattern or a shape and the database tries to find that shape inside the data. In particular, in *graph matching*, all the subgraphs with a given shape are searched for inside the graph of the instances of the database.



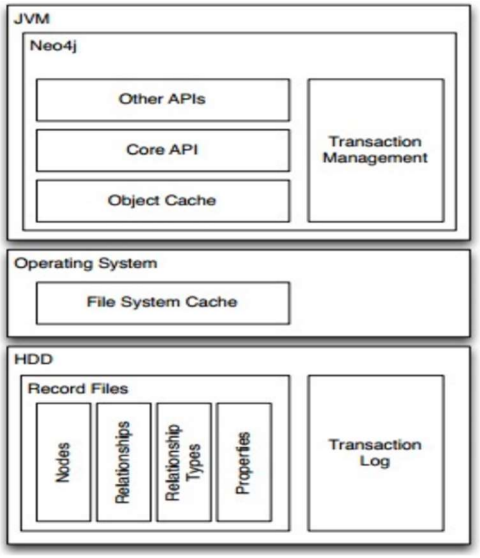
Additional advantages lie in the fact that graphs structures are easy to extend, and also easy to change.



Neo4j

Neo4j is a graph database management system, described as an ACID-compliant transactional database with native graph storage and processing. It's the most popular graph database and it's open source.

It's implemented in java, as the philosophy of its creators was to evade from all the established database monopolies, and using a language such as java goes in that direction. Support for a wide variety of languages is in any case available.



Neo4j is fully schema free, meaning that data does not have to adhere to any convention and flexibility is gained.

ACID properties are enforced by transactions.

Regarding the architecture, it's worth noticing that different kind of data is stored in separate files (Nodes are stored separately from relationships etc.). As it is difficult and expensive to look for data in disks, caching is severally exploited to keep as much graph in memory as possible.

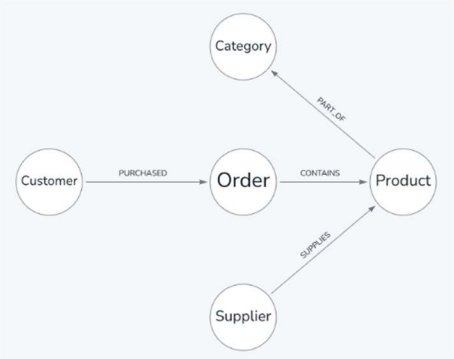
Neo4j is meant to be an operational DB and wasn't developed specifically for analytics. In fact, ACID properties are present, and the efficiency is more on single nodes than in whole graph analysis.

The data model is composed of edges and nodes with labels, making the native implementation. Additional components such as indexes and constraints can be added.

To query Neo4j, the language to use is **Cypher**, a custom proprietary language that is now under standardization. Cypher is a declarative language with the aim of making it easy to formulate queries based on relationships.

Getting started with Neo4j

A graph database stores nodes (Supplier, Product, etc.) and their relationships (e.g., Supplier SUPPLIES Product).



Other types of databases may use tables and documents, but in a graph, data is stored in the same way as you may sketch ideas on a whiteboard.

In relational databases, you are often required to add explicit joins to your queries. In doing this, relationships are calculated at runtime, which can be an expensive and slow operation. In a graph, where the relationships are stored, many powerful operations are faster and simpler.

In a relational database, you must define your schema (the structure of the data). In a graph, your data is managed without restricting it to a predefined schema. This allows more flexibility in thinking about the data and in evolving it. Graph databases are schema-free.

The nodes and relationships in your graph can have types and properties.

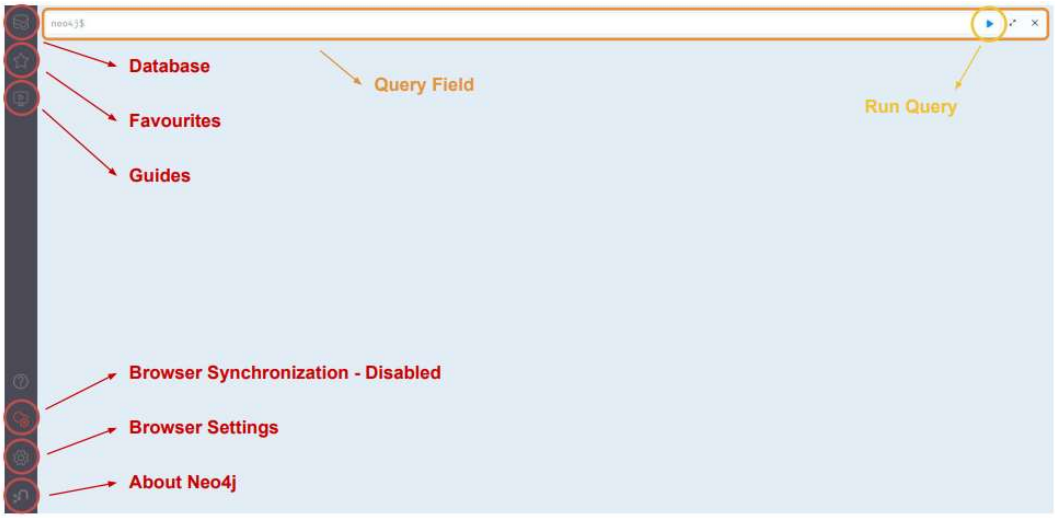
Nodes can be tagged with *labels*, representing their different roles in your dataset (e.g., Supplier, Employee, Customer). They can also have any number of key-value pairs as *properties* (e.g., name: "Camembert Pierrot").

Relationships provide directed, typed, or attributed, connections between two node entities (e.g., Shipper SHIPS Order). Relationships always have a *direction*, a *type*, a *start node*, and an *end node*. They can also have properties, just like nodes.

Nodes can have any number of relationships without sacrificing performance.

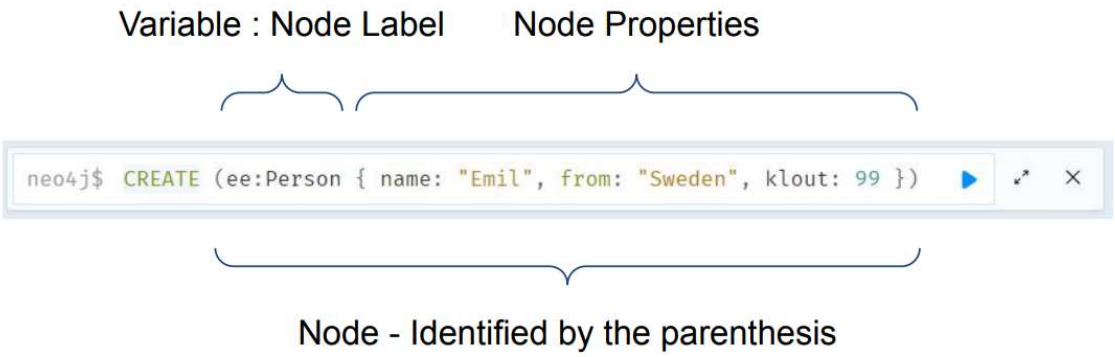
Although relationships are always directed, they can be navigated efficiently in any direction, although this may cause problems.

Neo4j interface is the following:



As previously said, Cypher is used to query a Neo4j graph, as well as update it. It focuses on the clarity of expressing what to retrieve from a graph, not on how to retrieve it, using pattern matching.

The **CREATE** clause allows the creation of nodes and relationships.



When specifying the pattern to search for in the graph

- nodes are indicated inside round brackets (…)
- attributes inside curly brackets {…}
- Relationships inside square brackets […]

The semicolon ":" identifies the following word as a Label of a node or as the type of a relationship.

It is possible to CREATE multiple nodes and/or relationships at once.

```
CREATE (ee:Person { name: "Emil", from: "Sweden", klout: 99 }),
       (a:Person { name: "Richard", from: "UK", klout: 80 }),
       (j:Person { name: "Francis", from: ["Sweden","UK"], klout: 75 })
```

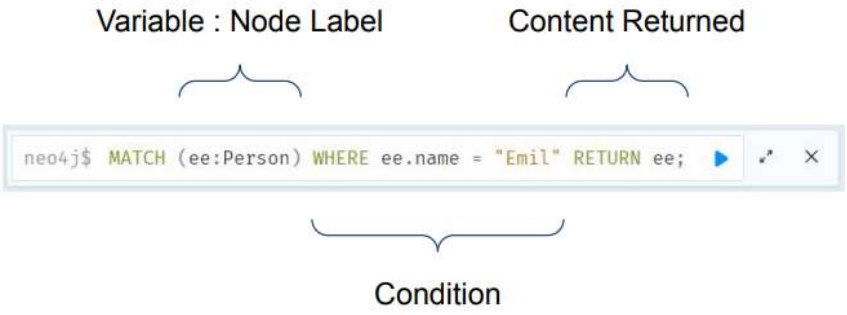
It is possible to CREATE simpler or even more complex nodes.

```
neo4j$ CREATE (ee:Person:Fisherman), (a)
```

The **RETURN** clauses return nodes and/or the properties of a node after its creation.

```
neo4j$ CREATE (a {name: 'Andy'}) RETURN a.name
```

In order to perform queries and thus return some results base on some input, the MATCH clause is used.



The **MATCH** clause returns all the nodes that match the conditions in query.

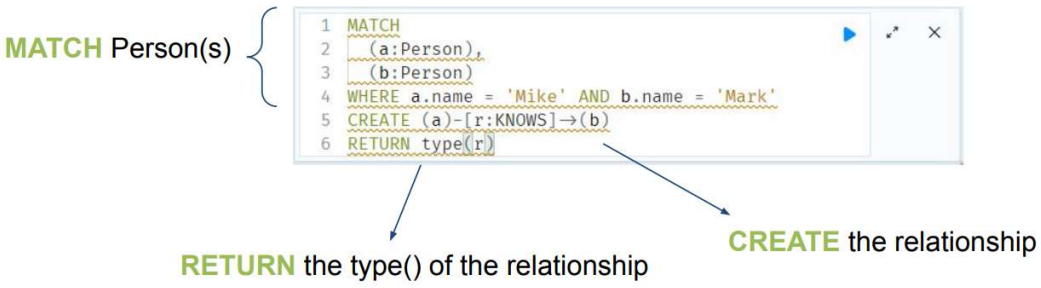
The **WHERE** clause identify all the conditions the nodes should match.

The **RETURN** clause identify what the query should return to the user.

To make an example, we create two Person(s) named Mark and Mike:

```
neo4j$ CREATE (e:Person {name: "Mark"}), (a:Person {name: "Mike"})
```

And then we CREATE the relationship KNOWS between Mark and Mike.



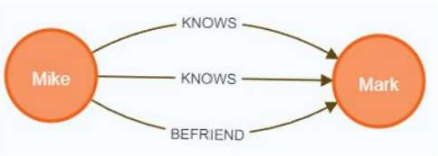
Note that the specified direction of the relationships matters with relation to the results produced by the query.

The **type()** function returns the Label of the node it has as argument.

It is possible to CREATE multiple relationships between the same entities, and relationships can also be created with properties.

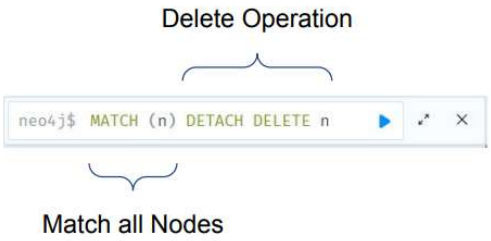
```
MATCH
  (a:Person),
  (b:Person)
WHERE a.name = 'Mike' AND b.name = 'Mark'
CREATE (a)-[r:KNOWS {since: 2015}]→(b), (a)-[h:BEFRIEND]→(b)
RETURN type(r)
```

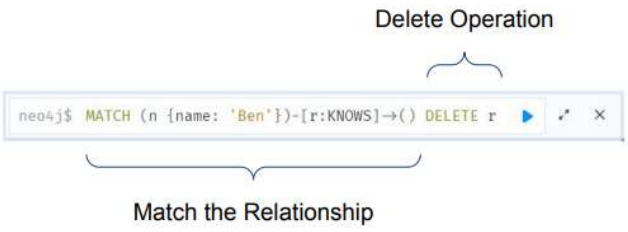
The two KNOWS relationships we created are different even though they have the same name



So, it's possible to create different relationships between two nodes with the same name, but of course this is a bad practice, except in some cases when the created relationships have different attributes, and this fact may be useful.

The **DELETE** clause allows the removal of nodes and relationships.



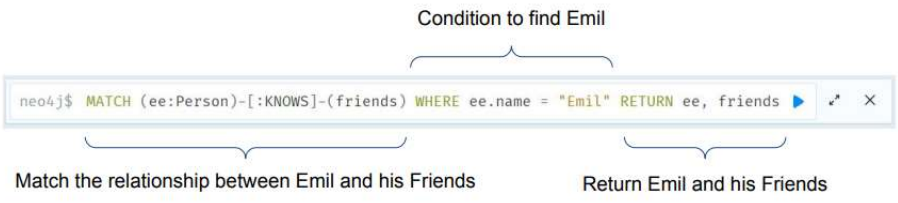


DETACH removes all the relationships before removing the nodes.

When deleting nodes, leaving behind relationships that are not connected to nodes leads to errors thrown in Neo4j. We can either: delete the relationships first and then the nodes, or use DETACH DELETE instead of only DELETE to automatically delete the relationships attached to the deleted nodes.

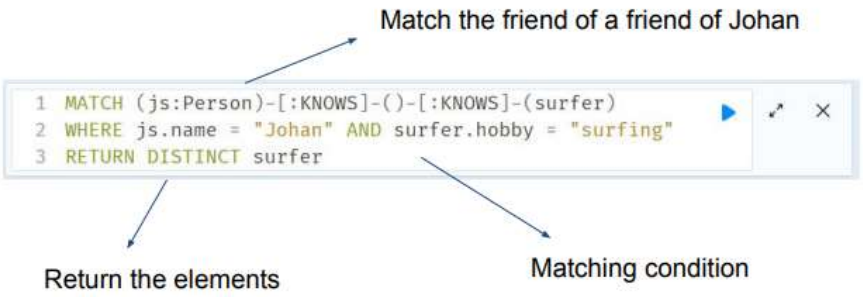
Note that writing () means that any node can be taken, without conditions.

Let's find all of Emil's friends:



Note that the relationship *:KNOW* is undirected because we assume that friends both know each other's, but usually relationships in graphs are directed. Forgetting to specify a direction can cause problems in the results of a query, because it will collect and return more relationships than the ones wanted.

Let's find a new friend that can teach surf to Johan:

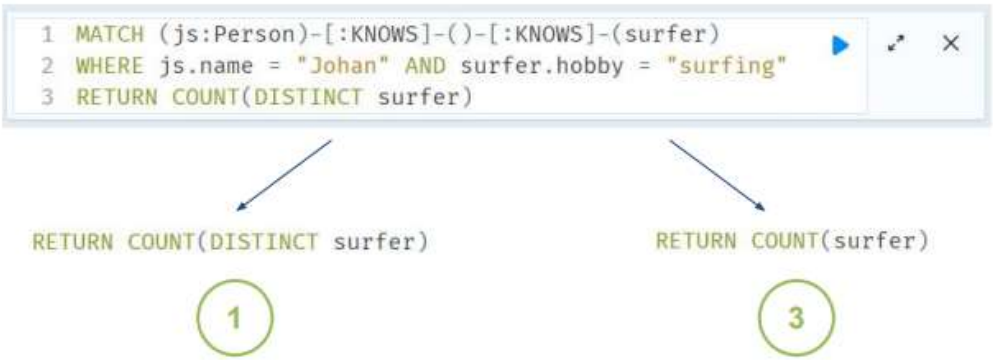


The keyword **DISTINCT** is needed because we want to avoid the case in which we get to the same person through a different friend and this person would be returned more than one time.

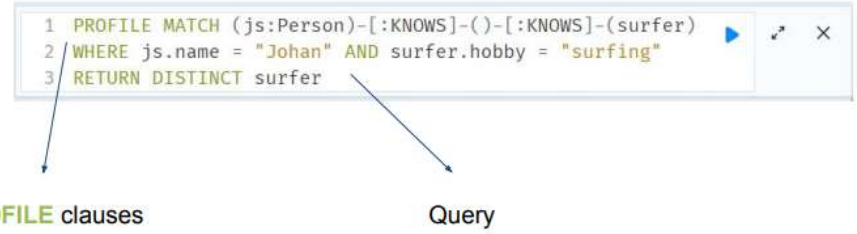
Note that in this query assumptions are made. First, we are assuming that the graph is made of people and that each node is a person, so we are not specifying the labels for all the nodes. Secondly, we are assuming that every node connected to a person with the :KNOWS relationship is a person.

Generally talking, it's better to avoid making assumptions and prefer writing queries as specific as possible.

Going back to the example, if there is only one friend that is a surfer, we would expect to get only one result. Let's use the COUNT clause to get the amount of nodes returned by the query when using the DISTINCT clause and when not doing it:

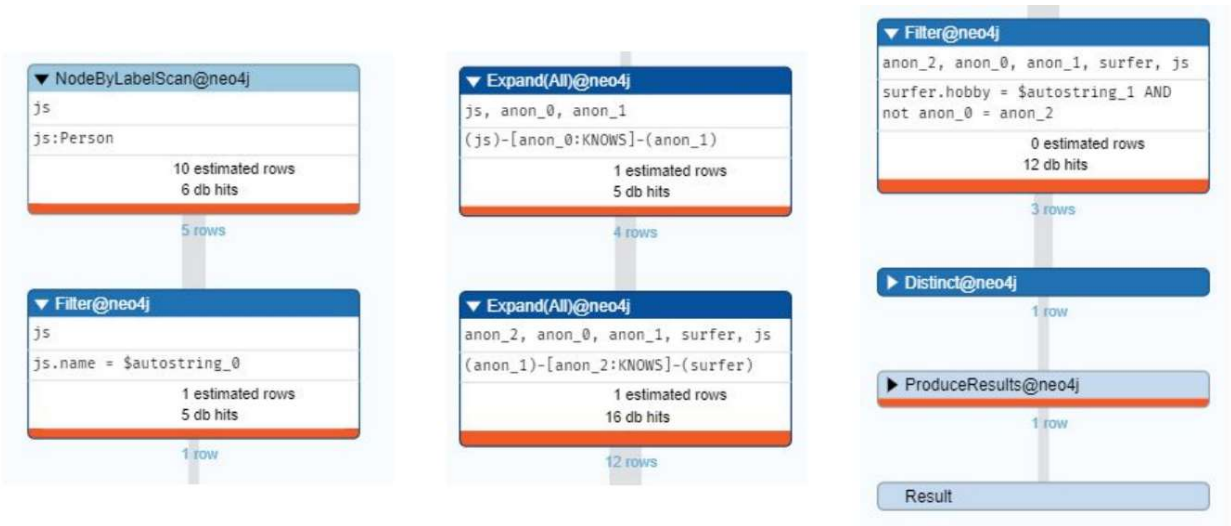


The **PROFILE** provides the complete set of operations provided to perform the query.



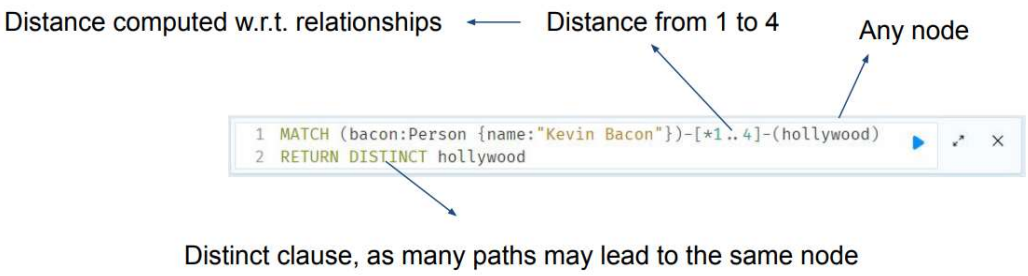
It's important to understand that Neo4j queries are executed as a pipeline, so that operations are performed sequentially to get to the results.

Let's see the execution of the least query, run with the profile:



1. The DB identifies all the nodes whose label is 'Person'
2. The DB filters the nodes from the previous step, removing those whose name is not equal to the autostring that in this case is 'Johan'. (Autostrings are values used by Neo4j to perform operations)
3. The DB looks for all the relationships from the nodes filtered at the last step to other nodes
4. The DB looks for the relationships from the nodes found connected to the initial person, found at last step, to other nodes and calls these new nodes 'surfer'
5. The DB filters 'surfer' nodes and checks that they have surfing as a hobby. It also checks that we are not connecting nodes already considered. (this check is *not anon0 = anon1*)
6. The DB checks that this result set of nodes does not contain equal nodes
7. The results are prepared
8. The results are returned

Neo4j embeds the concept of **nodes distance**, allowing the user to specify a range of distances between nodes in terms of number of relationships to cross.

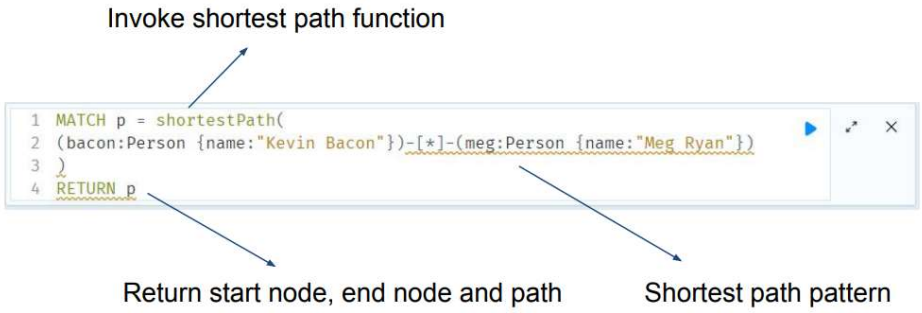


The * symbol can be used to allow any distance, while a precise interval with lower and upper bound can be expressed with the syntax: **lower…upper*

Note that in the example we are not putting any constraint on the nodes or relationships we are crossing with the hops we are making to reach our destination node. When there is only a kind of

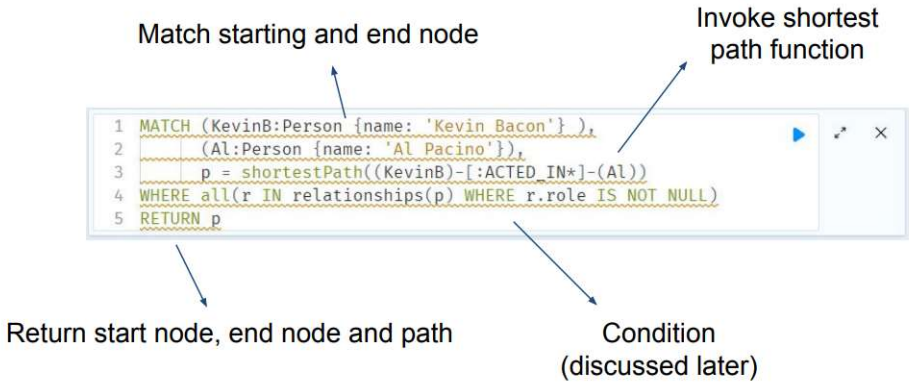
relationship this is fine, but when relationships of heterogeneous kinds are present problems may arise.

The shortest path between two nodes can be also found with the **shortestPath()** function:

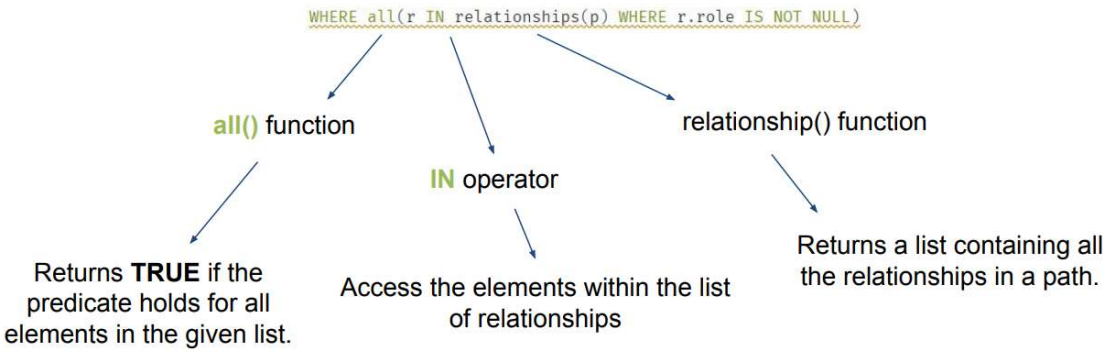


The shortest path between the two specified nodes is found and assigned to a variable, called *p* in this case.

It's possible to specify the kind of relationship to move through:



The **all()** function is used to check a condition on more instances,



Using **WITH**, it's possible to manipulate the output before it is passed on to the following query parts.

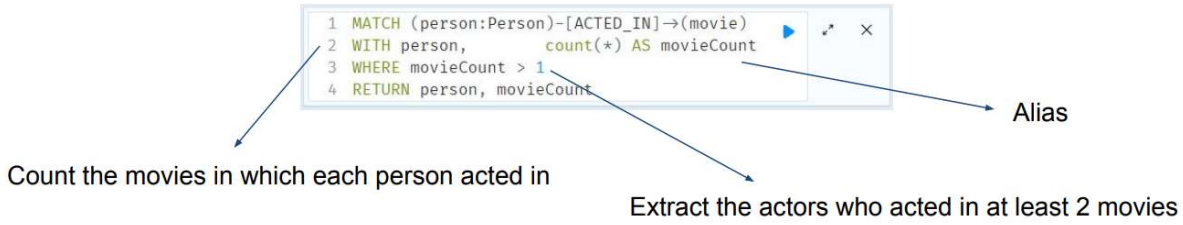
it is usually combined with other clauses, like

- **ORDER BY** - Sort the result of the query.
- **LIMIT** - Limit the amount of results provided by the query

It can also be used to

- Introduce aggregates which can then be used in predicates in WHERE.
- Alias expressions that are introduced into the results using the aliases AS the binding name.
- Separate reading from updating of the graph.

Using the with statement, it's possible to filter based on a value of an aggregate function.

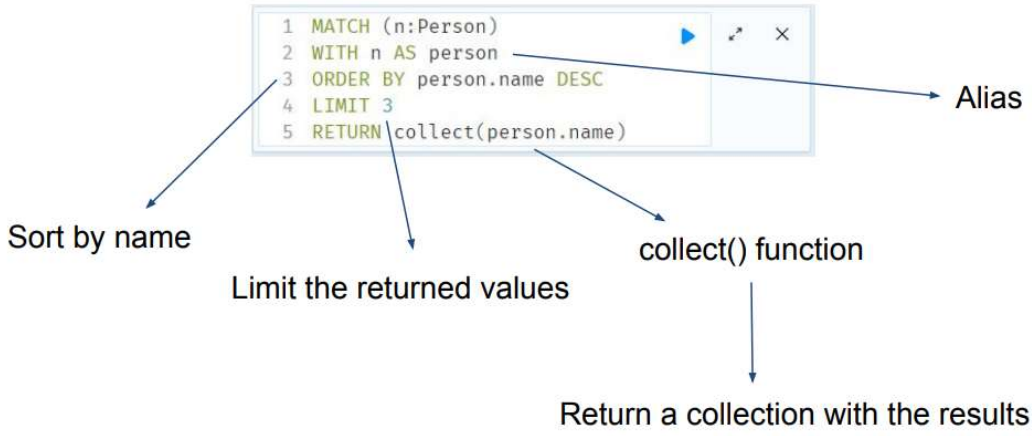


In the example, for every person which acted in a movie, the query counts how many patterns it has found (so how many movies that person has acted in), and filters out the person(s) who acted in 1 or less movies.

count(*) counts the number of found matching patterns.

When using the WITH statement, the query will afterwards only be able to read the variables stated within the WITH clause. For example, if the person variable wouldn't have been included within the clause, I wouldn't have been able to RETURN it and an error would have been thrown. This happens because of the pipeline-like execution of queries in Neo4j.

The following is an example of the use of ORDER BY and LIMIT clauses, that are used in combination with the WITH clause.



The **collect()** function returns a collection with the results in form of array, instead of the usual list in tabular form that is returned by the RETUN without the use of this function.

Sometimes when Cypher queries are performed, *WARNINGS* may arise. They may arise for different reasons. For example, some operators may be deprecated or the query is performing a lot of operations e.g., Cartesian products among the nodes or unbound number of relationships.

Advanced Cypher

As a last note, we must be careful when using the shortest path in Neo4j, as the kind of algorithm utilized can influence the results. The choice of algorithm depends on the predicates to be evaluated in the queries.

By default, Neo4j uses a **Fast Bidirectional Breadth-first Search Algorithm** if the conditions can be evaluated whilst searching for the path.

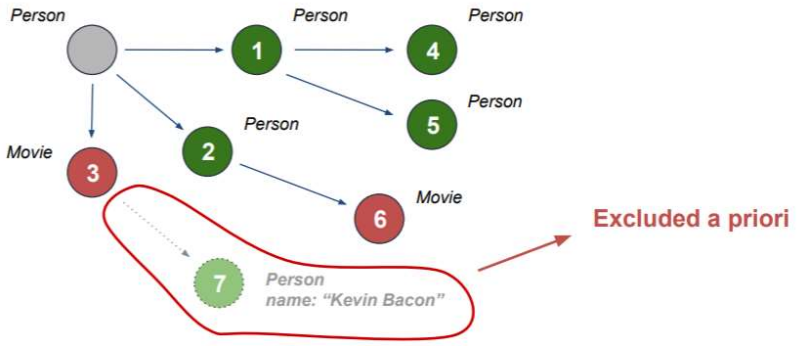
e.g., all nodes must have the Person label.
e.g., no nodes should have a name property.

If the predicates need to inspect the whole path before deciding on whether it is valid or not, Neo4j may have to resort to using a **Slower Exhaustive Depth-first Search Algorithm** to find the path.

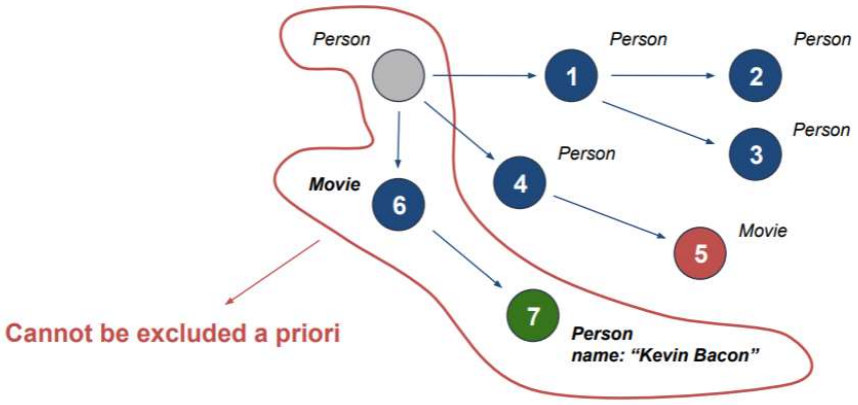
e.g., at least one node contains the property name = “Kevin Bacon”.

When the **Exhaustive Search** is planned, it is still only executed when the Fast Algorithm fails to find any matching paths.

Fast Algorithm - All nodes must have the *Person* label.



Exhaustive Algorithm - At least one node contains the property *name* = *“Kevin Bacon”*.



When the Fast algorithm is used, it means that the conditions can be evaluated as the function is performed. Predicates used in the WHERE clause that apply to the shortest path pattern are evaluated before choosing the shortest matching path.

When the exhaustive algorithm is utilized, it means that Neo4j must check that the whole path follows the predicate before validity is evaluated.

```
MATCH (KevinB:Person {name: 'Kevin Bacon'} ),
      (Al:Person {name: 'Al Pacino'}),
      p = shortestPath((KevinB)-[:ACTED_IN*]-(Al))
WHERE all(r IN relationships(p) WHERE r.role IS NOT NULL)
RETURN p
```

Fast Algorithm
Always returns a result, if exists

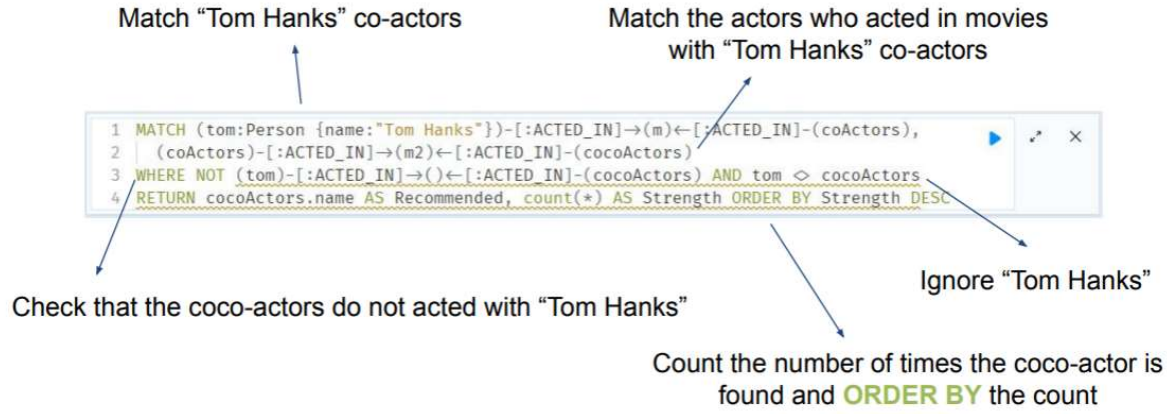
```
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
      (Al:Person {name: 'Al Pacino'}),
      p = shortestPath((KevinB)-[*]-(Al))
WHERE length(p) > 1
RETURN p
```

Exhaustive Algorithm
Always returns a result, if exists

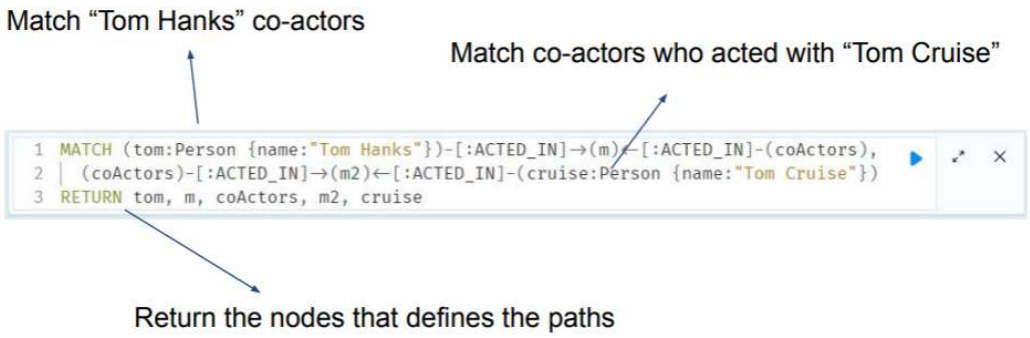
```
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
      (Al:Person {name: 'Al Pacino'}),
      p = shortestPath((KevinB)-[*]-(Al))
WITH p
WHERE length(p) > 1
RETURN p
```

Fast Algorithm
Not guaranteed to return a result, even if it exists.

Find actors that "Tom Hanks" hasn't yet worked with, but his co-actors have. Find someone who can introduce Tom to his potential co-actor.



Find someone to introduce Tom Hanks to Tom Cruise



6. Document Databases – MongoDB

While in graph databases relationships are more important than objects, document databases focus on the object as the most prominent feature of the data.

Some document databases are MongoDB, Couchbase, MarkLogic. We will focus on MongoDB.

Data is stored in the form of **documents**, and every document can have a different and complex structure with many values and inner elements in sub-levels of the object.

The reason for this choice is that, when dealing with business objects in a business context, problems may arise from the granularity of stored data (especially with relational models). Document databased aim at overcoming the limits of generic databases where granularity is fixed at a very small level for which every table contains elements at a small granularity. Instead, document databases keep a granularity and a level of containment of data that is as aligned as possible with the user requirements. So, document databases handle objects of the dimensions of business objects at the exact granularity used by business operations.

Documents in the database correspond to business objects and this may however lead to overlapping of information between documents. The problem to transform data in a consistent and coherent way is well present.

The advantages, apart from the ones already disclosed, are that schema changes are handled very well, basically because document databases have a schema-less approach in which the schema is completely determined by the user requirements when creating the documents. This means that there are no issues of flexibility. Secondly, saving data in document objects solves the problem of impedance mismatch with object-oriented programming. Lastly, the use of JSON to store the documents simplifies the interaction of the database with programming languages.

A document contains:

- Many fields
- Different structures
- Sub-elements
- Sub-documents
- Sub-objects

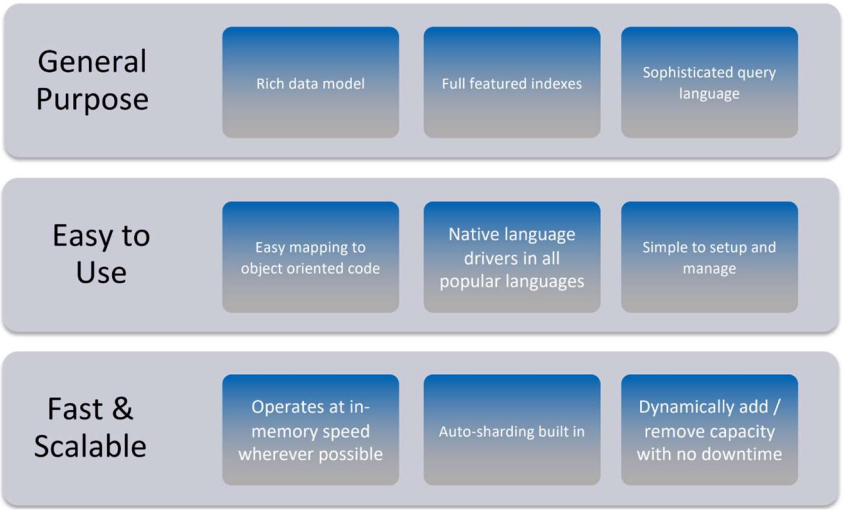
A document is an independent and self-standing object. There is no need for joins to get the data, as documents are already complete.


```
{
"business_id": "rncjoVoEFUJGCUoC1JgnUA",
 "full_address": "8466 W Peoria Ave\nSte 6\nPeoria, AZ 85345",
"open": true,
"categories": ["Accountants", "Professional Services", "Tax Services",],
 "city": "Peoria",
"review_count": 3,
"name": "Peoria Income Tax Service",
 "neighborhoods": [],
 "longitude": -112.241596,
 "state": "AZ",
"stars": 5.0,
"latitude": 33.581867000000003,
"type": "business"
}
```

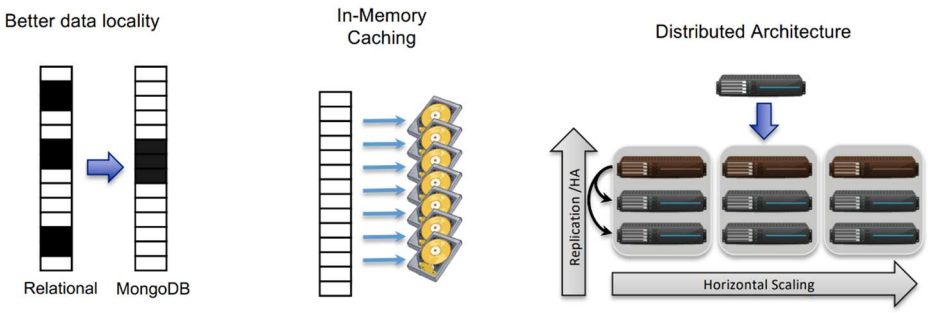

As mentioned, documents are saved in **JSON** format (Java Script Object Notation). A JSON file is a text file containing the information in textual form, and every document is a sequence of attributes. Every attribute is composed by a name and a value. There is no need to specify types for data.

Text files are however space-expensive to store and operations on them are also not efficient. To overcome this problem, documents are not actually saved as text files but in their binary format, called **BSON** format. BSON is the Binary-encoded serialization of JSON-like documents, optimized for space and speed.

MongoDB is an open source and document-oriented database, allowing dynamic schemas and performing automatic data sharding, designed with both scalability and developer agility.



MongoDB aims to be fast and scalable, with high performance and high scalability.



To enable big scalability level, the key concept utilized is the one of sharding.

Sharding consists in data partitioning, that in NoSQL databases is a bit different than traditionally because it's fully automatic, configurable, transparent in the sense that queries are written as if the database was in a single place and follows the BASE principles.

Sharding enables scale, geo-locality (optimize with respect to geography by putting data near where they are needed), hardware optimizations and lower recovery times.

To decide where to put data, the systems performing sharding make use of shard keys, defined by a data modeler that describes the partition space of a data set. Data is partitioned into data chunks by the shard key, and these chunks are distributed evenly across shards that reside across many physical servers. Some rules, defined based on the designer experience on the data, determine how to spread the data by using the shard key. For example, the attribute "country" of the saved users can be used as a shard key to decide whether to put the data in a database in Europe or in the USA.

MongoDB implements sharding through horizontal partitioning. The list of documents and collections of documents are split across different places, but the single document is never split, even if it contains sub-documents.

Partitioning enables MongoDB to increase the number of documents easily.

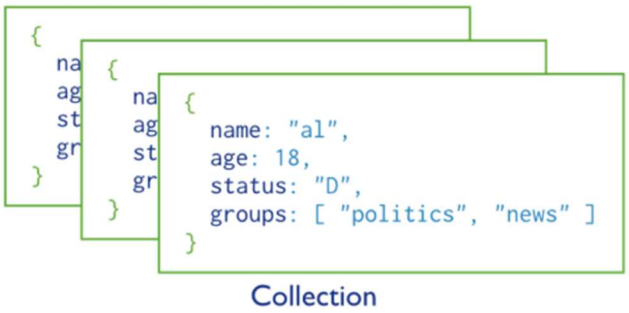
Security in MongoDB makes mainly use of SSL(Secure Sockets Layer) between client and server and in intra-cluster communication and authorization mechanisms are present at the database level.

To better understand MongoDB, we can make a comparison between the concepts in a SQL database and in a MongoDB database:

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document
column	field
index	index
table joins (e.g. select queries)	embedded documents and linking
Primary keys	_id field is always the primary key
Aggregation (e.g. group by)	aggregation pipeline

Documents are saved into **collections**, that are containers of documents. A collection has no semantical or syntactic meaning, it's just an "empty box" that must be defined before the documents it will contain.

There are no constraints on the type of documents to put in a collection: its' possible to put extremely heterogeneous documents in the same collection. However, best practices suggest putting documents that have some kind of similar meaning together.



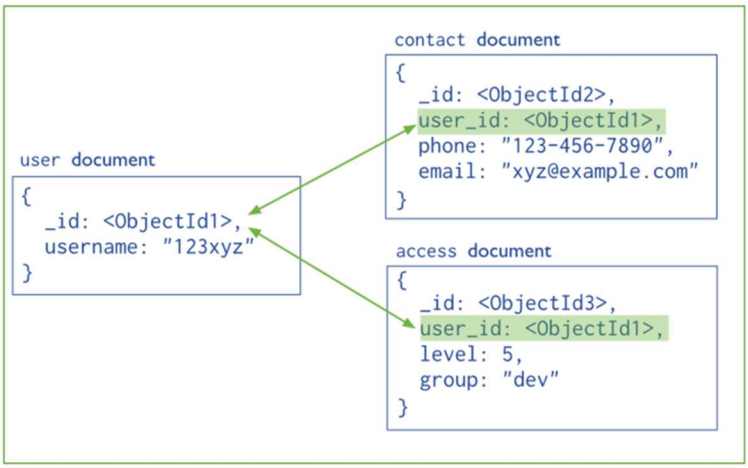
The structure of each document is decided by the designer according to the business needs and to his expertise. Data are designed based on how the people are going to read them. Each field of a document may contain native data types, arrays or other documents contained in the main document.

The only rule is that every document must have an id, usually indicated by the field *_id*. The id is a technical identifier, it is not the user that decides how to univocally identify a document.

In a document no NULL value can be found, because an attribute is created with its value and if there is no value to put, then it's not possible to create the field.

Documents embedded in other documents:

There is still a way to connect documents with each other, and consists in linking documents using a reference to their identifier:



Reason suggests however to use this structure of linked documents as less as possible and to try to create documents as independent as possible as the document databases' philosophy contemplates.

In fact, contrarily to relational technologies where the approach is that of schema on write and so every query needs to take different pieces of the schema and to put them together to formulate the answer, a document is a sub-selection of the data and it's like a consolidated and stored query in which no data should need to be put together. The structure of a document is the one that is used most of the times, just opening a document instead of needing to write a query.

MongoDB queries overview and CRUD operations

As NoSQL data structures are driven by application design, we need to consider necessary CRUD operations to handle this structure. In addition, the shape of documents is dynamical and potentially in continuous change.

While SQL and Cypher are declarative query languages, for MongoDB there is not a query language but only a list of operations to be used by developers in their object-oriented programs with dedicated libraries to interact with MongoDB.

To CREATE a collection:

db.createCollection(name, options)

To insert a document:

db.<collection_name>.insert({"name": "nguyen", "age": 24, "gender": "male"})

An insertion can happen only inside a collection, and every time an *insert* is run a new document is created as result.

Queries are always run in a single collection at a time, thus the importance of putting documents in collections in a coherent way.

To take all the documents in a collection:

db.<collection_name>.find().pretty()

Queries return entire documents.

Conditions can be added:

db.<collection_name>.find(
{ "gender": "female", "age": {$lte:20} }).pretty()

Conditions are written as parameters to the find() function and are written in a document-like format. We are practically passing a document as parameter to the find() to express the conditions. This means we are using a pattern matching paradigm in which the document database inspects the collection and looks for documents matching the pattern provided.

In the example above, it can be seen how the value associated with name "age" is a sub-document inside the document used to express conditions, to embed "entities html" to add conditions such as "less than 20".

It's interesting to note a sort of correspondence to the relational world, by making a mapping to SQL:

SQL Statement	MongoDB commands
SELECT * FROM table	db.collection.find()
SELECT * FROM table WHERE artist = 'Nirvana'	db.collection.find({Artist:"Nirvana"})
SELECT* FROM table ORDER BY Title	db.collection.find().sort(Title:1)
DISTINCT	.distinct()
GROUP BY	.group()
>=, <	$gte, $lt

It can be noticed that while SQL has a declarative approach, MongoDB has a procedural approach (performed via APIs) that consists in expressing a sequence of steps one after the other.

Available comparison operators are:

Name	Description
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a loginal AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

A further and useful feature when reading is the possibility to work with aggregates. There are SQL-like aggregation functionalities that allow to pipeline documents from a collection through an aggregation pipeline in which output documents are produced based on calculations performed on input documents.

An example is the following:

db.parts.aggregate ({$group : {_id: type, totalquantity : { $sum: quanity} } })

Another feature is the Map-Reduce:

db.collection.mapReduce(, , { out: , query: , sort: , limit: , finalize: , scope: , jsMode: , verbose: })

Moving on to updating data, the commands look like the below:

db.<collection_name>.update(<select_criteria>,<updated_data>)

db.students.update({'name':'nguyen'}, { $set:{'age': 20 } })

In the example, in the collection students, the documents with a certain condition on the "name" are picked and on them a replacement is performed in the field "age".

To, instead of updating some fields, replace the existing document with new one, there is the save method:

db.students.save({_id:ObjectId('string_id'), "name": "ben", "age": 23, "gender": "male"}

Deleting documents, collection and database is of course also possible:

db.<collection_name>.remove({"gender": "male" })

db.<collection_name>.drop()

Show database: show dbs
Use a database: use <db_name>
Drop it: db.dropDatabase()

MongoDB architecture

It's possible to distinguish between two kinds of processes:

- Mongod: the database instances, the main daemons that manage data, get queries, and return data.
- Mongos: the sharding processes and sharding managers, aware of where the data is located.

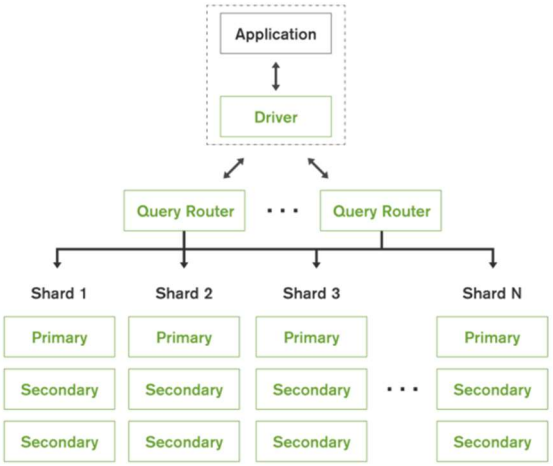
Mongos are analogue to database routers/balancers and process all requests by deciding how many and which Mongod should receive the query, then collect the results and send them back to the client. Note that the presence of Mongos adds and overhead to operations.

It's possible to have architectures with more than one Mongos.

To facilitate access to documents, indexes are used in the form of B+ tree indexes. An index is automatically created on the _id field (the primary key), but users can create other indexes to improve query performance or to enforce Unique values for a particular field.

Partitioning happens through sharding using shard keys that are defined by the users. To distribute documents across shards, by default MongoDB automatically splits and migrates chunks when the default max size of 64mb is reached, but automatic rebalancing can be enabled so that the system can perceive if a server is overloaded and decide to move the data or create a new server when needed.

As already said, a **Shard** is a node of the cluster, consisting of a single Mongod or a replica set composed of data distributed across more physical servers.

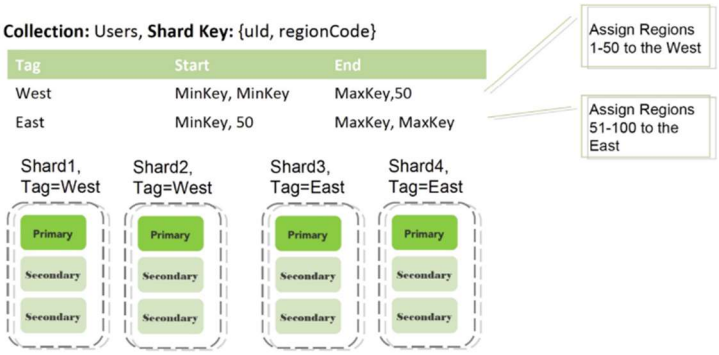


Replication has the aim to improve robustness: when deploying a MongoDB system the designer can choose how many copies of data to keep in different disks as far as possible from one another. The rule of thumb is to have 3 copies with a main one among them. Every node of the replica set has a local Mongod server that has 1 copy of the data of interest.

Possible sharding strategies are:

- Range sharding: shards are split based on sub-range of a key (or also multiple keys combined). For example: "*if attribute A is in range (a,b) then put the document in shard 1, if in range (b,c) put it in shard 2*". However, the distribution of a key may not be uniform, causing the distribution not to be balanced.

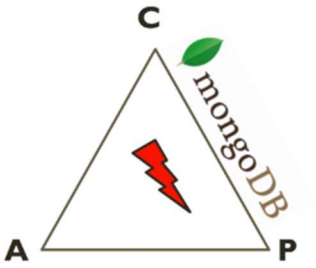
- Hash sharding: shards are split based on the MD5 hashed value of the used key. This approach solves the problem of non-uniform distribution of a key, because data is distributed randomly within the range of MD5 values.
- Tag sharding: Tag-aware sharding allows subset of shards to be tagged and assigned to a sub-range of the shard-key.



The choice of the sharding strategy to use depends on the use cases:

Usage	Required Strategy
Scale	Range or Hash
Geo-Locality	Tag-aware
Hardware Optimization	Tag-aware
Lower Recovery Times	Range or Hash

As a final remark, MongoDB positions itself on the CP side of the CAP theorem, ensuring consistency and partition tolerance, but giving no complete guarantee of availability.



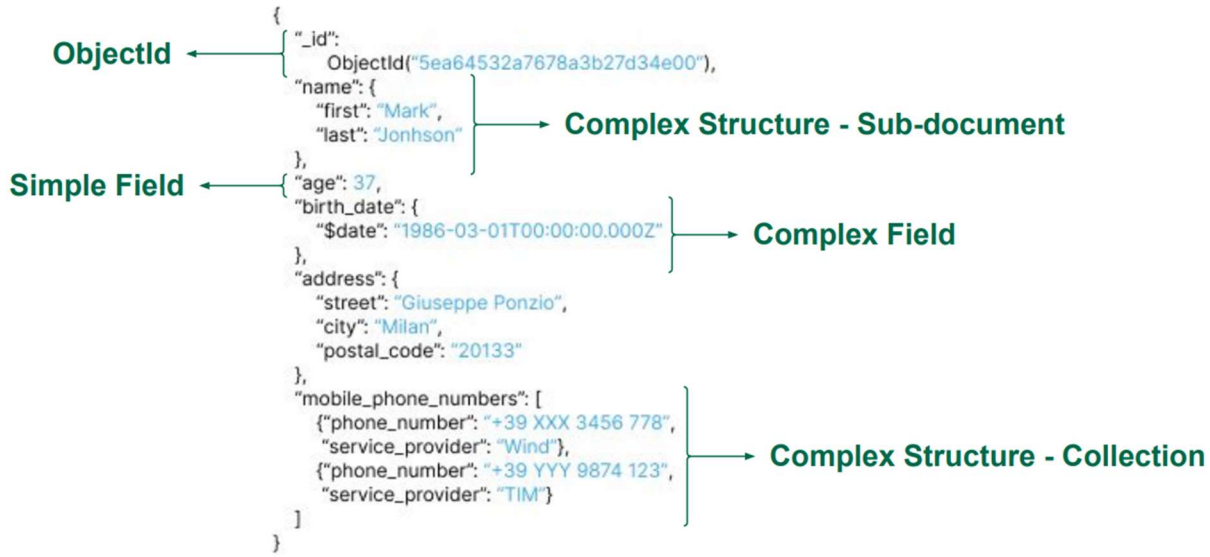
Getting started with MongoDB

MongoDB is a document-oriented database that stores data within Collections as Documents.

Collections contain sets of documents. Databases are made by one or more collections.

Documents consist of key-value pairs which are the basic unit of data in MongoDB. They are JSON-like structures that can contain different types of data and sub-documents and are not rigid.

To every document is assigned and unique object id. This id is assigned only to the main structure of a document, not to its sub-structures.



ObjectId is the type associated with the predefined field created by MongoDB to uniquely identify the documents within a collection, like a Primary Key in a relational database. Such field is always named _id.

The 12-byte ObjectId value consists of three different elements: a 4-byte timestamp value, representing the value creation, measured in seconds since the Unix epoch, a 5-byte random value generated once per process that is unique to the machine and process, a 3-byte incrementing counter, initialized to a random value.

A document can be added to a collection using the **InsertOne(...)** method.

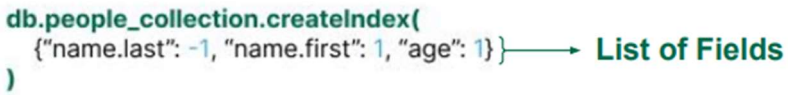
Multiple documents can be added using the **InsertMany(...)** method instead.

While the first one accepts only one document, the latter may involve a list of comma-separated documents to be added to a specific collection.

In the example above, “people_collection” is the name of the collection in which we are inserting a document.

Indexes are data structures that store a small portion of the collection's data set in an easy to traverse form, ordered by the value of the field. The objective of Indexes is to support the efficient execution of some types of queries. With indexes, it is possible to create and store some ordering on the fields of the database.

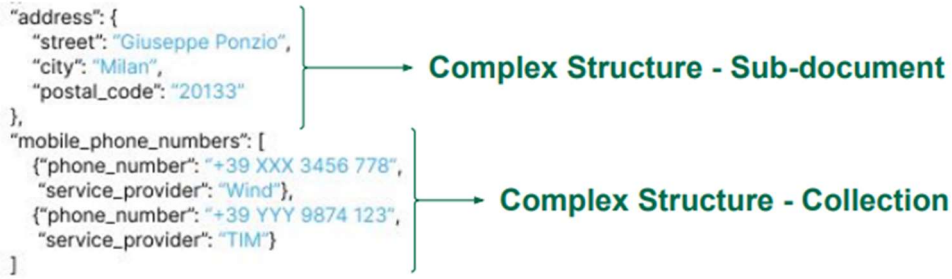
Indexes are created with the **createIndex(...)** operator which accepts a list of the fields with respect to which create the index and their corresponding ordering, i.e., ascending (1) or descending (-1).



When creating an index, we store some data in the database, that is the data about the index itself like information about the sorting of the data.

Documents can contain complex structures, like sub-documents or even collections of documents. In both cases, access to these structures is achieved through the dot notation. An example could be to write: *address.street*

When accessing a sub-document, the chosen attribute is accessed directly. Instead, when accessing collections, the chosen attribute is accessed for each of the documents included in the collection.



How to look for data? A document can be collected using the **findOne(...)** method. It collects the first document that satisfies one or more conditions defined in a filter, that is a set of conditions to put on the documents to return.

Multiple documents can be collected using the **find(...)** method. It behaves exactly like its individual counterpart, although it collects all the documents rather than the first one.

Whenever it is necessary to return the number of documents collected instead of the documents themselves, the **countDocuments(...)** method can be applied.

A document can be updated using the **updateOne(...)** method. It collects the documents that satisfy one or more conditions defined in a filter and updates the first one found according to a list of comma-separated fields' updates.

Multiple documents can be updated using the **updateMany(...)** method. It behaves exactly like its individual counterpart, although it updates all the collected documents rather than just the first one.

A document can be deleted using the **deleteOne(...)** method. It collects the documents that satisfy one or more conditions defined in a filter and deletes the first one found.

Multiple documents can be deleted using the **deleteMany(...)** method. It behaves exactly like its counterpart, although it deletes all the collected documents.

When collecting documents, it is possible to restrict, explicit, or expand the fields to be returned through projections. Projections are lists of key-value pairs made by the field name and a boolean value representing whether the field will be returned (1) or not (0). Among the values listed in a projection, the ones with a 1 will be returned, while the ones with a 0 won't.

Whenever a list specifies a subset of fields to be returned, the other ones won't be returned. Conversely, whenever a list specifies a subset of fields not to be returned, the other ones will be returned by default. This way, instead of specifying whether to return or not every element, it's possible just to list the ones to return or the ones not to return.

Furthermore, it is possible to shape projections to include fields from subdocuments and arrays or create new fields.

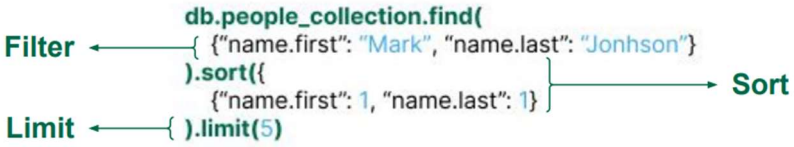
In the example, we are stating what fields to keep and return. I particular, we are saying we want to return the birth date, the phone number and create a new temporary field which includes the year of the birth date.

The execution is the one of a pipeline: in the example the database first takes all the documents in the collection that respect the filter, so the document about "Mark Jonhson", and then removes the attributes that are not in the projection from these documents before returning the results.
When performing any find(...) operation, it is important to notice it can only perform filters and projections in that exact order. Hence, it won't be possible to project and then filter.

When collecting documents, it is possible to sort and limit the results. These operations can be performed through the $sort and $limit stages or using the sort(...) and limit(...) methods.

The **sort(...)** method (and its equivalent stage) accepts a list of fields and their ordering, i.e. descending (-1) and ascending (1). The earlier a field is referenced, the more relevant it is for the ordering.

The **limit(...)** method (and its equivalent stage) accepts a number representing the number of elements to collect.



When collecting documents, these are filtered based on conditions evaluated on the value of their fields. Several types of operators can be employed in filtering stages.

Logical Query Operators: operators that return documents based on expressions evaluated as true or false.

- **$and** – returns documents that match all the conditions of multiple query expressions.
- **$not** – returns documents that do not match the conditions of a query expression.
- **$nor** – returns documents that do not match at least one condition of multiple query expressions.
- **$or** – returns documents that match at least one condition of multiple query expressions.

Comparison Query Operators: operators that return documents based on value comparisons.

- **$eq** – matches values equal to a specified value.
- **$gt** (**$gte**) – matches values greater (greater or equal) than a specified value.
- **$lt** (**$lte**) – matches values smaller (smaller or equal) than a specified value.
- **$in** – matches any of the values specified in an array.
- **$ne** – matches values not equal to a specified value.
- **$nin** – matches values not contained in a specified array.

Element Query Operators: operators that return documents based on field existence or type.

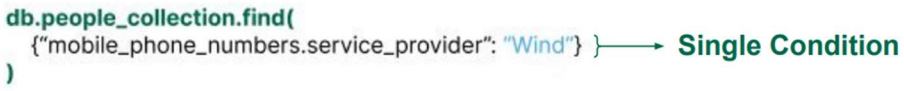
- **$exists** – matches documents with a specified field.
- **$type** – matches documents whose chosen field is of a specified type.

It's important to remark the difference between a field not existing, meaning that it is not included in the document, and a filed being empty, meaning that it is included in the document but has no assigned value (which is impossible in a document based database).

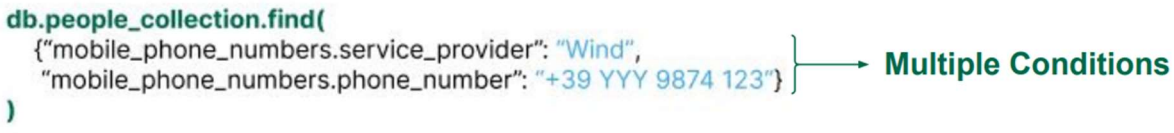
Evaluation Query Operators: operators that return documents based on evaluations of individual fields or documents.

- **$text** – matches documents based on text search on indexed fields.
- **$regex** – matches documents based on a specified regular expression.
- **$where** – matches documents based on a JavaScript expression.

Filtering operations may behave differently based on the type of complex field a query is accessing (e.g., subdocuments, arrays, etc.). Queries evaluating one or more conditions on the fields of a subdocument field are not subject to any behavior change. On the other hand, queries evaluating a single condition on the fields of the documents of an array will return the main document if *at least one* of the documents in the array satisfies the condition.



Whenever multiple conditions are evaluated on the documents in an array field, they will be assessed individually on the array's documents, hence returning the main document if, for each condition, there exists at least one document that satisfies it. It doesn't matter whether there's only one document satisfying all conditions or multiple documents satisfying one each.



The execution for the previous example would be:

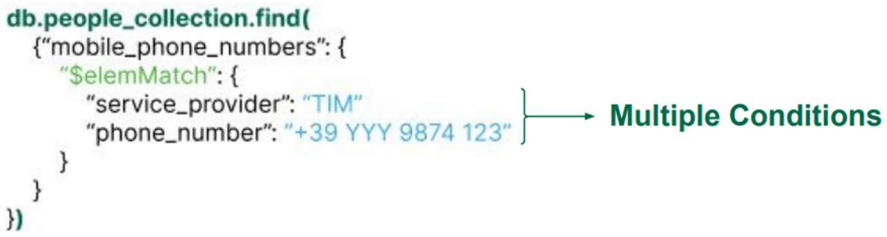
1. From the documents in the collection, filter out all documents that do not have in the array of the mobile phone numbers at least one subdocument having as service provider the value "Wind".

2. From the remaining documents, filter out all documents that do not have in the array of the mobile phone numbers at least one subdocument having phone number of the value “+39 YYY 9874 123”.

So, if we have a document having 3 mobile phone numbers as sub-documents, among which one has “Wind” as operator but doesn’t have the specified number, one has that number but a different operator, and one doesn’t match any of the two conditions, then the main document containing the three will be returned.

Not in this case, but in general, the order in which conditions are specified does count and makes a difference in the results.

Whenever a query is targeted at evaluating multiple conditions on the fields of the same document of an array, it is necessary to apply the **$elemMatch** stage. It matches documents containing an array field with at least one document that satisfies simultaneously all the specified query criteria.



When retrieving information, it is possible to work with aggregation in order to perform some operations on the data.

When a collection is made of documents containing arrays, retrieving the array’s content may be useful. Applying the **$unwind** stage can achieve such an outcome. It shapes the collection so that each document is replaced with a set of new ones, i.e., one for each element in the document’s array on which the unwind stage is applied. These new documents contain all the fields from the main one and a field with the name of the array field that contains one of its documents.

In the example, we are unwinding the main document in more documents, each one containing one sub-document from the mobile phone numbers array and the rest of the main document, instead of returning one main document containing the array of mobile phone numbers sub-documents.

When applying $unwind or $group stages, it is necessary to apply the **aggregate(...)** method, i.e., a method to compute aggregate values for the documents in a collection.

Aggregate operations, i.e., operations aimed at grouping with respect to one or more fields, are achieved by applying the **$group** stage within the aggregate(...) method. Such a stage requires defining the list of fields to perform the aggregation and the aggregation functions to be applied. MongoDB supports many aggregate functions, e.g., sum, avg, min, max, etc.

Note that whenever a $group stage is applied, only the fields used to perform the aggregation or created by it will be available in the next stages. So, fields that are not specified will not be usable later on in the pipeline.

In the example, we are first unwinding the main document in more documents, one for each mobile phone number. Then, we are grouping them using the service provider field to make these groups. So, we will have some documents, one per provider saved in the newly created field "provider", each one with as id a subdocument containing the provider field. In addition, in each new document we are counting the number of phone numbers with that provider and saving it in the field "phone_numbers_per_provider".

Whenever a grouping operation is to be performed on the whole dataset, it is possible to apply a *dummy _id* in the $group stage. It's enough to set the _id to *true* or a fixed value. The latter is why the **$** in the grouping stage is important!

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": true,
      "phone_numbers_per_provider": {"$sum": 1}
    }
  }
])
```

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": {"provider": "$mobile_phone_numbers.service_provider"},
      "phone_numbers_per_provider": {"$sum": 1}
    }
  }
])
```

Not putting the $ symbol like in the second case means grouping all the providers together, it's like writing TRUE instead of the condition.

Whenever an aggregation pipeline is to be employed, it is necessary to explicitly specify all the different pipeline stages (e.g., filtering, projections, etc.). In particular, besides the previously explained $group, $unwind, **$sort**, and **$limit** stages, **$match** defines filters while **$project** defines projections. These stages can be applied interchangeably in the aggregation pipeline.

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": {"provider": "$mobile_phone_numbers.service_provider"},
      "phone_numbers_per_provider": {"$sum": 1}
    }
  },
  {"$match": {"phone_numbers_per_provider": {"$gt": 10} },
  {"$sort": {"phone_numbers_per_provider": -1} },
  {"$project": {"phone_numbers_per_provider": 0} }
])
```

7. Key-value databases – Redis

Key-value databases are a technology with focus on performance. Performance is a business need, for example a 1/10 second delay results in 1% loss of sales for Amazon, half a second delay causes 20% drop in traffic for google, and so on.

Key-value stores are built upon the assumption that any entity can be seen as a value pointed by a key and searching a value means searching that key. Keys are just used in key-value databases to look for some value, with small access times. In many key-value stores, like in Redis, most commands are executed in $O(1)$ and with minimal lines of code: retrieving values usually takes the same time independently on the size of the interested data.

Key-value stores are employed at different levels: at database level or to support other databases, at caching level and at message brokers' level.

Redis

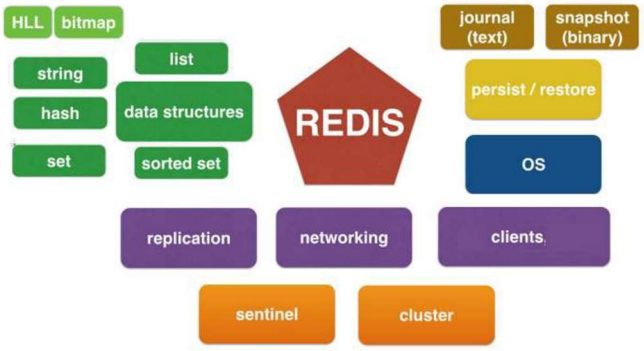
Redis is an advanced key-value store, where keys can contain data structures such as strings, hashes, lists, sets, and sorted sets, and atomic operations on these data types can be performed.

Redis can be used as Database (it can persist data to the disk), as a Caching layer (because it is fast to retrieve data) or a Message broker (it is not only a key-value store). Redis is not a replacement for Relational Databases nor Document Stores but can be used in a complementary way to a SQL relational store or to a NoSQL document store. It's best used for rapidly changing data with a foreseeable database size (should fit mostly in memory).

The use cases of Redis can then be caching, counting things, blocking queues, as a service bus, MVC Output cache provider, as backplane for SignalR, for online user data, etc.

Redis must be considered when speed is a critical requirement, and the dataset is not critical. In fact, the dataset is held in memory, where it must fit, allowing very fast operations. As a consequence to the position where the data is stored, a single shutdown of a non-redundant system will comport the loss of all the data. So, Redis is not very fit for a general-purpose long-term storage system.

Something about Redis' architecture: it is written in C as a single thread program that performs atomic operations and executes most commands with $O(1)$ complexity on data that is stored in-memory. The two fundamental commands are the $\mathrm{SET}(\mathrm{key},\mathrm{value})$ and the $\mathrm{GET}(\mathrm{key})$ and based on them all the others command are built. Unfortunately, there is not official support for windows, but Microsoft develops and maintains an open-source Win-64 port of Redis. Redis is however meant to work in POSIX systems like Linux, which is the recommended one.



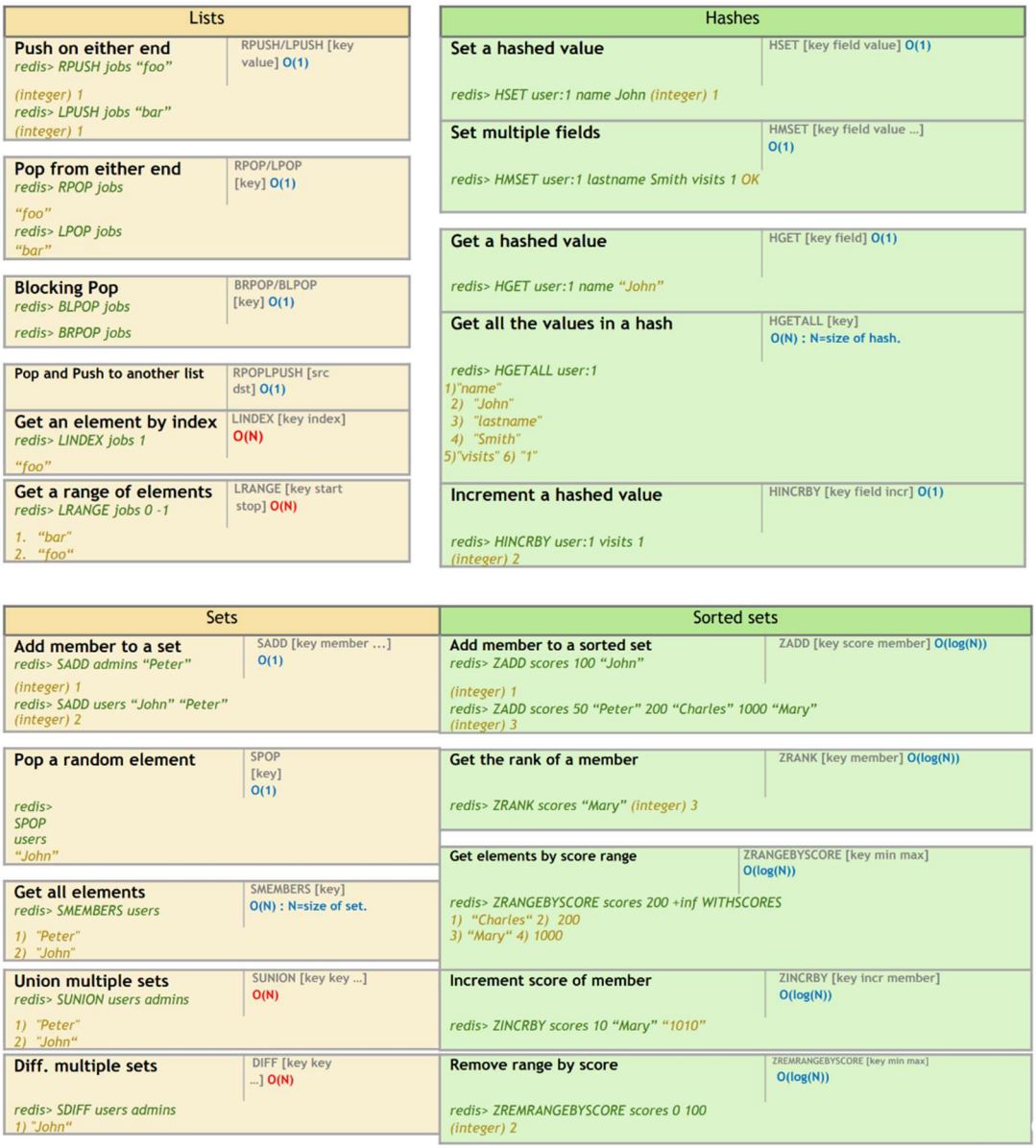
Another important feature is the ability to set an expiration time to a key, after which the key is deleted from the database. Since Redis is often used as a caching layer, this feature is very important.

Redis is a different evolution path in the key-value databases where values are complex data types that are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers.

When searching for a value, it's possible to specify what kind of value we are looking for. The main data types in Redis, from which all the complex types can be derived, are listed below.

Redis Data		Contains	Read/write ability
String	I m a string! 0 1 1 0 0 0 0 ...	Binary-safe strings (up to 512 MB), Integers or Floating point values, Bitmaps.	Operate on the whole string, parts, increment/decrement the integers and floats, get/set bits by position.
Hash	Key1 Value1 Key2 Value2	Unordered hash table of keys to string values	Add, fetch, or remove individual items by key, fetch the whole hash.
List	A C B C	Doubly linked list of strings	Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value.
Set	D B C A	Unordered collection of unique strings	Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items.
Sorted Set	B: 0.1 D: 0.3 A: 250 C: 250	Ordered mapping of string members to floating-point scores, ordered by score	Add, fetch, or remove individual items, fetch items based on score ranges or member value.
Geospatial index	Value Lat.: 20.63373 Lon.: -103.55328	Sorted set implementation using geospatial information as the score	Add, fetch or remove individual items, search by coordinates and radius, calculate distance.
HyperLogLog	0 1 1 0 0 0 0 1 0 1 ...	Probabilistic data structure to count unique things using 12Kb of memory	Add individual or multiple items, get the cardinality.

Operators available are specific to deal with different kinds of data types. With them, it's possible to retrieve a value by searching its key and once we have the value is retrieved then it is treated as a more complex data structure.

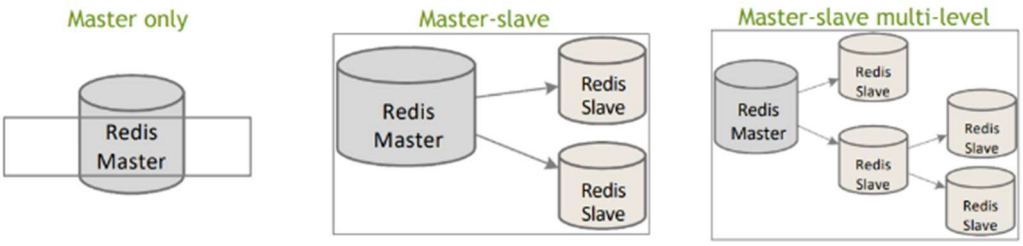


Redis is scalable, this allows to obtain more fault tolerance and ease to deployment and elasticity, so it can be used in Big Data applications.

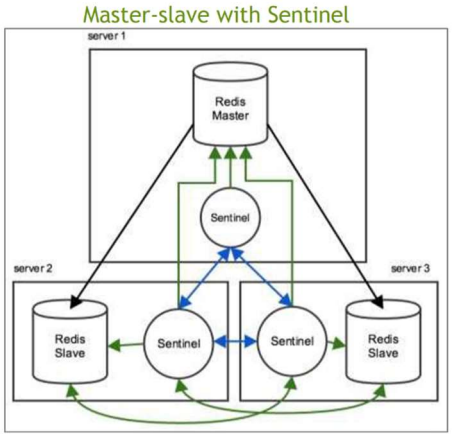
- Persistence: Redis provides two mechanisms to deal with persistence: Redis database snapshots (RDB) and append-only files (AOF).
- Replication: a Redis instance known as the master, ensures that one or more instances known as the slaves, become exact copies of the master. Clients can connect to the master or to the slaves. Slaves are read only by default.
- Partitioning: breaking up data and distributing it across different hosts in a cluster; can be implemented in different layers:
    - Client: partitioning on client-side code.
    - Proxy: an extra layer that proxies all Redis queries and performs partitioning
    - Query Router: instances will make sure to forward the query to the right node. (i.e Redis Cluster).
- Failover: it can be either manual, automatic with Redis Sentinel (for master-slave topology) or automatic with Redis Cluster (for cluster topology)

There are different kinds of possible topologies for Redis.

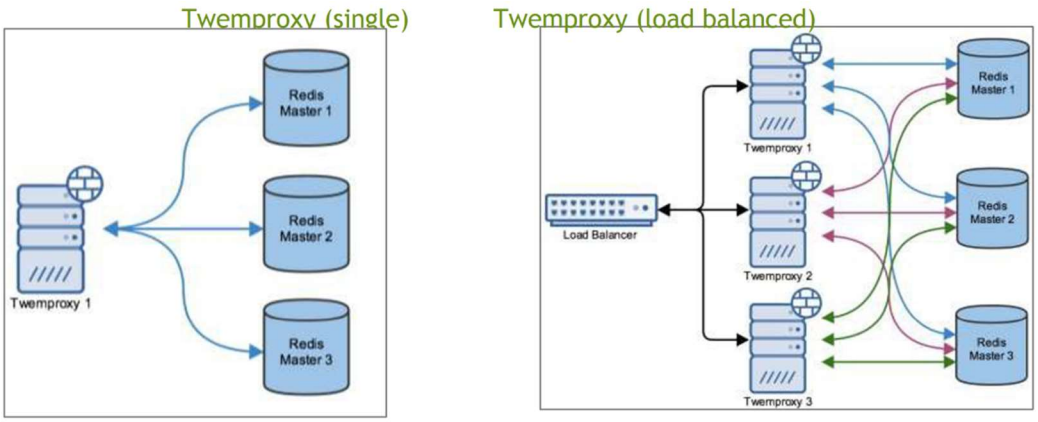
In the **Standalone** topology, the master data is optionally replicated into a hierarchy of slaves, which provide data redundancy, reads offloading and save-to-disk offloading. Clients can connect either to the master or to a slave for read operations but need to connect to the master to perform writes. With this topology, there is no automatic failover.



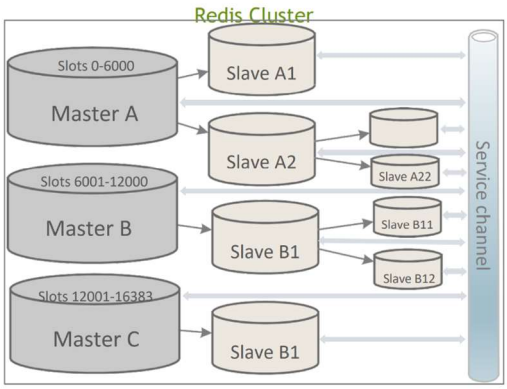
In the **Sentinel** topology, an automatic and reliable failover mechanism is provided in a master-slave architecture, by promoting slaves to masters when the masters fail. In Sentinel, data is not distributed across nodes.



Twemproxy is a project from Twitter and works as a proxy between the clients and many Redis instances. It is able to automatically distribute data among different standalone instances and supports consistent hashing.

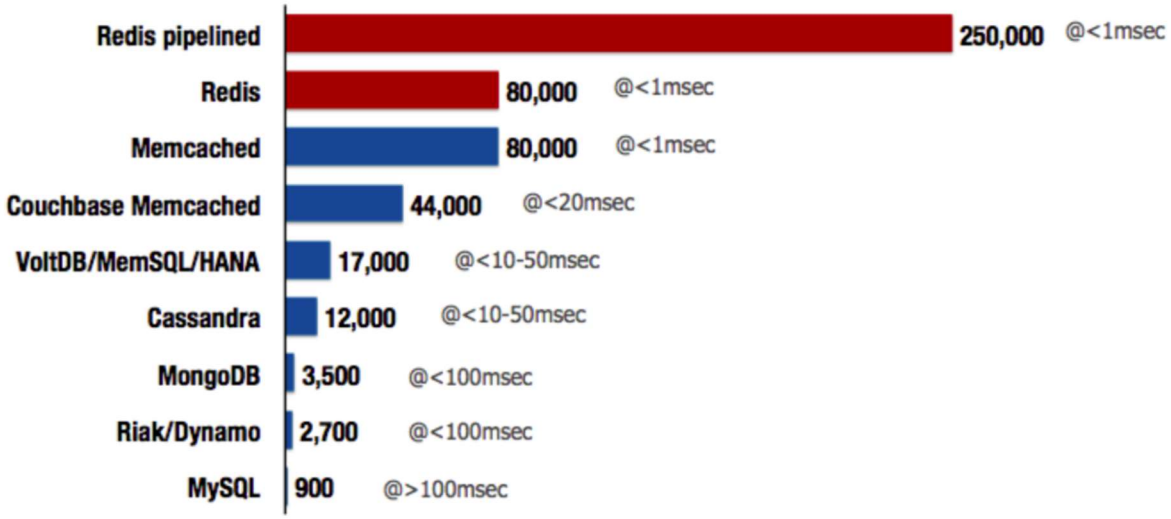


Redis **Cluster** distributes data across different Redis instances and perform automatic failover if any problem happens to any master instance. All nodes are directly connected with a service channel and the key-space is divided into hash slots. Different nodes will hold a subset of hash slots.



To recap the advantages of Redis, we can list: Performance, Availability, Fault-Tolerance, Scalability (adaptability), Portability.

NoSQL and SQL response performance comparison highlights Redis on top:



Key-value and Caching

Caching is the solution for achieving better performances.

A **cache** is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (owing to longer access time) or to compute, compared to the cost of reading the cache.

Queries to the database can pass through the cache first, that checks if the answer is already available in cache itself and if it is available than it returns the answer immediately, without needing to involve the database. Otherwise, the request is submitted to lower levels of cache until it reaches the database.

The cache works as a simple key-value storage, in which every saved object has a key that is a direct pointer to a data value. Searching by key works fast. Moreover, the cache “database” is easy to operate because there are simple operations like set, get and delete.

Some terminology: • storage cost • retrieval cost (network load / algorithm load) • invalidation (keeping data up to date / removing irrelevant data) • replacement policy (FIFO/LFU/LRU/MRU/RANDOM vs. Belady's algorithm)	• cold cache / warm cache • cache hit and cache miss typical stats: • hit ratio (hits / hits + misses) • miss ratio (1 - hit ratio)

Caches are only efficient when the benefits of faster access outweigh the overhead of checking and keeping your cache up to date. This means that it's desirable to have way more cache hits than cache misses.

Cache is used:

- at hardware level (cpu, hdd)
- operating systems (ram)
- web stack
- applications (cache as an additional level at application level, implemented appositely for performance reasons)

The Key-value model can be used as standalone, integrated with other databases, or in a cache system.

Memcached

Memcached is one of many dedicated technologies to implement in a native way a key-value database.

It's a free & open source, high-performance, distributed memory object caching system. It's implemented by a key-value dictionary, only and exactly implementing key-value features.

Technically Memcached is a server to which clients can access over TCP or UDP and can run in pools of servers.

It's convenient to use Memcached when dealing with data that is:

- High demand: data that is requested a lot and used often,
- Expensive: data that is hard and complex to compute,
- Common: data that is shared across users and that many different people ask for.

Best scenario is when the data is of all three kinds.

Applications where typically there are these conditions are user sessions, user data and homepage data. Memcached is now used by Netlog, Facebook, Flickr, Wikipedia, Twitter, YouTube, and many others.

In order to make these databases as fast as possible, Memcached has been stripped out of every additional feature that could slow down the processes.

The principles of Memcached are:

- Fast network access: Memcached servers close to other application servers
- No persistency: if your server goes down, data in Memcached is gone. It's not possible to put data on disks as data is meant only to be in cache memories.
- No redundancy / fail-over
- No replication: single item in cache lives on one server only
- No authentication: not in shared environments
- 1 key is maximum 1MB
- keys are strings of 250 characters: in application typically MD5 of user readable string
- No enumeration of keys: thus no list of valid keys in cache at certain moment
- No active clean-up: only clean up when more space needed, LRU: Least Recently Used

The operations available are basic operations and operations that are an optimization for their use:

Memcached::add — Add an item under a new key
Memcached::addServer — Add a server to the server pool

Memcached::decrement — Decrement numeric item's value
Memcached::delete — Delete an item
Memcached::flush — Invalidate all items in the cache
Memcached::get — Retrieve an item
Memcached::getMulti — Retrieve multiple items
Memcached::getStats — Get server pool statistics

Memcached::increment — Increment numeric item's value
Memcached::set — Store an item

An example of code using the cache is the following:

```php
<?php

function getUserData($UID)
{
    $key = 'user_' . $UID;
    $userData = $cache->get($key); //Search in the cache
    if (!$userData) //if (I don't find what I want in the cache) //if(cache miss)
    {
        $queryResult = Database::query("SELECT * FROM USERS
WHERE uid = " . (int) $UID);
        $userData = $queryResult->getRow();
        $cache->set($userData);
    }
    return $userData;
}

?>
```

All this part is skipped if data is found in the cache

Cache is however not trivial to manage, because challenging probabilistic problems do arise. **Cache invalidation** is a process in a computer system whereby entries in a cache are replaced or removed, because they are not trusted anymore by some cache coherence protocol. To determine what to keep in the cache, statistics from the past and about data locality are utilized.

A possible strategy could be to store data forever, i.e., without an expiration time, and invalidate them on certain events that happen in the system.

A possible optimization strategy is that of the **multi-get** optimizations. Database access is reduced by the presence of the cache and Memcached is faster, but access to Memcached still has its price. The solution is to fetch multiple keys from Memcached in one single call and return an array of items.

More tips:

- Be careful when security matters: there is no authentication in cache.
- Caching is not an excuse not to do database tuning (cold cache still can happen).
- Make sure to write unit tests for your caching classes and places where you use it.

Getting started with Redis

Redis is a Key-Value database whose values can be accessed through the name of the key.

The stored values can be simple types (e.g., numbers) or complex structures (e.g., lists).

Storing a single data instance on Redis is performed by using the SET operator:

```
> SET key_name key_value
```

To collect a value, the GET operator is available and requires a key to search for:

```
> GET key_name
```

The GET operator is correctly executed if the key actually exists, and to check it we use the EXISTS operator, that checks whether a key-value pair associated to a given key exists. It returns 1 if the field exists, 0 otherwise.

```
> EXISTS key_name
```

Also, temporary fields can be created, and they have an expiration time. The EXISTS operator can be used to see if they still exist or are expired.

To delete a key-value pair, the DEL operator is used: it takes as input the key and deletes the pair.

```
> DEL key_name
```

Numerical key-value pairs can be updated using a series of available operations, that are performed atomically:

- INCR – Increases the value by 1.

  ```
  > INCR key_name
  ```

- DECR – Decreases the value by 1.

  ```
  > DECR key_name
  ```

- INCRBY – Increases the value by the amount set.

  ```
  > INCRBY key_name value
  ```

- DECRBY – Decreases the value by the amount set.

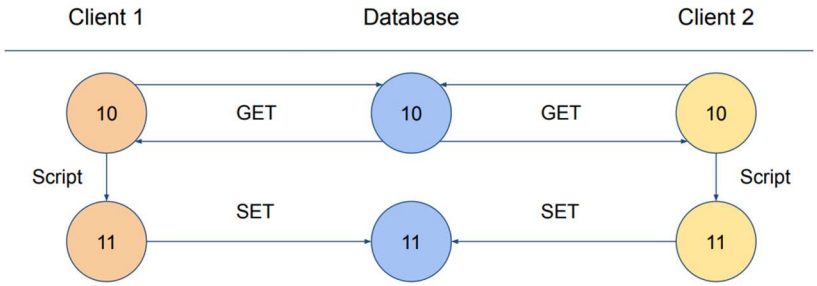
  ```
  > DECRBY key_name value
  ```

Scripting can be employed in Redis. For example, the above operations could be executed by the following code.

```
> x = GET key_name
  x = x + 1
  SET key_name x
```

Note that this code is not executed atomically like the dedicated operations.

Atomic operations are not affected by problems in case of concurrent access. If, for example, two clients access the same resource to read it and get some value, then they both execute a script to modify this value and want to update the local value to the common store, the result of the second updated operation will overwrite the result of the first, leaving the database in a state like if only one of the two operations happened.



Concurrent accesses to the data need to be correctly managed. When possible, it's better to always use atomic operations and avoid scripting.

Redis supports the creation of temporary key-value pairs. This can be achieved using the following operators:

EXPIRE – Set the lifespan of a variable (in seconds). The effect is applied to the least created variable.

```
> SET key_name key_value
> EXPIRE time_to_live
```

TTL – Returns the remaining lifespan of a variable (in seconds).

```
> TTL key_name
```

PEXPIRE and PTTL achieves the same operations but using milliseconds as a unit of measurement for the expiration time.

The TTL operator returns the following values:

- remaining_lifespan – The remaining time the key will be maintained.
- -1 if the key will never expire.
- -2 if the key has already expired.

Furthermore, not only it is possible to set the TTL when creating the key, but whenever the key is SET, the TTL is reset.

To set the expiration directly when creating a variable:

```
> SET key_name key_value EX time_to_live
```

If the user realizes that he needs to permanently store a variable that is set as temporary, he can always remove the expiration time.
The PERSIST operator can be used to solve such a problem. It permanently stores a temporary variable on the database.

```
> PERSIST key_name
```

Of course, PERSIST can be run only on variables that are not expired yet. If a variable has expired, its value is lost.

Redis allows to store and manage complex data structures, like LISTS.

To create a list, the following operators can be used:

- RPUSH – right-push: puts one or more new elements at the end of the list.
- LPUSH – left-push: puts one or more new elements at the beginning of the list.

N.B. If the key does not exist, it will be created. A key with an empty list will be automatically deleted

```
> LPUSH key_name key_value
```

The LRANGE operator can be used to retrieve a subset of values from a list.

```
> LRANGE key_name first_index last_index
```

The values of *first_index* are positive numbers (e.g., 0, 1, etc.).
The values of *last_index* can be positive or negative numbers (e.g., -1, etc.)

Negative numbers (e.g., $-N$) means that the operator will return all values besides the last $N-1$ values (e.g., -2 means that we are collecting all the elements, besides the last one). If setting $-N$, we collect all the values until the $-N$ element from the end of the list.

The LPOP and RPOP operators are used to remove items from lists.

- LPOP – Removes the first item of the list and returns it.
- RPOP – Removes the last item of the list and returns it.

```
> LPOP key_name
```

It is also possible to obtain the length of the list using the LLEN operator.

```
> LLEN key_name
```

Sets are similar to lists but they don't have a specific value order and each value can only appear once per set. They are useful as it is very quickly to test the existence of a value, which is not as easy when it comes to lists.

Sets can be managed using the following operators:

- SADD – Adds one or more values to a list. Returns 1 if the element is correctly added, 0 otherwise.
- SREM – Removes one or more values from a list. Returns 1 if the element is correctly removed, 0 otherwise.

```
> SADD key_name key_value_1 key_value_2 etc.
```

Other possible operations on sets are:

- SISMEMBER – Returns 1 if the value is part of the set, 0 otherwise.

```
> SISMEMBER key_name key_value
```

- SMEMBER – Returns all the members in the set.

```
> SMEMBER key_name
```

- SUNION – Combines two or more sets into one set.

```
> SUNION key_name_1 key_name_2
```

Since sets are not ordered, Redis introduced the concept of ordered set. In an ordered set, each element of the set is assigned to a score that is used to define the order in the set.

An ordered set is created using the ZADD operator:

```
> ZADD key_name key_score key_value_1 etc.
```

and its elements are retrieved using the ZRANGE operator:

```
> ZRANGE key_name first_index last_index
```

Hashes are the best way to store objects. They are mappings between string fields and string values. They are managed using the following operators:

- HSET – Creates a hash and assigns one or more string fields with their values to the hash. Numbers are managed in the same way.
- HGETALL – Returns a hash with its fields.
- HGET – Returns a single field from a hash.

```
> HSET key_name field_1 value_1 field_2 value_2 etc.
```

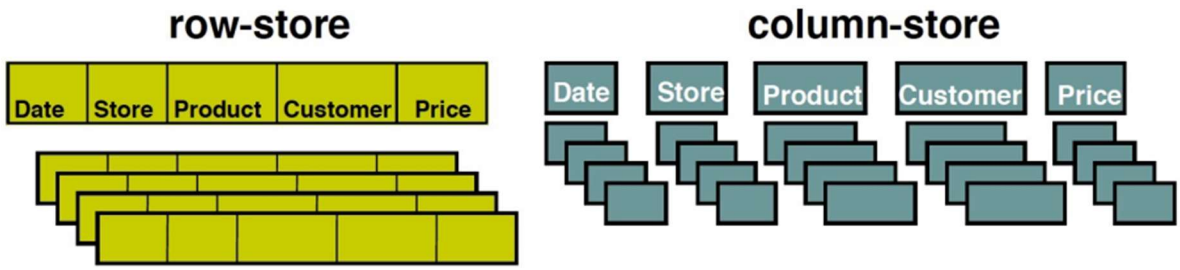
8. Columnar databases – Cassandra

Columnar databases are the most recent technology to be born among the ones seen. It can be found referenced to as: columnar database, wide column store, column oriented, big table approach, and others.

Columnar databases are a class of databases that doesn't make sense if huge amount of data is not involved but are designed specifically for those case of large data.

Column Oriented Databases store data in columns instead of in rows (the name column is tricky), and are mainly used in OLAP(online Analytical Processing), Data Mining operations.

While in row-store databases entities are saved as a row of attributes all together, in column-store databases each individual attribute is saved in an independent and dedicated column with respect to the other attributes and stored in a physically separated storage. This means that when only a subset of the attributes is needed, there is no need to retrieve the entire row but only the separately saved columns containing the data of interest. So, only relevant data is read and there is no need to retrieve the entire "tuple" just to use part of it.



However, when an entire "tuple" is needed or a big subset of its attributes, these attributes are to be retrieved individually and the tuple must be reconstructed. This operation is of course expensive and requires more accesses. We are talking about the *reconstruction problem*.

Moreover, tuple writes also require multiple accesses to different locations.

Column storage databases are thus suitable for read-mostly, read-intensive, large data repositories. An ideal application could be statistical and analytical processing.

This is why columnar technologies is made for big data and is not convenient to use for small amounts of data: the overhead makes them non-competitive when dealing with small amounts of data for which other technologies have better performances. These technologies are never deployed locally but always used in scalable and distributed multi-server architectures.

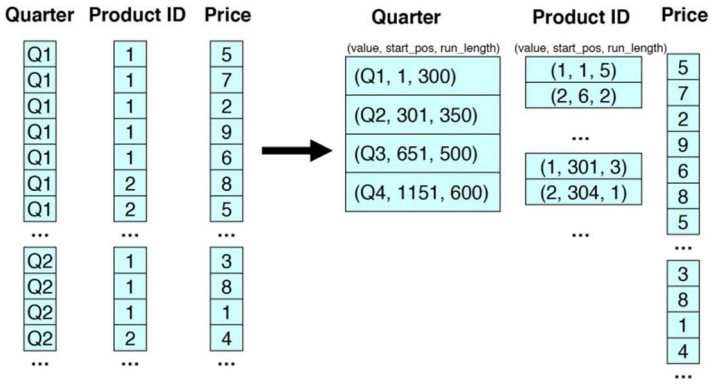
The advantages of Columnar approaches are mostly about performance when dealing with Big Data: improved bandwidth utilization, improved code pipelining, and improved cache locality.

In addition, **data compression** is facilitated by the technology. This makes a big difference when dealing with huge sizes of data and it's possible to compress this size. Thanks to higher data locality in column stores, data of the same type is near and so the same pattern can be compressed and used less times, saving space. Extra space can be used to store multiple copies of data in different sort orders.

The compression is of course loss-ness, so that no part of the data is lost (as it would happen in a lossy compression like to jpg or between video formats).

Data compression is not only a problem of storage space but more importantly a problem of efficiency. In fact, moving a smaller, because compressed, amount of data means less time is required by operations.

Run-Length Encoding is utilized: repeated values are saved once and an interval in which they are found are save, instead of saving these values as many times as they appear.

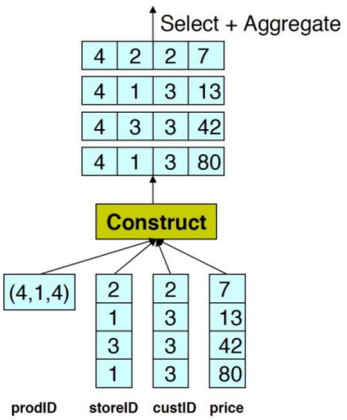


It's more probable to have repeated values in the same columns than in the same row. This is why this compression is facilitated by the columnar technology.

The cons of the columnar technologies are mainly due to the fact that data is spread:

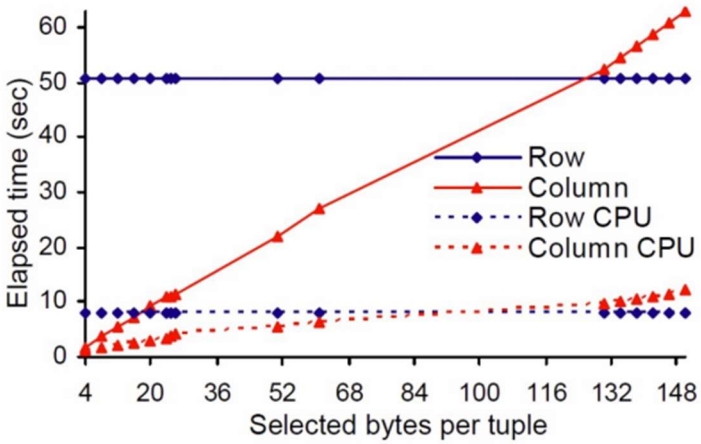
- Increased disk seek time,
- Increased cost of inserts,
- Increased tuple reconstruction costs.

Tuple reconstruction is the data access problem arising when a tuple needs to be reconstructed from its attributes that are spread around in different locations. In fact, every row is a bunch of values spread around different servers and every element needs to be fetched in order to rebuild the structure of the row.



In the row-based approach, independently of the number of columns composing a tuple, reading a row has always the same cost. Instead, in the columnar approach, there is a linear increase in the cost with relation to the number of columns composing a row that are needed, because every additional column has a fetch cost.

However, large prefetch hides disk seeks in columns. There is a point from which even with a large number of columns to be fetched the columnar approach is more convenient when facing tuple reconstruction than the row-based approach. How big is the data that most of the times is going to be reconstructed determines what approach is best in terms of performance when it comes to reconstruction.



Note that in the graph above, the horizontal dimension could also be expressed in Columns per Tuple.

Available columnar databases are: Cassandra, Vertica, SybaselQ, C-Store, BigTable, MonetDB, LucidDB.

Cassandra in a way mitigates the cons of columnar technologies because of its partial key-value approach beside the main columnar one.

Cassandra

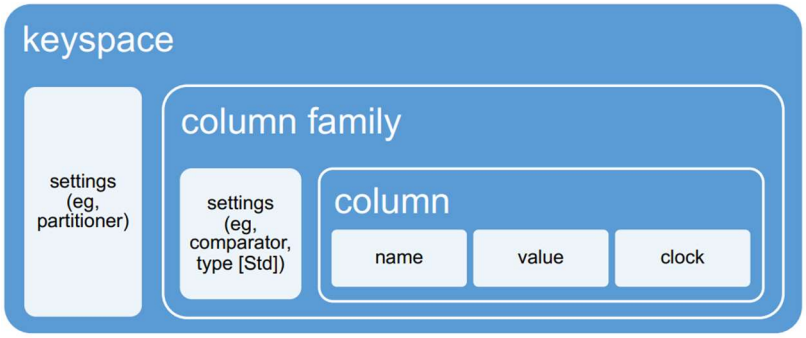
Originally designed at Facebook, **Cassandra** is an open-sourced columnar database technology now adopted by many companies such as eBay, Spotify, Netflix, Ibm, Twitter, Adobe, and others.

It was born by combining two projects together: Google BigTable that is a columnar database and AmazonDynamoDB that is a key-value database. As a consequence, Cassandra is a blending between a columnar and a key-value database.

The atomic element is the **Column**, an independent element corresponding to a single data item composed by a name, a single value, and a timestamp.

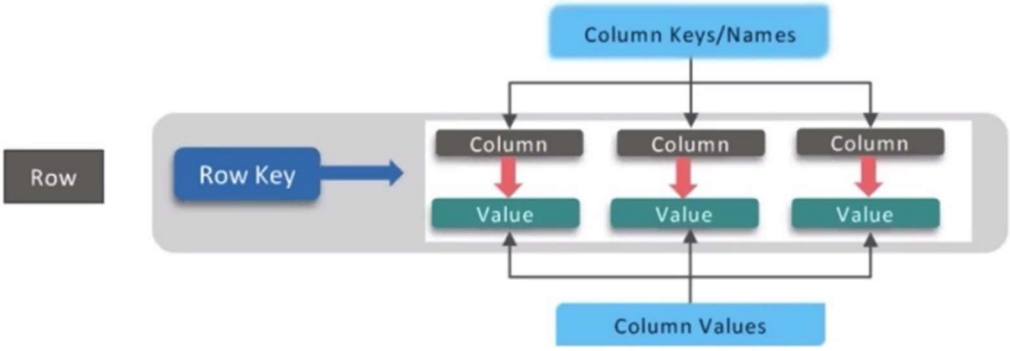
Columns are grouped together in **column families**, that collect records of similar kind. Columns in a column family are somehow similar and grouped coherently, but not necessarily they have the same structure and are of the same kind. A columns family is the position of a set of columns for application purposes. Column families are often defined when queries are executed in a schema on read fashion. They can additionally contain settings and properties to apply to combinations of columns.

The set of all the column families plus some other settings, so basically the while database, is the **keyspace**. Typically, there is one keyspace per application and some settings are configurable only per keyspace.



Some columns may be missing from some entries (or some "rows"). If that value is not known or doesn't exist, the corresponding column just doesn't exist either, and there is no need to put a NULL value somewhere.

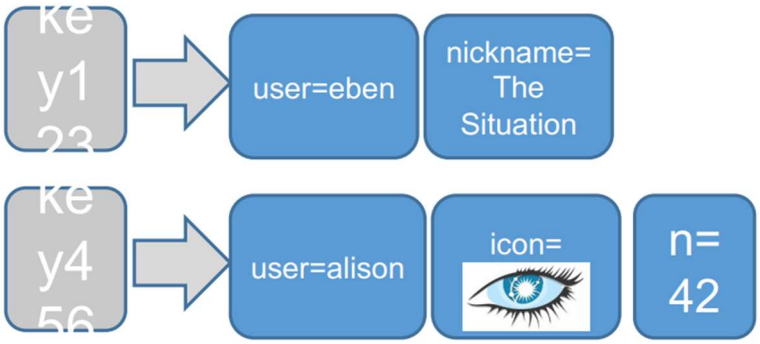
To solve the reconstruction problem, Cassandra adopts an approach from the key-value family by exploiting the column family idea and defining specific settings to help find the right combination of columns. In order to do so, the columns belonging to the same "row" are bonded together by a **Row Key**, containing the pointers to all the columns that are part of the row corresponding to that key.



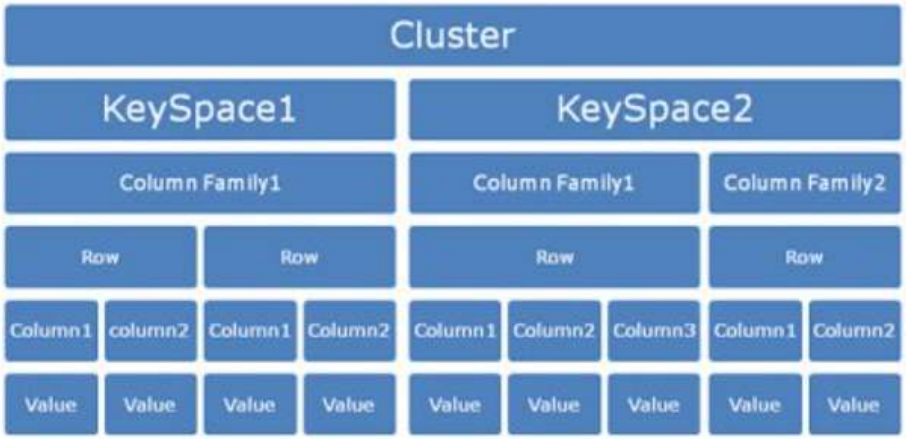
Each Row Key points to a set of columns that put together compose an entry of what would be a table in a row-based approach.

So, the structure of a columnar database is only composed by elements pointing to others, and from this fact we can say that Cassandra is **schema-less**.

If, for the sake of the reasoning, we think of Cassandra as a row-oriented approach, we can see each row as uniquely identifiable by its key and each row as a grouping of columns representing the attributes.



A column family usually collects the row keys and the columns that, with their row key, represent instances of a kind of entity. There is total flexibility in deciding which column to use to define an object and the shape of each row, even inside the same column family, is totally independent from the shape of the others.

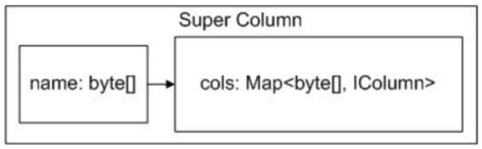


The approach is in some way similar to document databases, just replacing the concept of row with the one of a document containing other subdocuments in a flexible way. A row could hypothetically be stored in JSON-like format to resemble a document as follows:

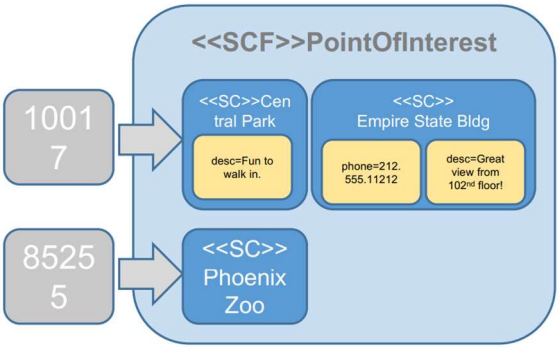
```
User {
    123 : { email: alison@foo.com,
             icon:

    456 : { email: eben@bar.com,
             location:  The Danger Zone}
}
```

There are more advanced ways to treat data in Cassandra, like *Super Columns* that are a sort of intermediate structure between a table and a column, grouping more columns under a common name.



Then there are *Super Column Families* grouping rows composed by Super columns and their keys.



Cassandra is a **Not only SQL** database, meaning that it can be queried both with SQL and dedicated non-relational APIs.

API functions are listed below:

get() : Column
- get the Col or SC at given ColPath

```
COSC cosc = client.get(key, path, CL);
```

get_slice() : List<ColumnOrSuperColumn>
- get Cols in *one* row, specified by SlicePredicate:

```
List<ColumnOrSuperColumn> results =
    client.get_slice(key, parent, predicate, CL);
```

multiget_slice() : Map<key, List<CoSC>>
- get slices for *list of keys*, based on SlicePredicate

```
Map<byte[],List<ColumnOrSuperColumn>> results =
    client.multiget_slice(rowKeys, parent, predicate, CL);
```

get_range_slices() : List<KeySlice>
- returns multiple Cols according to a *range*
- range is startkey, endkey, starttoken, endtoken:

```
List<KeySlice> slices = client.get_range_slices(
    parent, predicate, keyRange, CL);
```

```
client.insert(userKeyBytes, parent,
    new Column("band".getBytes(UTF8),
    "Funkadelic".getBytes(), clock), CL);
```

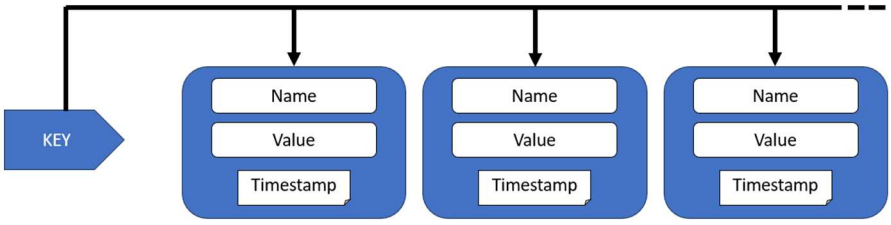
batch_mutate
- void batch_mutate(
  map<byte[], map<String, List<Mutation>>> , CL)

remove
- void remove(byte[],
  ColumnPath column_path, Clock, CL)

Given a key and a column, it's possible to retrieve a value with the get() function, or given a key and a predicate, it's possible to retrieve a column with the get_slice() function. It's even possible to retrieve columns associated to more keys and more predicates using other functions. Then there are of course functions to write new columns.

If the user prefers to use a declarative approach, SQL is available. However, the expressive power in a columnar database of SQL-like queries is lower than the one of SQL. For example, multi-level or nested queries are not possible.

This is due to the fact that queries are not written on the values of the data, like in a classical SQL query. In fact, from an efficiency perspective, values are not a good way to start a query in columnar approaches because the entire structure is based on keys that let the user find the data, but the opposite way is non contemplated.

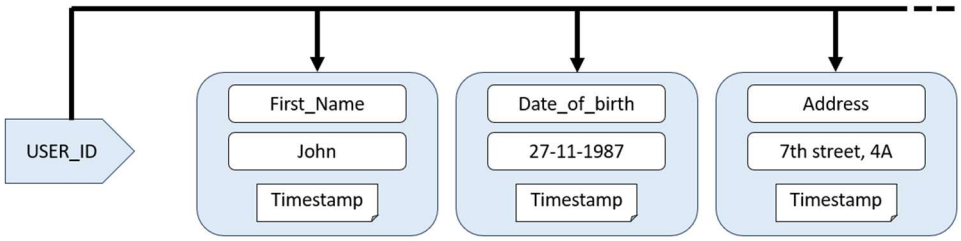


So, the queries look for data by searching the keys, not the values. Looking for a value to retrieve the whole row would mean to first find the value in a column, then find the correct key that points to that column (and there is no other way to find it than a sequential scan of the keys) and then access the other columns pointed by that key. This process would be utterly inefficient.

To query on a value, that value must then be the key. Data should then be designed in a query-based model, in which first the designer or the system automatically considers what questions will be

asked and then designs the model based on that. The data model is designed based on the queries that will be asked on it. We are talking about **schema on read**.

To make an example:



The following query is a query on the key:

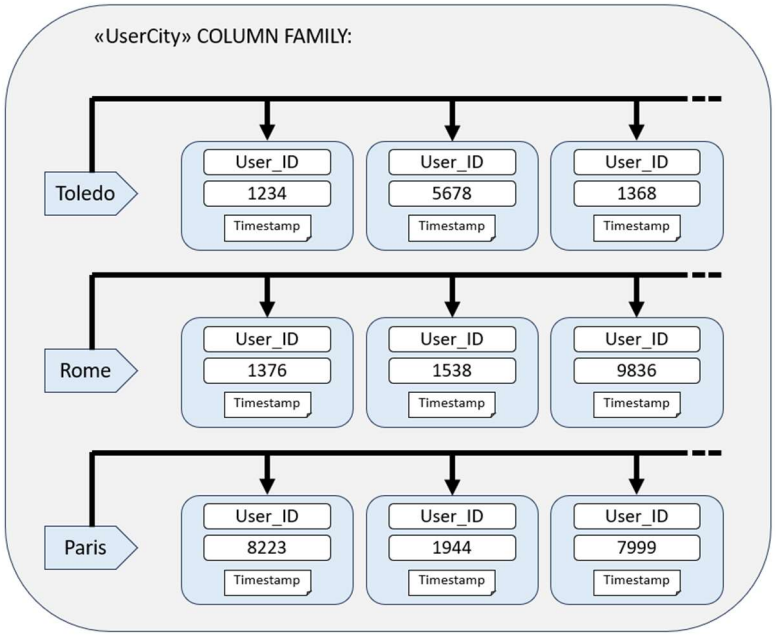
SELECT * FROM User WHERE User_ID='…'

And is resolved by searching that key and retrieving the associated columns.

Instead, the following query is a query on a value:

SELECT * FROM User WHERE city = 'Scottsdale'

To resolve it, the database appositely creates a new column family, called for example "UserCity" in which the key is the city and points to the columns containing info about the IDs of users of that city.



Now the database has a new schema for the data that is based on the requested query, which can be performed efficiently.

Being a column family a sort of index over the data, we can say that Cassandra is an index factory. Columnar databases let the user create huge number of indexes over a huge amount of data.

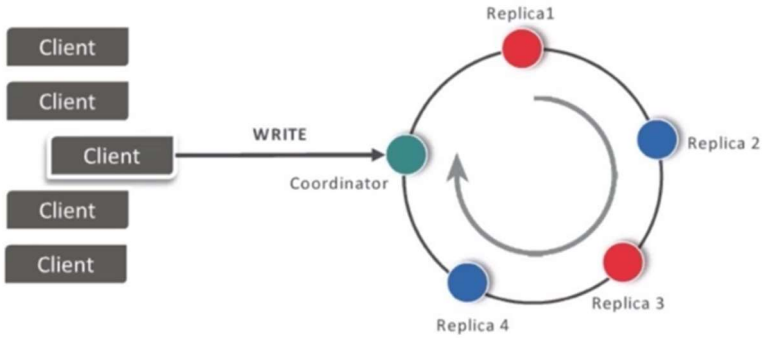
Unfortunately, creating a column family has an overhead in terms of both space and efficiency: indexes need to be saved and updated when data changes.

To recap some properties of Cassandra:

- "tuneably" consistent
- very fast writes
- highly available
- fault tolerant
- linear, elastic scalability
- decentralized/symmetric
- different client languages
- almost automatic provisioning of new nodes
- $0(1)$ dht
- Big data

Cassandra architectures are implemented in a way called *Master-less*, in a peer-to-peer fashion. This way, there is no point of bottleneck and no single point of failure, but interactions need to be coordinated in some way.

To manage coordination and replicas of data spread around nodes, the **ring strategy** is employed. All the nodes are organized in a ring fashion, meaning that every node is preceded and followed by another node. To avoid conflicts, the order of execution of operations follows this ring in a sequential order.



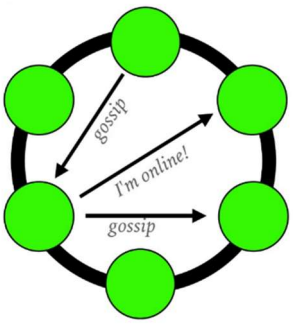
When a client asks to do something, the request reaches one of the nodes (it can be any node, possibly the nearest one to the client or the less overloaded) and this node becomes the coordinator for the requested operation. The coordinator executes and then passes the request along the ring to other nodes that execute, which in turn pass along the request to the next node, until it is needed. For example, the request is passed until all the replicas have performed the write operation.

If a server is offline and thus the ring is broken, the coordinator keeps the request in a buffer, waiting for the needed node to return online.

In particular, for a write operation, if any replica is down, the coordinator writes to all other replicas, and keeps the write until the downed replica comes back up. When all replicas are down, the coordinator buffers write operations.

When the ring is broken because a node is offline, for disservice not to happen, every node should know the status of other nodes. To do so, every node should communicate to other nodes its good status, but doing this could cause the network to be overloaded or flooded with status updates, making the approach not scalable and not suitable for a high number of nodes as desired. The solution is to use the gossip protocol for status updates.

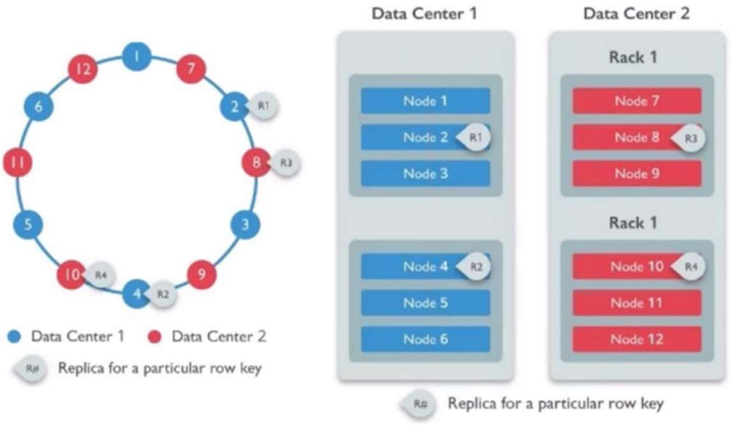
The **gossip protocol** (or epidemic protocol) ensures that data is disseminated to all members of a group. Every node picks three discussants and sends them information. These three peers will then spread the information to other three nodes each, until every node that is meant to receive it has actually received it.



This process allows obtaining "convergence" of data shared between the two interacting nodes much faster.

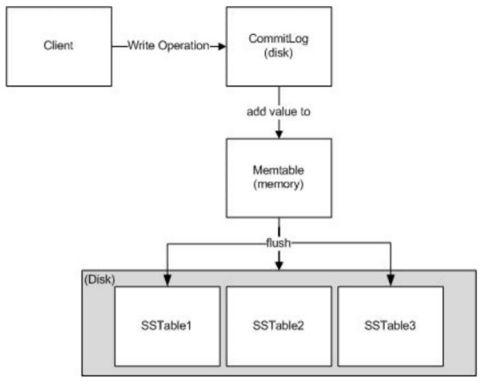
In any case, except for gossip storms, there should be always a constant amount of network traffic.

The network topology is structured in a way for which there are multiple datacenters supporting local queries. The nodes inside these datacenters are logically organized with the ring strategy.



Writes need to be lock-free and fast, and as already said happen with the client sending a write to one front-end node and with this elected coordinator node sending it to all replica nodes via a partitioning function.

To be faster, data is put in a log and in a RAM memory and the operation is confirmed, even if the actual write from RAM to disk is not completed yet but will be performed later. If the system fails, the RAM is re-filled from the log so that the operations will eventually be performed.



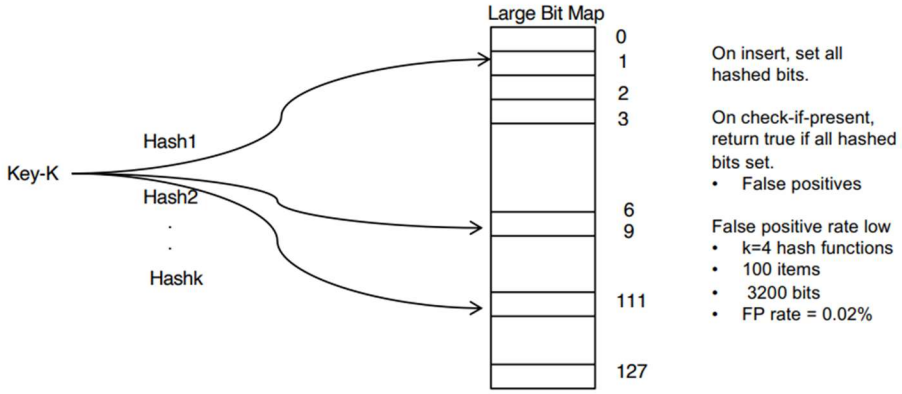
On receiving a write:

1. log it in disk commit log.
2. Make changes to appropriate memtables: in-memory (RAM) representation of multiple key-value pairs.
3. Later, when memtable is full or old, flush to disk.

Periodically perform Compaction: Data updates accumulate over time and sstables and logs need to be compacted, to put everything in a correct status, putting data in disks, merging updates, etc.

As a downside, *Reads* need to interact with the log and multiple SSTables, and thus may become slower than writes.

To gain efficiency, **bloom filters** are utilized. A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with the counting Bloom filter variant); the more items added, the larger the probability of false positives.



When carrying out a *Delete* operation, the strategy is to not delete the item right away but to add a tombstone to the log, so that periodical compaction will remove the tombstone and delete the item.

Read operation are similar to writes, except that to read it is needed to check first in memory, then in the RAM and in the log for the most updated version of the data. This is a consequence of how writing operations work. As mentioned, this process implies that reading is generally slower than writing data.

The coordinator contacts the closest replica for the data to read, then fetches from the other replicas and checks consistency in the background between replicas and with RAM and logs. If two values are found to be different, the read-repair process is initiated to re-establish consistency. The read-repair process uses the gossip protocol.

In the CAP theorem balance, Cassandra positions itself in the AP side (Availability and Partition tolerance). However, Cassandra lets the user choose the level of consistency desired. This property is called **tunable consistency**.

Higher consistency affects availability and performance, so there is a trade-off between consistency and these two aspects.

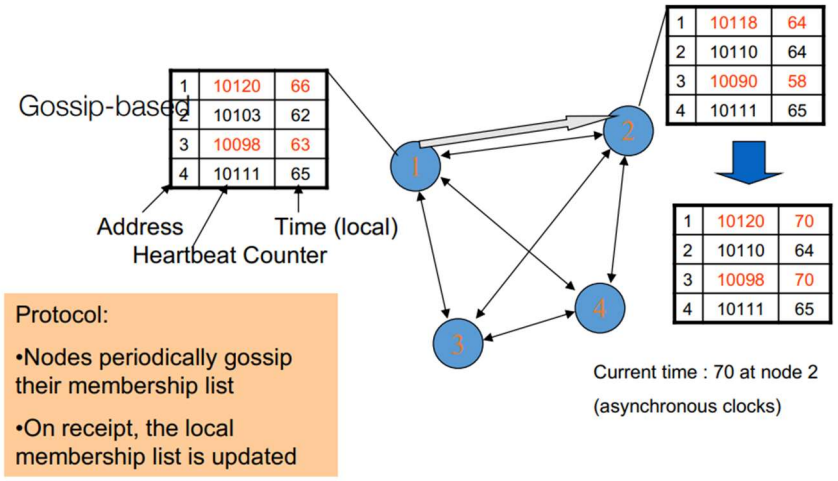
A client can choose one e of these levels for a read/write operation, listed from the one with lower consistency to the one with the highest:

- ANY: any node (may not be replica)
- ONE: at least one replica
- QUORUM: quorum across all replicas in all datacenters
- LOCAL_QUORUM: in coordinator's DC
- EACH_QUORUM: quorum in every DC
- ALL: all replicas all DCs

The **heartbeat protocol**, that is a gossip-based protocol, is used to detect if a node goes offline. Every node owns a *membership table*, containing for each node of the net a value and a timestamp. This value is the heartbeat of the corresponding node, and is a counter emitted periodically along with the timestamp from each node.

In particular, every node sends to the others its own membership table, containing both their updated heartbeat and the ones of other nodes in record. When receiving a membership table from another node, a node compares it with its own and updates it with the most recent values if needed, so everyone has the updated values of the heartbeats of the other peers.

When the nodes see that the value of a certain node is not updated by no received table for a while, they know that that node is probably offline.



A system like Cassandra, guaranteeing BASE properties and positioning itself on the AP side with relation to the CAP theorem, is good when:

- you need really fast writes
- you need durability
- you have lots of data (a lot of GBs and more servers)
- your app is evolving (startup mode, fluid data structure, evolving requirements, columnar approach is schema-less)
- loose domain data, “points of interest”
- your programmers can deal with: documentation, complexity, consistency model, change, visibility tools
- your operations can deal with: hardware considerations, can move data, JMX monitoring

To make a comparison with SQL, when dealing with more than 50 GB of data:

MySQL	Cassandra
• Writes 300 ms avg • Reads 350 ms avg	• Writes 0.12 ms avg • Reads 15 ms avg

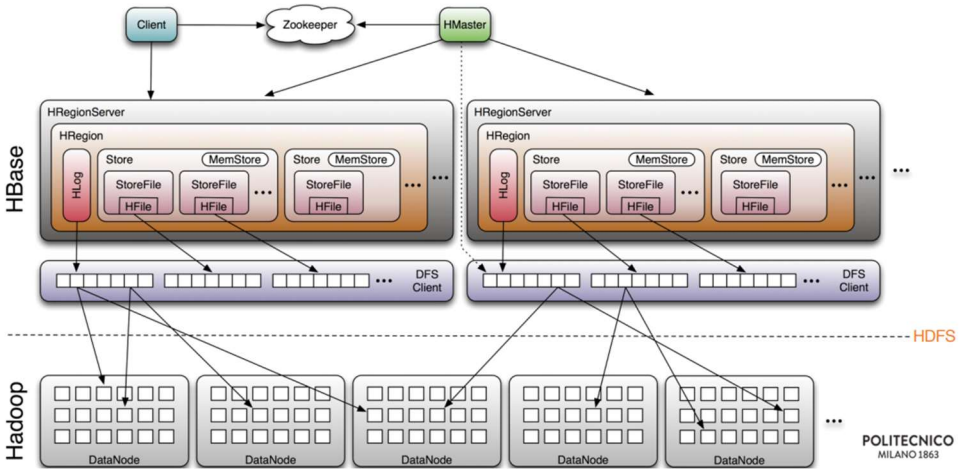
Notice how reads in Cassandra are 100 times slower than writes.

Other columnar DBs

If strong consistency is preferred over availability, a system like **HBase** could be a better solution.

HBase was born as Google's storage system and was then open sourced by Yahoo and is now an Apache project, part of Hadoop and used by Companies like Facebook. It is a columnar database system executed by Hadoop Distributed File System (HDFS), providing a fault-tolerant system suitable for big data.

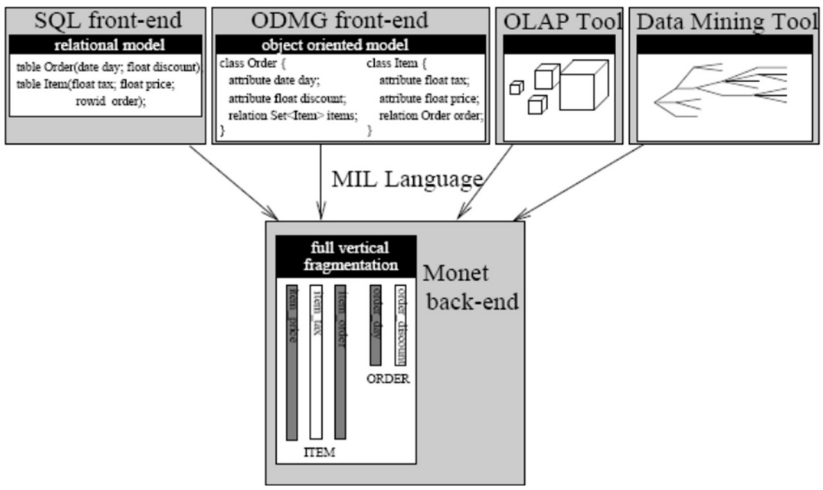
It is based on Get, Put, Scan and MultiPut operations and has a storage hierarchy based on HBase tables and HFiles.



The main property of HBase is to enforce strong consistency. To do so, it uses a Write-Ahead Log, by writing to HLog before writing to MemStore. It can this way also recover from failure.

MonetDB is an open-source column-oriented relational database management system (RDBMS), designed to provide high performance on complex queries against large databases, such as combining tables with hundreds of columns and millions of rows.

MonetDB has been applied in high-performance applications for online analytical processing, data mining, geographic information system (GIS), Resource Description Framework (RDF), text retrieval and sequence alignment processing.

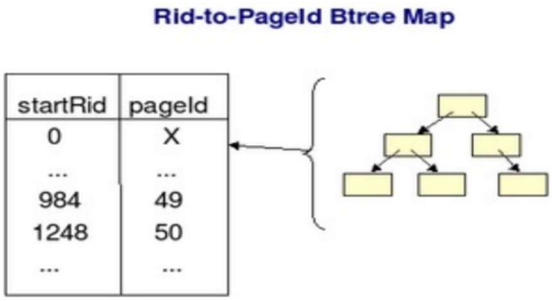


LucidDB is an open-source database purpose-built to power data warehouses, OLAP servers and business intelligence systems. Its architecture is based on column-store, bitmap indexing, hash join/aggregation, and page-level multi-versioning.

LucidDB tables are column store tables, consisting of sets of clusters, and data in LucidDB is stored in Operating System in a file name as “db.dat”.

Each column maps to a single cluster and single cluster page, therefore, stores the values for a specific set of rowIDs for all columns in that cluster.

Each cluster also has a b-tree index associated with it, that maps rid values to page-Ids.



Within a cluster page, column values, by default, are stored in a compressed format, which allows LucidDB to minimize storage requirements. The idea here is instead of storing each column value for every rid value on a page, we instead store just the unique column values. This way, column architecture doesn’t read unnecessary columns.

LucidDB avoids decompression costs and thus performs operations faster.

The use of compression schemes allow us to lower our disk space requirements.

Getting started with Cassandra

Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data, providing high availability with no single point of failure.

It's a column-based NoSQL database created by Meta (ex. Facebook).

Cassandra has the following features:

- Highly and Linearly Scalable
- No Single Point of Failure (i.e., no single part of the system can stop the entire system from working)
- Replicas of data are allowed
- Quick Response Time
- Flexible Data Storage (i.e., supports structured, unstructured and semi-structured data)
- Easy Data Distribution (i.e., supports flexible data duplication)
- BASE Properties
- Fast Writes

To query the data stored within Cassandra, a dedicated query language named **Cassandra Query Language** (CQL) was developed.

CQL offers a model similar to MySQL under many different aspects

- It is used to query data stored in tables
- Each table is made by rows and columns
- Most of the operators are the ones used in MySQL

CQL commands and queries can either be run in the console or by reading a textual file with the corresponding command.

In any case, it's important to keep in mind that usually in columnar databases we do not look for single rows, but query for big amounts of data.

The first operation to perform before creating the table is creating the keyspace. A **keyspace** is the outermost container in Cassandra.

Keyspaces are created using the **CREATE KEYSPACE** command.

```
cqlsh> CREATE KEYSPACE <identifier> WITH <properties>;
```

Let's create a keyspace with the name population.

```
cqlsh> CREATE KEYSPACE population
       WITH replication = {'class': 'SimpleStrategy',
                           'replication_factor': 3};
```

Notice that with the "replication_factor" parameter, we are specifying how many replicas we want to have for the data.

The **DESCRIBE** command can be used to check whether a keyspace (or a table) has been correctly created. It can also be applied to other elements.

```
cqlsh> DESCRIBE keyspaces;
```

To be able to perform the operations on the tables (that we still have to create), we must choose in which keyspace we want to work. The command **USE** covers such need.

```
cqlsh> USE <keyspace_name>;
```

Let's USE the keyspace we just created.

```
cqlsh> USE population;
```

Keyspaces can be also modified (**ALTER**) and deleted (**DROP**) with the corresponding commands.

```
cqlsh> ALTER KEYSPACE <identifier> WITH <properties>;
```

```
cqlsh> DROP KEYSPACE <identifier>;
```

Let's now learn how to **CREATE** a table. The command is the following.

```
cqlsh:keyspace> CREATE TABLE <table_name> (
                    <column_definition>,
                    <column_definition>,
                    …
                )
```

Optionally, some options can be included by using **WITH <options>**.

The definition of the columns is performed as follows.

```
                    <column_name> <column_type>
```

Let's try to create a simple table named person with name, age, birth date and gender.

```
cqlsh:population> CREATE TABLE person (
                    personal_id text,
                    name text,
                    age varint,
                    birth_date text,
                    gender text,
                    PRIMARY KEY (personal_id, text)
                );
```

Notice how for each field we specify its name and type, and at the end the primary key is explicitly declared.

Let's check whether the table has been created or not.

```
cqlsh:population> DESCRIBE tables;
```

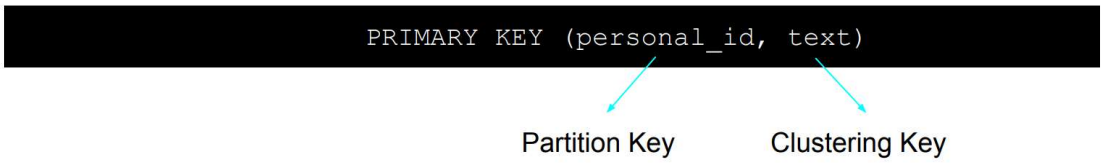
Now that the table has been created, let's take a look at its description.

```
cqlsh:population> DESCRIBE person;
```

This command prints a lot of details about the table. All of these details can be customized when creating the table (as mentioned before).

When creating the **PRIMARY KEY** of the table as the last definition within the **CREATE TABLE** operation, the columns that you put within the **PRIMARY KEY** statement have different meaning depending on the order and the brackets.

The first value (or set of values) is named Partition Key(s). It defines the way in which the data is partitioned within the Cassandra nodes. The second value (or sets of values) is named Clustering Key(s). It is used to define the way in which the data is stored within a partition (i.e., the sorting).



- Partition key: how the data is partitioned across different nodes.
- Clustering key: how the data is sorted inside each partition.

A table can employ many different Clustering and/or Partition Keys.

Attention must be paid at brackets format for a correct command syntax.

When creating a table, clustering keys can be used to define an ordering.

```
cqlsh:population> CREATE TABLE person (...)
                  WITH CLUSTERING ORDER BY (text ASC, ...);
```

Tables can be also modified through the **ALTER** command.

```
cqlsh:keyspace> ALTER TABLE <table_name> <instructions>;
```

For example, we can add a new column to our table:

```
cqlsh:keyspace> ALTER TABLE <table_name> ADD
```

or remove a column from it:

```
cqlsh:keyspace> ALTER TABLE <table_name> DROP <column_name>;
```

Let's try add two new columns to the person table named address (text) and salary (float).

```
cqlsh:population> ALTER TABLE person
                  ADD address text;
```

```
cqlsh:population> ALTER TABLE person
                  ADD salary float;
```

Let's drop the salary attribute from the person table.

```
cqlsh:population> ALTER TABLE person
                  DROP salary;
```

Tables can be also deleted through the **DROP** command.

```
cqlsh:keyspace> DROP TABLE <table_name>;
```

Rather than deleting the table, it is possible to empty it through the **TRUNCATE** command. This way, the structure of the table is kept but all the data inside is deleted.

```
cqlsh:keyspace> TRUNCATE TABLE <table_name>;
```

N.B. by looking at the documentation, you may notice that the keyword TABLE can be interchanged with COLUMNFAMILY. There is no difference between them. Indeed, COLUMNFAMILY is still supported for "historical" reasons.

Indexes are one of the most important elements of a table in Cassandra. They allow to query the column efficiently. It is kind of hard to notice such an advantage on a small set of data, while it is essential in big datasets.

Secondary Indexes are created with the following command.

```
cqlsh:keyspace> CREATE INDEX <identifier>
                ON <table_name> (<column_name>);
```

Let's create an index on the column name of the table person:

```
cqlsh:population> CREATE INDEX person_name
                  ON person (name);
```

Indexes can also be deleted through the DROP command.

```
cqlsh:keyspace> DROP INDEX index_name
```

Let's add a new index on the address column of the table person:

```
cqlsh:Population> CREATE INDEX person_address
                  ON person (address)
```

then remove it:

```
cqlsh:Population> DROP INDEX person_address
```

Let's see how to **INSERT** data within our tables.

```
cqlsh:keyspace> INSERT INTO <tablename>(<column_name1>,
                                        <column_name2>, ...)
                VALUES (<column_value1>, <column_value2>....)
                USING <option>;
```

Let's try and insert a new person in our table:

```
cqlsh:population> INSERT INTO person(personal_id, address, age,
                                     birth_date, gender, name)
                  VALUES ('FRNTRZ95E12F675T', 'Via Milano 12',
                  26, '12-05-1995', 'Male', 'Francesco Terzani');
```

Let's see how to **SELECT** the data within our tables.

```
cqlsh:keyspace> SELECT <field_list>
                FROM <table_name>
                WHERE <conditions>
```

Let's select the person we just inserted within or database using their *personal_id*.

```
cqlsh:population> SELECT *
                  FROM person
                  WHERE personal_id = 'FRNTRZ95E12F675T'
```

Let's now retrieve the person we inserted within our database through their age.

```
cqlsh:population> SELECT *
                  FROM person
                  WHERE age = 26
```

But it won't run: it's only possible to write WHERE conditions on the fields that are a primary key or own an index.

So, an **Invalid Request Error** is shown as the age column has no associated primary or secondary index! Indeed, being Cassandra a column-oriented database, all the operations are optimized to

extract data from columns. To solve this issue, it's necessary to query with respect to the attributes included in the primary key or to create a secondary index. Be careful that not all the operations are supported (e.g., most comparison operators need the additional statement ALLOW FILTERING).

Using the ALLOW FILTERING is another way to solve the problem, but it is not treated here.

Let's see how to **UPDATE** tuples within our database.

```
cqlsh:keyspace> UPDATE <table_name>
                SET <column_name> = <new_value>, ...
                WHERE <condition>;
```

Let's update Francesco's address to 'Via Milani 13':

```
cqlsh:population> UPDATE person
                  SET address = 'Via Milani 13'
                  WHERE personal_id = 'FRNTRZ95E12F675T';
```

Let's see how to **DELETE** the data from our tables.

```
cqlsh:keyspace> DELETE
                FROM <table_name>
                WHERE <condition>;
```

Let's try deleting Francesco using their address:

```
cqlsh:population> DELETE
                  FROM person
                  WHERE address = 'Via Milani 13';
```

An **Invalid Query Error** is displayed as we are not performing a DELETE operation using a primary key, which is against Cassandra's standard operations pattern.

Let's then perform a proper **DELETE** operation using the primary key.

```
cqlsh:population> DELETE
                  FROM person
                  WHERE personal_id = 'FRNTRZ95E12F675T';
```

Let's check whether our operation was successfully performed.

```
cqlsh:population> SELECT * FROM person
```

A set of INSERT, UPDATE and DELETE operations can be organized in **BATCH**. In that way, they are executed one after another with a single command.

```
cqlsh:keyspace> BEGIN BATCH
                 <insert_statement>;
                 <update_statement>;
                 <delete_statement>;
                APPLY BATCH;
```

We are setting a list of operations, separated by semicolons, that will be executed all together.

When the amount of data within a database grows, it can be really tough to visualize it within a terminal. Cassandra provides us with a few commands to overcome this problem.

The **CAPTURE** command followed by the path of the folder in which store the results and the name of the file.

```
cqlsh> CAPTURE D:/Program Files/Cassandra/Outputs/output.txt;
```

Now, whatever is run after the CAPTURE command will be stored in the specified path in the *ouput.txt* file.

To interrupt the CAPTURE, you can run the following command.

```
cqlsh> CAPTURE off;
```

Usually as result of queries a simplified output is provided, but the **EXPAND** command provides extended outputs within the console when performing queries. It must be executed before the query to enable it.

```
cqlsh> EXPAND on;
```

To interrupt the EXPAND you can run the following command.

```
cqlsh> EXPAND off;
```

It's also possible to load the queries from some files: the **SOURCE** command allows you to run queries from textual files. The command accepts the path to the file with the query.

```
cqlsh> SOURCE D:/Program Files/Cassandra/Queries/query_1.txt;
```

How about **FOREIGN KEYS** and relationships between tables? The answer to this question is pretty simple: in Cassandra there is no concept of FOREIGN KEYS and/or relationships, if you want any cross-table check to be performed, you have to manage it by yourself. In Cassandra there is nothing like a JOIN operator.

As mentioned before, Cassandra wasn't created to perform such operations, but to be able to query a lot of data quickly and efficiently. If such operations are needed, you'd better reconsider your DB choice.

Cassandra supports many different data types, like text, varint, float, double, Boolean, etc.

In particular, it supports two particular data types

- Collections
- User-defined data types

Collections are pretty easy to define and update:

```
cqlsh:keyspace> CREATE TABLE test(email list<text>, ...)
```

```
cqlsh:keyspace> UPDATE test SET email = email + [...] WHERE ...
```

When it comes to **user-defined data types** the complexity increases, as it is necessary to define the data type before using it.

```
cqlsh:keyspace> CREATE TYPE <type_name> (
                    <column_definition>
                    ...
                )
```

To check that the new type has been properly created, you can use the DESCRIBE operator. User-defined data types support the ALTER and DROP operations.

```
cqlsh:keyspace> DESCRIBE TYPE <type_name>
```

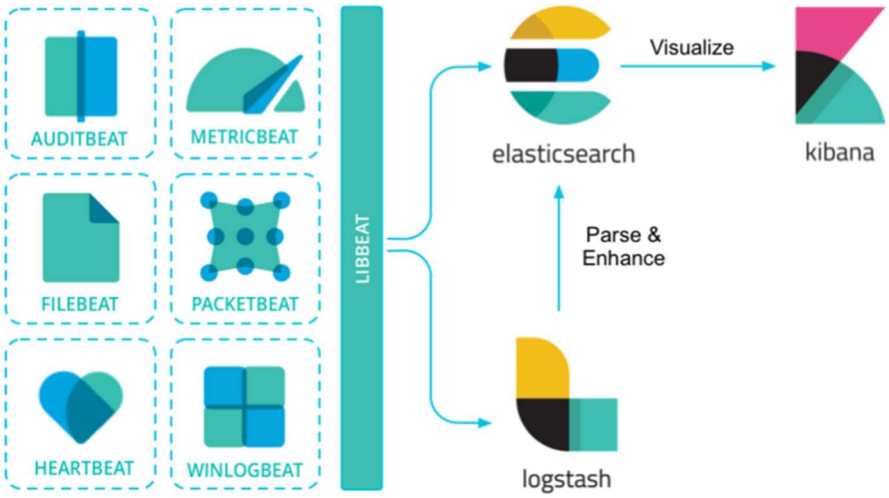
9. IR based databases – ELK stack

Information retrieval (IR) based databases are suited for those tasks that consist in searching for data. They do not come from databases technologies but have been developed starting from information retrieval technologies, and later evolved to also cover the need for data storage.

The **ELK** stack is a system composed by three components:

- **Kibana**: to visualize data
- **Elasticsearch**: to store, search and analyze data
- **Logstash** (+ Beats): to ingest data

The objectives are to store big data while keeping data search capabilities and building data analysis on top, being able to explore and visualize data in real-time with dashboards.



Elasticsearch is an open-source software that stores data, runs queries and performs analysis and extraction, and composes the core of the stack. It's a search and analytics engine for statistics, trends and others that works near real-time (meaning that the time for a document to be searchable after indexing is very short).

Elasticsearch is a distributed and restful system based on an older search engine called Apache Lucene that enables full-text search. It's originally a search engine meant to be used over proprietary data (e.g. the data inside a big organization, over the data owned by the company); these technologies are called *Enterprise search engines*.

Logstash is the Streaming ETL engine of the ELK stack, which provides centralized data collection, processing, and enrichment on the fly. It is data agnostic and is compatible with a wide range of integrations and processors.

Kibana is an open-source data visualization dashboard for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster.

Kibana is simple and intuitive to begin with. Despite such simplicity, it is highly customizable, allowing complex and detailed representations.

Elasticsearch

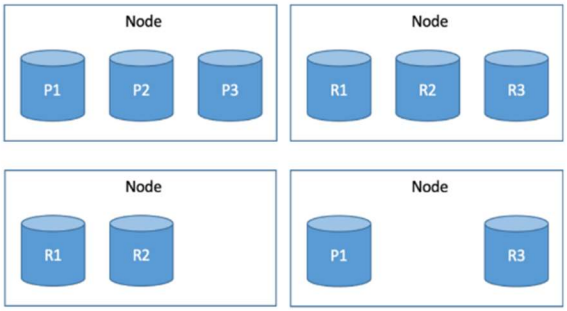
The core of the elastic stack, Elasticsearch wasn't originally designed as a database, but only as a search and analytic engine.

It's distributed and RESTful.

Elasticsearch stores data structures in JSON documents, that can be accessed from any node when in a cluster. When stored, new documents are indexed and made fully searchable.

The built indexes need to be fit for the "best match" approach that is used instead of the "exact match".

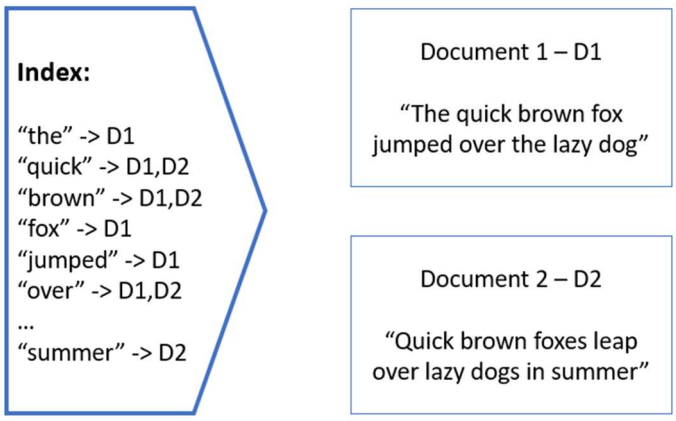
Shards architectures are employed to distribute operations and increase resistance to faults and improve performance, gaining full scalability. Copies of shards, or replicas, are stored in different nodes than the original one.



Writes and reads respect the Master-Slave approach, for which every write first writes in the primary replica shard and then in the copies, while the read is routed to the primary replica.

As said, Elasticsearch relies on indexes as a data organization mechanism. Indexes define the structure of documents via mappings and are partitioned in shards.

In particular, Elasticsearch is based on the concept of **Inverted index**. Such index starts from the value, not from the key. So, considering a document as a set of words, an inverted index lists every unique word from every document and identifies which documents a word appears in. Every word of the index points to the list of documents containing it.



This index structure is very useful for information retrieval because what is searched are words, and the words provide direct access to where they come from.

This index is pre-computed and updated every time the data, so a document, is inserted, and is then almost immediately available for use.

It's possible to enrich the model by adding the position of the words inside the documents along with the list of documents in which they are present.

Let's introduce three key concepts:

The results of a search engine can be less or more relevant with respect to the query that is ran. A relevant answer is the one that is most significant for the needs of the inquirer, and the more the answer is resolving the need, the more relevant it is. In traditional databases, **relevance** is not contemplated as a concept, while in Elasticsearch it has a strong role in determining the results.

When returning answers, they should be ordered in terms of relevance. **Ranking** is the process of ordering the answers based on some criteria so that the top element is the best answer and then the other answers are presented in decreasing order of importance. Note that the word "ordering" can be misleading, as ranking is not an ordering but a "listing by best match".

Best match is the approach for matching in which the match between two string, values, or objects is not exact, but complementary to it and this match is approximate. In some cases, like when doing textual search, best match must be employed.

The problem of how to compute a relevance metric for the data in order to rank them arises. We must define a way to specify the relevance for every data type, the most frequent being text. The final aim is to have a way to determine what document is a good match for a search based on some inputted text, by retrieving the best documents that are about that text.

Results are scored with Lucene's Practical Scoring Function, which uses **TF-IDF**: Term Frequency-Inverse Document Frequency. TF-IDF measures the (textual) relevance of a term inside a document and penalizes the words that appear too often across different documents.

The Term Frequency (TF) is the frequency of term i in document j:

$$tf_{i,j} = \frac{n_{i,j}}{|d_j|}$$

Where $n_{i,j}$ is the number of times the term i appear in document j and $|d_j|$ is the total number of terms in document j.

So, if a document j mentions a word i a lot of times, probably that document is a good answer.

However, using only TF would be biased as not all the words have the same global frequency in the language. An example are *stop-words* like articles and pronouns, that should have less or zero importance.

The Inverse Document Frequency (IDF) measures how common a term is across the collection of documents; the lower, the less important that term is.

$$idf_i = \log \frac{|D|}{|\{d: i \in d\}|}$$

So the IDF is the logarithm of the quotient between the number of documents in the collection $|D|$ and the number of documents containing the term i.

If a word is spread everywhere, it's like it is nowhere as it carries low informative weight. If a word in instead very rare, then it probably is very informative and deserves a higher weight.

The TFD-IF score is calculated as:

$$(tf - idf)_{i,j} = tf_{i,j} \times idf_i$$

It's important to be aware that we are this way operating under a *Bag Of Words* (B.O.W.) model, in which only the words are considered and their organization together inside the text is ignored.

Mapping is the process that assigns to every value its type and its indexes. For every filed, it is defined its type and so the kind of search to run when looking for it.

By defining a mapping for an index you can tell Elasticsearch the types of your documents' fields and so decide which fields and types are searchable and enable full-text search, time and geo-based queries. The data type assigned through mapping implies the kind of index and the kind of search that will be used when that field is involved.

Depending on the mapping of the value of interest, the relevance can be defined in different ways. As a consequence, the computation of the ranking of results depends on the data type we are looking for and so on the mapping that was performed.

The user must careful as it is not possible to change the mapping on an existing index that already has documents.

Elasticsearch is schema-less as it tries to guess the structure of the documents using dynamic mapping (which is sometimes risky as for example dates could be parsed as strings).

If we use the API to request the mapping for "nyc-restaurants" index, we obtain:

```
GET /nyc-restaurants/_mapping
```


```
{
  "nyc-restaurants" : {
    "mappings" : {
      "_meta" : {
        "created_by" : "ml-file-data-visualizer"
      },
      "properties" : {
        "@timestamp" : {
          "type" : "date"
        },
        "ACTION" : {
          "type" : "text"
        },
        "BBL" : {
          "type" : "long"
        },
```


In the mapping we specify all the types for the fields of the documents in the database.

Interactions with Elasticsearch happen through requests to REST endpoints through REST APIs.

Actions depend on HTTP verbs: GET, POST&PUT, DELETE and requests can be sent from command line, software packages like Postman or developer tools built in Kibana.

Beware of the differences between POST and PUT: POST doesn't require the ID of the resource, so Elasticsearch takes care of creating and assigning IDs to documents and duplicates can be created, while PUT requires the ID of the resource, so that resource if already exists is updated and no new one is created.

POST /index_name/_doc	PUT /index_name/_doc/document_id

Flexibility in the structure of documents allows to define different structures every time.

```
POST /my_index/_doc
{
  "author":"Andrea",
  "title":"Set up Elasticsearch and Kibana in 15 minutes!",
  "date":"Tue, 23 Feb 2021 11:40:00 +0000",
  "categories": ["elasticsearch","tutorial","data science","bigdata"],
  "lang":"en-US"
}
```

The Id of documents is always present and can be user specified or generated automatically.

```
{
  "_index": "blog",
  "_type": "_doc",
  "_id": ”jYpU_XQBkGeBYJKltRh8",
  "_version": 1,
  "result": "created",
  ...
}
```

ID auto-generated by Elasticsearch

```
{
  "_index": "blog",
  "_type": "_doc",
  "_id": "my_id",
  "_version": 1,
  "result": "created",
  ...
}
```

User-specified ID

To create an index, we use PUT /index_name:

```
PUT /my_index
```

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Then, a mapping needs to be defined, with PUT index_name/_mapping:

```
PUT /my_index/_mapping
{
  "properties": {
    "field_name": {
      "type": "text"
    }
  }
}
```

The type of document is this way specified.

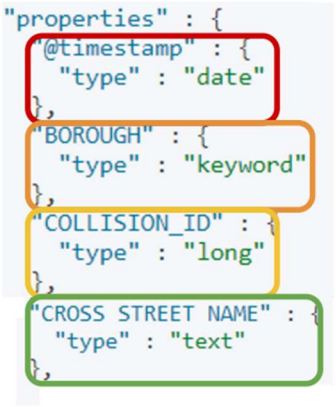
To retrieve a document:

Query:

```
GET /nyc-restaurants/_doc/pZpozncBDaS7kSRanan2
```


```
"_index" : "nyc-restaurants",
"_type" : "_doc",
"_id" : "pZpozncBDaS7kSRanan2",
"_version" : 1,
"_seq_no" : 382695,
"_primary_term" : 1,
"found" : true,
"_source" : {
  "DBA" : "BENTO HOUSE",
  "RECORD DATE" : "02/22/2021",
  "VIOLATION CODE" : "22G",
  "PHONE" : "9173063883",
  "Latitude" : 40.760198166197,
  "BBL" : 4049800055,
  "BUILDING" : "136-75",
  "BORO" : "Queens",
  "CUISINE DESCRIPTION" : "Chinese",
  "CAMIS" : 50071889,
  "Community Board" : 407,
  "INSPECTION TYPE" : "Administrative Miscellaneous / Initial Inspection",
  "BIN" : 4112507,
  "ZIPCODE" : 11354,
  "INSPECTION DATE" : "02/17/2021",
  "STREET" : "ROOSEVELT AVENUE",
  "Longitude" : -73.827606355065,
  "Census Tract" : 87100,
  "@timestamp" : "2021-02-17T00:00:00.000+01:00",
  "ACTION" : "Violations were cited in the following area(s).",
  "Council District" : 20,
  "NTA" : "QN22",
  "location" : "40.760198166197,-73.827606355065"
```

An example of a mapping in which attributes of different types are created is the following:



Date is used for dates you can specify the format.

Keyword is used for structured content such as email, tags, postcode etc…

Long is used for 64-bit integers.

Text is used for full-text search. With **analyzer** you can specify how the field is pre-processed.

When the **Keyword** type is chosen, we are referring to categories we do not want to alter and exact match is applied, character by character.

When the type is **text**, it is possible to enable full text search, by defining inverted indexes to search for words via best matching approaches.

The **language analyzer** is the component that embeds the way in which a text field is pre-processed. It performs pre-processing of the text and prepares the text for the search in it.

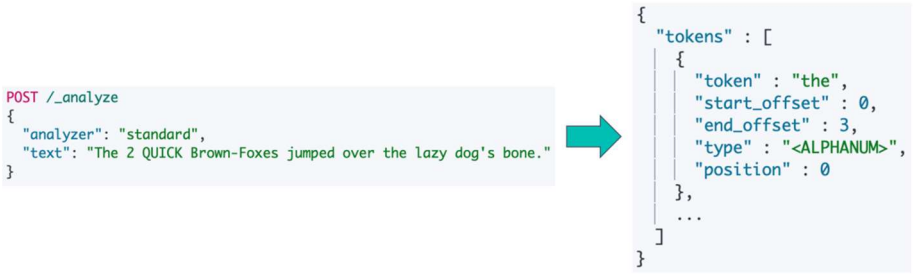
It could remove stop-words of a given language and perform stemming, which is a technique used to reduce an inflected word down to its word stem. For example, the words “programming”, “programmer” and “programs” can all be reduced to the common word stem “program”.

Elasticsearch provides several analyzers for different languages, that may need different pre-processing. (We are in the natural language processing field)

Some built-in analyzers:

- Standard Analyzer: the standard analyzer divides text into terms on word boundaries, as defined by the Unicode Text Segmentation algorithm. It removes most punctuation, lowercases terms, and supports removing stop words.
- Simple Analyzer: the simple analyzer divides text into terms whenever it encounters a character which is not a letter. It lowercases all terms.
- Whitespace Analyzer: the whitespace analyzer divides text into terms whenever it encounters any whitespace character. It does not lowercase terms.
- Stop Analyzer: the stop analyzer is like the simple analyzer, but also supports removal of stop words.
- Keyword Analyzer: the keyword analyzer is a “noop” analyzer that accepts whatever text it is given and outputs the exact same text as a single term.
- Pattern Analyzer: it performs the analysis according to a custom *regexp* pattern.

If the user is unsure what analyzer to use, it’s possible to test them. The analyzer creates a structure for each token in the text.



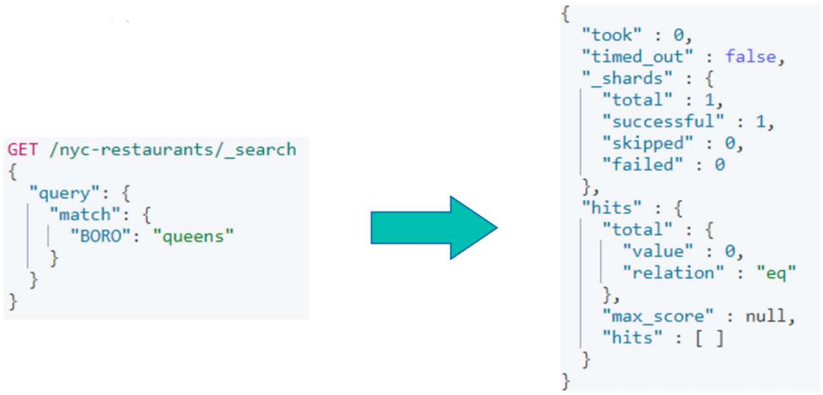
A search query is composed of the query and the body. We specify the index in which to query after the GET, and inside the body we specify the method, usually a "match", the attribute and the value to search for that attribute.

Except for text search, the queries behave very similarly to traditional databases queries, performing exact match with regards to the specified field.

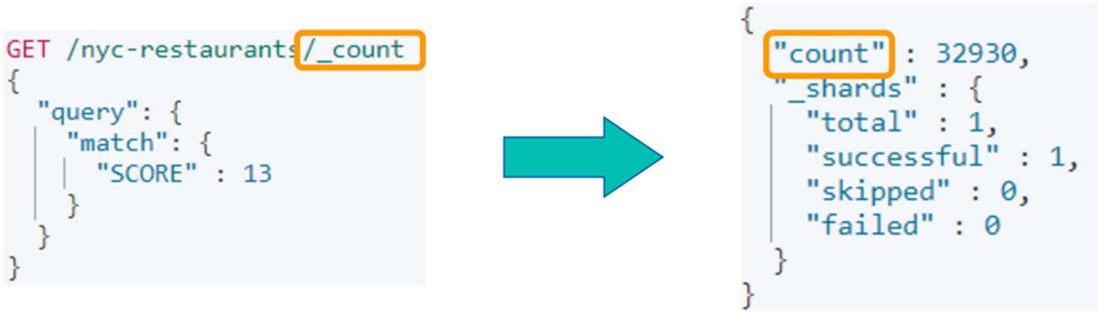
The result of the above query is all the documents that have "INSPECTION TYPE" equal to "Cycle Inspection / Initial Inspection".

In the last query, Borough (BORO) was mapped as a keyword. A keyword does not have any analyzer, so exact matching is used to determine matching of values.

Metadata for the query can be seen below:

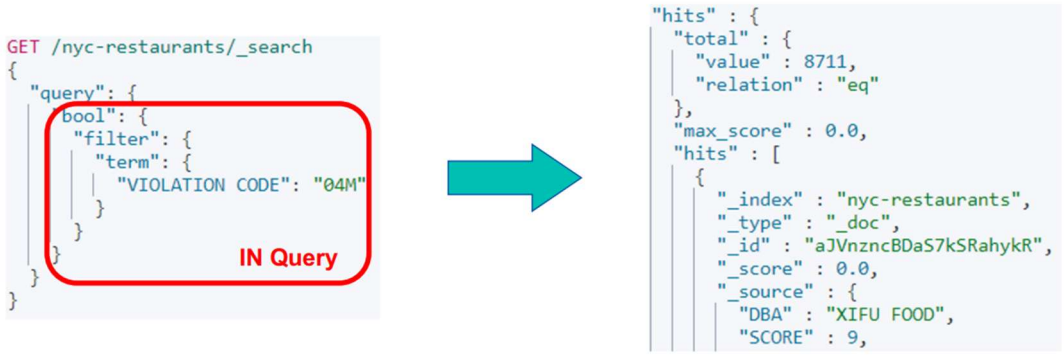


Aggregations are possible, and they allow operations such as counting:



Attention must be paid when using the match identifier, as it can only match one field at a time. An error would be produced:

For fields that are not analyzed and are matched with exact matching, and for which no relevance is involved, filters can be used. Filters are either satisfied or not.



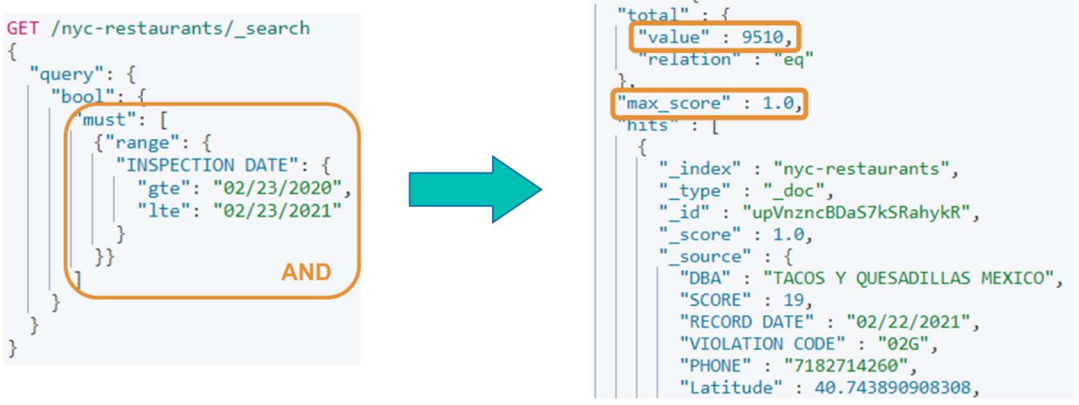
Boolean conditions can be put together, and also an imprecise match is possible. In fact, the *should* operator checks a condition that doesn't necessarily need to be respected, but if a result respects that condition then this fact contributes positively to the answer and this result has a better score.

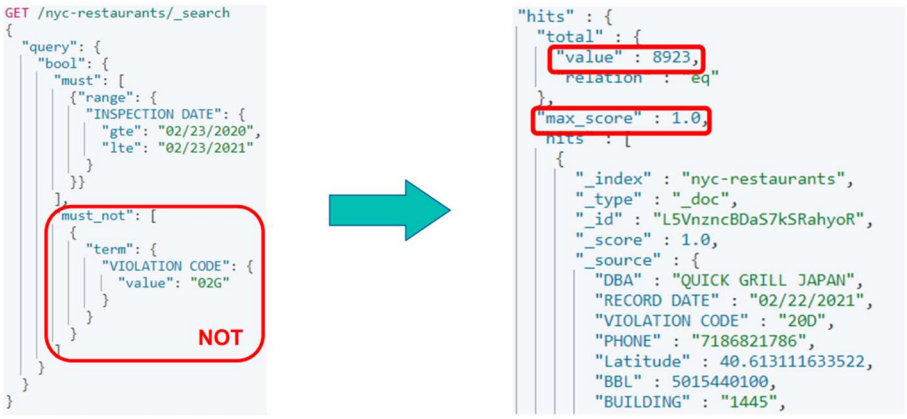
In the above query we are asking to find all the restaurants that:

- Have not included the violation code of value "02G".
- Have been inspected in the range of dated between 02/23/2020 and 02/23/2021.
- Better if they are in Queens.

If restaurants that are in an area different than Queens are found as result of the query, they are still a valid answer but restaurants that are also in Queens are a better answer than they are.

If conditions are Boolean "must" (AND) or "must not" (NOT), then the results can be scored 0 or 1 according to whether these conditions are respected.



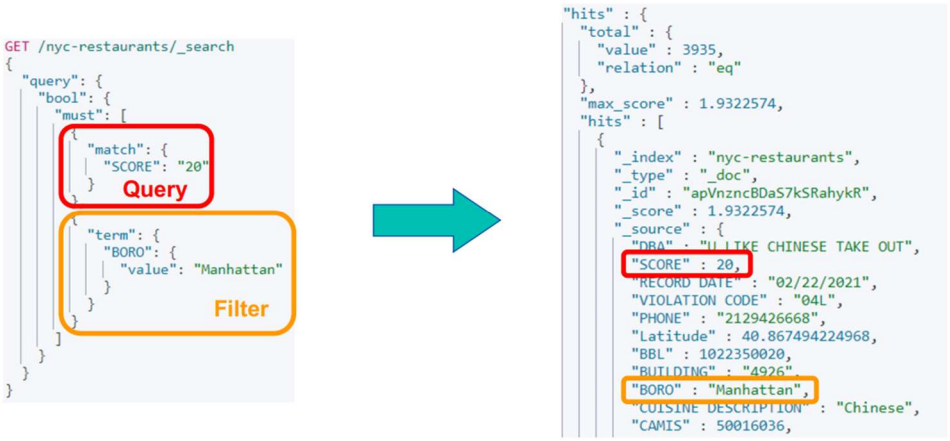


The score is an indication of relevance. Every answer has a score and answers will be ranked by score.

A “should” (OR) condition gives a differential score that is used for the ranking.

Filters can be also about prefixes:

It’s possible to combine filter and queries: it’s more efficient than doing only queries.

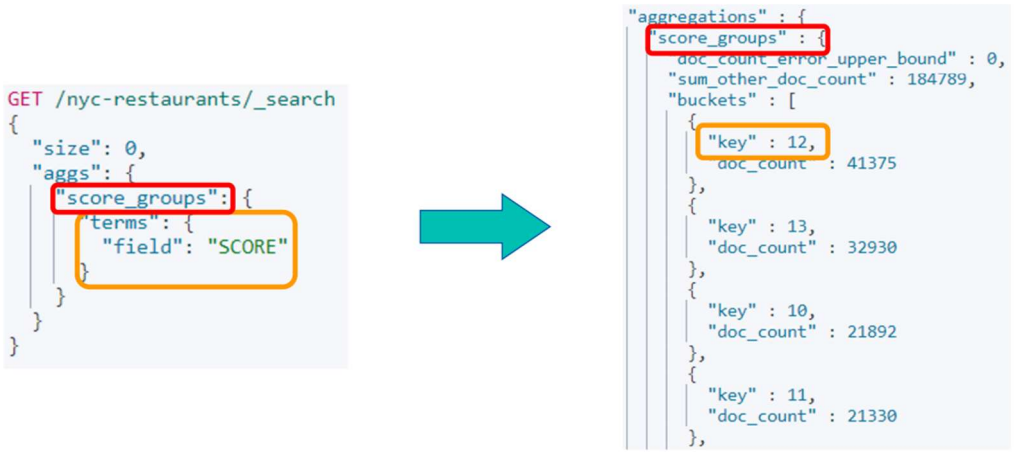


Elasticsearch allows to search in multiple index at the same time:

/_search
index_1, index_2/_search
prefix*/_search

To perform aggregations in Elasticsearch:

In the above example, we are grouping by “SCORE”.



Elasticsearch provides many types of aggregations: Metrics aggregations, Bucket aggregations, Pipeline aggregations, Matrix aggregations.

Getting started with Elasticsearch

The ELK stack is composed by three components:

- **Kibana** for data visualization
- **Elasticsearch** for storing data, querying the different indexes and analysis
- **Logstash** and **Beats** for data ingestion, processing and aggregation

Elasticsearch is the core of the Elastic Stack. It's a Search and Analytic Engine that is:

- Near real-time – Short latency,
- Full-text search (this is the strong point of elasticsearch!),
- Distributed,
- JSON storage format,
- RESTful,
- Fast, Scalable, Resilient.

An **index** is a data organization mechanism that contains documents and defines the structure of documents via mappings. New data is partitioned in shards and automatically indexed. Elasticsearch is based on the concept of **Inverted Index**.

Data is sharded across different nodes and replicated to guarantee fault tolerance. **Shards** distribute operations to increase resistance to faults and hardware failures, improve performances, increase capacity to serve read requests.

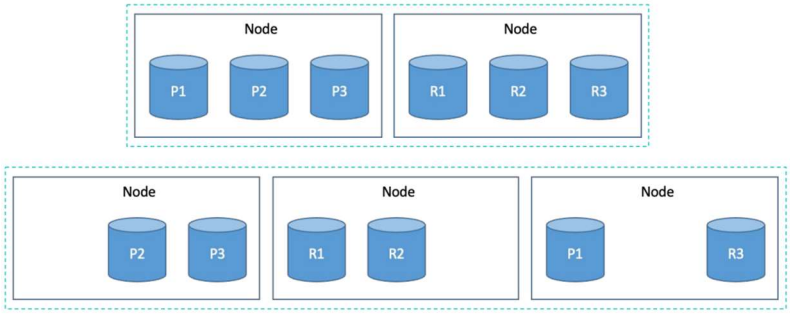
There are two types of shards: primaries and replicas, that are stored in different nodes. Each document in an index belongs to one primary shard and replica shards are copies of primary shards stored on a different node.

Write Operation are performed on a Primary shard then on Replicas.

Read Operation are either performed on a Primary shard or a Replica. There are synchronization mechanisms to ensure correct parallelization of operations and to avoid the reading of invalid (because old) values.

The number of Primary Shards and the size of shards influence performances:

- The more shards you have, the more the management overheads.
- The larger the shard size, the more it takes to move them (if needed).
- Querying a lot of small shards makes the processing per shard faster, but more queries means more overheads, consequently it may be more efficient to query a small amount of shards.



A **document** is described by a set of fields.

A **mapping** describes the data type of a document's fields. There are two types of mapping:

- *Dynamic Mapping* – Elasticsearch takes care of adding the new fields automatically when a document is indexed. Elasticsearch understands the structure of a document by itself.
- *Explicit Mapping* – Makes you define the mapping. You must take care of any changes in the structure of the index manually. Types for all fields are explicitly stated, and the index is structured like a table.

There are potential problems with Dynamic mapping: first, data types interpretation errors can occur and for example dates can be recognized erroneously, and secondly when uploading a document Elasticsearch creates attributes in the mapping for fields, but if the documents are all different there would be a large number of attributes created.

So, when using the Dynamic Mapping, you must be aware of the possible drawbacks:

- Whenever a new document is indexed, Elasticsearch adds its fields to the mapping, if they are not already present. Therefore, if a wide number of heterogeneous documents are added, you may end up with a huge number of fields which may end up breaking up the index.
- Elasticsearch is Schema-less, i.e., if no mapping is provided and dynamic mapping is applied, it will try to guess the structure of the document and the type of its fields.

N.B. you can't change the mapping on an existing index that already has documents! (It is possible with scripting, but it almost always happens that it is deleted and re-created)

Interactions with Elasticsearch happen through requests to **REST** endpoints.

The actions that can be performed depend on HTTP verb, namely:

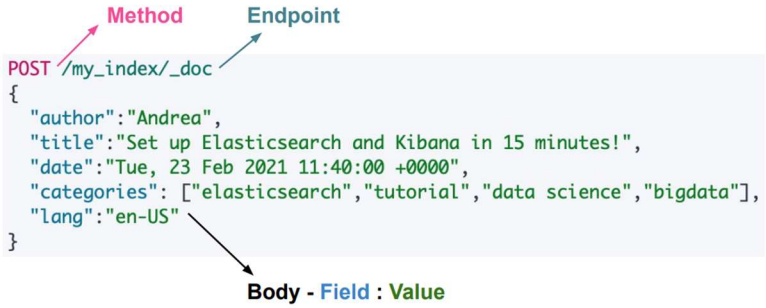
- GET is used to read document, indices metadata, mappings, etc.
- POST and PUT are often used to create new documents, indices, etc.
- DELETE is used to delete documents, indices, etc.

Requests can be sent from command line, software packages (e.g., Postman) or developer tools (Kibana).

Be aware of the differences between POST and PUT

- POST doesn't require the ID of the resource: If omitted, Elasticsearch takes care of creating and assigning IDs to documents.
    - POST /index_name/_doc
- PUT requires the ID of the resource.
    - PUT /index_name/_doc/document_id

To add a document to an index called “my_index”:



If the index doesn’t already exist, it will be automatically created.

To create a new index, it’s enough to use the PUT operator:

```
PUT /my_index
```

As an output, Elasticsearch will provide some metadata describing whether the index was successfully created.

```
"acknowledged": true,
"shards_acknowledged": true,
"index": "my_index"
```

When defining an index, it is also possible to setup some parameters, namely one or more aliases, number of shards, number of replicas, and many more.

```
PUT /my_index
{
  "settings": {
    "number_of_replicas": 3,
    "number_of_shards": 3
  }
}
```

When creating an index, it is possible to define the mapping straight away. Otherwise, it can still be defined after the index has been created.

```
PUT /my_index
{
  "mappings": {
    "properties": {
      "age": {"type": "integer"},
      "email": {"type": "keyword"},
      "name": {"type": "text"}
    }
  }
}
```

When asked to provide the complete mapping of a document, the parameters in “settings” and the mapping in “mapping” should be both provided.

Given the rate at which the shape and complexity of the data evolves, sometimes it may be necessary to add new fields to an index:

```
PUT /my_index/_mapping
{
  "properties": {
    "surname": {"type": "text"},
    "index": "false"
  }
}
```

Index defines whether a field is indexed and queryable.

Fields with index equal to “false” are returned only if returning the whole structure, but are not researchable.

Given the rate at which the shape and complexity of the data evolves, sometimes it may be necessary to update the mapping of some fields. A convenient way to do so is to use aliases.

Be aware of the following when such updates are applied:

- Updating the mapping of a field could invalidate the indexed data. If an update has to be made, it is necessary to create a new the index and reindex the data.
- Renaming a field could invalidate the indexed data. If a field must be renamed, add a new alias to that field instead.
- Each field has a set of mapping properties (e.g., index, and many more) which can be updated without any problem arising.

When Dynamic Mapping is applied, it is useful to visualize the mapping assigned by Elasticsearch to look for some flaws.

```
GET /my_index/_mapping
```

Sometimes, it is not even necessary to have a look at the whole mapping. Instead, it could be interesting to take a look at the mapping of a single field.

```
GET /my_index/_mapping/field/my_field
```

The field types provided by Elasticsearch are the followings:

binary	Binary value encoded as a Base64 string.
boolean	Either true or false.
keyword	Used for structured content (e.g., IDs, email addresses, hostnames, zip codes, etc.).
long	A signed 64-bit integer.
double	A double-precision 64-bit floating point number, restricted to finite values.
date	A string containing formatted dates. Internally managed as UTC.
object	A JSON object.
text	The traditional field type for full-text content. Text fields are **analyzed**.
geo_point	Latitude and longitude points.

In particular, the two most relevant are **keyword and text.**

Keyword type fields, used for content with a specified structure, are not analyzed, and are treated with exact matching, meaning that the string to match must be exactly the same of the one specified in a query.

Text fields instead are analyzed, and so when queried these fields are temporarily "cleaned" by removing for example commas and dots, depending on the employed analyzer. No exact matching is performed on text fields, but a more flexible kind.

The **analyzers** available are many, some of which are:

Standard Analyzer	It divides text into terms on word boundaries (using the UTS algorithm). It removes most punctuation, lowercases terms, and supports removing stop words.
Simple Analyzer	It divides text into terms whenever it encounters a character which is not a letter. It lowercases all terms.
Whitespace Analyzer	It divides text into terms whenever it encounters any whitespace character. It **does not** lowercase terms.
Stop Analyzer	It is like the **Simple Analyzer**, but also supports removal of stop words.
Pattern Analyzer	It uses a regular expression to split the text into terms. It supports lower-casing and stop words.
Language Analyzer	Language-specific analyzers (e.g., english, etc.)

If you are not sure about which analyzer is best suited for a specific case, they can be tested on sample texts to observe their behavior. Moreover, if none of the pre-built analyzers is suited, custom analyzers can be built.

```
POST _analyze
{
  "analyzer": "standard",
  "text": "I really like sunsets."
}
```

From now on, all the queries will be performed on a small, sample dataset which contains data about video games on Steam.

Field	Type
Developer	Keyword
Overview	Text
Publisher	Keyword
Tags	Text
Title	Text

→ Dynamic Mapping

A first observation is that Elasticsearch dynamic mapping assigned the types in a probably erroneous way. "Developer" and "Publisher" are associated with Keyword type, but they do not have a a-priori

specified structure to motivate this choice. On the other hand, “Tags” could have been keywords, but this is debatable. It’s important to inspect data when uploading it to the system.

Retrieving a single document can be done using its unique identifier:

```
GET /steam_overviews/_doc/0f5ERIIB_2yY7-gczBye
```

Elasticsearch returns a set of metadata (e.g., whether the document has been found, the name of the index, etc.) and a source field. Such a field contains all the data about the document that has been retrieved.


```
"_source": {
  "overview": "EverQuest® II is the epitome of massively multiplayer gaming - the ultimate blend of deep
    features, heritage, and community.Explore an enormous online game where friends come together for
    adventure and community. Immerse yourself in a living, breathing fantasy world filled with exciting
    locales, mysterious lore, monsters, gods, and dragons. Vast, beautiful, and dangerous, EverQuest II
    sets the standard for MMORPG online gaming. EverQuest II is free to play.Your character. Your story.
    Your adventure. 110 levels of unparalleled gameplay  Strong heritage and lore spanning 14 years and 15
    expansions  A thriving, friendly community with thousands of guilds  Thousands of creatures to battle
    and quests to complete Hundreds of gorgeous, expansive, and dangerous environments to explore
    Independently leveled tradeskill system with deep quest lines  Full support of different play styles
    from, including Solo, Group, Raid, and PVP  Robust housing system with dozens of styles, thousands of
    items, ratings, and more  *Optional content available for purchase.",
  "publisher": "Daybreak Game Company ",
  "developer": "Daybreak Game Company ",
  "title": "EverQuest II",
  "tags": "['Free to Play', 'Massively Multiplayer', 'RPG', 'MMORPG', 'Fantasy', 'Crafting', 'Open World',
    'Adventure', 'Exploration', 'Multiplayer', 'Action', 'Character Customization', 'Singleplayer',
    'Classic', 'Sandbox', 'FPS']"
}
```


Elasticsearch supports two different categories of query clauses:

- **Leaf Query Clauses** look for a particular value in a particular field. They can be used by themselves. This category includes *match*, *term*, and *range* queries.
- **Compound Query Clauses** wrap other leaf or compound queries and are used to combine multiple queries in a logical fashion or to alter their behavior. This category includes *bool* queries and many more.

By default, Elasticsearch sorts matching search results by **Relevance Score**, which measures how well each document matches a query. Each query type can calculate relevance scores differently. Score calculation also depends on whether the query clause is run in a Query or Filter Context

Query Context	Filter Context
“How well does this document match this query clause?”	“Does this document match this query clause?”
Decides whether the document matches the query or not. The relevance score IS computed. Applied with the *query* parameter.	Decides whether the document matches the query or not. The relevance score IS NOT computed. Applied with the *filter* or *must not* parameter.

Match Queries return documents that match a provided text, number, date or boolean value. Whenever a text field is provided, it is analyzed before matching.

```
GET /steam_overviews/_search
{
  "query": {
    "match": {
      "publisher": {
        "query": "Daybreak Game Company "
      }
    }
  }
}
```

Note that the above is the extended version of writing a query, in which “query” is repeated indicating the two different query contexts. The compact style would be the following:

```
GET /steam_overviews/_search
{
  "query": {
    "match": {
      "publisher": "Daybreak Game Company "
    }
  }
}
```

Match Queries are of type boolean. It means that the text provided is analyzed and the analysis process constructs a boolean query from the provided text.

Depending on the value of the operator parameter, the boolean query is either built using the OR operator or the AND operator.

- The **OR** operator matches the documents that contain at least 1 word among the ones returned by the analyzer.
- The **AND** operator matches the documents that contain all the words returned by the analyzer.

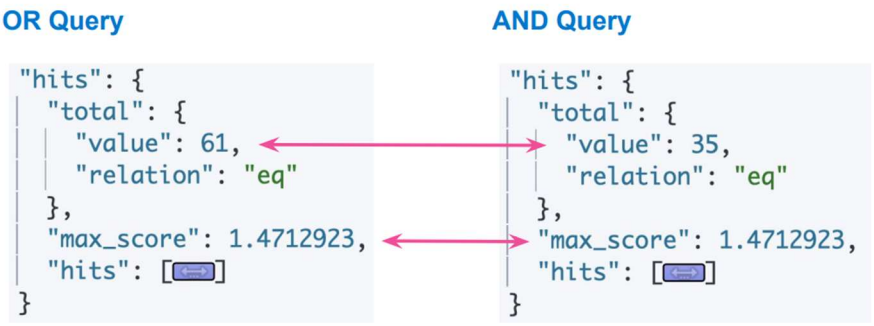
```
GET /steam_overviews/_search
{
  "query": {
    "match": {
      "overview": {
        "query": "free-to-play",
        "operator": "or"
      }
    }
  }
}
```

OR Query
free OR to OR play

AND Query
free AND to AND play

The above query searches for results that contain either “free” or “to” or “play”.

The AND query would usually bring less results and the average score with the OR is lower, because there are results that do not exactly match and have a lower score than others that are uniquely considered in the AND.



Term Queries return documents that contain an exact term in a provided field. Fields are not analyzed, and so no full-text search is performed.

DO NOT use term queries for text fields. Remember that text fields are analyzed, and such an approach make it difficult to find exact matches with strings inside the query that are not analyzed if we are using a term query.

```
GET /steam_overviews/_search
{
  "query": {
    "term": {
      "publisher": {
        "value": "Ubisoft "
      }
    }
  }
}
```

The value parameter contains the term to look for in the document.

Match queries:	Term queries:
Both the text inside the query and the text inside documents are analyzed before they are compared.	The text inside the query is not analyzed, while the text of documents is, and the two are compared.

Term Queries support the following parameters:

- *boost* is a float number used to decrease or increase the relevance score of a query. Useful for searches with multiple queries (default 1.0)
- *case_insensitive* is a boolean that allows insensitive case matching (default true)

These parameters must be added within the field, together with the value parameter.

Range Queries return documents that contain terms within a provided range. They are usually employed with numerical fields or dates.
Range Queries support the following operators:

- gt, i.e., greater than
- gte, i.e., greater or equal than
- lt, i.e., less than
- lte, i.e., less or equal than

```
GET /steam_overviews/_search
{
  "query": {
    "range": {
      "publish_date": {
        "gte": "now-10d/d",
        "lte": "now",
        "boost": 2.0
      }
    }
  }
}
```

When performing ranged queries involving dates, it can be useful to know a little bit of the so-called Date Math. There are three main parts, namely:

- An anchor date which is either now or a date followed by II.
- A math expression made of a math operator, a number, and a time unit (e.g., d for days, M for months, Y for years, etc.).
- (optionally) An operator to round the date to the closest, chosen time unit.

e.g., now +1d /M, now -1Y /Y, 2022.07.28\I\I +2M /M

In the first example, we start from the present moment and add a day, the round up by month. In the third, after the anchor date the \ symbols are used for escaping.

Boolean Queries match documents matching boolean combinations of other queries. It is built using one or more boolean clauses, each clause with a typed occurrence.

The table below highlights the available boolean clauses.

must	The query must appear in matching documents and it will **contribute** to the score.
filter	The query must appear in matching documents. However, unlike **must** the score of the query will be **ignored**.
should	The query should appear in the matching document. If so, it will **contribute** to the score.
must_not	The query must not appear in the matching documents. The scoring is **ignored**.

When the “should” is utilized, documents having the specified field condition will have higher score then the ones that do not.

Each of the clauses described may contain multiple conditions.

```
GET /steam_overviews/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"tags": "Multiplayer"} },
        {"match": {"tags": "Action"} }
      ]
    }
  }
}
```

You can write Boolean Queries that include all or just a few of the clauses described.

```
GET /steam_overviews/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"tags": "Multiplayer"} }
      ],
      "filter": [
        {"term": {"publisher": "Ubisoft "} }
      ],
      "should": [
        {"match": {"tags": "Action"} }
      ]
    }
  }
}
```

Elasticsearch supports three different types of aggregations, namely:

- **Metric Aggregations** calculate metrics from field values.
- **Bucket Aggregations** group documents into buckets, based on field values, ranges, or other criteria.
- **Pipeline Aggregations** take input from other aggregations instead of documents or fields.

Aggregations return the set of documents before the aggregations themselves. The *size* parameter defines how many of these documents are returned.

Aggregations can be run as part of a search by specifying the **aggs** parameter:

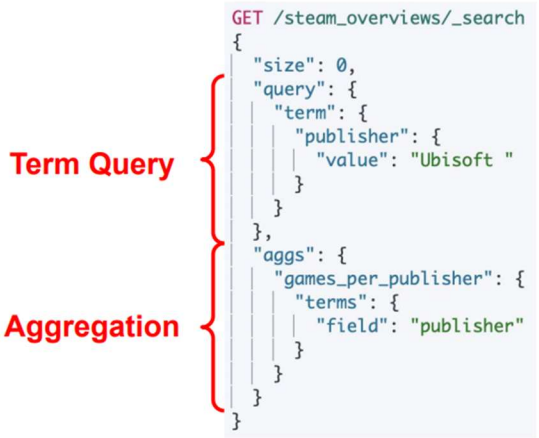
```
GET /steam_overviews/_search
{
  "size": 0,
  "aggs": {
    "games_per_publisher": {
      "terms": {
        "field": "publisher"
      }
    }
  }
}
```

```
"aggregations": {
  "games_per_publisher": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 44,
    "buckets": [...]
  }
}
```

```
{
  "key": "Daybreak Game Company ",
  "doc_count": 3
},
```

When writing “size”: 0 we are specifying not to return any document but just the aggregation that is built later.

Aggregations can be combined with normal queries to filter the data and then perform the aggregation.



N.B. What would happen if we were to perform the Aggregation and then the Term Query?

Within the same aggs operator, it is possible to perform multiple aggregations on different fields.

```
GET /steam_overviews/_search
{
  "size": 0,
  "aggs": {
    "games_per_publisher": {
      "terms": {
        "field": "publisher"
      }
    },
    "games_per_developer": {
      "terms": {
        "field": "developer"
      }
    }
  }
}
```

```
"hits": {
  "total": {
    "value": 64,
    "relation": "eq"
  },
  "max_score": null,
  "hits": []
},
"aggregations": {
  "games_per_publisher": {...},
  "games_per_developer": {...}
}
```

Elasticsearch supports Sub-Aggregations which are computed for each aggregation by considering the documents in each one of them.

```
GET /steam_overviews/_search
{
  "size": 0,
  "aggs": {
    "games_per_publisher": {
      "terms": {
        "field": "publisher"
      },
      "aggs": {
        "average_games_per_publisher": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

```
{
  "key": "Daybreak Game Company ",
  "doc_count": 3,
  "average_games_per_publisher": {
    "value": null
  }
},
```

Note that in the example “Price” does not exist in the mapping, so it is set to null.

Example queries

Some examples of requests are the followings:

- Define the full mapping describing a Person with 10 shards and 3 replicas.

```
PUT /people
{
  "settings": {
    "number_of_shards": 10,
    "number_of_replicas": 3
  },
  "mappings": {...}
}
```

```
"mappings": {
  "properties": {
    "personal_id": {
      "type": "keyword"
    },
    "name": {
      "type": "text"
    },
    "surname": {
      "type": "text"
    },
    "birth date": {
      "type": "date",
      "format": "yyyy-MM-dd"
    },
    "address": {
      "type": "text"
    },
    "eye_color": {
      "type": "keyword"
    },
    "height": {
      "type": "integer"
    }
  }
}
```

- Extract all the games which publisher is “Ubisoft”.

```
GET /steam_overviews/_search
{
  "query": {
    "term": {
      "publisher": "Ubisoft "
    }
  }
}
```

- Extract all games which tags field contains the “Free-to-Play” tag.

```
GET /steam_overviews/_search
{
  "query": {
    "match": {
      "tags": "Free-to-Play"
    }
  }
}
```

- Extract all games which tags field contains the “Free-to-Play” tag, prioritizing those whose developer is “Daybreak Game Company”.

```
GET /steam_overviews/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"tags": "Free-to-Play"} }
      ],
      "should": [
        {"term": {"developer": {"value": "Daybreak Game Company "} } }
      ]
    }
  }
}
```

- Extract all games which publisher is exactly “Lag Studios” without computing the relevance score.

```
GET /steam_overviews/_search
{
  "query": {
    "bool": {
      "filter": [
        {"term": {"publisher": "Lag Studios "} }
      ]
    }
  }
}
```

- For each developer, extract the count of the games which tags field contains the “Free-to-Play” tag.

```
GET /steam_overviews/_search
{
  "size": 0,
  "query": {
    "bool": {
      "must": [
        {"match": {"tags": "Free-to-Play"} }
      ]
    }
  },
  "aggs": {
    "f2p_per_devs": {
      "terms": {
        "field": "developer"
      }
    }
  }
}
```

Logstash

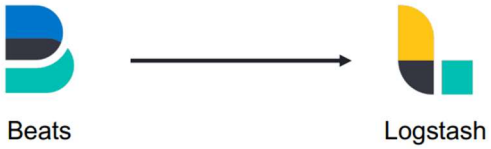
Logstash constitutes that part of the stack that is devoted to data ingestion. It is in fact a centralized platform implementing a streaming ETL (Extract Transform Load) engine that is simple and efficient. It provides centralized data collection and processing and enrichment of data on the fly.

It is also data agnostic and compatible with a wide range of integrations and processors.

Beats are platforms for data shippers, able to collect and ship log and metrics from hosts or containers. Many beats are available, the main ones are:

- Filebeat: Collects log data.
- Metricbeat: Gathers system and service metrics.
- Packetbeat: Monitors network traffic and protocols.
- Heartbeat: Checks availability by monitoring services.

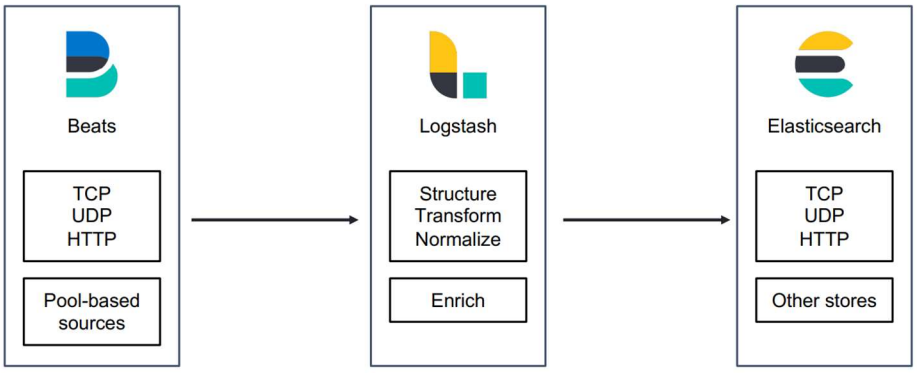
While Beats focus on data collection and shipping, Logstash focuses on processing and data normalization that is performed before the data is put inside Elasticsearch.



Of course, Logstash can also receive data from devices for which Beats are not deployed, via TCP, UDP, HTTP protocols or thanks to pool-based inputs like JDBC(Java Database Connectivity, java API to allow java applications to interact with databases).

Filter plugins, also called Processors, help Logstash with data wrangling and enable the building of pipelines to structure, normalize and enrich data. For example, a filter can be used to derive geographic coordinates from IP addresses or simply to exclude sensitive fields from the ingested data.

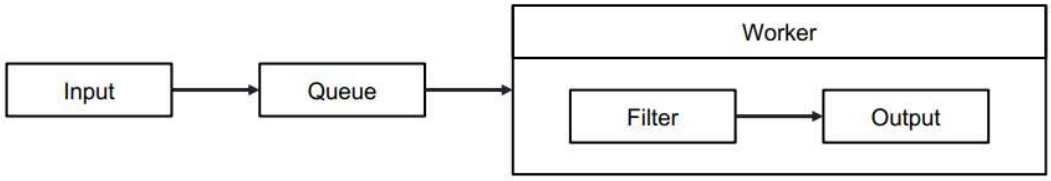
Other plugins are available to emit data to Elasticsearch or other data stores from Logstash in many ways, for example via TCP, UDP and HTTP.



The primary unit of data in Logstash are **events**: structured in a similar way to JSON documents, they flow through the pipelines of the Logstash system.

```
{
  "@timestamp" => 2021-16-02T01-01-01,
  "message" => "hello",
  "other_field" => {
    "nested_field" => 5678
  }
}
```

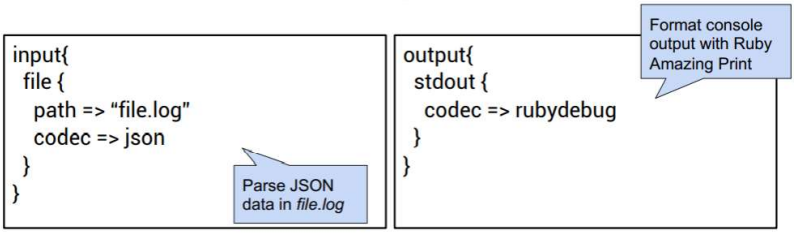
Logstash works as a pipeline, implementing a logical flow of data supporting multiple inputs and exploiting queues to buffer data.



Executers take data from the input buffer and execute operations and print out the outputs.

Multiple pipelines are supported in a single Logstash instance, that is a process inside the system, and so different and separate data flows can co-exist.

Codecs are available and used to change the data representation of an event through serialization and de-serialization, to define how data is encoded or decoded as it moves through the Logstash pipeline. They can be applied to input data from various sources to convert data in a common format or can be applied to output data to adapt to the needs of external systems by converting data in a suitable format.



Filters are also available, for example there are:

- mutate: field manipulation filter
- split: divide a single event into multiple events
- drop: delete an event

But also enrichment filters, like:

- geoip and dns: enrich IP addresses
- useragent: record information like browser type from web logs
- translate: use local data to translate parts of the events
- elasticsearch: query Elasticsearch
- jdbc: query databases that support Java

An example of a Logstash pipeline that receives data from a host that uses Beats, then processes that input via a filter and outputs directly to Elasticsearch is the following:

```
input {
  beats { port => 5043 }
}
filter {
  mutate { lowercase => ["message"] }
}
output {
  elasticsearch {}
}
```

Message management in Logstash follows the "*At Least Once*" message delivery policy, ensuring a message will be delivered to the recipient at least once, though it might result in duplicate messages being received. In fact, in most conditions, messages are delivered exactly once but unclean shutdowns can lead to duplicates due to the retry mechanisms implemented.

The **Dead Letter Queue** (DLQ) is a mechanism used in messaging systems or queues to handle messages that cannot be delivered successfully to their intended recipients. When a message fails to be processed or delivered after a certain number of retries, or encounters an error that prevents its successful delivery or processing, it is moved to the Dead Letter Queue, where configurable policies are applied to it.

Kibana

Kibana is the software in charge of building dashboards once the data is retrieved and available in Elasticsearch. It's a drag and drop open-source platform to automatically create dashboards for visualization without the need for coding.

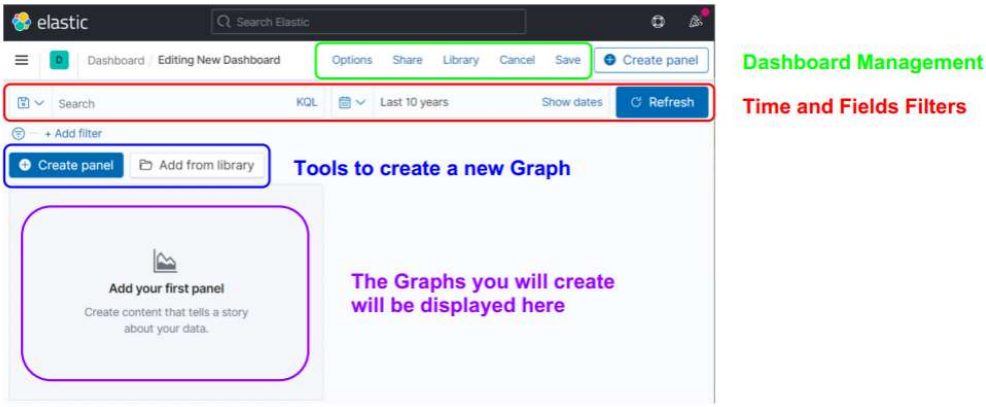
Kibana is simple and pretty intuitive to begin with. Despite such simplicity, it is highly customizable, allowing complex and detailed representations.

Through Kibana you can aggregate, organize, filter and represent data using many different data types and representations. A wide variety of graphs is available, although some data types, analyses and graphs are only available through a paid subscription.

Kibana is really powerful. It allows the creation of customized dashboards combining together basic representations, location-based analyses, time series, machine learning and graphs/networks.

For data ingestion, some modules are available to periodically collect the data and send it to Elasticsearch. Elastic agent instead offers a centralized way to set up integrations through integrations modules. The integrations ship with dashboards and visualizations, so insights into the data can be quickly generated.

Through Fleet mode, it's possible to add and manage integrations for popular services and platforms, providing an easy way to collect data.



Pre-defined components are available ready to use inside the platform.

Kibana is a very interesting tool to explore, organize and visualize huge amounts of data.

Kibana is easy to learn thanks to its intuitiveness and the tools it provides, but really hard to master. It can be noticed from the complexity it allows to manage while creating all the different panels.

10. DWH and Snowflake

OLTP and OLAP

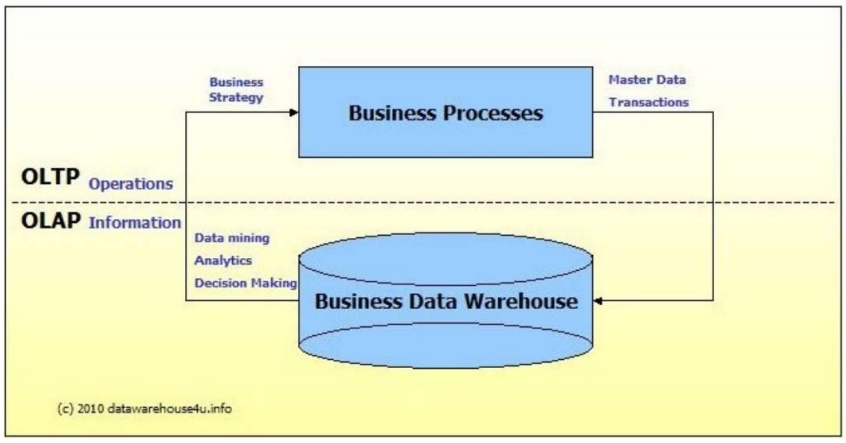
OLTP and OLAP are two different types of systems used in the field of databases:

- **OLTP** (Online Transaction Processing):
  - OLTP systems are transactional solutions designed for transaction-oriented applications, and describe processing at operational sites.
  - They handle day-to-day, routine operations of a business, such as order processing, banking transactions, or inventory management.
  - These systems emphasize fast query processing, data integrity, and concurrent access by multiple users.
  - OLTP databases typically have normalized data structures to minimize redundancy and ensure efficient transaction processing.
- **OLAP** (Online Analytical Processing):
  - OLAP systems are used for analytical and decision-making purposes.
  - They allow users to analyze large, integrated and multidimensional data in data warehouses from different perspectives, perform complex calculations, and support ad-hoc querying.
  - OLAP databases are optimized for read-heavy operations and complex analytical queries.
  - They often use de-normalized or star/snowflake schema structures to facilitate faster query processing and data analysis.

In summary, OLTP focuses on efficient transaction processing for daily operations, while OLAP emphasizes analytical capabilities for decision-making and data analysis.

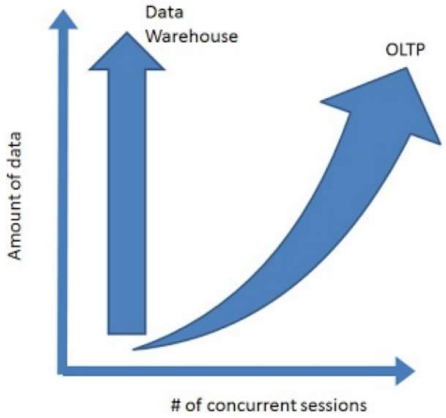
So, we can divide IT systems into transactional (OLTP) and analytical (OLAP). We can assume that OLTP systems provide source data to data warehouses, whereas OLAP systems help to analyze it.

Most of the technologies discussed so fare (except for some uses of the columnar databases and the ELK stack) are oriented to OLTP.



Talking about the interaction of these two kinds of systems, we can say that data flows from operations to analytics, while decisions are taken in analytics and applied to operations. So, OLTP systems provide the data to the data warehouse, where it is used to produce analytics results that have impact on operations.

While the number of people involved at managerial level does not usually grow so much with the dimension of a company, the same cannot be said about the workers at operational level, and with them the amount of data produced. So, the amount of available data does not significantly increase the need of more user sessions and more contemporaneous analytics of OLAP systems, while OLTP systems need to become more and more performant and capable with the company growing.

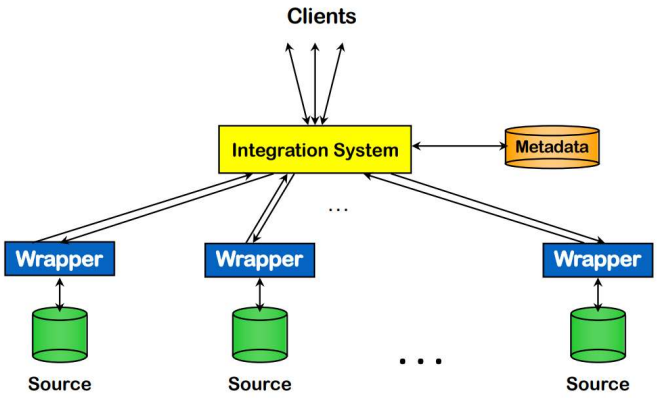


Historically, the processes of OLAP with analytics and decision making took the name of **Data Warehousing**.

Standard DB (OLTP)	Warehouse (OLAP)
Mostly updates to record operations Many small transactions Mb - Gb of data Current snapshot (of the company) Index/hash on p.k. Raw data Thousands of users (e.g., clerical users)	Mostly reads, to create analytics Queries long and complex Gb - Tb of data History Lots of scans Summarized, reconciled Hundreds of users (e.g., decision-makers, analysts)

Heterogeneous and distributed data sources need to be integrated in order to run analytics that need to produce coherent insights.

The traditional analytics approach is based on the integration of heterogeneous sources that are put together in a single storage by using wrappers and an ad-hoc integration system, with metadata as side product to use later in analyses.



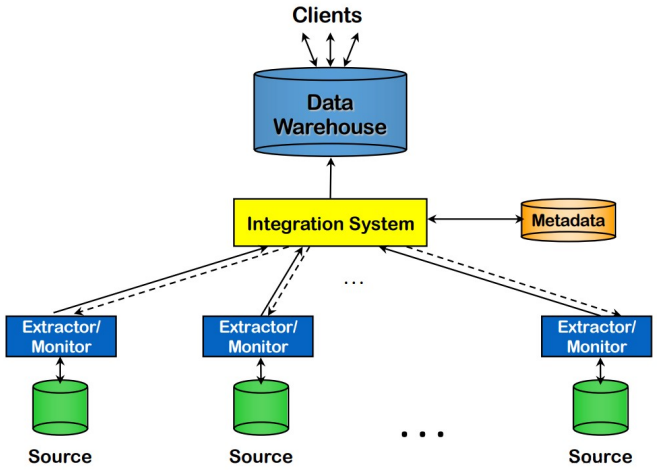
This approach is query driven, meaning that data analysis and insights extraction are primarily initiated by specific queries or questions posed by users. As a consequence, this approach works well for targeted analysis or when specific information is required. However, it can be limiting when exploring data for unforeseen insights or when the questions evolve or change over time. It also relies heavily on analysts' ability to ask the right questions.

Modern analytics, such as those employing machine learning or AI-driven approaches, often transcend this query-driven method by allowing systems to learn patterns, generate insights, and uncover correlations without explicit queries.

Traditional analytics are also lazy, in the sense that they prioritize efficiency by deferring actions or computations until necessary, avoiding unnecessary processing.

The other aspect to evaluate is that integration systems like those needed are on-demand: being them custom built integration solutions for a specific domain and for specific data sources, they need to be rebuilt every time they are needed or modified if something changes, for example if a new data source is added.

The solution to some of these issues is provided by the modern warehousing approach, in which both the data warehouse and the integration system are standard solutions, and information is integrated in advance before reaching the integration system, that performs less operations and sends integrated data to a storage facility (the data warehouse) where direct querying and analysis happen.



The main advantages of the warehousing approach can be summarized with:

- High query performance, but not necessarily on the most current information.
- Doesn’t interfere with local processing at sources, as complex queries are executed at warehouse while OLTP operations are executed at information sources.
- The information is copied at the warehouse, allowing both to store historical information and to be able to modify, annotate, summarize, restructure it without interfering with operations.

These are the main reasons why data warehousing is still widely adopted in industry, despite being a concept born in the 80s.

Shifting from the data warehousing process to the **data warehouse** as storage of data, we can define it as "a single, complete, and consistent store of data obtained from a variety of sources and made available to end users in a way they can understand and use it in a business context".

From an analytical viewpoint it is a:

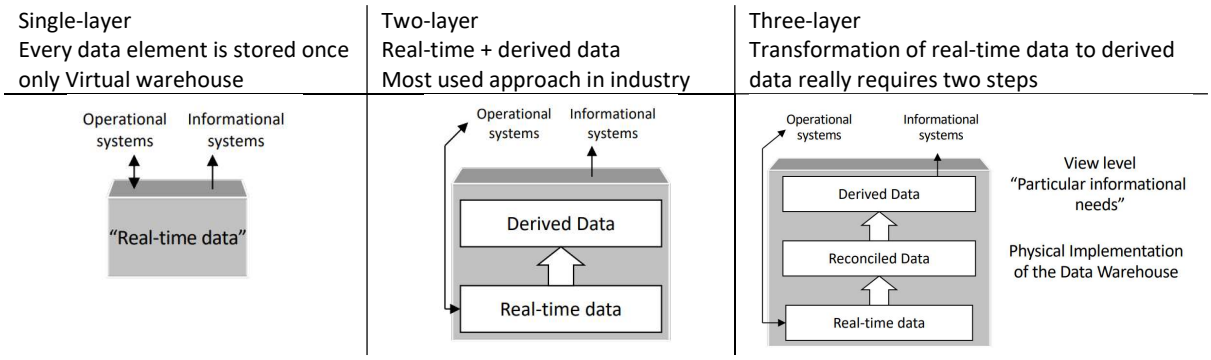
- subject-oriented,
- integrated,
- time-varying,
- non-volatile

collection of data that is used primarily in organizational decision making.

Data in a data warehouse is non-volatile, undergoes infrequently to updates that are instead periodic, and may be append-only, meaning that the only allowed operation is to add data to the existing one.

Moreover, data is decoupled from the operational data. This is important not only to allow all kinds of operations on analytical data, but also for performance reasons: operations must not be slowed down when running analytics.

Architectures of data warehouses can vary.

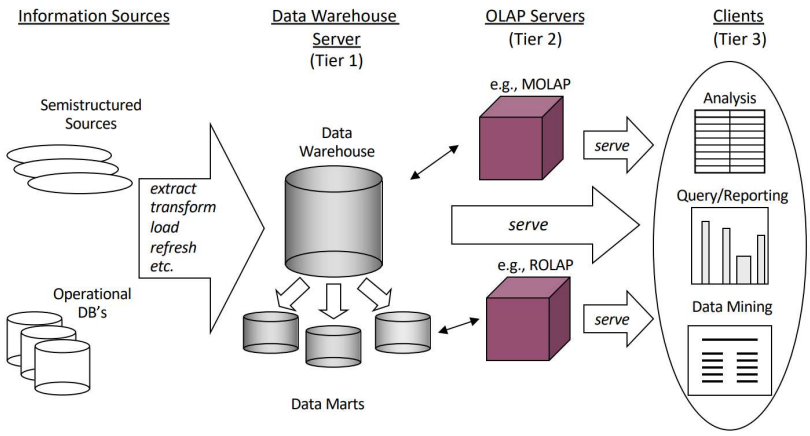


Issues in data warehousing regard:

- Warehouse Design: how do we design the schema for efficient data analysis?
- Extraction of data: how to copy the data in an efficient way? When to do it? With what frequency? What amount per time? Is it better to aggregate it or not?
- Integration: Cleansing & merging
- Optimization
- Maintenance

A **Decision Support Systems (DSS)** is an information system designed to help individuals and organizations, in particular knowledge workers, make faster and better informed decisions by providing relevant and timely data, tools, and models for analysis. DSS are used across various industries and functions to support decision-making processes.

On-line analytical processing (OLAP) is an element of decision support systems (DSS).



A Decision Support System usually relies on different **data marts**, each one focusing on a different perspective inside the company.

A data mart is a specialized subset of a data warehouse that focuses on specific business lines, departments, or functional areas within an organization. It's a smaller, more focused portion of a larger data warehouse, containing a subset of data that is tailored to the needs of a particular group of users. Organizations often use data marts to decentralize data access and analysis, allowing different departments or teams to have dedicated access to the specific data they need without having to navigate through the entire data warehouse. This facilitates faster decision-making and more focused analysis tailored to individual business needs.

There are two main possibilities for OLAP servers:

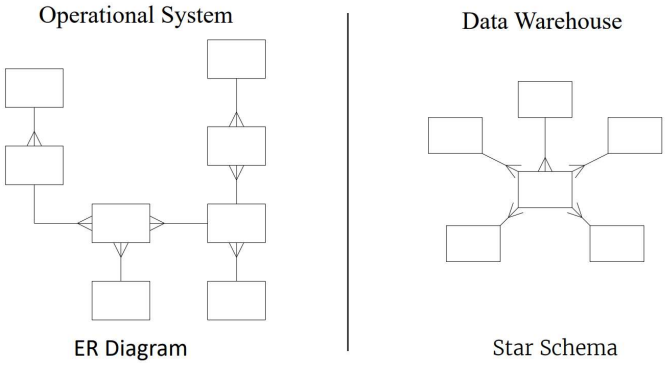
- Relational OLAP (**ROLAP**): Relational specialized DBMS to store and manage warehouse data with a OLAP middleware to support missing pieces.
- Multidimensional OLAP (**MOLAP**): Array-based storage structures meant to provide direct access to array data structures while implementing multidimensional data and operations.

When cloud computing services and infrastructure to create, manage, and maintain data warehouses are employed, we talk about **Cloud based data warehousing**.

Data warehouse (OLAP)	Operational system (OLTP)
Subject oriented	Transaction oriented
Large (hundreds of GB up to several TB)	Small (MB up to several GB)
Historic data	Current data
De-normalized table structure (few tables, many columns per table)	Normalized table structure (many tables, few columns per table)
Batch updates	Continuous updates
Usually very complex queries	Simple to fairly complex queries

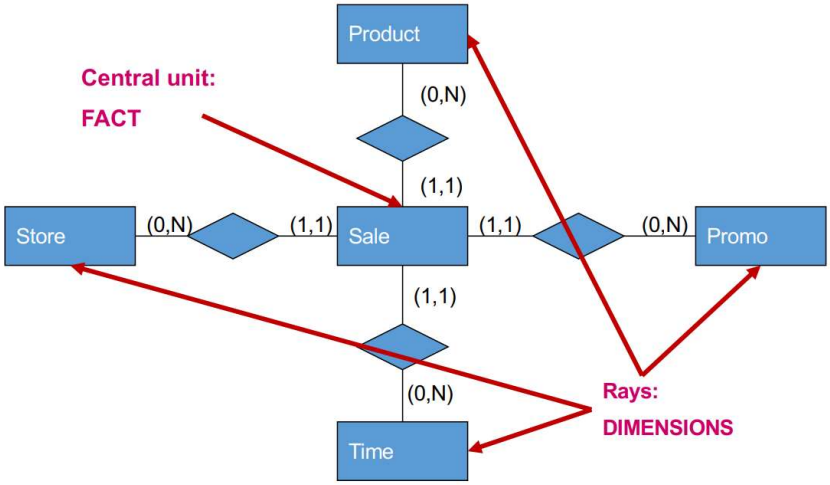
Data warehouse models

Because the required analyses are all very similar in structure, there is in most cases no need for a different schema every time one needs to be run, but a schema that works in the majority of the cases is enough. So, if in an operational system based on ER diagrams the structure of the database changes with the queries, in a data warehouse instead the structure is fixed, usually with the star schema.



The **star schema** is a schema that organizes data into a star-like structure with a central fact table surrounded by dimension tables.

The key elements are **facts** and **dimensions**, which determine the pre-defined directions (queries) of exploration of the schema.

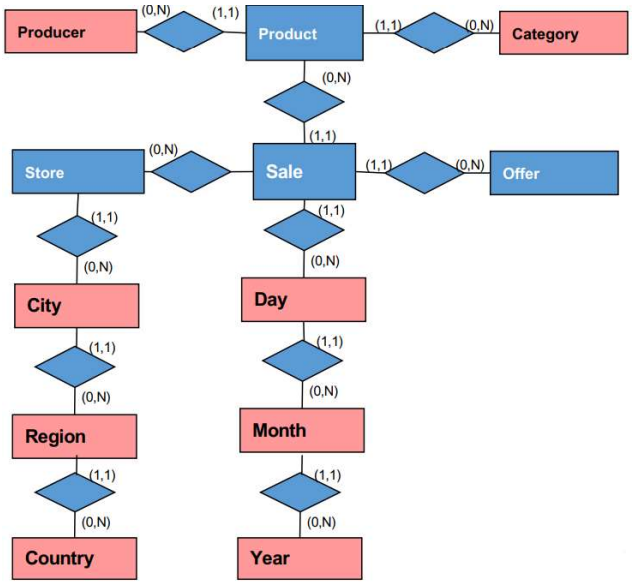


The central fact is an aggregate that has a composed key (from dimensions) and is normalized.

If, on the other hand, the capability to expand each dimension in more dimensions is needed to explore complex or varied data relationships, the snowflake schema might be more appropriate than the star schema.

While the star schema consists of a central fact table surrounded by de-normalized dimension tables, the **snowflake schema** further normalizes these dimension tables into multiple related tables, resembling a snowflake when visually represented.

In a snowflake schema, not-normalized dimensions are organized in a hierarchy.



The decision to use a snowflake schema versus a star schema often depends on factors like:

- Data Complexity: Snowflake schemas are useful when dealing with highly normalized data or complex relationships between entities.
- Data Integrity Needs: If maintaining data integrity and reducing redundancy are critical, a snowflake schema might be preferred.
- Query Performance vs. Query Complexity: While snowflake schemas offer data normalization, they might introduce increased query complexity and potentially impact query performance due to multiple joins.

Snowflake schemas are effective in certain scenarios where data normalization and maintaining strict relationships are prioritized, but they might require more careful query optimization compared to star schemas.

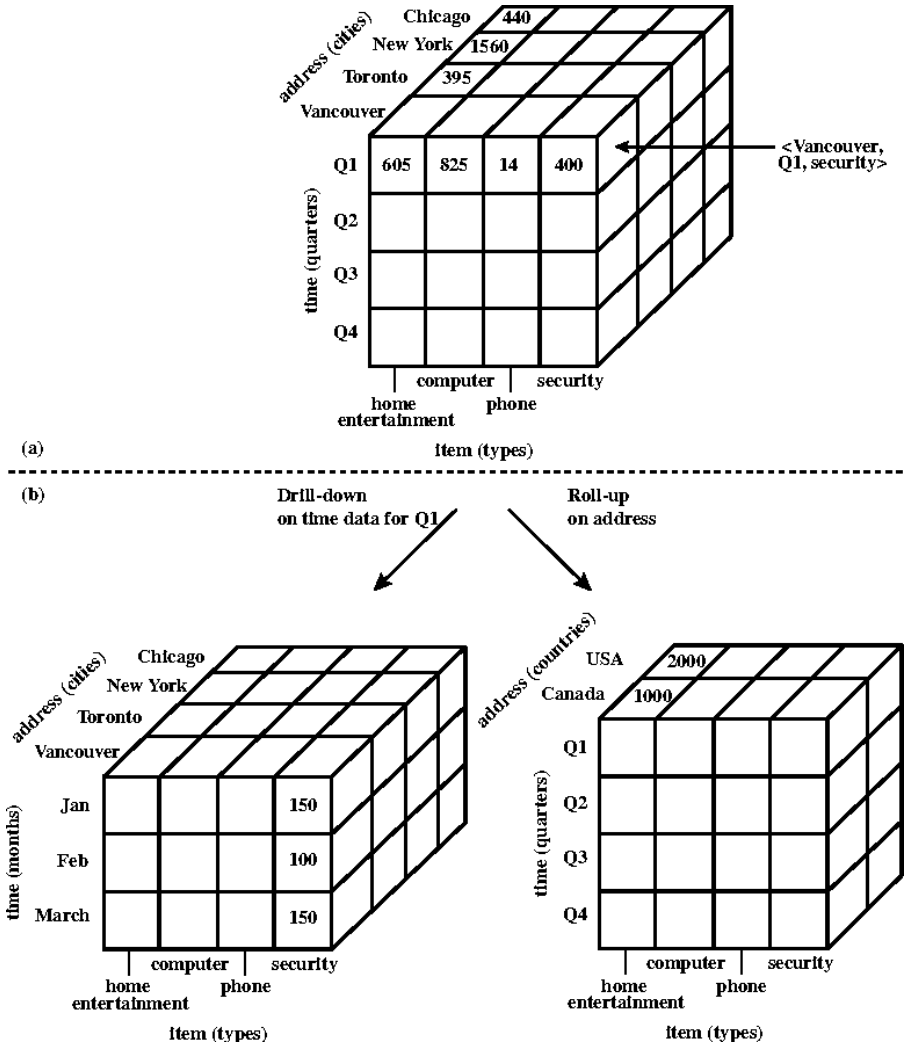
Star schema	Snowflake schema
Simplicity and Performance oriented structure De-normalized data Efficient for less Complex Relationships Aggregation and Reporting of summarized data	Data Integrity and Consistency, with reduced redundancy Highly Normalized Data Storage Optimization Handle Complex Data Relationships

Often, a hybrid approach or a mix of schemas might be used within a data warehouse to cater to different types of data and query needs.

When working with a hybrid or snowflake schema, basic operations are:

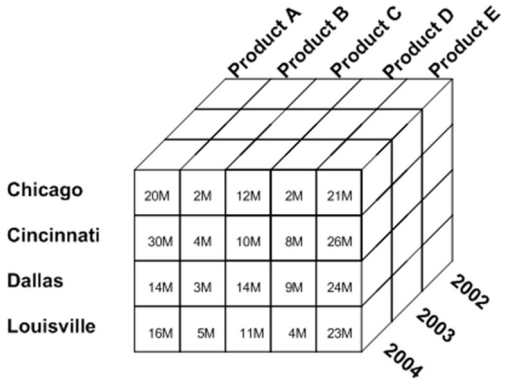
- **Drill-down**: add one analysis dimension (disaggregation)
- **Roll-up**: remove one analysis dimension (aggregation)

These operations can be visualized on a datacube:

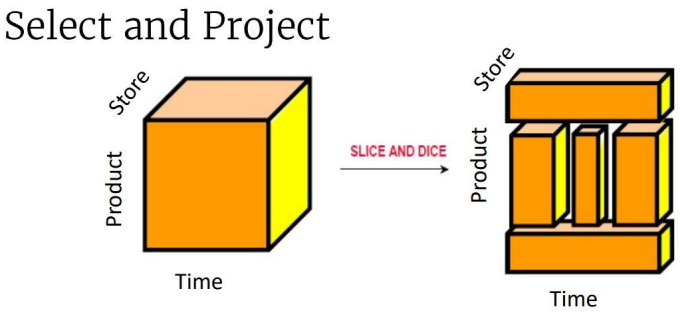


In the example, a drill-down divides the facts for Q1 in more months, while a roll-up on the address dimension groups the cities in their countries.

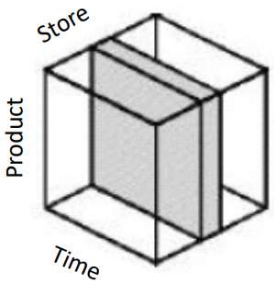
In the multidimensional cube, every small cube corresponds to a value for each of the dimensions (3 in the tridimensional case, but the dimensions can be more than 3) according to its position and projection on these dimensions. Moreover, every cube has one or more additional values that are the attributes of the fact that cube represents.



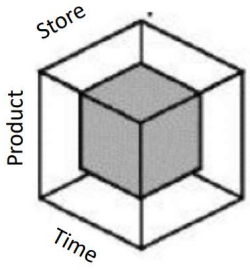
Slice and dice operations help in examining and analyzing data from different perspectives by selecting subsets of data along specific dimensions.



A **slice** operation involves selecting a single value along one dimension of the data cube to view a 2D subset of the cube.



The **dice** operation involves selecting and viewing a subset of the data cube by choosing specific values along multiple dimensions. It's similar to slicing but involves selecting multiple values across different dimensions simultaneously.



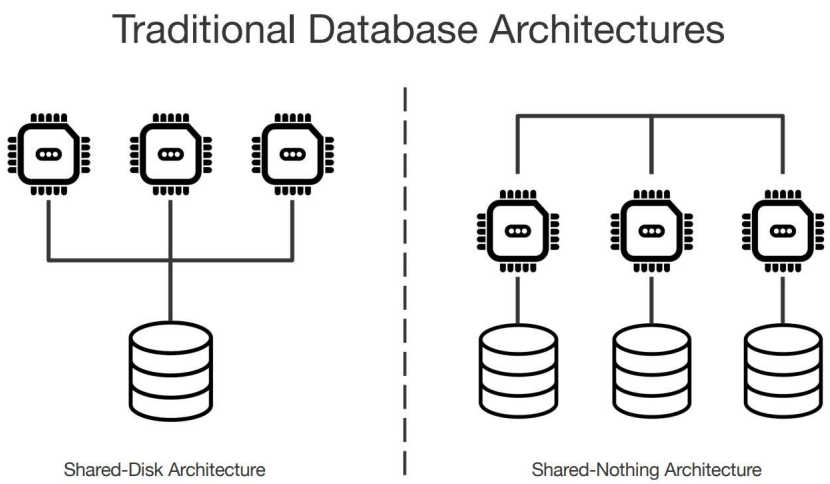
For example, using slice and dice it’s possible to create a sub-cube by specifying a range of values for each dimension.

Snowflake

Snowflake is an enterprise-ready, scalable and cloud-based data warehouse solution that automatically scales for balancing performances and costs.

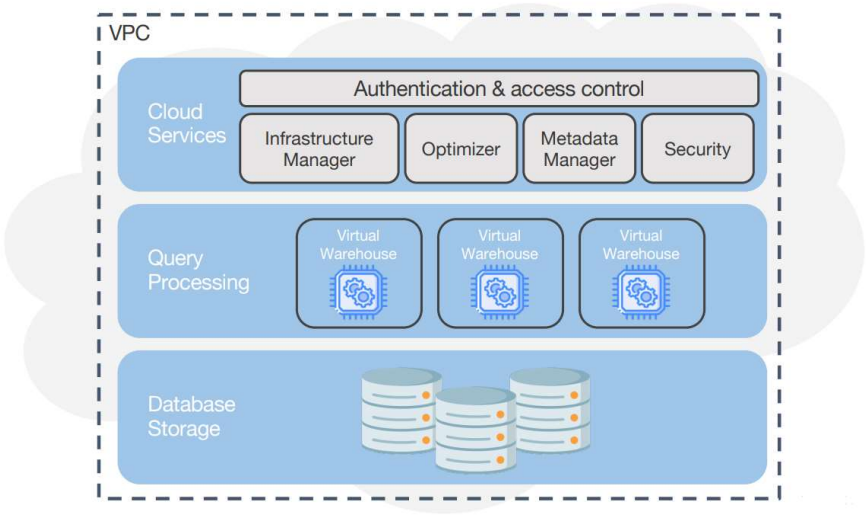
Snowflake implements a separation between compute and storage, that are independent in terms of configuration, when other databases combine the two together, meaning you must size for your largest workload and incur the cost that comes with it.

Traditional database architectures can be divided in two types:



With the **Shared-Disk Architecture**, multiple computing nodes or servers share access to a common storage area. Instead, in the **shared-nothing Architecture**, each node has its own dedicated storage and doesn't directly share data with other nodes, but only works on its own.

The architecture of Snowflake is structured in layers:



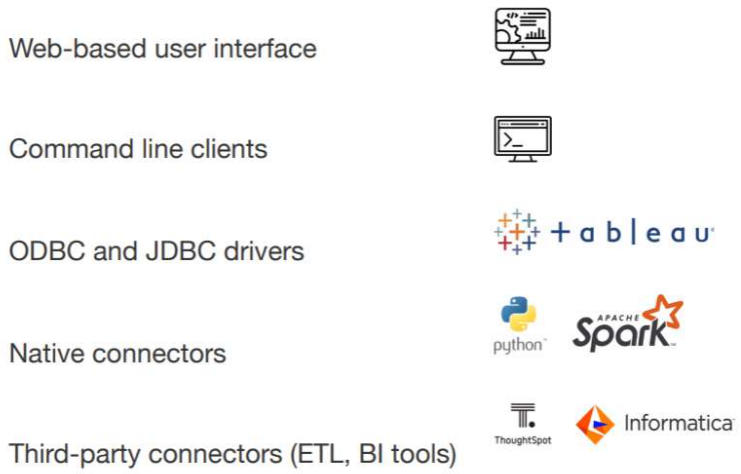
Data in Snowflake is organized into the internal optimized, compressed, columnar format. While this is its physical format, the conceptual format is the data warehouse schema.

This data is visible or accessible by customers only through a simplified and ad-hoc version of SQL queries writable by the users and executed in Snowflake. The idea is that the only skill required to the user is to know the basics of SQL and nothing more about databases.

Snowflake processes queries using virtual warehouses, that are independent compute clusters that do not share compute resources with other virtual warehouses, so to have no impact on the performance of the others.

Several services are managed as cloud services: authentication, infrastructure management, metadata management, query parsing optimization, access control.

Snowflake can be connected and integrated with other modern tools:



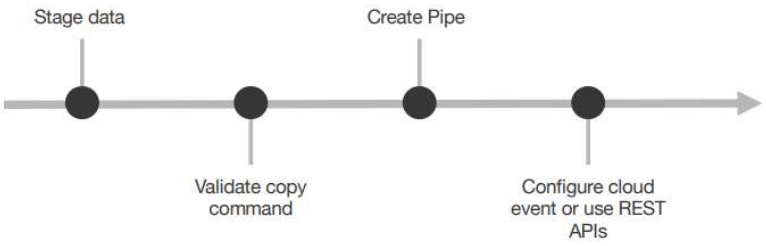
When ingesting data, Snowflake acts in an agnostic way with relation to the inputted data. The types of data supported for loading are: data with supported character encoding, compressed files, CSV, TSV, etc., JSON, Avro, ORC, Parquet, and XML format,Amazon S3, Google Cloud Storage, or Microsoft Azure.

Always talking about data ingestion, there are two possible behaviors on the data for Snowflake:

- Bulk load (Batch): read all data in a certain moment periodically, manually using Snowflake COPY command and exploiting virtual warehouses resources, also allowing transformations on the data.
- Continuous load (Stream): set up a "Snowpipe" used for loading streaming data.

When the continuous load is chosen and a snowpipe is set up, data is loaded as soon as it is introduced in the source operational system. Note that virtual warehouse computing resources are not utilized by the snowpipe.

To load bulk data from cloud and local storage: prepare the files to make the data consistent, then "stage" the data to make Snowflake aware of the data, then execute the COPY command to copy the data into the table and at last manage regular loads with scheduling.

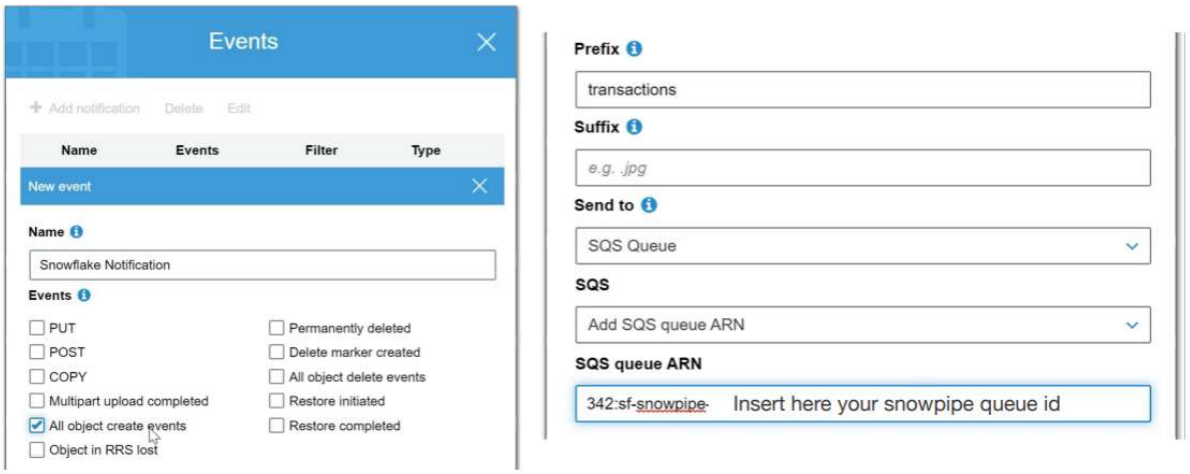


Staging the data in particular means informing the data warehouse about where the data lies. Staging area is an intermediate, transient place used to process data for extracting, transforming and loading processes. Cleaned data is moved to an area called the "stage", that is a convenient space from which to read data when moving it into the data warehouse. The classic method is to make use of cloud solutions as a cheap storage for raw data to use as staging area. Of course, data can also be staged on local file systems.

The snowpipe is a pipeline in which data can enter and at the end finishes in the data warehouse, this mechanism enables also the loading of streaming data. In this case, stage and copy steps are always performed, but in a pipeline that performs them continuously.

Performance optimization is a key point for Snowflake, in order to do so:

- Use dedicated virtual warehouses for different workloads
- Scale up for known large workloads
- Scale up with virtual warehouses for unknown workloads
- Design the production workflow to maximize cache usage
- Use cluster keys for partitioning large tables (partitioning data across servers)



11. MapReduce – Hadoop

MapReduce

In many cases, there is a task that is simple and easy to perform once, and the process to do it is known. But performing that task at industrial level, with big number of products and results might be complex.

Even if the operation themselves may be conceptually simple, the data to process may be huge and sequential execution wouldn't scale up and be suitable for the case.

The usual solution is to employ parallel execution to achieve greater efficiency.

However, parallel programming is hard, expensive and has some issues with relation to fault tolerance, data distribution and load balancing.

The idea of **MapReduce** is: instead of re-inventing the parallel processes (with their connected issues) every time a process needs to be scaled up, use a framework that can be used in different situations in which computation needs to be parallelized.

MapReduce is a programming model and an associated implementation for processing and generating large data sets. It's a blueprint and a style of development with the aim to easily scale up with the parallel computation and parallelization.

Every algorithm can be re-written using the MapReduce model and transformed into a highly parallel algorithm.

The advantages of MapReduce include:

- Automatic parallelization
- Automatic distribution
- Load balancing
- Network and data transfer optimization
- Fault tolerance

The name of MapReduce is due to the fact that it is based on the presence of two key actors, or more precisely workers:

- Mappers: read in data from the filesystem, and output (typically) modified data.
- Reducers: collect all of the mappers' outputs on the keys, and output (typically) reduced data, by aggregating values.

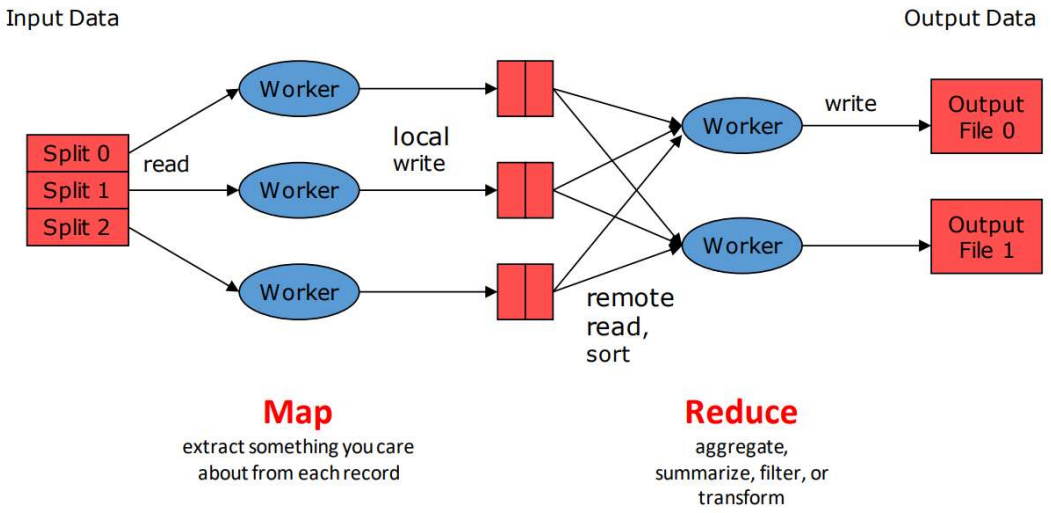
The data we are talking about is in terms of *key-value* pairs.

The workflow of MapReduce is the following:

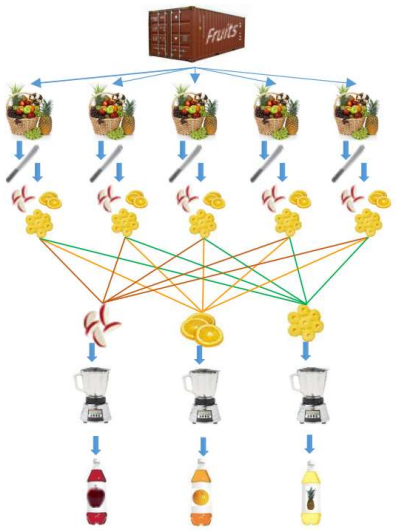
1. Read a lot of data, this big input is split into pieces by a splitter.
2. These pieces are processed separately by the mappers, that have the task to extract something you care about from each record.
3. Combine the results into intermediate items and use a shuffler to move data to the reducers.
4. The reducers aggregate, summarize, filter, or transform the data.

5. Thanks to a Committer, the result, composed by the union of all results produced by the reducers, are written in the output.

The strategy of MapReduce to solve problems related to moving data is to keep it as much as possible in the same place and run workers as close as possible to it, possibly even on the same machine holding the data. Thus, workers are moved, but not the data, except for one case in which it is inevitable, that is when it needs to be moved with the shuffler into the reducers. For this reason, the mappers should try to generate outputs that are as small as possible, so to then move as less data as possible.



To sum up the process, the task is split in parallel independent equal tasks that are executed in parallel with no dependencies, in an infinitely scalable way. Then, the results are shuffled and blend together to produce the final output.



Lets' see the key components more in detail.

The **Input Splitter** is responsible for splitting your input into multiple chunks that are then used as input for your mappers. Splits usually happen on logical boundaries, and the default is 64MB per chunk but can be changed according to needs.

A **Mapper** reads in input a pair $< K, V >$ and outputs another pair $< K', V' >$ by applying the defined mapping function. The mapper generates intermediate key-value pairs, where the key signifies a category or grouping, and the value represents some associated data, that will later be aggregated and processed by the reducer.

Ex. For our Word Count example, with the following input: "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am."

The output would be:
<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

A **Reducer** accepts outputs from multiple mappers, and collects values on the key, and then can apply a user-defined function to these groups. This means that all inputs with the same key must go to the same reducer.

For our example, the reducer input would be:
<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>
The output would be:
<The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Between the mappers and the reducers, we can find the Partitioner, or **Shuffler**, in charge of deciding which pairs are sent to which reducer. The strategy to do so can be chosen, but the basic and simplest strategy is a random strategy, that unfortunately doesn't always do the case. Custom partitioning is often required, for example, to produce a total order in the output.

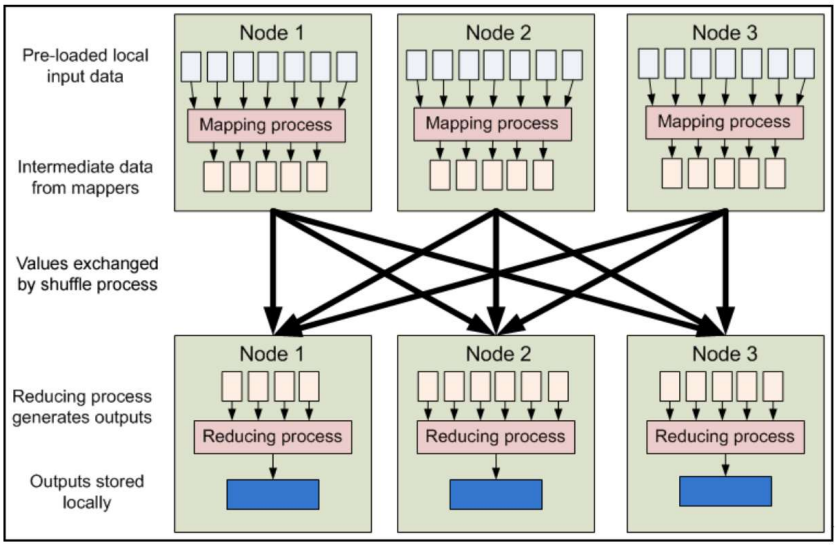
Some values might need to be sent to the same reducer
Ex. To compute the relative frequency of a pair of words <W1, W2> you would need to make sure all of word W1 are sent to the same reducer
Binning of results

The **Combiner** is an optional component, that acts as an intermediate reducer and positions itself just after the mappers along the pipeline. It reduces output from each mapper, reducing bandwidth and sorting. Its ultimate aim is optimization.

The **Output Committer** is the final step of the process. It is responsible for taking the reduce output, and committing it to a file. Typically, this committer needs a corresponding input splitter (so that another job can read the input).

In charge of monitoring the whole process there is a **master**, responsible for scheduling & managing jobs. It follows the principle that scheduled computation should be close to the data if possible. If a task fails to report progress, crashes, or something happen, the machine is assumed to be stuck, and is killed, and so the step is re-launched.

The advantage in using a Master is that the Master is handled by the framework, no user code is necessary. So, when using MapReduce, the only two things that need to be implemented are the Mapper and the Reducer.



Hadoop

Hadoop is a highly scalable computational platform that provides highly parallel computation for nodes of a network. It was built by taking standard efficient platforms and using them combined for scalability.

This software platform to easily process vast amounts of data includes:

- MapReduce – offline computing engine (for parallel scalability)
- HDFS – Hadoop Distributed File System
- Hbase – online data access (scalable database)

The main features of Hadoop are that it is:

- Scalable: It can reliably store and process petabytes.
- Economical: It distributes the data and processing across clusters of commonly available computers (in thousands).
- Efficient: By distributing the data, it can process it in parallel on the nodes where the data is located.
- Reliable: It automatically maintains multiple copies of data and automatically redeploys computing tasks based on failures.

Hadoop implements Google's MapReduce, using HDFS. MapReduce divides applications into many small blocks of work, while HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster. MapReduce can then process the data where it is located.

The target is to run on clusters of the order of thousands of nodes.

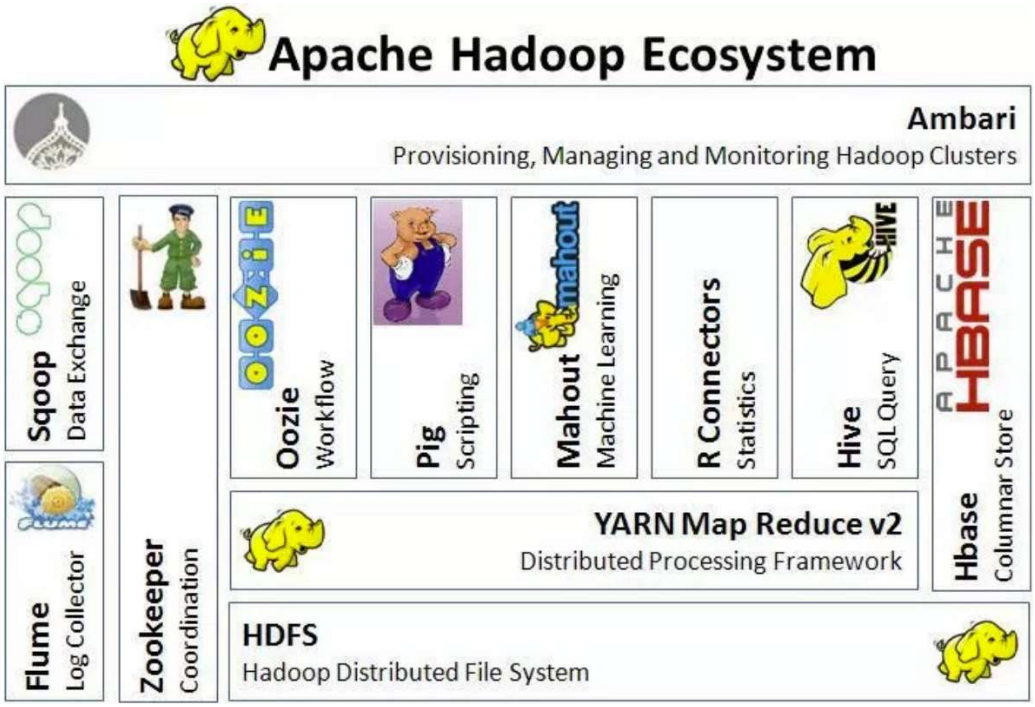
On an architecture so big, however, it's certain that at some point there will be a failure, because such numbers mean many potential failure points. But we want a platform that is scalable and resistant at the same time!

Some assumptions:

- Processing will be run in batches: High throughput as opposed to low latency.

- Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size.
- It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster.
- Applications need a **write-once-read-many** access model, called **WORM** model, typical in modern systems.
- Moving Computation is Cheaper than Moving Data.

Apache Hadoop ecosystem is set up as follows:



The main components are:

- **HDFS**: Hadoop Distributed File System, is a file system that is natively built to be distributed over thousands of servers.
- **YARN Map Reduce v2**: it's the Distributed Processing Framework constituting the computation part implementing MapReduce and running parallel tasks across nodes.
- **Hbase**: a columnar database optimized to save data on HDFS.
- **Zookeeper**: a coordinator to monitor the behavior of all components in terms of performance, execution, etc.
- **Sqoop**: handles data exchange and ingestion from relational databases.
- **Flume**: a log collector platform handling data ingestion from streaming data sources.
- **Ambari**: management platform used for provisioning, managing, and monitoring Hadoop clusters through a web-based interface.

The presence of different layers in this layered architecture facilitates the interaction with MapReduce, allowing multiple languages and models to be used transparently. The other components, allowing this compatibility and interaction, are:

- **Hive**: a SQL layer over MapReduce that gives a relational view over non-relational distributed data.

- **R Connectors**: allow to use R language to build statistics over the data, translating the R code and applying it to MapReduce.
- **Pig**: high-level scripting platform allowing to write complex MapReduce tasks using an SQL-like language called Pig Latin to analyze large datasets through a series of data transformations and operations.
- **Oozie**: workflow scheduler system to manage and coordinate complex workflows, allowing for the automation of job execution, scheduling, and coordination of various tasks.
- **Mahout**: a scalable machine learning library that provides various algorithms and tools to implement and execute machine learning tasks on large datasets using Hadoop.

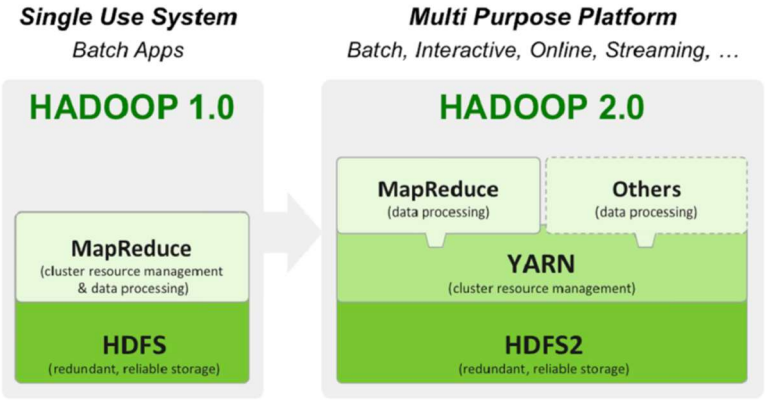
By making a quick comparison between Relational database systems and Hadoop, we notice that, being a Big Data platform, Hadoop wouldn't perform well on small data. In fact, Hadoop works by executing processes over big amounts of data in a single block, in batch.

	RDBMS	Hadoop
Data size	Gigabytes	Petabytes
Access	Interactive & Batch	Batch
Updates	Read & write many times	Write once, read many times
Integrity	High	Low
Scaling	Non Linear	Linear
Data representation	Structured	Unstructured, semi-structured

Hadoop is an Apache project, often offered as a service by companies on their cloud. For example, we can list: Cloudera Distribution for Hadoop (CDH), MapR Distribution, Hortonworks Data Platform (HDP), Oracle Big Data Appliance.

The original version of Hadoop had some limitations, having a Maximum cluster size of 4000 nodes, meaning that at maximum only 4000 instances of the same process could run in parallel, and exposed a single point of failure in a component that was the manager of the entire infrastructure.

To overcome these problems, **Hadoop 2.0** was created.



The YARN layer was added, focused on resource management and oriented at ensuring resilience. Moreover, different approaches to scalability, other than MapReduce, that weren't previously allowed, now were possible.

HDFS

HDFS or Hadoop File System is the distributed file system responsible for storing data on the cluster, built to be ready and fully compatible with MapReduce scalable distribution of processes.

Data files are split into blocks and distributed across the nodes in the cluster and each block is replicated multiple times. This way, HDFS provides redundant storage for massive amounts of data.

HDFS works best with a smaller number of large file (Millions as opposed to billions of files, typically 100MB or more per file) and most of the files are write-once.

In fact, HDFS is optimized for streaming reads of large files and not for random reads: it's optimized for reading huge batches at a time, not for looking for single information.

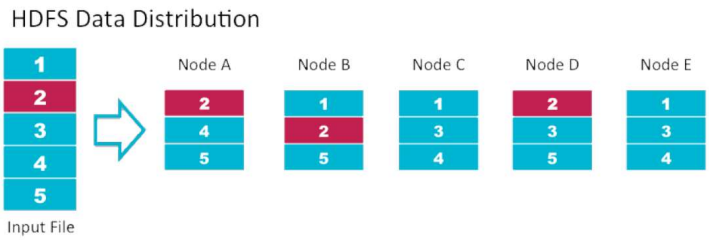
To store files, they are split into blocks (typically 64MB block size) that are in turn split across many machines at load time. So, different blocks from the same file will most likely be stored on different machines, and then they are replicated on multiple DataNodes, always across multiple machines.

The **NameNode** keeps track of which blocks make up a file and where they are stored. There is a single namespace for entire cluster, thus one single way to describe folders and files, even if the resources are actually distributed.

The client, said to be intelligent client, can find the location of blocks and access data directly from a DataNode.

Data coherency is achieved with the write-once-read-many access model, for which clients can only append to existing files.

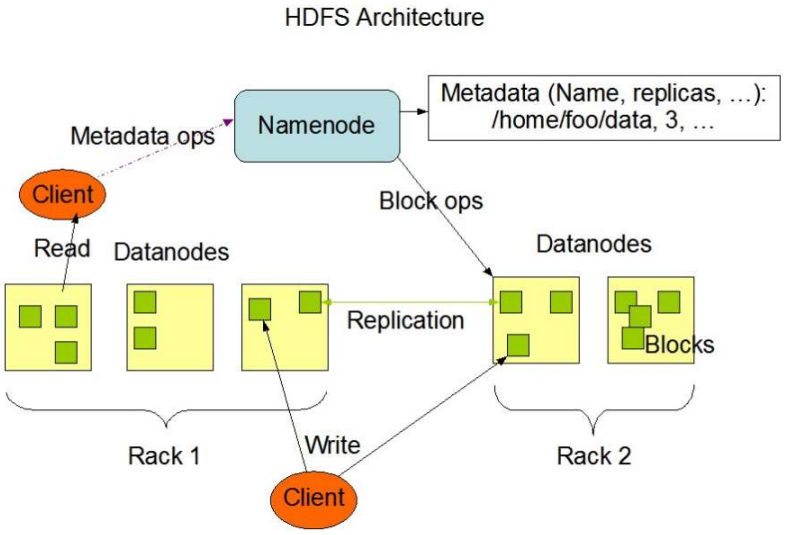
The default replication factor is of 3, so default duplication is said to be 3-fold:



The block placement strategy is to put one replica on a local node, then a second on a remote rack and the third on the same remote rack with the second. Additional replicas are randomly placed. A client can read from any of the replicas, choosing the nearest one.

When a client wants to perform a READ operation, he asks HDFS system where some data are and, once he gets a response, he goes on to read it in the right node. In particular, it queries the NameNode, that provides information about where the data are stored in form of metadata. The metadata for every searched file includes the list of blocks composing it, their location in DataNodes and the location of their replicas.

When a WRITE is due, data is written with its replicas through HDFS interface, logs are saved and metadata about this new data is generated and saved in the NameNode.



More in detail, the **NameNode** manages the file system namespace, by mapping a file name to a set of blocks and by mapping a block to the DataNodes where it resides. It also has a role in cluster configuration management and acts as replication engine for blocks by choosing new DataNodes for new replicas and balancing disk usage and communication traffic.

The entire metadata used by the NameNode is in main memory, so there is no demand paging of metadata. Metadata can be of different types: list of files, list of blocks for each file, list of datanodes for each block, file attributes, etc.

A transaction Log records file creations, deletions and other events.

A **DataNode** is a block server, storing data in the local file system and storing metadata of a block, in order to serve data and metadata to clients when required. It periodically sends a report of all existing blocks to the NameNode. It also forwards data to other specified DataNodes if needed.

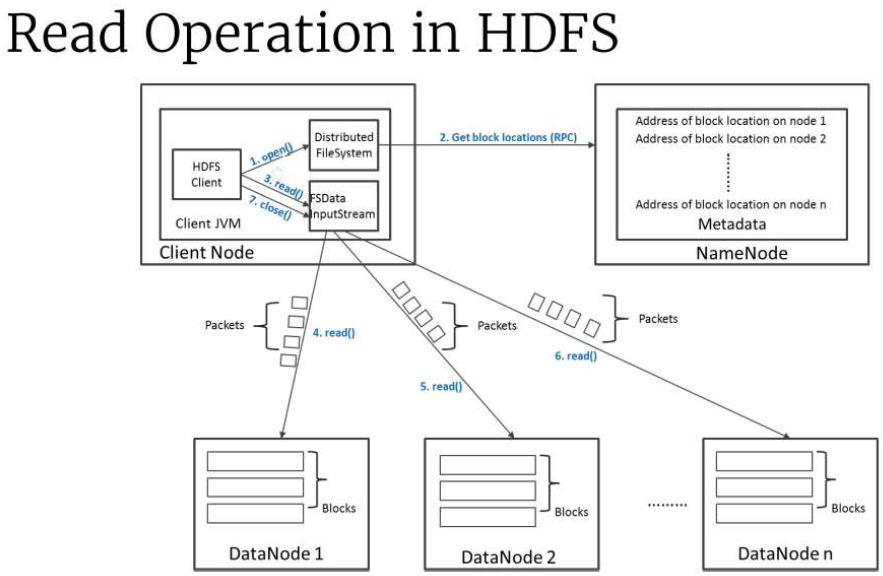
To guarantee resiliency, the Heartbeat strategy is exploited. DataNodes send a heartbeat signal to the NameNode, once every 3 seconds, and the NameNode uses heartbeats to detect DataNode failure.

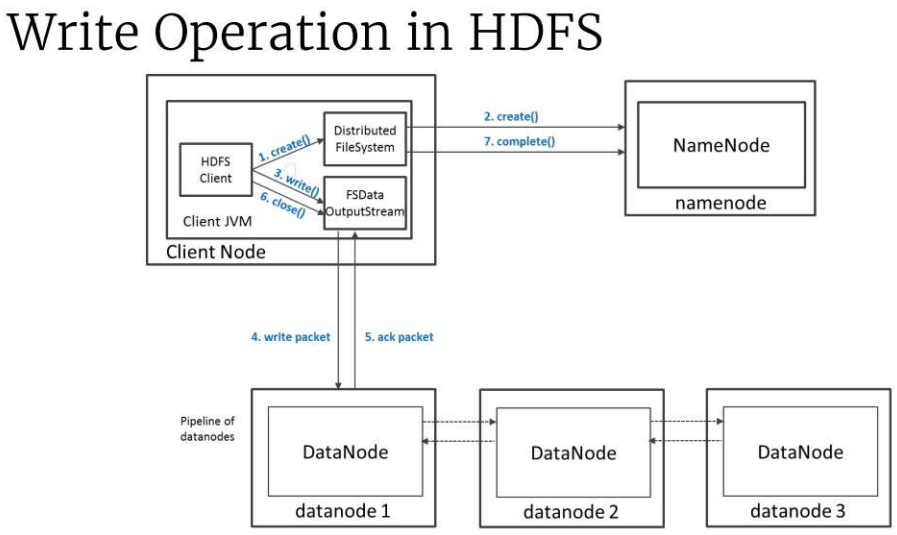
To check data correctness, data is validated with Checksums, in particular with CRC32. The DataNode stores the checksum after a file is created and upon file access it compares it with the newly computed one to see if some data has changed.

Despite being very useful, the NameNode was the single point of failure of HDFS 1.

The **Rebalancer** has the task to check that all nodes are loaded in a balanced way, and, if not, to move data in the moments in which the network has low activity. The goal is to have a percentage of occupation of disks on DataNodes that is similar across all DataNodes.

A **Secondary Name Node** copies FSImage and Transaction Log from NameNode to a temporary directory, merges them into a new FSImage and uploads it to the name node. The transaction Log on the NameNode is purged. The Secondary NameNode is a standby complementary Node for the primary one, to use in case of failure of the first.





HDFS can be accessed with different APIs, some of which are:

- Java API (DistributedFileSystem)
- C wrapper (libhdfs)
- HTTP protocol
- WebDAV protocol
- Shell Commands

However, the command line is one of the simplest and most familiar.

HDFS shell commands are similar to Linux File System commands, divided in User commands and Administration commands.

HDFS is a good fit to work with MapReduce, that exploits its behavior. Map tasks are in fact scheduled close to data and these tasks are small and natively organized to process the data in a distributed way.

Hadoop 2.0

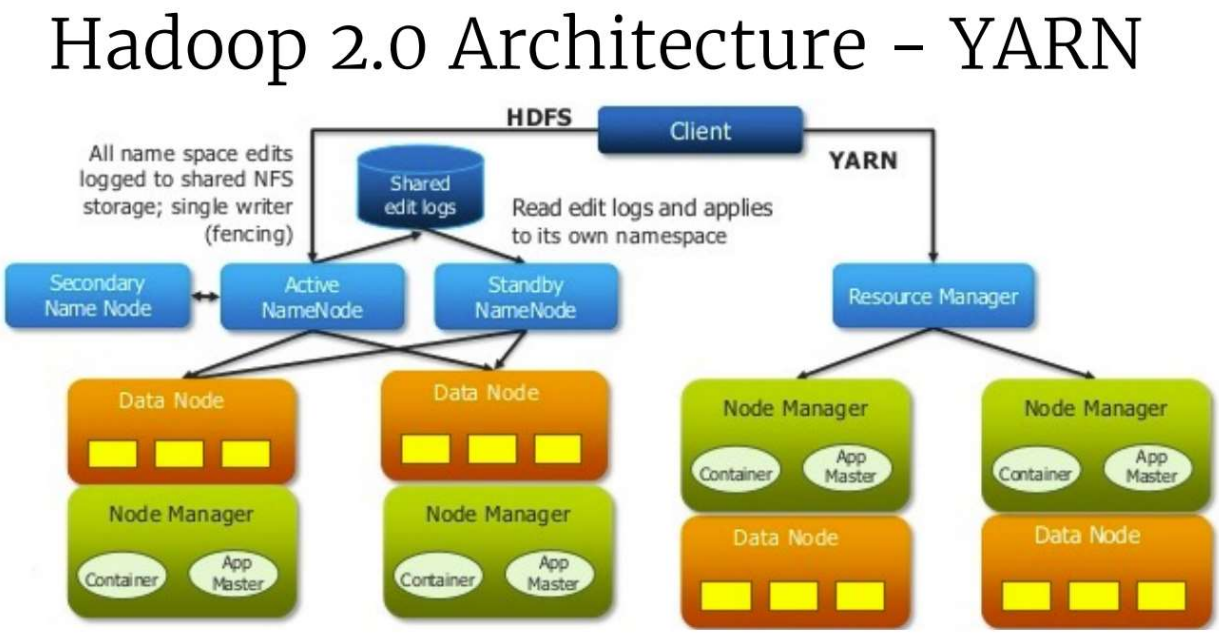
As already mentioned, Hadoop 2.0 was introduced to overcome the limitations of the first version, allowing more processes to run in parallel.

YARN (Yet Another Resource Negotiator) was added to the system. It's a resource management layer responsible for managing resources in a Hadoop cluster and scheduling user applications. YARN was introduced in Hadoop 2 to separate the resource management and job scheduling functionalities from the MapReduce engine, which was tightly coupled in Hadoop 1.

The two major functions of JobTracker are split:

- Global Resource Manager - Cluster resource management.
- Application Master - Job scheduling and monitoring (one per application). The Application Master negotiates resource containers from the Scheduler, tracking their status and monitoring for progress. Application Master itself runs as a normal container.

In addition, with NodeManager (NM), a new per-node slave is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting to the Resource Manager.



From the client we can split a data perspective (in the image, in the left side) and a computational perspective (in the image, on the right side). We can see how the NameNode can find the needed nodes in the datanodes, interacting with the data part of the servers, while the resource manager receives instructions from the client and creates the tasks to send to the Nodes where they will be executed, interacting with the processing part of the server (moving a piece of software is always cheaper than moving data).

When a client asks to perform some process, the tasks, that should be as small as possible, are sent to the nodes and each node talks to the local data.

By default, Hadoop uses FIFO to schedule jobs.

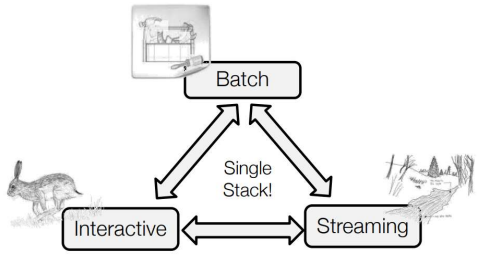
12. Spark

Big Data is used to:

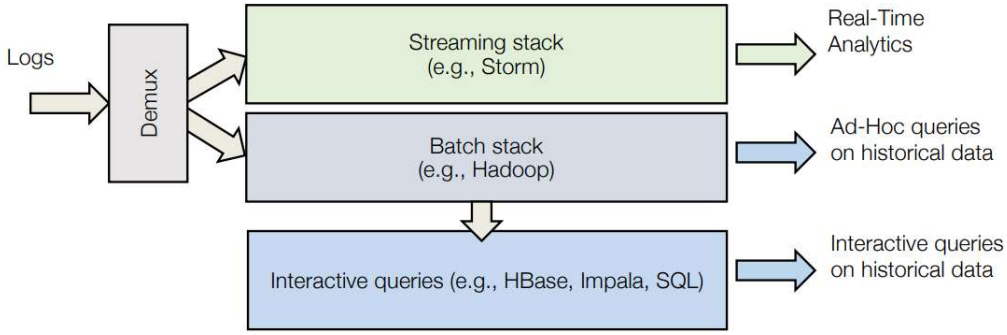
- Generate reports
- Run diagnosis
- Make decisions

But data is only as useful as the decision it enables.

The goals of data processing are **Interactive data processing**, with queries on historical data to enable faster decisions, and **Streaming data processing**, to enable decisions on real-time data. These two, plus a third scenario that is the **Batch data processing** scenario, that is when huge bulks and amounts of data are taken and a process is run on it, compose a single stack around data usage.

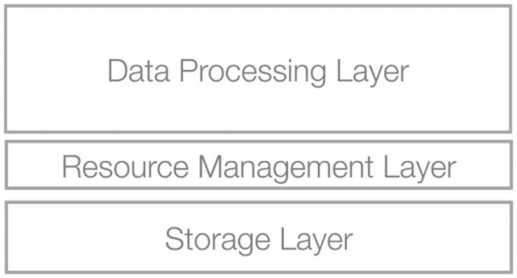


To cover these three scenarios simultaneously, we usually would need three different tools, one for each. We would find ourselves with an analytics stack, composed by three stacks, and its related challenges: the need to maintain these three separate systems and handle the cost of sharing data across them.



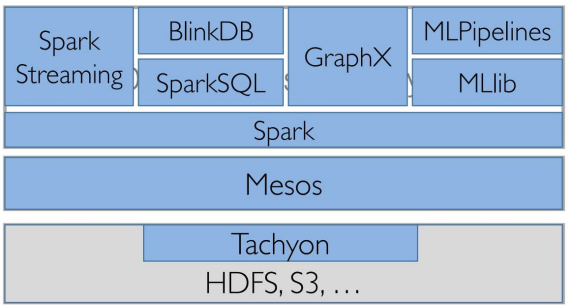
In addition, some real-time decisions need all the three kinds of analysis to get a complete answer. For example, to detect a DDoS attack we would need to detect the attack pattern in real time (streaming), to know if the traffic surge is expected or not (interactive queries) and to make queries fast with pre-computation (batch), not to talk about the complex algorithms we would need to implement.

A unified vision for every type of data processing would be ideal, organized in a data processing stack with the three needed layers.



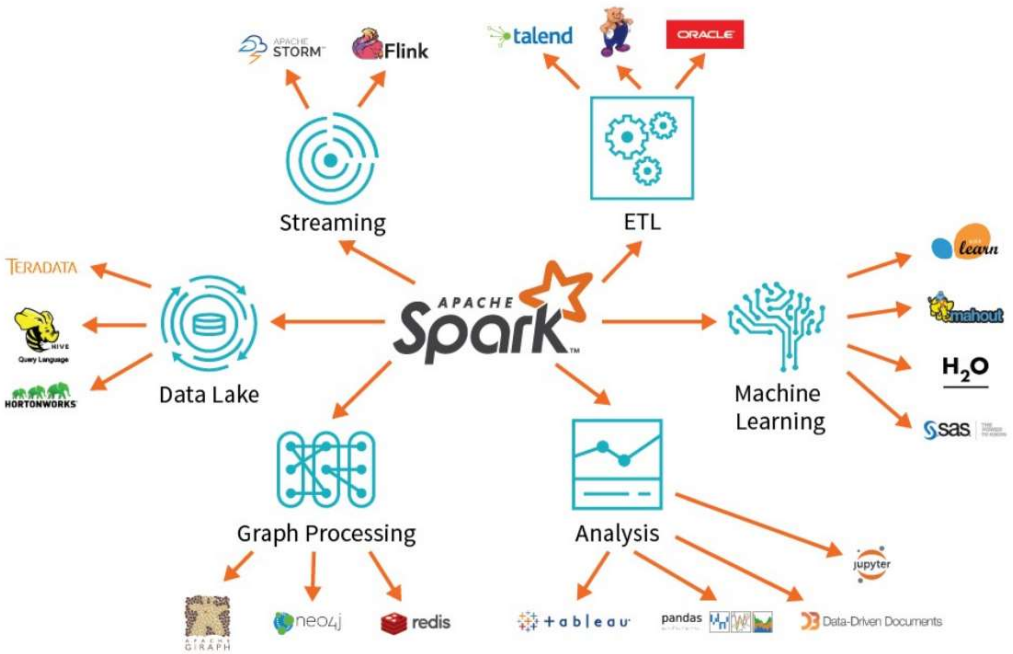
Spark is the architecture to bridge the three scenarios over the same stack. It's an open-source project on Apache, released in 2013 and famous for ease of use and speed.

Berkley Data Analytics Stack is composed by some components on top, covering different scenarios corresponding to different kinds of data needs, and just below them Spark works as the distribution engine, acting as a unifying layer.



What is hidden under Spark is the low level state of the art, that we try to preserve as much as possible.

The technology of spark is the underlying technology on top of which code for different scenarios is based.



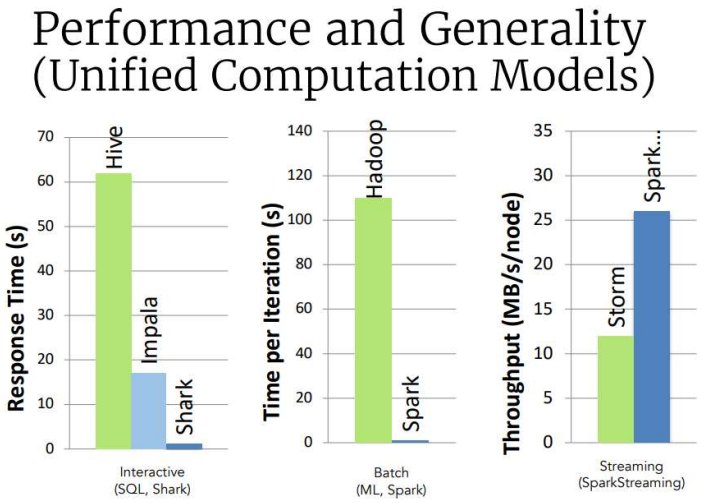
Spark acts as distributed execution engine, being fault tolerant and offering efficient in-memory storage. It can be up to 100 times faster than Hadoop in some cases and requires less code, while supporting interactive and iterative applications.

Spark Streaming is the component of Apache Spark that enables real-time stream processing, allowing large scale streaming computation. In order to do so, it is needed to see streaming data as the rest of data that is handled by the underlying architecture, so as static data.

The solution employed by spark streaming is to split streaming data and consider it as a sequence of static data. This way, streaming data can be processed as if it was static. Spark streaming in a way

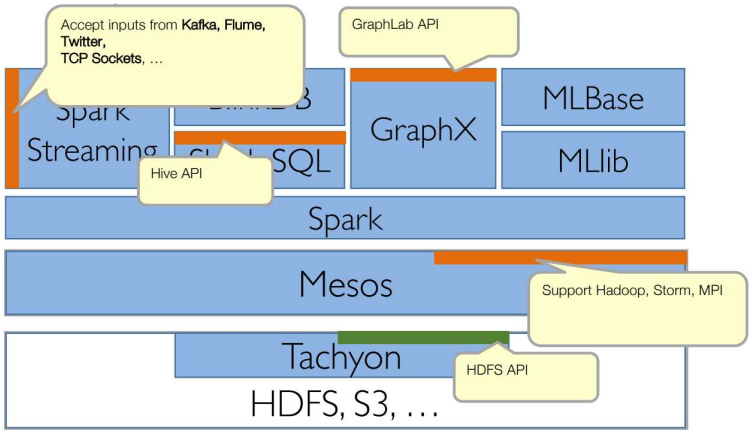
unifies Stream, interactive and batch computations, so that for the three it's possible to code the processes in the same way.

Spark SQL (that substituted the older Shark) is an integral part of Apache Spark that provides a higher-level abstraction for working with structured and semi-structured data. It enables users to execute SQL queries, manipulate data using DataFrames (structured APIs), and seamlessly integrate SQL queries with Spark's functional programming capabilities.



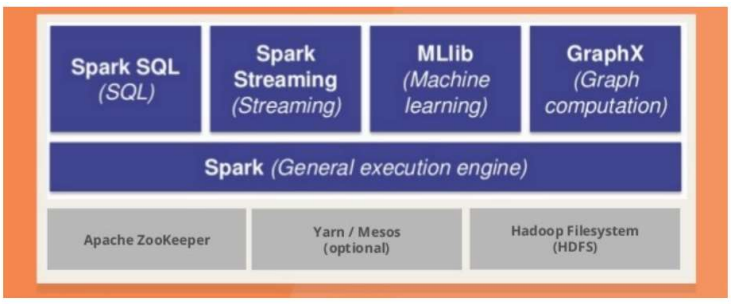
Unified programming models allow to work with a unified system for SQL, graph processing, machine learning, which all share the same set of workers and caches.

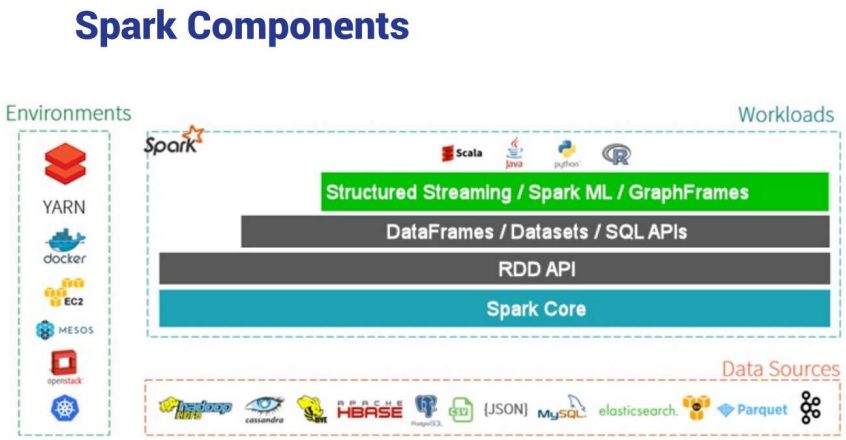
Some components of Apache Spark are replaceable with other libraries or components:



It's possible to think of Spark as scalable computation platform.

Spark can use data stored in a variety of formats: Cassandra, AWS S3, HDFS, and more.

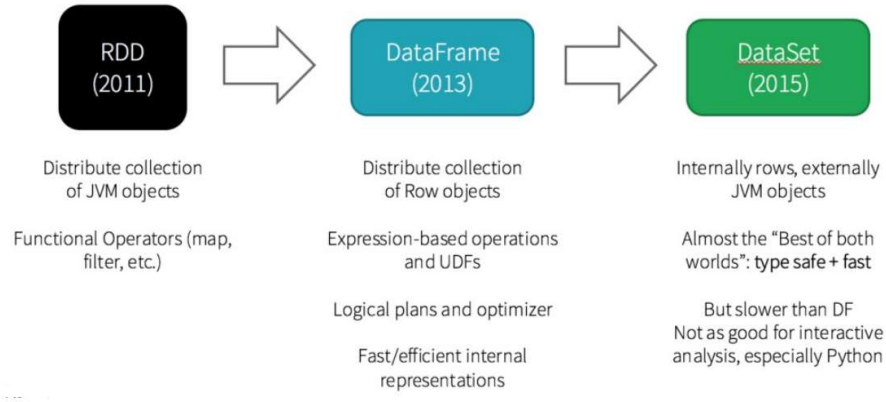




Looking at Spark components, we can notice how Spark Core constitutes the engine on top of which APIs are placed to give it commands.

Different Spark APIs have existed, to interact with the data in Spark. Thanks to them, traditional data structures are replaced with data types that are compatible with scalability and distribution.

Historically, DataFrame replaced RDD, and later DataSet expanded what DataFrame had to offer.



Resilient Distributed Dataset (**RDD**) constitutes the core idea of Spark to distribute data in a resilient way. A RDD is a distributed collection of objects that can be operated on in parallel across a cluster, and it is the primary abstraction in Spark for representing data.

The idea is to build an infrastructure that distributes data and lets the user write code that is automatically distributed for execution on data, and RDDs are one the basic blocks.

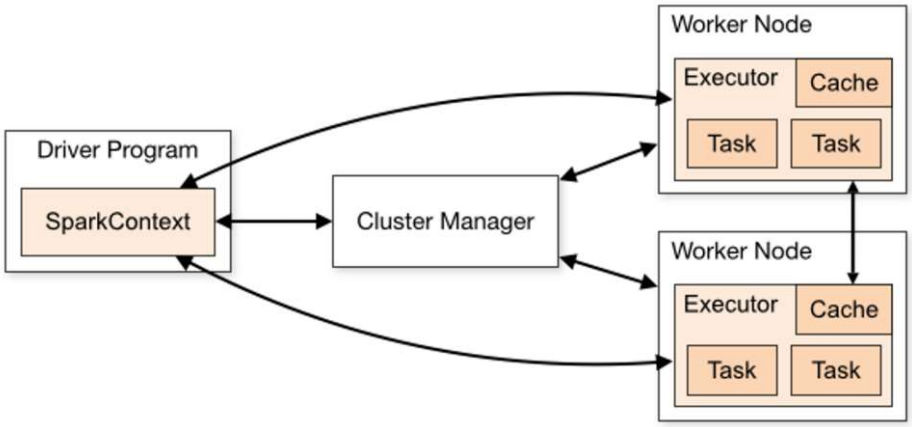
Resilient Distributed Dataset (RDD) has 4 main features:

- Distributed Collection of Data
- Fault-tolerant
- Parallel operation - partioned
- Ability to use many data sources

Spark RDDs is based on a component enabling access to the system, the **SparkContext**, which represents the connection to a Spark cluster and allows you to create RDDs (Resilient Distributed Datasets), perform operations on those RDDs, and control the configurations of your Spark application.

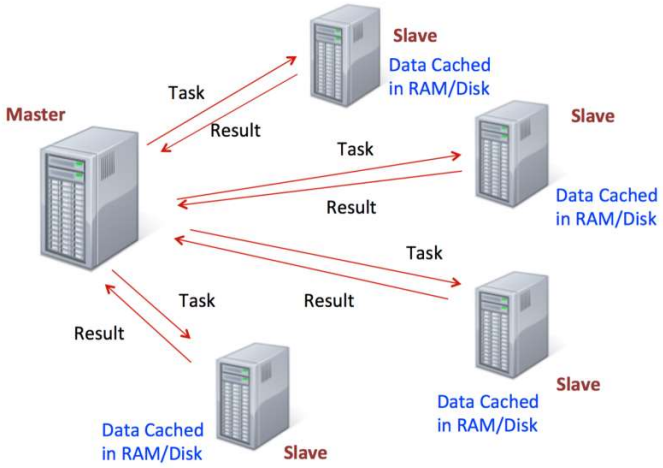
A central engine manager, the **cluster manager,** sends tasks to the worker servers.

Every virtual server is seen as a worker node, meaning that it is a node able to run code. Every worker has an executor environment, able to execute spark code locally and every worker may be assigned a list of tasks that it will run on the local data.



For efficiency purposes, most of the operations will be run in main memory (cache).

So, we have an architecture in which the Master sends tasks to nodes that execute them on local data and send back the results.



RDD is a data structure that is used directly by programs to process data in different locations.

RDDs are:

- Immutable: when a value is assigned to a variable, that value cannot be modified in memory. When changing a variable value, a new value is created but the existing one is preserved.
- Lazily evaluated: the policy is to run the minimum amount of tasks as possible, and a process is not run unless it comes the moment when it is really needed.
- Cacheable: even though the data is saved in the disks, the same data structures can be moved in main memory and the code can be executed there, in the cache, where it can be executed faster.

There are two types of Spark operations:

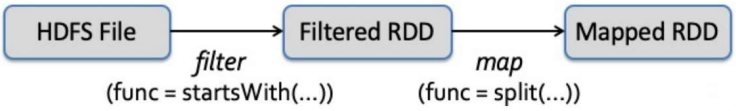
- **Transformations** are a recipe to follow, they are “lazy”. Transformations include map, filter, join.

- **Actions** perform what the recipe says to do and returns something back. Actions are executed in a forced way, they must be executed when requested. Actions include count, collect, save.

Transformations do not produce any result or output to the user, so the engine decides to not run them until they are needed by some action, that on the contrary produces some values for the user. So, the strategy is to execute transformations only when an action needs to be executed and this action depends on the transformations. Transformations are executed only in order to produce the output of an action, not for the sake of themselves.

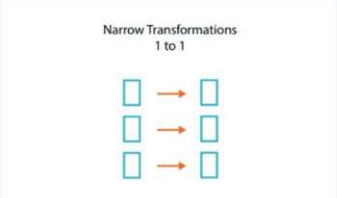
Transformations *(lazy)*	Actions
select	show
distinct	count
groupBy	collect
sum	save
orderBy	
filter	
limit	

The reason to delay the execution of transformations as much as possible is tied to the fact that transformations and actions are run in a pipeline as a sequence of concatenated steps. After each transformation, intermediate results are produced, and every step produces an RDD.



Some of the steps performed in the pipeline might result not useful at the end, when requesting an action and so executing the transformations. Delaying their execution allows for optimization by skipping unnecessary operations. Moreover, executing together transformations can save resources.

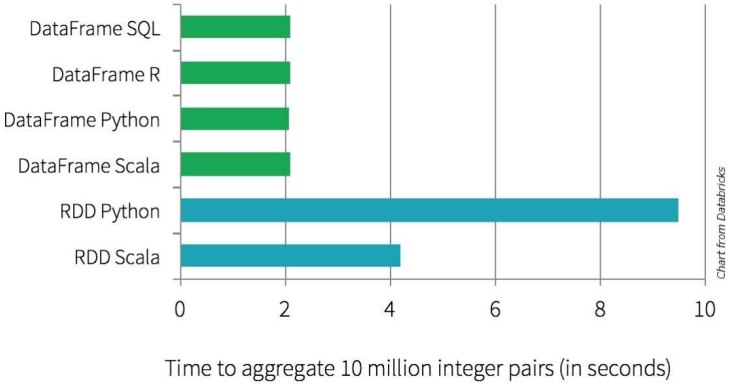
It's possible to distinguish between Narrow transformations, for which data is local and they can be performed by looking at a single piece of the dataset, and Wide transformations, that need to look for data around potentially the entire dataset.

Narrow Transformations	Wide Transformations
The data required to compute the records in a single partition reside in at most one partition of the parent RDD.	The data required to compute the records in a single partition may reside in many partitions of the parent RDD.
	
* `filter(..)`	* `distinct()`
* `drop(..)`	* `groupBy(..).sum()`
* `coalesce()`	* `repartition(n)`

With the introduction of **Spark 2.0**, RDD syntax is substituted by a DataFrame based syntax. But the way files are being distributed can still be thought of as RDDs, it is just the typed-out syntax that is changing.

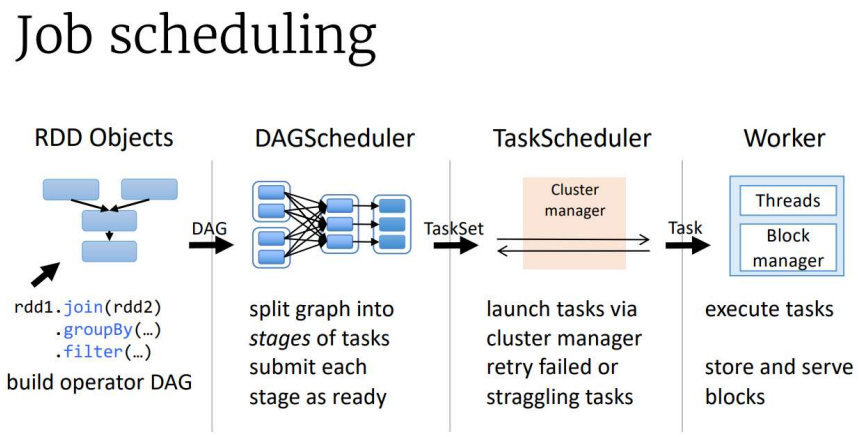
A **DataFrame** is a programming data structure that represents a table on which it's possible to run operations. However, in a hidden way, a DataFrame implements an RDD, not a table.

The characteristics of a DataFrame are to have a user-friendly API that is also uniform across languages, and that is offers improved performances via available optimizations.



DataFrames can also be used with SQL.

Fault recovery is implemented in Spark by keeping a log of all operations (lineage information) and using it to be able to re-compute missing data after potential failures.



Spark works with different formats: text/CSV, Json, Avro, Parquet, DataFrames, ORC, and others.

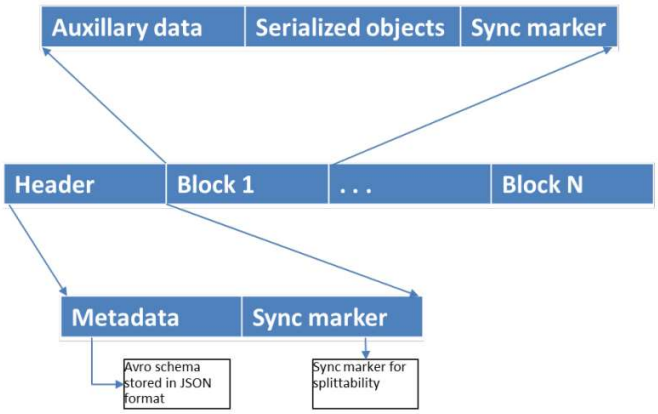
AVRO is a data serialization system which provides a data serialization format. The idea of AVRO is to be independent from technology and platform, so everyone can easily write it and move it without any loss of information.

AVRO data is described using language independent schema written in JSON and then data is encoded in binary format. Each file contains a description of its own format and a description of the shape of the data.

Sample AVRO schema in JSON format

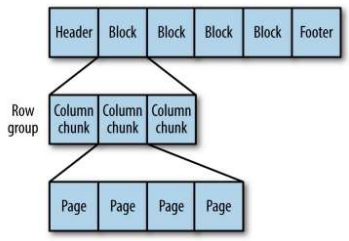
```
{
  "type" : "record",
  "name" : "tweets",
  "fields" : [ {
    "name" : "username",
    "type" : "string",
}, {
    "name" : "tweet",
    "type" : "string",
}, {
    "name" : "timestamp",
    "type" : "long",
} ],
  "doc:" : "schema for storing tweets"
}
```

Avro file structure



A variation of AVRO, but specifically designed to work into a distributed environment, is **Parquet**. Parquet is state-of-the-art, open-source columnar format the supports most of processing frameworks and is optimized for high compression and high scan efficiency. In a parquet file, data is saved in a columnar format.

Internal structure of parquet file



Configurable parquet parameters

Property name	Default value	Description
parquet.block.size	128 MB	The size in bytes of a block (row group).
parquet.page.size	1MB	The size in bytes of a page.
parquet.dictionary.page.size	1MB	The maximum allowed size in bytes of a dictionary before falling back to plain encoding for a page.
parquet.enable.dictionary	true	Whether to use dictionary encoding.
parquet.compression	UNCOMPRESSED	The type of compression: UNCOMPRESSED, SNAPPY, GZIP & LZO

The columnar storage format offered by Parquet is a 1:1 mapping to the data distribution, ideal for Spark. The key strength is to store nested data in truly columnar format using definition and repetition levels. This makes Parquet the most efficient way to store and read data in a DataFrame.

Nested schema

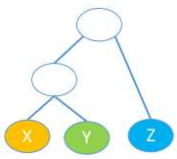
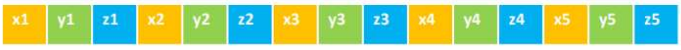


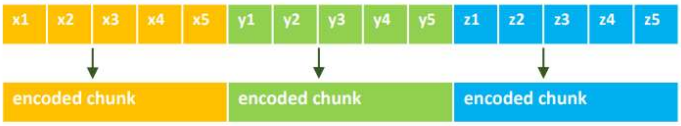
Table representation

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	Y5	z5

Row format



Columnar format



Parquet formats match perfectly the way data is saved in HDFS.

13. Data-driven innovation

Data-driven innovation refers to the process of using data to create new products, services, or insights that drive business, societal, or technological advancements. It involves leveraging large volumes of data, often from diverse sources, to gain valuable insights, make informed decisions, and create innovative solutions.

Data-driven innovation starts from the problems it has to solve. Framing these problems is the first step to accomplish.

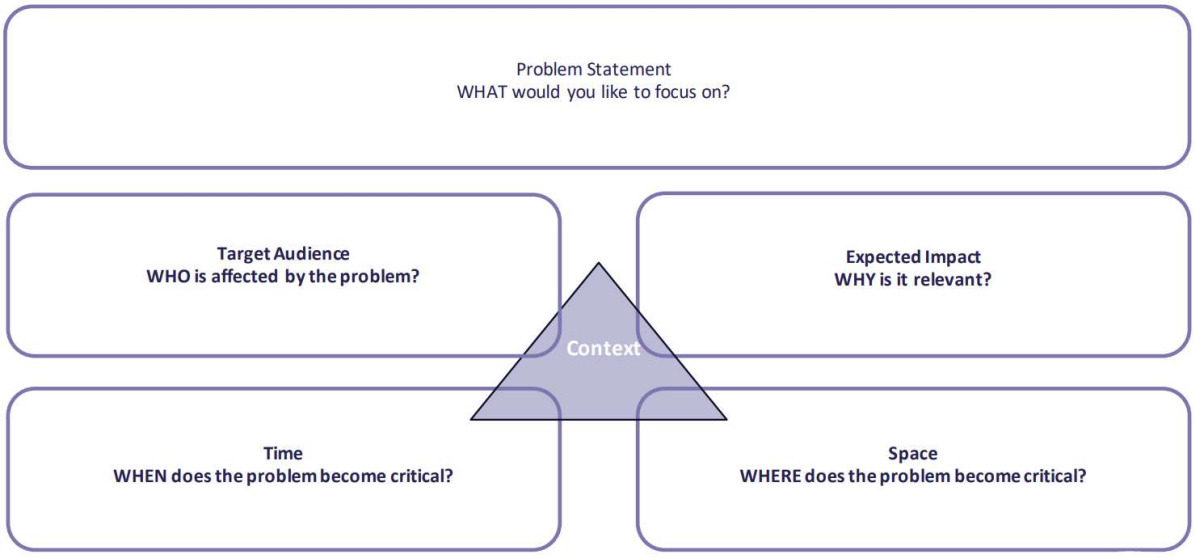
Frameworks to frame problems do exist, with the final aim to keep the work going in the right direction in the right context.

To handle the complexity of the different problem, framing the problem is an essential step to start with. Even when a brief or a goal is provided, a visual synthesis helps to see thing in a specific and sharable light.

The "5 Ws" of the journalisms and problem solving come in useful in identifying for a problem:

- One or two actors affected by the problem (WHO)
- The specific problem we want to answer (WHAT)
- The long-term impact of the problem, should it persist (WHY)
- The specific timeframe in which the problem is most relevant (WHEN)
- The specific place where the problem is most relevant (WHERE)

If some of these aspects is missing, then a project is probably going towards building the wrong solution. We need to make sure that a project is addressing the right question!

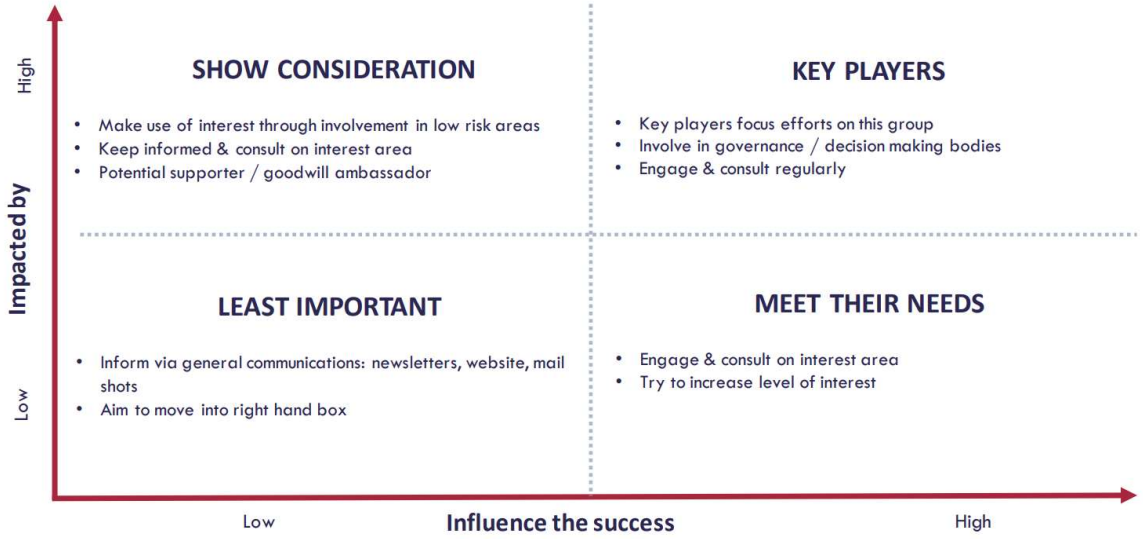


For example, a company facing the problem of deciding what products are most profitable and which products to retire from the markets in which they underperform, will identify as responsible a Global Sale Manager, expected to increase global profit and having results monthly before the board meeting.

It's important also to understand the problem in terms of who are the target users. The **stakeholder map** approach facilitates this understanding with a visual representation of all the stakeholders

involved in a system according to how much they can influence the situation under investigation or be impacted by it.

This tool helps in brainstorming all the possible categories of people that can be of interest for a project, also considering data-related stakeholders (data providers, controllers, stewards, etc.), and in taking apart the impact a problem might have on them, and the influence they have on solving the problem.



Positioning stakeholders on the map helps understanding their importance.

Notice that usually IT people are positioned in a place where they are not highly impacted, but have a high influence on the success of the project, because if they do not care about it then the project will probably fail.

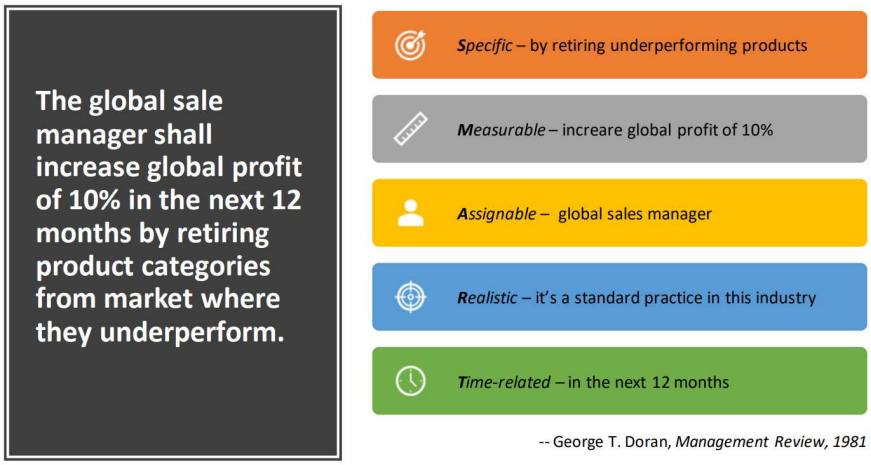
A recurring problem is that of communication between the business part of a project and the technical people, that need to understand each other's. The solution is to make goals as objective as possible, so to have quantitative goals.

An approach is to write down the goals of the project and transform them into specifications. The **S.M.A.R.T. Goal methodology** takes its name from the characteristics of the goals it aims to define:

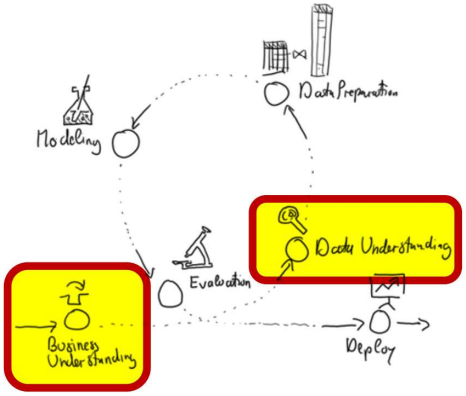
- **S**pecific – target a specific area for improvement,
- **M**easurable – quantify or at least suggest an indicator of progress,

- **A**ssignable – specify who will do it,
- **R**ealistic – state what results can realistically be achieved, given available resources,
- **T**ime-related – specify when the result(s) can be achieved.

By taking for example the goal "The global sale manager shall increase global profit of 10% in the next 12 months by retiring product categories from market where they underperform.", we see that it respects the S.M.A.R.T. characteristics:

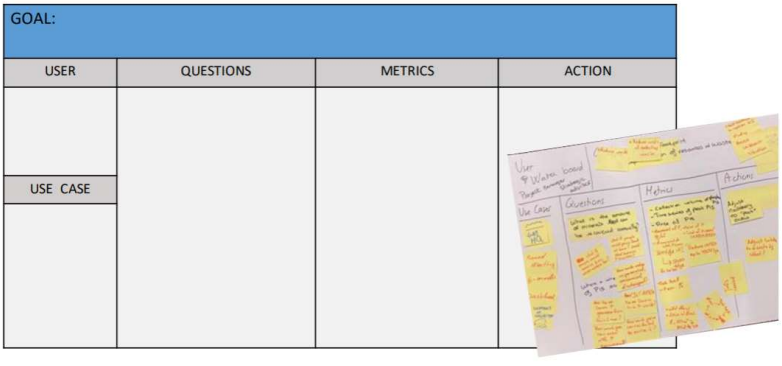


Notice that in a project the riskier step is that from business understanding to tech understanding. If this passage is done wrong, then the process will need to go back and adjust many times.



To transform the goal into technological specifications, the **goal poster approach** was invented.

It involves clearly defining the goal, the context in terms of user-centered design talking about user and use cases, and then the goal and context need to be transformed into the so-called QMA specification (Questions, Metrics, Actions, and also Methodology).



In a data science problem, we need to ask ourselves: which questions we want to answer? Which metrics are going to answer these questions? Which actions will derive from the obtained answers?

Because S.M.A.R.T. goals need to be assignable, a user centered design is employed also in the goal poster approach. A *Persona*, that is a fictional character created to represent a user type that might use the product in a similar way, is taken as the user, and the use case is defined in terms of why, when, how often, the product is used by a persona.

For our example we could have:

Who: Alice

Global sales manager

- **Why**:
  - support her in making better decisions
- **When**:
  - before meeting with the strategic board
- **How often**:
  - monthly
- **What**:
  1. understand for sales & profits
     1. the geographical dimension
     2. the thematic dimension
     3. the temporal dimension
  2. suggest products to retire

And then we would need to define:

- *Questions* specify which aspects of the study the persona investigates to understand how she is progressing toward the goal.
- *Metrics* define which data answer the questions in an objective (quantitative) way by comparing the collected data with a target.
- *Actions* define which behaviors derive from reading the metrics.

The result would be of the type:

What kind of reasoning is stimulated by reading the metric? What are the actions that derive from the reasoning?

GOAL: The global sale manager shall increase global profit of 10% in the next 12 months by retiring product categories from market where they underperform

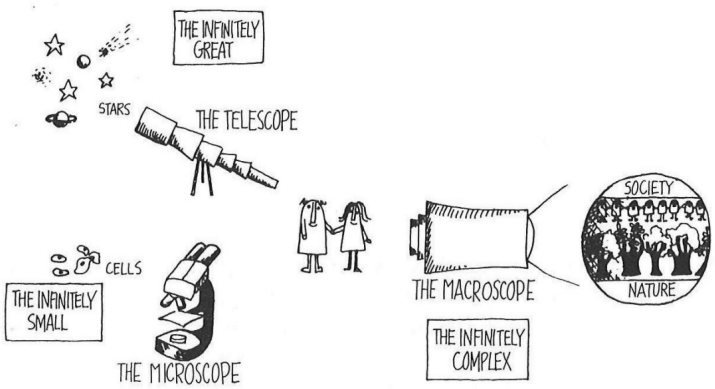
USER	QUESTIONS	METRICS	ACTION
Alice Global Sales Manager	In which geographical areas are we underperforming?	Sales & Profits per region, and country Target: largest sales & losses	
USE CASE before meeting with the strategic board monthly	In which categories areas are we underperforming?	Sales & Profits per category Target: largest sales & losses	Recommend to retire the under-performing product category
	What's the trend over time? Is there any seasonality to account for?	Sales & Profits per month and year Target: negative trends, no seasonality	Recommend to leave the under-performing product category in the portfolio for strategic reasons

14. Data Risks and Challenges

Challenge 1: the complexity of reality

The high complexity of reality is dealt with in engineering through modeling. However, a model cannot capture everything about reality, and some choices need to be made about what to include in the representation and what not. Making the right decisions is crucial for the success.

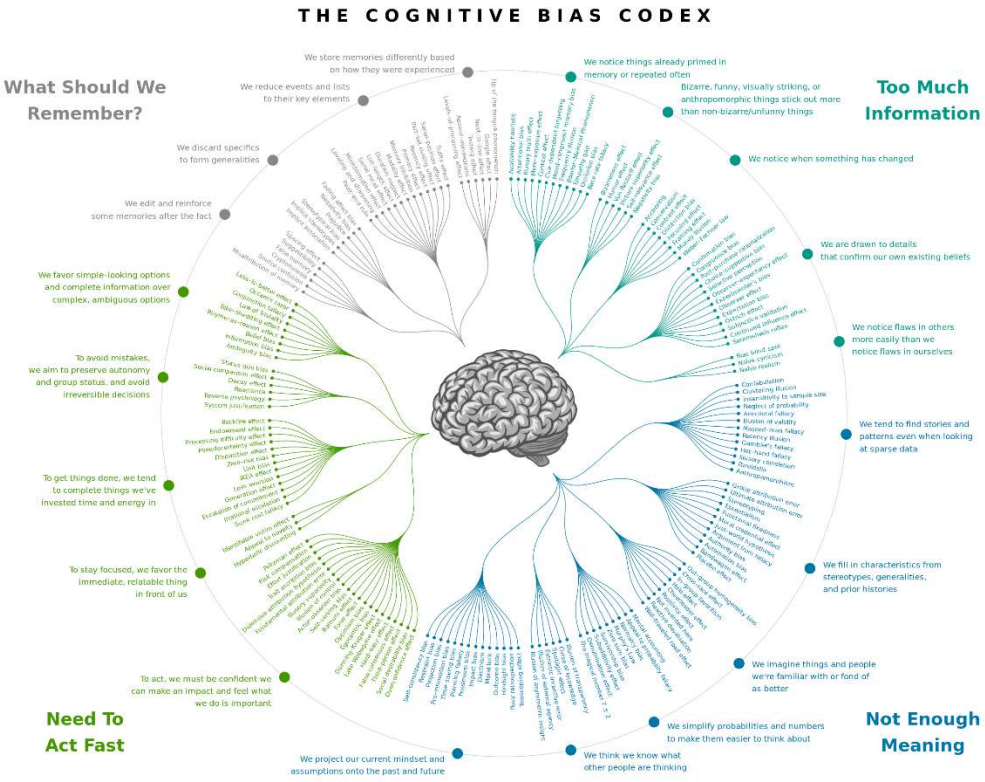
A way to observe the infinitely complex reality is through data. An observer can use data and Big Data to observe reality in a multi-perspective approach, so to avoid missing out some detail. This idea takes the name of the **Macroscope**.



Multiple perspectives to combine can include: sentiment, locations, topics with semantic analysis, word clouds, different granularities, analytics, and so on.

Challenge 2: Cognitive Bias

Humans have biases and they put their alteration of realities in their work.



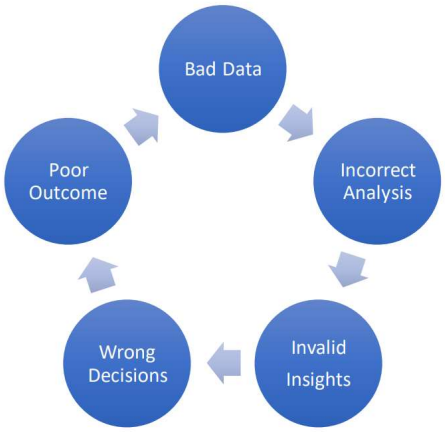
One part of the human bias is the **confirmation bias**, that is the tendency of individuals to favor information that confirms their existing beliefs or hypotheses while ignoring or downplaying contradictory evidence.

Examples of experimenter bias include conscious or unconscious influences on subject behavior: actions that influence subjects or altered or selective recording of experimental results themselves.

Challenge 3: Data Quality

Data need to have enough data quality to be trusted. If data quality isn't high enough, results might be misleading.

"Garbage in, garbage out" (GIGO) is the fundamental concept in data analysis stipulating that the quality of output or results is determined by the quality of the input.



Conventional dimensions of data quality are:

- Accuracy: the data was recorded correctly.
- Completeness: all relevant data was recorded.
- Uniqueness: entities are recorded once.
- Timeliness: the data is kept up to date (and time consistency is granted).
- Consistency: the data agrees with itself

The reasons why data can be dirty are several: dummy values, absence of data, multipurpose fields, cryptic data, contradicting data, shared field usage, inappropriate use of fields, violation of business rules, reused primary keys, non-unique identifiers, data integration problems, and so on.

To prepare data for analysis and visualization, **data wrangling** comprises: Data Preprocessing, Data Preparation, Data Cleansing, Data Scrubbing, Data Munging, Data Transformation, data Fold, Spindle, Mutilate, and (good old) ETL.

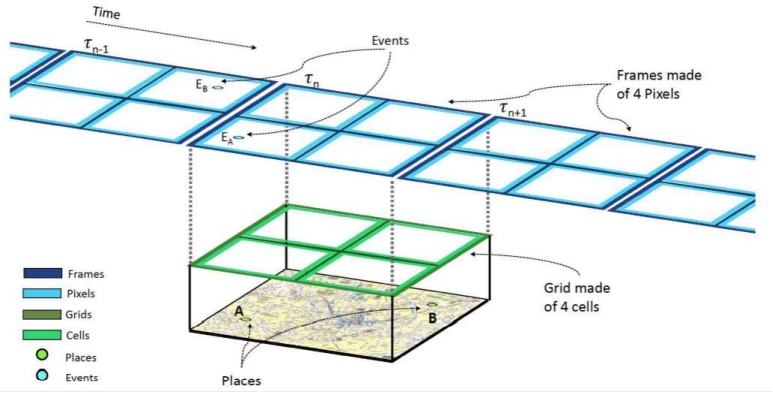
Challenge 4: Content Bias

Biases can be found also in the data, and depend on the source and the utilized sample.

It's important to check if the data and the question are aligned, so if the data and the question we want to answer are compatible. Then, there is the problem of representativeness of the sample, in terms of how much the sample is a good representation of reality, in terms of scale and completeness.

Challenge 5: Granularity

Every source has a different level of granularity in terms of space and time.



Challenge 6: Availability & Access

The data "divide" who has the data and who hasn't and so cannot compete. There is a competitive difference between the ones that have (or can access) the data and the ones who don't.

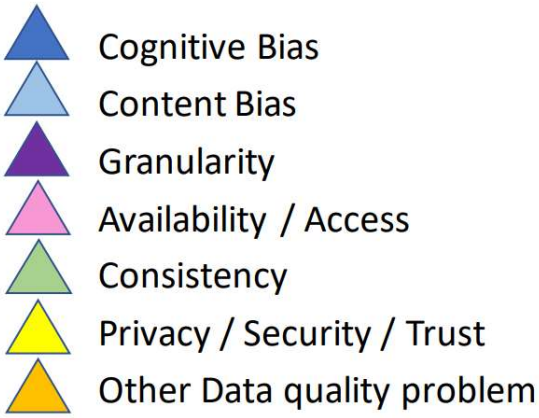
Moreover, often available APIs do not give us the data we want, so we need to fill those gaps. On quantity or time (Twitter) or on quality/content (Foursquare).

Challenge 7: Consistency

Data that can be found is not consistent, we need a way to make checks.

Challenge 8: Privacy, Security, Trust

Privacy and Security are problems that involve evolving regulations and user trust that is crucial for the success of a company.



A. Databases technologies comparison

Graph DB - Neo4j

Graph database ideal for handling highly interconnected data. Its strength lies in managing relationships between data points efficiently.

Graph databases are very fast for associative data sets, like social networks. They map more directly to object-oriented applications.

ACID properties are ensured.

Neo4j is efficient on nodes and relationships but not so efficient in whole-graph analysis.

Use Neo4j when your data structure involves complex relationships, such as social networks, recommendation engines, network and IT operations, fraud detection, or any scenario where the connections between data entities are critical.

Documental DB - MongoDB

A flexible, document-oriented NoSQL database. It's schema-less, allowing easy scalability and handling of unstructured data.

Scalability is a strong point for MongoDB, allowing the creation of a lot of entities.

MongoDB is suitable for applications where flexibility and scalability are essential, such as content management systems, real-time analytics, IoT applications, and scenarios requiring frequent updates to the data model.

Key-value DB - Redis

An in-memory data store known for its speed and versatility. It supports various data structures and is often used for caching, session management, real-time analytics, and queuing.

Choose Redis for scenarios requiring high-speed data access, caching, real-time analytics, or as a message broker. It's beneficial when rapid data access is crucial.

Columnar DB – Cassandra

Designed for scalability and fault-tolerance. Cassandra is excellent for handling large amounts of data across distributed clusters and ensures high availability.

Consider Cassandra for applications needing high availability, scalability, and a distributed architecture. It's suitable for time-series data, IoT, logging, and other scenarios that require high write throughput and scalability.

- Highly and Linearly Scalable
- No Single Point of Failure (i.e., no single part of the system can stop the entire system from working)
- Replicas of data are allowed
- Quick Response Time
- Flexible Data Storage (i.e., supports structured, unstructured and semi-structured data)
- Easy Data Distribution (i.e., supports flexible data duplication)
- BASE Properties

- Fast Writes

Cassandra wasn't created to perform JOIN operations, but to be able to query a lot of data quickly and efficiently. If such operations are needed, you'd better reconsider your DB choice.

IR based DB - Elasticsearch

A powerful search and analytics engine. Elasticsearch is built on top of Lucene and excels in full-text search, log analytics, and real-time data analysis.

When there are a lot of entities with textual fields on which indexing and search will be performed, Elasticsearch is the best solution.

Use Elasticsearch for applications requiring advanced search capabilities, log analytics, monitoring, and real-time data analysis. It's beneficial in scenarios where fast search and analysis of large volumes of data are critical.

Recap

	Operations/Analytics	Scalability	Unstructured data	Fault tolerance	Capabilities	CAP Theorem
Neo4J	Efficiently manage relationships. Studied for operational systems		Schema-free		Handle complex relations, very fast for associative data	ACID properties are enforced by transaction
MongoDB	Operational systems	Highly scalable	Documents schema-less, so very good for unstructured data and evolving schemas		Granularity at the level of business objects	CP
Redis	In support of other databases or as cache			Can be implemented, but normally a failure means the loss of data	Very fast	
Cassandra	OLAP and data mining. Red-mostly and read-intensive applications in which entire	Easily scalable, made for huge data	Schema-less, with a row-key pointing to arbitrary columns	High fault tolerance	Fast writes	AP With tunable consistency

	tuples are rarely needed.					
Elasticsearch	OLAP	Easily scalable	Schema-less with dynamic mapping		Best matching, ranking	