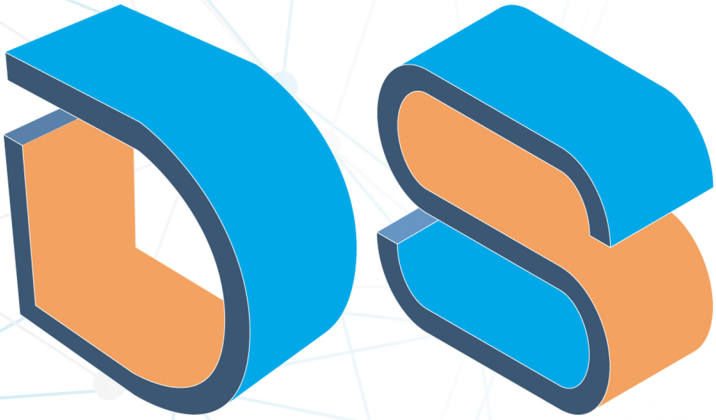


POLIMI
DATA SCIENTISTS

Unstructured and Streaming Data Engineering

Edited by:

Alessandro Messori

Credits

This document was made by a a student from its notes, and as such it may contain errors or be incomplete in some parts.

This document is not to be intended as the unique source of study material for the exam preparation, but rather as a quick way to review the theoretical part of the course.

To get the full course experience and the best exam preparation it's recommended to read the course slides and watch the recordings of the lectures.

Most of the work is based on the slides and lectures of professors Emanuele Della Valle and Marco Brambilla, and so all credits goes to them.

Contents

1 Introduction

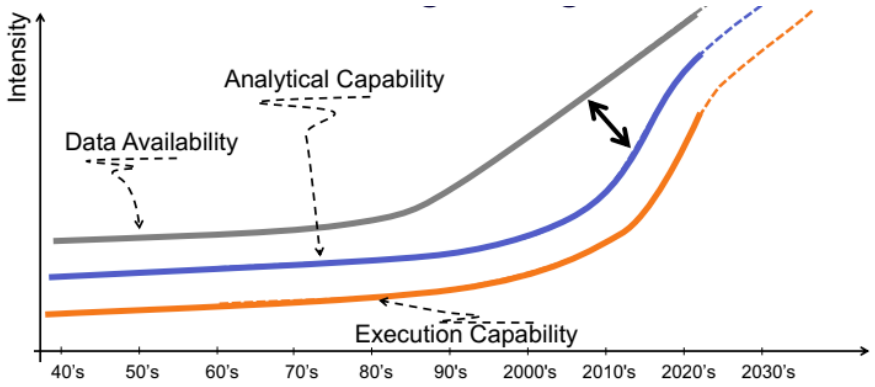
1.1 Data Driven Decision making

Nowadays companies need a reliable source of data for performing decision making, and big enterprises have huge amounts of data from which they need to extrapolate valuable information.

With the amount of data derived from IT systems companies can now structure their organization in a data-driven way, as opposed to in the past when they had to take decisions based on instinct.

Data-driven decisions are more effective, predictable and profitable. Such an approach is only possible if we have a great amount of data to analyze and if data is 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.



1.2 The Definition of Big Data

So in order to be able to have a data-driven approach we need to be able to handle the so called "Big Data", a term used to describe data with the following characteristics ("The 4 V's"):

- 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)
- 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)
- Velocity: information that flows continually creating streams, that often needs to be analyzed in a time frame of a few seconds
- Veracity: big data can often be imprecise and unpredictable, and this unreliability needs to be addressed and managed

1.3 The Roles in the Data World

The data from the operational systems of a company is a raw material that needs to be refined using Data Science techniques in order to extract valuable information that can fuel the decisional process of a company and so the duties of a Data Scientist are:

- Obtaining predictable, actionable insights from data
- Creating data product with immediate impact
- Communicating relevant business stories from data
- Building confidence in decisions that drive business value

Instead Data Engineers are the ones who build the underlying system that allows Data Scientist to analyze, develop and deploy working and ever-evolving data models.
In other words a Data Engineer handles the lifecycle of data in a processing pipeline, ensuring that it is always available and usable by whoever needs it.
To summarize, these are the typical responsibilities of a DE:

- Design and build data processing systems
- Operationalize machine learning models
- Ensure solution quality

2 Unstructured Data Models

2.1 NOSQL General Concepts

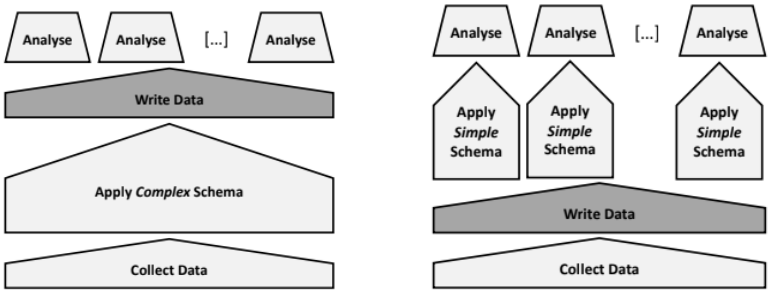
2.1.1 Differences with the traditional data model

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 premise 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.

2.1.2 Schema Less Approach

This new types of data model require lot of flexibility, which is usually implemented by getting rid of the traditional fixed schema of the relational model, using a so called "schema-less" approach where data doesn't need to strictly respect a predefined structure in order to be stored in the database.
This way the definition of the schema is postponed from the moment in which the data is written to the moment when the data is read. This approach is called Schema-On-Read as opposed to the traditional Schema-On-Write. This new philosophy has many advantages over the old one:

- 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 particular task
- The same data can assume a different schema according to the needs of the analytical jobs that is reading it, optimizing performance.

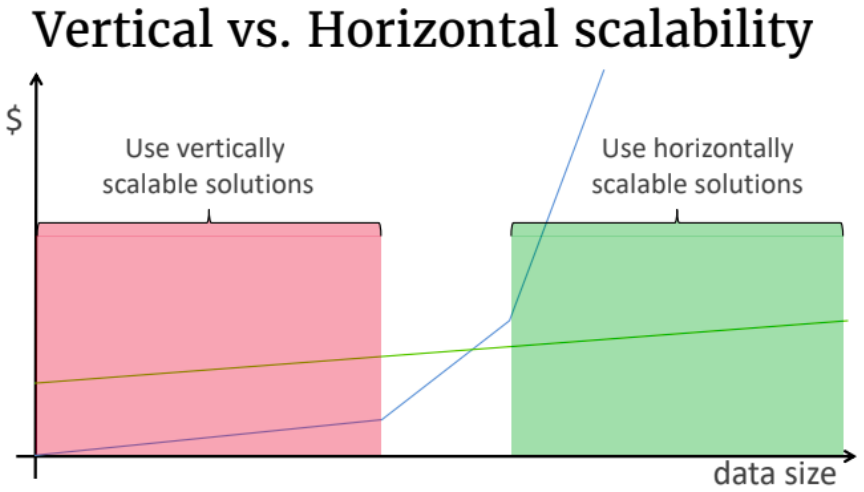


2.1.3 Data Lakes

The better define this new kind of data storage systems, the concept of Data Lake was invented: a reservoir of data, which is just a rough container, used to support the needs of the company.
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.
Building such a huge and centralized data storage system has many challenges: it's important to organize and index well all the incoming data, or we could end up with a huge Data Swamp where it's impossible for everyone to find what they're looking for.
And so usually in a Data Lake the incoming raw data gets incrementally refined and enriched in order to improve accessibility, by adding metadata and indexing tags and by performing quality checks.

2.1.4 Scalability

As we previously mentioned, traditional data systems scale vertically, meaning that when the machine doesn't have enough ram or disk to store all the data it just gets replaced with a more powerful and bigger one.
This approach only scales up to a certain point and after that it just isn't efficient anymore, so when dealing with Big Data we need to scale horizontally instead, meaning that the computing system is composed of many commodity machines and when there's a need for more resources more machines are added without replacing the old ones.



As can be seen from the graph, in vertical scalability cost scales exponentially with respect to data size, while in horizontal scalability cost scales linearly.

2.1.5 Transactional Properties in NoSQL

In the relational world we are used to having the concept of a transaction, an elementary unit of work encapsulated by begin and commit commands characterized by ACID properties:

- Atomicity: the operations contained in a transaction either all fail or all succeed
- Consistency: the state of the db is respects the integrity constraints before and after the transaction is executed
- Isolation: every transaction doesn't affect and isn't affected by other concurrent transactions
- Durability: a transaction produces durable changes in a db

These properties are very important and often even fundamental in a traditional SQL based OLTP application
Big Data systems who have an architecture based on horizontal scalability and distributed systems unfortunately can't give complete ACID transactions guarantees.
This is what essentially stated by the CAP theorem.
CAP is a word composed by the initials of 3 important features in distributed systems:

- Consistency: all nodes see the same data at the same time
- 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 guarantees at the same time but not all three. In big data system who rely heavily on network comunication Partition Tolerance is essential, so designing a distributed system means having to handle a tradeoff between Availability and Consistency. The extreme choice would be to abandon completely one between this 2 features , but in real system the data engineer needs to finetune the level of Consistency and Availability to the need of the service.

- AP: A partitioned node returns a correct value, if in a consistent state; a timeout error or an error, otherwise e.g., DynamoDB, CouchDB, and Cassandra (typical use case: banking applications)
- CP: A partitioned note returns the most recent version of the data, which could be stale. e.g., MongoDB, Redis, AppFabric Caching, and MemcacheDB (typycal use cases:media streaming, statistics, social media, news websites)

The traditional relational systems, being monolithic don't need Partition Tolerance and so they are both Available and Consistent (AC). To address the need for distributed big data system new data model and technologies have been invented.

As we mentioned previously these new kind of technologies (which are called NoSQL, that can mean both not SQL and not nly SQL) cannot give the same ACID guarantees of SQL databases, so a new weaker and generic set of characteristics called BASE has been invented to describe features of big data systems.

- Basic Availability: the system can always fulfill requests, but the answer could be partially consistent
- Soft State:The solid state of relational system is abandoned
- Eventual Consistency : At some point in the future data will converge to a consistent state (consistent state is not granted immediately but eventually)

These features are voluntarily vague and generic since they can be finetuned to the needs of the application as stated by the CAP theorem.

2.2 Graph Stores

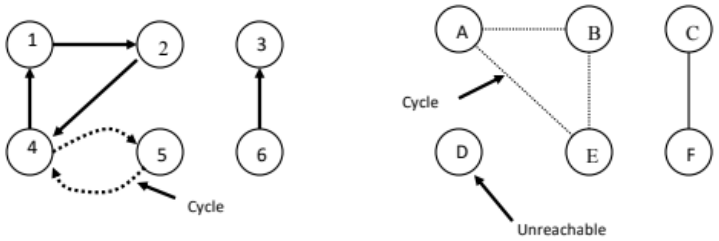
The first kind of databases to have success after the monopoly of SQL are the graph storage solutions.
We'll now have an introduction about graph as a data structure in general, and we will then go deeper into the theory behind the graph based databases and finally talk about a real world graph DB called Neo4J.

2.2.1 Graph Theory

The Graph is a data structure that was first used in 1736 to represent a city and its water canals and have been used ever since for a huge number of optimizations of problems that can be modeled has a set of entities connected by a relation of some kind. From a mathematical point of view, a graph is an ordered triple G:(V, E, f) where

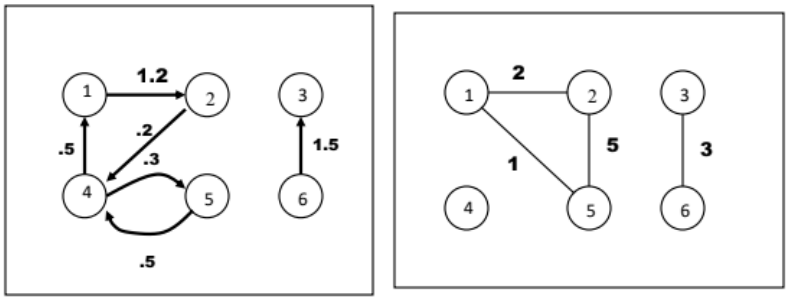
- V is a set f nodes/vertex.
- E is a set of arcs/edges.
- f is a function that maps every edge of E to an unordered pair of nodes of V

A path is a sequence of consecutives vertex, and a node A is reachable from another one B if exists a path that goes from A to B.
If a path starts from a node and returns to the same node its called a cycle, a graph is called acyclic if there it doesn't have any cycle, cyclic otherwise.
A graph is connected if every node is reachable from any other node and is strongly connected if every node it's directly connected to every other one.

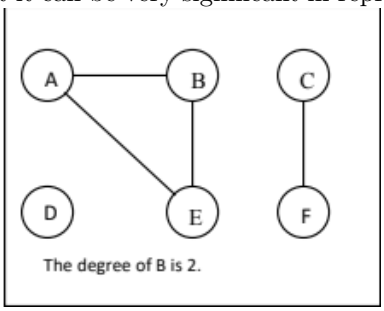


If a graph as n nodes then in can have up to v = n(n-1) edges, if v ≈ n(n-1) then the graph is called dense, otherwise it's called sparse.

Nodes and vertex can be enriched by putting additional data on them, the latter case for example is commonly used to represent some kind of cost of the connection (in this case we have a weighted graph).



If the edges can have a direction, meaning that given the two nodes A and B (A, B) $\neq$ (B, A) can be true, then the graph is called directed.
The number of incident edges in a node it's called the degree of a node; this is a very simple metric but it can be very significant in representing the importance of a node in a graph.



One last important concept is the subgraph, which is a graph whose vertex and edges are a subset of a supergraph. The physical implementation of a graph in a computer can be done in many ways, the most famous of which are the adjacency matrix (where aij = 1 if (i, j) belongs to E and aij = 0 otherwise) which is efficient for dense graphs and the edge list, more efficient for sparse ones instead.

2.2.2 Graph Databases

So why did the need for a graph based database arise? The answer is, somewhat surprisingly, that relational databases are not good at managing relationships! The table based structure of relational databases makes it hard to represent relationships between rows in the same table, and moreover whenever someone needs to find a relationship between records of different tables the db has to perform a JOIN operation, which is usually very expensive.
So for the use cases in which relationships are the most important feature of our data (e.g. social network friendships) it would be best to go with a technology who can implement relationships in a native and efficient way, and that's where graph dbs come in.
In these kind of storage systems data are represented as entities connected by information rich relations, just like in a real graph.

- Sailor [Reserves] Boat

- (:Sailor) –[:reserves]-> (:Boat)

The typical query based approach for managing data in relational databases isn't a good fit for graph dbs, in these cases it's preferred a different approach called Graph Matching, or more in general Pattern Matching. In these technique the user specifies a particular shape or structure he wants to find in the graph, and the system searches for and returns all the subgraphs that match that request.

2.2.3 Neo4J

The most popular graph database is Neo4J, implemented in Java by Neo Technologies, and it has the following characteristics:

- Operational db
- ACID guarantees
- Not efficient for large scale graph analysis
- Nodes and edges are a native feature
- Each node has an identifier tag and can have many properties
- Node can have types, called labels
- Schemaless (nodes with new types can be created an any time)
- traditional CA architecture

Queries are expressed with a custom made declarative language called Cypher. With Cypher is easy to express queries based on relationships (who are the main focus in a graph db), and it's much more efficient than SQL in doing operations equivalent to the JOINs in relational databases
Let's start now making some examples of Cypher Commands:
Create a node of type Crew and label Neo, with attribute name of value 'Neo':

```
CREATE (Neo:Crew {name:'Neo'})
```

Add an edge of type "Knows" from the node Neo to the node Morpheus

```
(Neo) - [:KNOWS] -> (Morpheus)
```

Create and index on a Node attribute (used as a starting point for a query

```
CREATE INDEX ON : Customer(customerID)
```

Create a constraint enforcing uniqueness of attribute customerID on nodes with type customer:

```
CREATE CONSTRAINT ON (c:Customer)
ASSERT c.customerID IS UNIQUE;
```

Import Data from CSV file:

```
LOAD CSV WITH HEADERS FROM "file:customers.csv" AS row
CREATE (:Customer {companyName: row.CompanyName,
customerID: row.CustomerID, phone: row.Phone});
```

Create Node with multiple labels:

```
CREATE (n:Actor:Director {name:'Clint Eastwood'})
```

Merge operator, creates new nodes only if it doesn't exist another node with the same label:

```
LOAD CSV WITH HEADERS FROM "file:///transfers.csv" AS Row
MERGE (player:Player {id:row.playerUri})
ON CREATE SET player.name = row.name,
              player.position = row.playerPosition
```

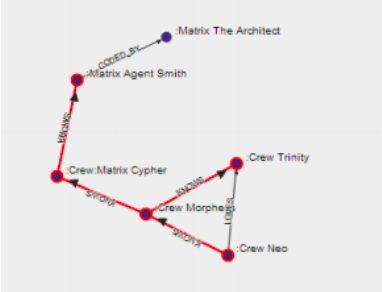
Let's now take a look at Cypher's query system, which is as previously mentioned based on pattern matching and has the following structure:

```
START
MATCH Pattern Matching
WHERE Expressions, Predicates
RETURN Output
```

Match describes the shape of the subgraph we are looking for (similar to SQL's FROM but much more flexible as it isn't limited to tables), Return descries the shape of the query output (similar to SQL's SELECT) and WHERE sets some conditions on the attributes of the nodes.
Example:search for all nodes who have a direct on indirect relation "Knows" to node of type Crew with attribute name "Neo";return the base node, the relation chain and the end node:

Cypher

```
Query:
MATCH (n:Crew)-[r:KNOWS*]-m
WHERE n.name='Neo'
RETURN n AS Neo,r,m
```



Neo	r	m
{name:"Neo"}	[(0)-[0:KNOWS]->(1)]	(1:Crew {name:"Morpheus"})
{name:"Neo"}	[(0)-[0:KNOWS]->(1), (1)-[2:KNOWS]->(2)]	(2:Crew {name:"Trinity"})
{name:"Neo"}	[(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3)]	(3:Crew:Matrix {name:"Cypher"})
{name:"Neo"}	[(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3), (3)-[4:KNOWS]->(4)]	(4:Matrix {name:"Agent Smith"})

(www.neo4j.org/learn/cypher)

This is a more general query structure with additional features such as aggregations, skip and limit:

- MATCH (user)-[:FRIEND]-(friend)
- WITH user, count(friend) AS friends
- ORDER BY friends DESC
- SKIP 1 LIMIT 3
- RETURN user

There are many possible pattern to be searched for, here's a list with the most popular ones:

(n:Person) Node with Person label.
(n:Person:Swedish) Node with both Person and Swedish labels.
(n:Person {name: $value}) Node with the declared properties.
()-[r {name: $value}]-() Matches relationships with the declared properties.
(n)-->(m) Relationship from n to m.
(n)--(m) Relationship in any direction between n and m.
(n:Person)-->(m) Node n labeled Person with relationship to m.
(m)<-[:KNOWS]-(n) Relationship of type KNOWS from n to m.
(n)-[:KNOWS|:LOVES]->(m) Relationship of type KNOWS or of type LOVES from n to m.
(n)-[r]->(m) Bind the relationship to variable r.
(n)-[*1..5]->(m) Variable length path from 1 to 5 rels. from n to m.
(n)-[*]->(m) Variable length path of any number of rels. from n to m
(n)-[:KNOWS]->(m {property: $value}) A relationship of type KNOWS from a node n to a node m with the declared property.

One additional feature is the possibility of adding custom pattern written in Java to Cypher by loading the JAR.

2.3 Key-Value Stores

2.3.1 Introduction

In many applications performance is an essential priority, and often a small delay in response time could mean a big financial loss.
For these kind of needs key-value stores were built, a type of db which g.uarantees the best possible lookup time of any storage system.
Key-value stores are build upon the assumption that any entity can be seen as a value pointed by a key, and so when you are looking for a certain value you need to search for the key associated to it.
In these contest keys are just used to retrieve a specific value, unlike relational databases where primary and foreign keys are means to do searches and mapping between tables; this obviously grants a significant speed boost during reads.
Key-Value stores can be used to improve performance at many different levels:

- Database
- Caching Layers
- Message Brokers

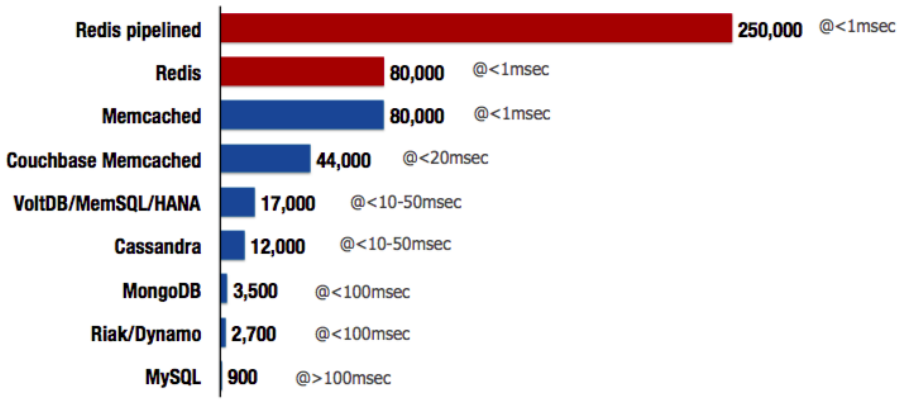
2.3.2 Redis

Redis is the most popular Key-Value store. It was built by a single developer, Salvatore Sanfilippo, it's performance and ease of use were made possible by simplifying its architecture as much as possible:

- Written in C
- Single Threaded
- Atomic operations only
- No official support for Windows
- Executes most commands with costant O(1) complexity
- Based on 2 fundamentals commands: SET(k, v) , which sets the value v to the key, v and GET(k), which returns the value associated to the key k
- In Memory in DB

These characteristics make Redis very fast on reads, but not very efficient for general purpose long term storage system (since being in-memory means that a single shutdown will cost you all your data).
Redis is the best technology for the use cases where:

- Speed is critical
- You need fast access to complex data structures
- Dataset can fit in memory
- Data is not critical

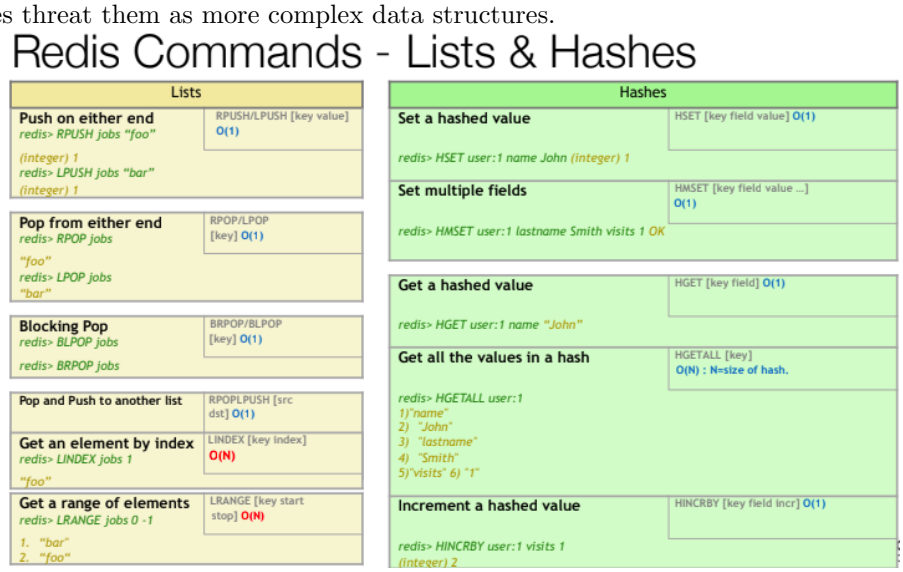


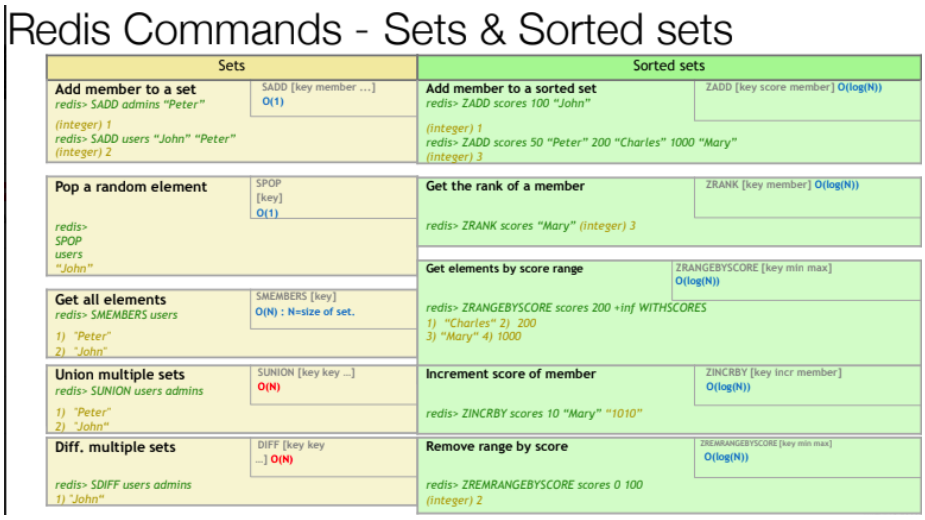
Other than the basic GET and SET commands there are a few other utility functions, but they only build upon the 2 original ones. There are commands to delete and check existence of a key and get the type of a value, for example. Another important feature is the ability to set an expiration time to a key, after which the key it's deleted from the db. Since Redis it's often used as a caching layer, this feature it's very important.

Upon the basic dictionary-like behaviour of Redis richer and more complex features are built.
One of these it's the possibility of handling the values in the key as more complex and abstract data types that can be associated to a specific key-value pair, such as:

- String
- Hash
- List
- Set
- Sorted Set
- Geospatial Set
- HyperLog

This way we can retrieve values by searching its keys and once we have those values threat them as more complex data structures.





Let's now talk about some of the ways to make Redis more scalable and fault tolerant, in order to be able to integrate it safely in Big Data applications:

- Persistence: while all main operations are done in main memory, Redis provides a couple of features to implement disk persistence. The first one allows you to take a snapshot of the db and save it to a file (RDB), while the second one works by keeping track of changes in append only log file (AOF)
- A Redis instance known as the master, ensures that one or more instances kwown 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)

2.4 Columnar Databases

2.4.1 Introduction

In the recent years there has been a ever growing need for technologies capable of handling large scala data analysis Thas need was born because dataset of "Big Data" size have often very different characteristics than the ones usually stored in realtional databases:

- Data Large and unstructured
- Lots of random reads and writes
- Less use of foreing keys in favour of denormalized data structures

What this new kind of data needs instead is:

- Incremental scalability
- Speed
- No single point of failure
- Low cost and administration
- Horizontal scalability

To address this new needs a new set of databases was invented:columnar databases, who owe their name to the fact that they store data by column, as opposed to traditional dbs that store data by row.
Row based databases work well in an OLTP application because it is easy to insert and update records and there isn't much need for bulk reads;however in OLAP environments where analytical jobs often read a huge number of records at once it's fundamental to be able to load only the necessary columns, otherwise the overhead of having to load the entire rows in main memory would be unsustainable.
Of course these aren't the only differences between the two approaches, here are some more pros and cons of choosing a columnar database:
Pros:

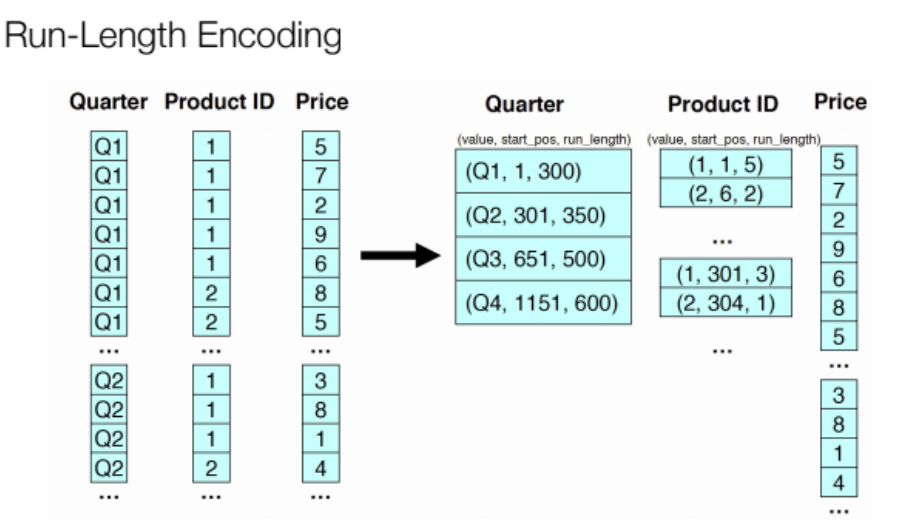
- Data compression
- Improved bandwidth utilization
- Improved code pipelining
- Improved cache locality

Cons:

- Increased disk seek times
- Increased cost of Inserts

- Increased tuple reconstruction costs

What we can gather from this is that columnar dbs are suitable for read-mostly, read-intensive, large data repositories like the ones used by OLAP systems.
Let's now focus on one of the biggest advantages of columnar databases, which is lossless data compression: since it happens very often that a column is composed of a few different values repeated many times, what we can do is to group all the equal values together and instead of storing that same values many times just store that datum once alongside the number of times that value is repeated. For example if we have one column called "Quarter" who only has the values "Q1", "Q2", "Q3" and "Q4" instead of repeating the full string for every record we can just sort the db by that column and save tuples of this sort ("Q1", 200), meaning that the value "Q1" appears 200 times (this approach is called Run-Length Encoding).



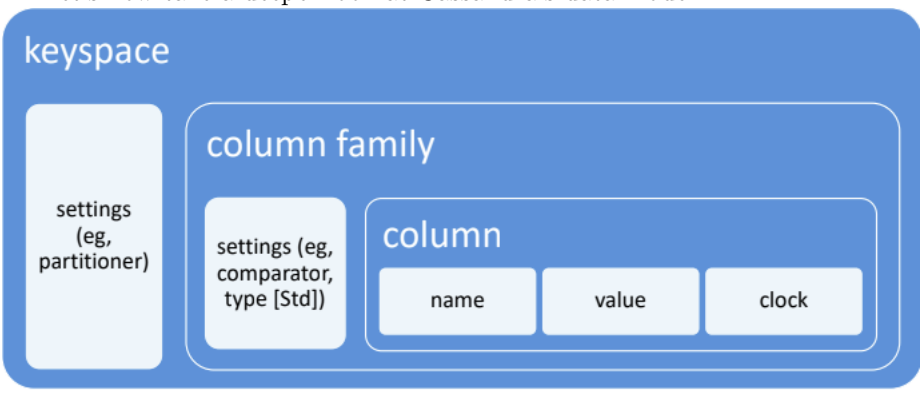
This approach exploits the so called spacial-locality, which means storing similar data values next to each other.

2.4.2 Cassandra

Cassandra is a columnar database with some key-value features. It was originally designed at Facebook and is now maintained by the Apache Foundation as an open-source project.
Its data model is based around the concept of column family, which is a group of columns. Column families are organized by rows indentified by an unique key (hence the key value approach). While this approach my seem similar to relational tables at firts glance, there are many significant differences that makes it a NoSQL db:

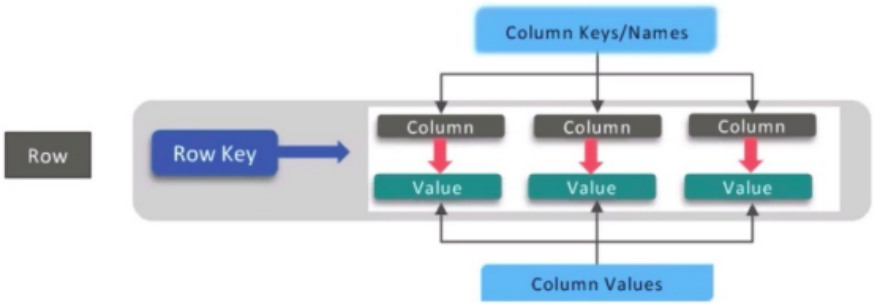
- Each cell in a table could be missing or different from the others
- Schemaless
- Increased tuple reconstruction costs
- Support for key getters and setters

Let's now take a deeper look at Cassandra's data model:

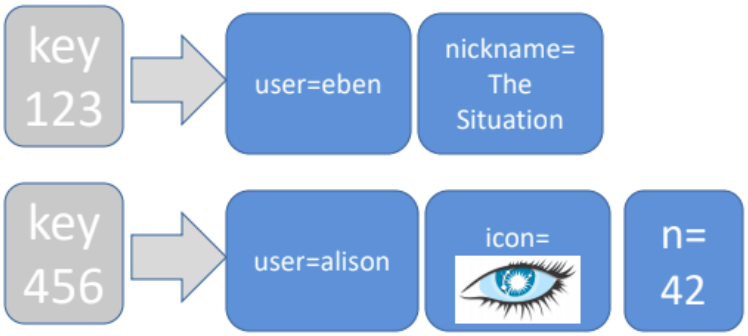


Each column is composed by a name, a value and a clock (which is the timestamp of the last update on that column). The value of a column can contain other columns inside him and in that case that column is called a super-column. Columns are grouped in families, and many group families make up a keyspace (which is the equivalent of a database in SQL)

We can see Cassandra also as a key-value store, where the key is the row identifier and the values are the schemaless column families.

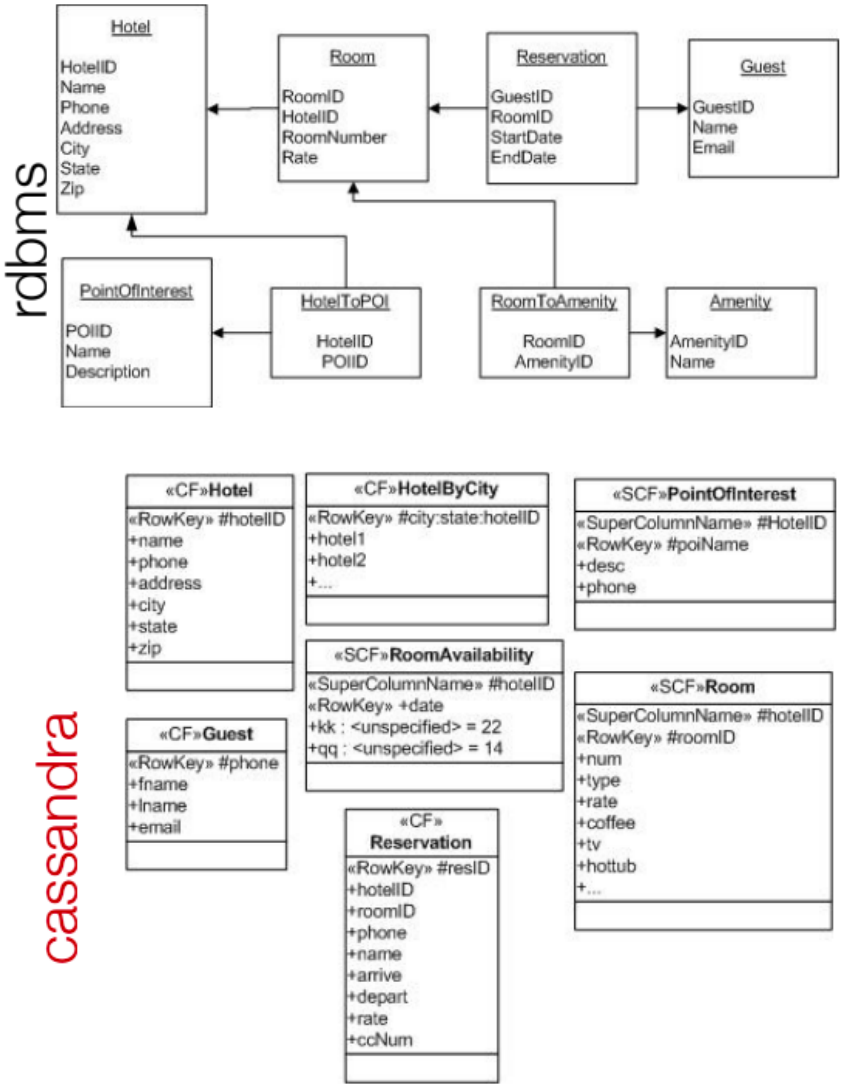


So a keyspace is a group of column families sharing the same application configurations, and the column family is a group of column with values similar or related to each other.
It's important to specify similar and not equal values because, unlike relational dbs, Cassandra can be seen as row-oriented, where every row is made up of many columns with values of similar kind but not necessarily equal to the ones in the other rows of the same cf.
Example of 2 rows in a User Column Family:



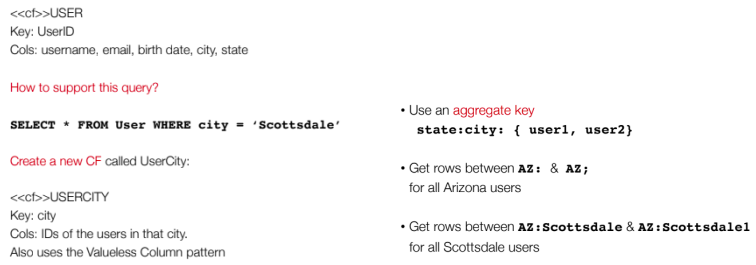
This flexible structure of the columns implies what is maybe the most peculiar characteristic of Cassandra, which is that you usually can't write queries on column values.
Cassandra in fact performs a complete paradigm shift from relational databases: while in traditional databases you design the db schema first and then you write queries over it (Domain-based-approach), in Cassandra it's used a Query First Approach, where designers first think about the requests the system will need to be able to answer to, and then they build the db structure around them.
The Domain-based and Query-based philosophies can be seen as the corresponding implementations of the schema-on-write and schema-on-read approaches we have seen in chapter 1

This is what a rdmbs schema looks like compared to a Cassandra database:



Cassandra can be seen as an "index factory", meaning that for each query there is a table optimized to handle it. For example if we have a User that we need to search by id or by city, we'll have to add a User cf indexed by id and another User cf with the same data but indexed by city, so that each query runs on an optimized table.
So the UserByCity cf would be a list of rows with the city as a key and a column for each id of the users living in that city. This way a row single row can have

a huge number of different columns and that is why columnar db are also often called Big Column stores.
If we need to search an Entity by more than 1 property at the same time then we'll need to create a cf with as key the aggregate value of all those properties.



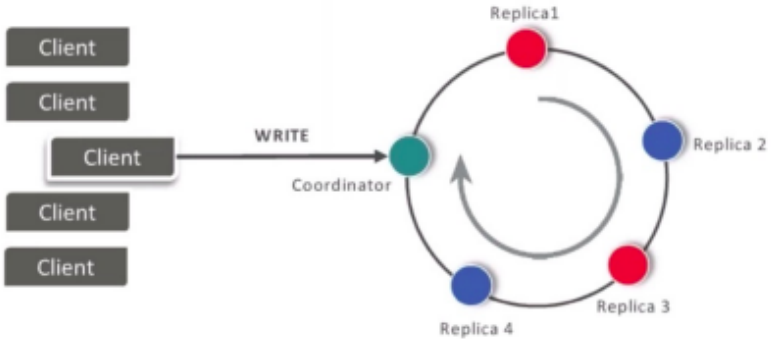
This big change of perspective while gives great advantages on reads implies data duplication, and that is one of the reasons why Cassandra is almost exclusively used in OLAP systems with multiple Terabytes sized data.

2.4.3 Cassandra's Architecture

One Cassandra's main difference from RDBMS it's that it relies on data duplication and denormalization and following it's query-first philosophy it doesn't support JOINs (instead of joining 2 different tables you just build an unique column family).
Here's a list of it's main characteristics:

- Tuneably consistent (we can't have full consistency as stated by the CAP theorem)
- Fast writes
- Highly available
- Fault tolerant
- Linear, elastic scalability
- Decentralized
- around 12 client languages
- O(1) dht

Cassandra is a distributed system implemented with a Ring Architecture: this means that the cluster is masterless and made up of equivalent nodes, and each of them as a predecessor and a successor. Each node can only send data to its successor and receive data from its predecessor.
When a client wants to write data to the database any one of the nodes can answer to that request and if it does it becomes the coordinator for that operation, and after it has finished writing all data to disk he passes the data to its successor so that it can be replicated in other nodes.

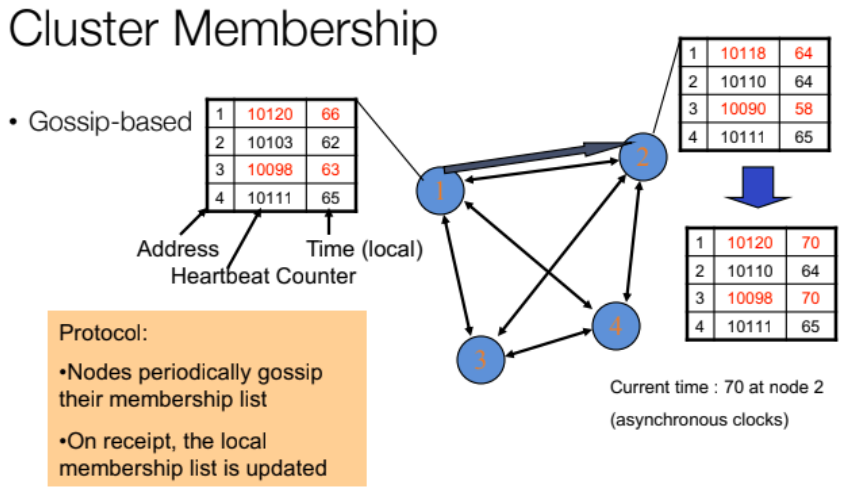


Cassandra was designed to enable very fast writes, in order to do so it had to abandon many transactional guarantees In order to avoid locks on resources a WORM(Write Once Read Many) methodology, were data in written only one time and then never updated again.
The write process works in an asynchronous way: the client sends the data to the coordinator which sends it to all the replica nodes responsible for that key via partitioning function. This method grants atomicity for a given key, since in the case of fault in a replica during the write, the coordinator can just keep sending the data to the working nodes which in turn will send the data again to the broken replica once it's up and running again.
If all replicas are down, the coordinator can buffer the data stream for up to an our
To speed up even more the write process all record are initially stored in main memory and then flushed to disk later on (in case of fault the node can restart interrupted operations since every action is writted to a changelog)

When dealing with reads and deletes we need to handle the problem of consistency (since we have many replicas of the data stored in different nodes there's always the risk of having inconsisten data between them).
A Delete operation works by adding a tombstone to the item in the changelog and later on during the compaction that item will be effectively deleted from all replicas. The Read operation is similar to the Write, but with some differences: the coordinator fetches the same data from multiple replicas and if there is any inconsistency the right data is determined by a quorum mechanism and then a read repair operation is performed. This makes the reads a bit slower than the writes but they're still very fast
As we mentioned before consistency in Cassandra can be tuned in order to maximize o minimize performance;this is done by changing the quorum policy

- `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

Heartbeat communication between nodes is performed by using the gossip protocol: since if each node were to ping every other node there would be an huge and unnecessary stress on the network, each machine chooses a small number of its peers to communicate with and in turn each of this peers will communicate with an equal number of different nodes. This way status updates can spread quickly in the cluster while keeping communications limited.

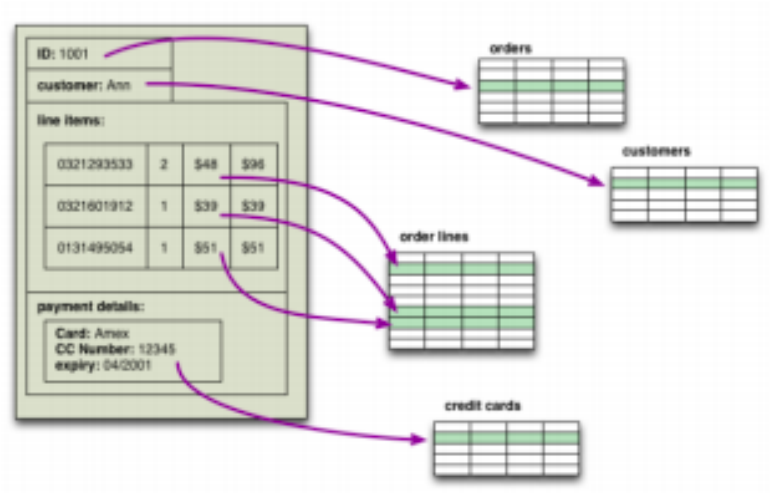


Every node keeps a membership list of the others machines in the cluster alongside the timestamp of the last heartbeat message that was received from them, after a certain period of time without has passed receiving any message from one of its peers that node is considered down.

2.5 Document Databases

2.5.1 Intoduction

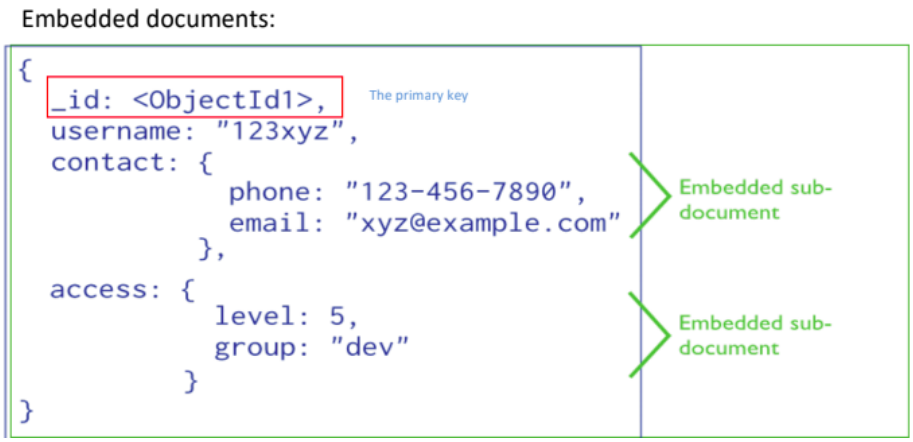
Document databases are probably the most widespread kind of NoSQL technologies. They are called in this way because they deviate from the entity-based denormalized data model of relational dbs and prefer an approach based on denormalized and aggregated data typical of business document.
A document is a complex object made up of what would be many different SQL tables, and so it can be composed of different pieces of data with different structures.
Some of the advantages of document stores are:



- Flexible structure that handles well schema changes
- Solves mismatches impedance problems
- full JSON compatibility, which guarantees very ease of integration with web technologies

2.5.2 MongoDB

MongoDB is the most popular document database, it's built using a CP architecture.
Its data model is based on documents structured just like a JSON file, many documents form a collection. Every document must have ad unique id and can have any number of nested documents inside him.



These data model is convenient for many applications (especially web based ones) since unlike relational databases you don't have to reconstruct the complex business objects from the normalized tables with expensive joins or aggregations techniques.
This approach gives you the possibility of structuring your data using the granurality which fits best your application needs.
It's however possible to extablish references between documents, but it doesn't make much sense to overuse this feature since that would be just like recreating a relational database, Queries and CRUD operations between document are made using built in functions that can filter and operate on the objects.

Read – 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

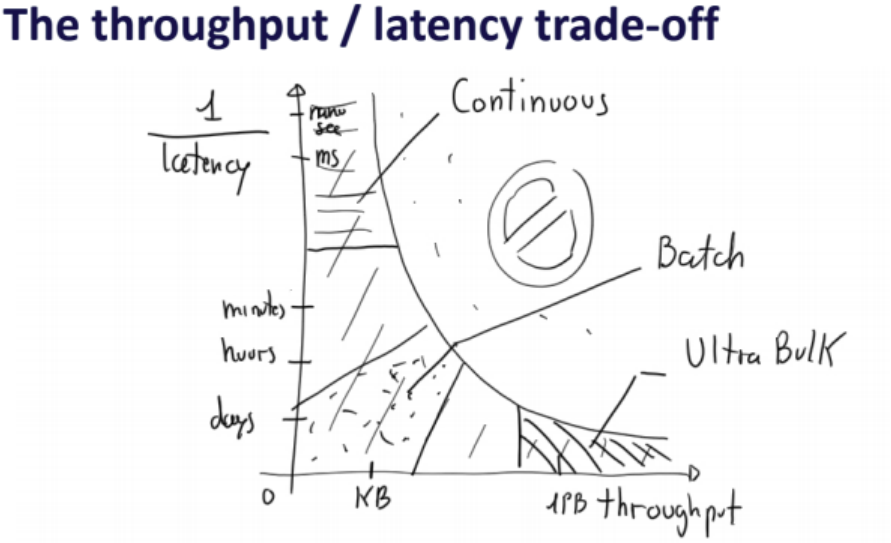
Comparison Operators

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

3 Streaming Data Engineering

3.1 Introduction

When a Data Engineer wants to design a scalable system, he needs to consider the dimensions by which processing systems are usually described by:Throughput, Latency and Message Size.
To understand these properties, let's make a comparison with the trucks of a transportation company: the time need for a truck to go through a certain road would be the latency (which means how much does it take to our server to answer to a request) and the amount of tracks that can travel at the same time would be the throughput (the amount of data our system can process in a given time frame).
Let's also say that before the company can send out a new track it needs to wait some time, the amount of which depends by how the speed and the load of the last sent truck (and in our analogy this would be the message size).
If we want to increase our performances there are many ways to do so: we can increase the speed of the trucks (vertical scaling), add lanes to the road (horizontal scaling) or decrease the time frame between trucks (reduce message size).
It's easy to understand that there is a tradeoff between throughput and latency:after a certain point, if you want to increase one you need to decrease the other.

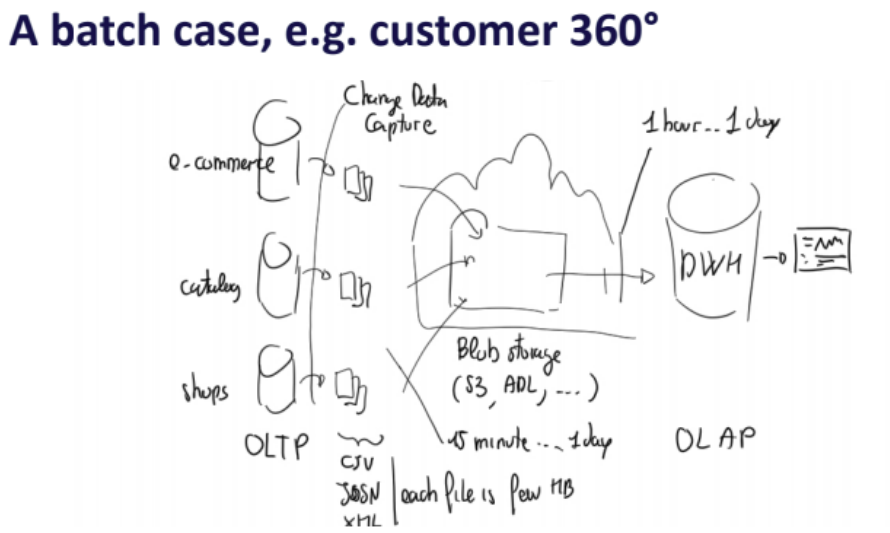


The graph above represents the solution space created by this tradeoff, the area below the curve represent the possible characteristic of a computing systems, the one above it's impossible to obtain.

The solution space can be divided into 3 main zones:

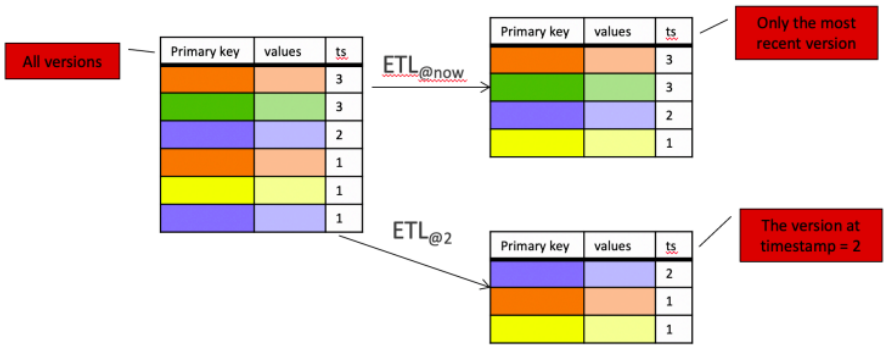
- The area with very high throughput but bad latency is called Ultra-Bulk
- The sector with extreme low latency obtained by sacrificing throughput is the Continuous zone
- The middle ground is called Batch

An example of Ultra Bulk is when companies decide to move their in-premise data to the cloud not by using the internet but by transporting the hard drives by car directly to their cloud provider's data centers. This solution may be counterintuitive but when the data size is very big its the fastest way to perform the migration (for example if the company had to move 1 PB of data it would take 37 days even with a 2.5 Gbps network, while with the transfer appliance the migration time doesn't depend on the size of the data but only on the distance from the datacenter).
Let's make now an example of a continuous case instead:let's say that a multinational operating all over the world has both retail stores and e-commerce websites, and this company wants to keep track of all sales and purchases in order to create a real time inventory
The sales could generate something like 200 MB/s, which isn't much for a network to handle but it could be difficult for the hard disks to have this kind of throughput if we have a single node architecture. In fact, in order to solve this problem we would need a continuous distributed system.
Let's now cover the last and more common case, the batch one: a company needs a data warehouse to host its analytical data coming from many sources (e-commerce, catalog, retail shops...);the data get periodically extracted from the source databases by using the data change capture technique (which means basically by reading the commits from the log file) and saved to a could blob storage in a raw text format (like CSV, JSON or XML), then a listener program loads all the data uploaded to the blob storage to the data warehouse.
The refresh period it's usually quite high and can vary from once every few minutes to even once a day.
Lastly Business Intelligence tools connect to the data warehouse and generate a report.



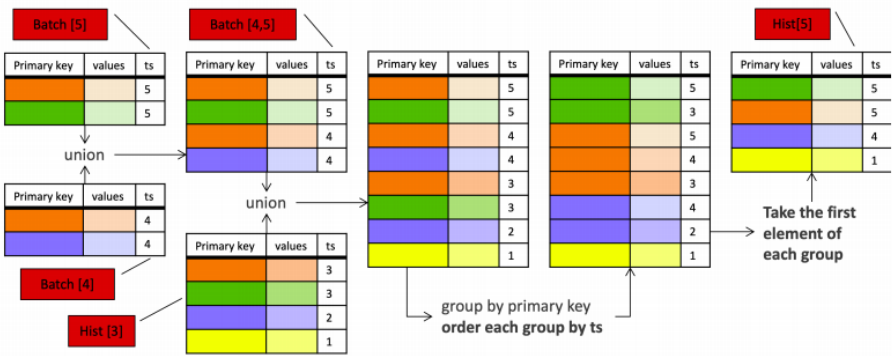
One of the biggest challenges when working with blob storage is the fact that files are immutable, following the WORM methodology, and so updates aren't as trivial as they would be in a more traditional storage system.
We need to choose a policy to use for handling the consecutive batches being pulled from the OLTP databases:

- Append Mode: this is the simplest and more popular batch update policy, changes are treated as new inserts on the table and are differentiated by the timestamp. So every version of the database in time is kept and as time passes rows keep on getting added. The ETL that reads the data later on has to create a consistent view of the database at a given timestamp itself when it applies the schema on read.



- Compaction Mode: in this, more complex policy changes are treated as updates, meaning that ETL processes only access the latest version of the data. This process works by grouping the record by primary key and sorting them by timestamp, then keeping only the record for each primary

key with the latest timestamp



Each of the 2 policies has its own advantages and disadvantages obviously, and often the Data Engineer implements an hybrid version of the 2 tailor suited for the needs of its data-lake:

Append vs. Compaction mode

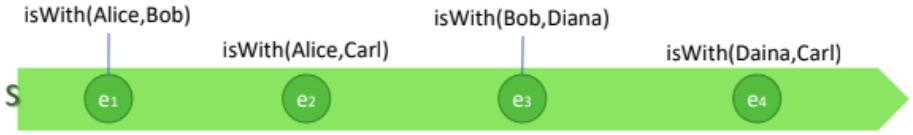
POLITECNICO MILANO 1863

	pros	cons
Append	• Minimal ingestion & wrangling cost • The version of the data in any point in time can be rebuilt	• It requires more space than keeping only the most recent version • If multiple ETLs access the same data the same computation may be repeated • Avoidable by introducing a view that computes the shared data (e.g. ETL@now)
Compaction	• It requires less space than keeping all versions as in append only mode • Multiple ETLs can access the same data (@now) without additional computation	• It requires more computation that ingestion and wrangling in append only mode • Only the most recent version is available, build version in arbitrary points in time is impossible

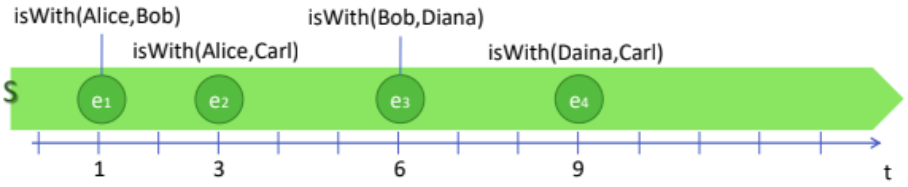
3.2 Streaming

We will now dive deeper into the continuous case, also called Data Streaming. This approach performs a complete paradigm shift from traditional data processing: if in traditional systems data gets stored first and then processed later on, in streaming data gets transformed and analyzed continuously as it is generated
Before going forward we need to talk about the Time Models which are the many possible ways in which the relation between the passing of time and information being generated can be interpreted.

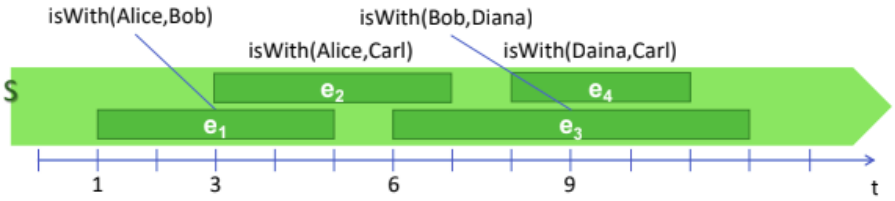
- Causal Time Model: the events happen one after the oher, it's possible to determine an order of the events but we can't now how much time as passed in between but we can still make meaningful queries (e.g. Does Alice meet Bob before Carl? Who does Carl meet first?)



- Absolute Time Model: like causal time model, but the system keeps track of the time passed between each event, and this gives additional expressive power to our queries (e.g. How many people has Alice met in the last 5 minutes? Does Diana meet Bob and then Carl within 5 minutes?)

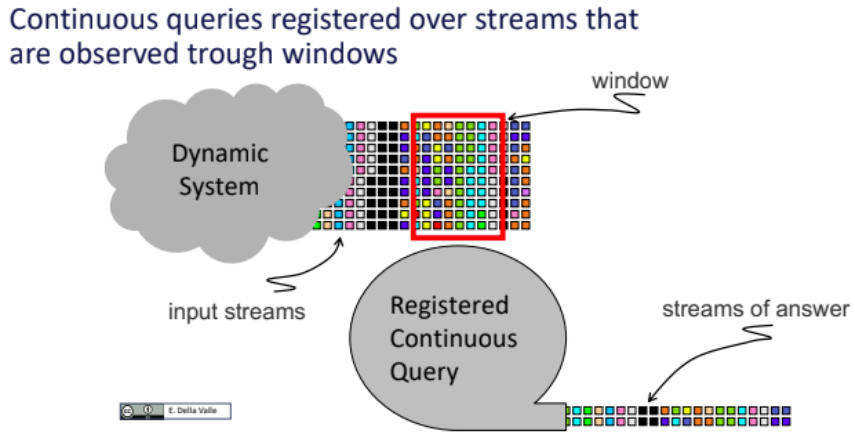


- Interval-based Time Model: like the absolute time model, but we also know the how long does an event last, and this give us the chance to write even more queries (e.g. What meetings last less than 5 minutes? Which meetings are overlapping?)



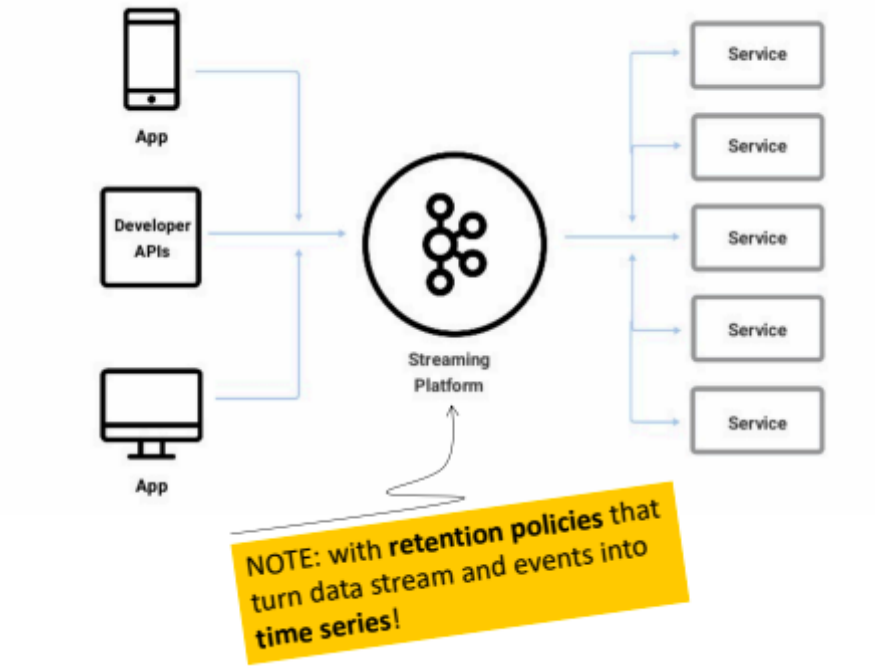
Different communities in the computer science world have implemented the streaming concept in different ways, let's take a look at them:

- Database Community:Standard DBMSs are passive, meaning that the storage system is static and the queries performed by the users are active and dynamic. Streaming systems works in an active way instead: the queries are always the same and they run on a continuous stream of ever-changing data, the so called Data Streams, who are unbounded sequences of time.varying data.
  This concept is implemented in practice by having analitycal jobs working on limited windows of the data stream, as it would be impossible to operate on an unbounded source.
  The database community uses heavily the time models to implement their streaming data systems.



- Event-based Community: this model is based on event processing systems, where each component publishes data about a certain topic and subscribes to the topics he's interested to via a middleware. This kind of communication is asynchronous, anonymous, message based, multicast and implicit. The Complex Event Processing (CEP) variant adds the ability to deploy rules that describe how composite events can be generated from primitive (or composite) ones

- Service Oriented Community:in this architecture component can communicate only by calling services, so there's a standardized protocol of communication in the system and no component can directly manipulate another one.This approach however is rigid and creates tight coupling between services, so a new version called Event-Driven Architecture (EDA) was developed in order to have a more flexible and extensible model. In EDA components communicate with services via data streams that act as a middleware, this way the messages don't get lost if the service goes down


App
Developer APIs
App
Streaming Platform
Service
Service
Service
Service
Service
NOTE: with **retention policies** that turn data stream and events into **time series**!

3.3 EPL

3.3.1 Introduction

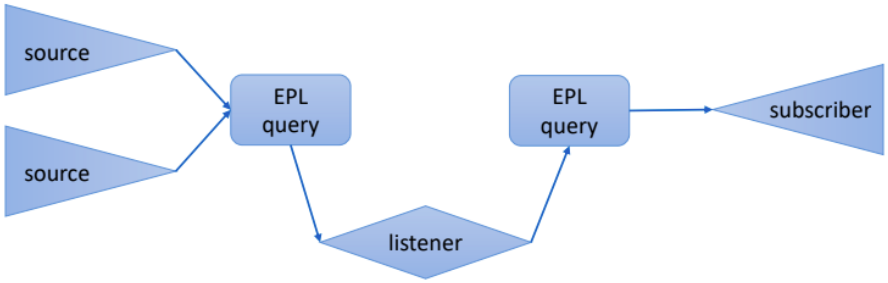
There are many languages for data stream processing, one of the first to be wildly adopted was EPL, developed by a German company called Esper.
EPL is a rich language used to express rules, it supports windowing, relational operators (select, join, aggregate, subqueries...) as well as Relation-to-Stream operators More advanced features includes combining more queries into a network and event recognition abstractions.
EPL runs on a streaming system called Esper. Being implemented in Java in can run in any JVM environment and it's designed for having high throughput and low latency. It has a sophisticated event recognition algorithm it's based on trees, indexes and finite state machines. It's also possible to interact with historical data in Esper, and it's easy to integrate it in data workflows since it has many I/O adapters (CSV, DBs, Socket, HTTP....) and a push/pull based communication.
From an architecture point of view, Esper ensures high availability and state recovery in case of failure (although this features are only available in the commercial version, not in the open-source one).
In order to better understand how EPL and Esper work we'll use as example a Fire and Some Detection system:let's say we need to count the number of fires detected using a set of smoke and temperature sensors in the last 10 minutes.
Events:

- Smoke event: String sensor, boolean state
- Temperature event: String sensor, double temperature
- Fire event: String sensor, boolean smoke, double temperature

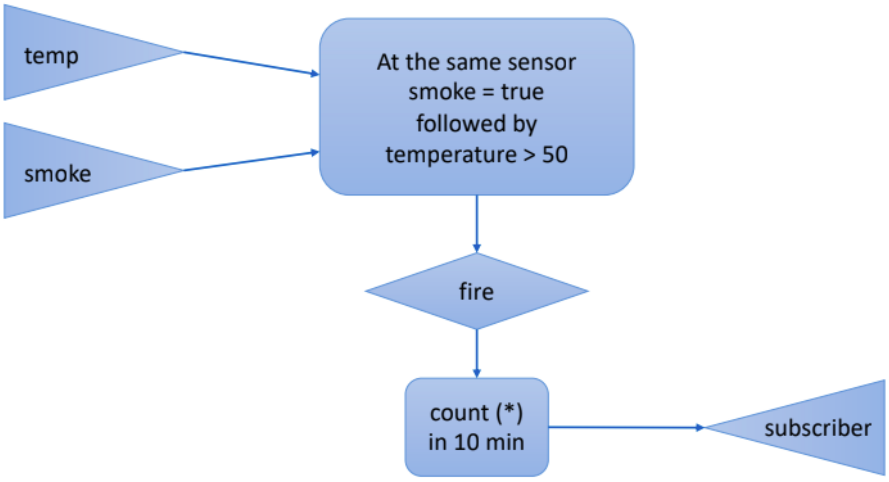
Condition

- Fire: at the same sensor smoke followed by temperature greaterthan 50

The data processing model will be the following: our data sources are the sensors output streams, a first set of EPL queries will process these streams, and a second set of queries will be listening for the outputs of the first set amd will get triggered if a certain condition occurs.
This way we have a data pipeline where data gets through various stages and at each step it gets processed until it reaches the end of the network and gets stored somewhere.



In our specific case, this is the graph:



Let's see now how to implement this system in practise. First of all we need to declare the events.
General sintax:

```
create schema
schema_name [as]
(property_name property_type
[,property_name property_type [,...])
[inherits inherited_event_type
[, inherited_event_type] [,...]]
```

In our example this are the commands needed:

```
create schema SmokeSensorEvent(
      sensor string,
      smoke boolean
);

create schema TemperatureSensorEvent(
      sensor string,
      temperature double
);

create schema FireComplexEvent(
      sensor string,
      smoke boolean,
      temperature double
);
```

In the data manipulation part, EPL is similar to SQL, but queries are run on streams and events instead that on tables:

```
[insert into insert_into_def]
select select_list
from stream_def [as name]
[, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

In our use case some useful queries could be the following:

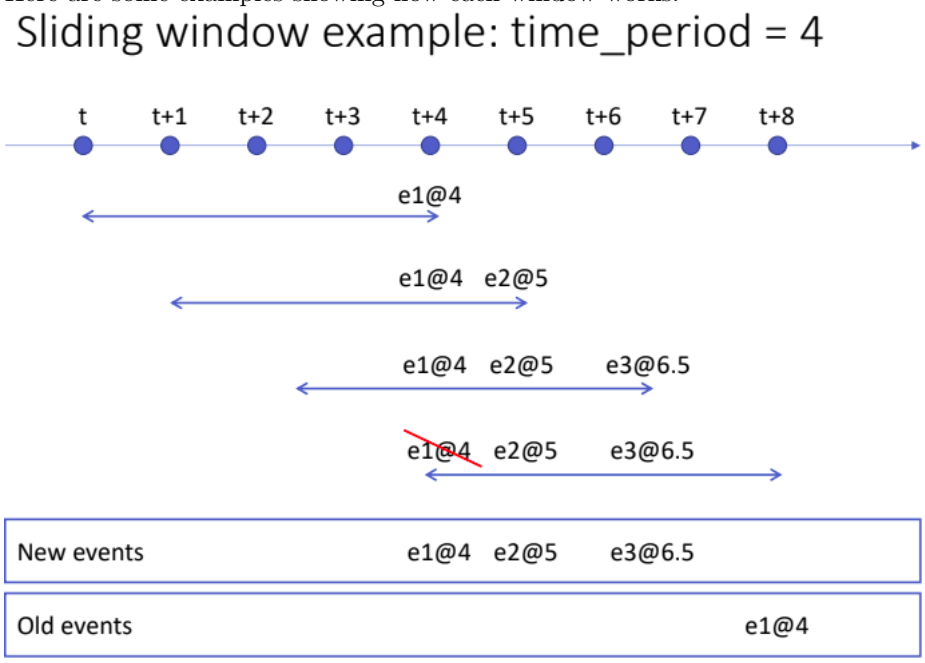
```
insert      into FireComplexEvent
select      a.sensor as sensor,
            a.smoke as smoke,
            b.temperature as temperature
from        pattern
            [every a=SmokeSensorEvent(smoke=true)
            ->
            b=TemperatureSensorEvent(
            sensor=a.sensor, temperature>50)];


select      count(*)
from        FireComplexEvent.win:time(10 min);
```

3.3.2 Windows

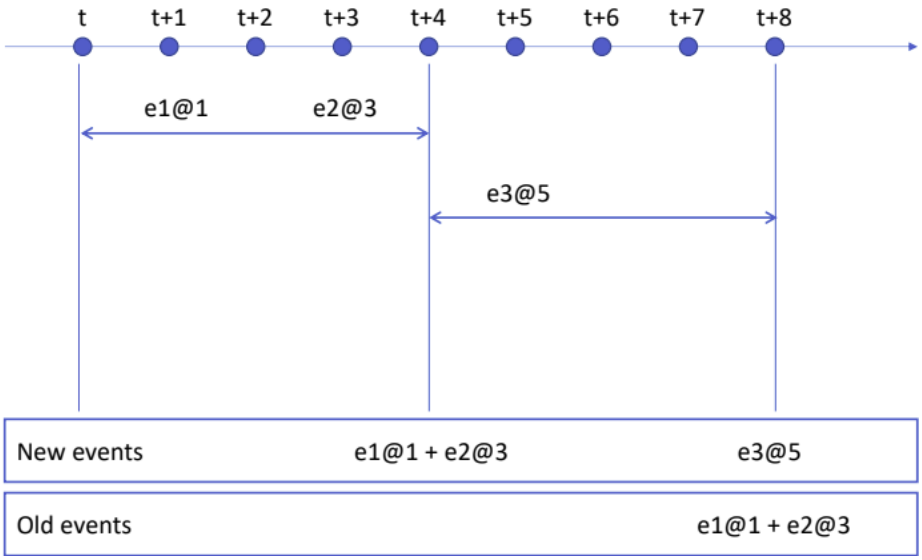
Since we are working with streams we need to specify how we want to translate our unbounded data into a batch, and for that purpose we use windows:

Type	Syntax	Description
Logical Sliding	win:time(*time_period*)	Sliding window that covers the specified time interval into the past
Logical Tumbling	win:time_batch(*time_period* [, *reference point*] [, *flow control*])	Tumbling window that batches events and releases them every specified time interval, with flow control options
Physical Sliding	win:length(*size*)	Sliding window that covers the specified number of elements into the past
Physical Tumbling	win:length_batch(*size*)	Tumbling window that batches events and releases them when a given minimum number of events has been collected

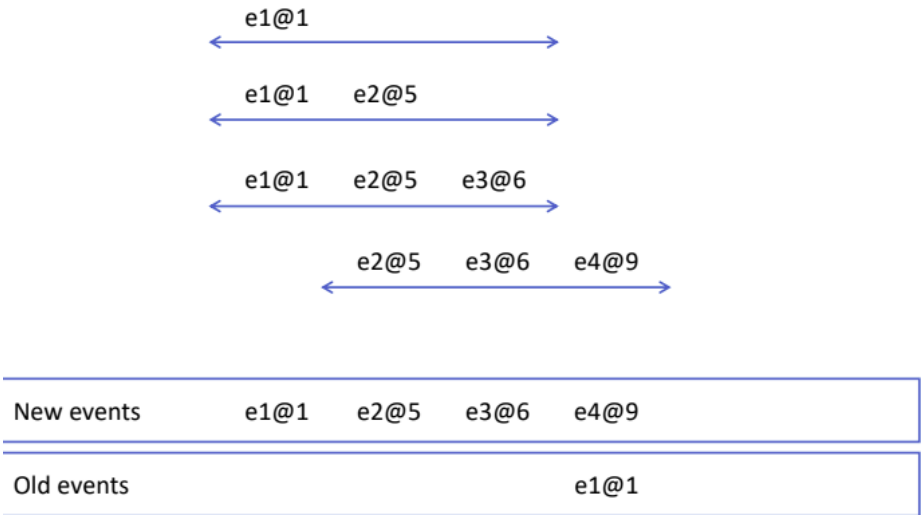
Here are some examples showing how each window works:



Tumbling window example: time_period = 4



Physical sliding window example: size = 3



There is one optional clause that can be added at the end of the query to manage some details about the query output. More specifically, this "output" command can be used to set the output rate and suppress output events. Here's its sintax:

```
output [all | first | last | snapshot]
every output_rate [seconds | events]
```

And here's a couple of examples of queries with windows and output control:

```
select avg(temperature)
from TemperatureSensorEvent.win:length(4)
output snapshot every 2 events
```

```
select avg(temperature)
from TemperatureSensorEvent.win:time(4 sec)
output snapshot every 2 sec
```

3.3.3 Pattern Matching

Let's now take a look at the pattern matching features in EPL and Esper, which allows you to emit a signal when an events that matches certain conditions happens:

- Content Based Event Selection:

  ```
  TempStream(sensor="S0", val>50)
  ```

- Time based events (they filter events basing on intervals or timers):

  ```
  timer:interval(10 seconds)  // fires after 10 seconds
  ```

  ```
  timer:at(5, *, *, *, *) // fires every 5 minutes
  ```

  Pattern Matching operators:

- Logical operators: and, or, not
- Temporal operators that operate on event order: $\rightarrow$ (followed-by)
- Creation/termination control: every, every-distinct, [num] and until
- Guards filter out events and cause termination: timer:within, timer:withinmax and while-expression

Example, search for SmokeEvents followed within 2 seconds by a Temperature Event from the same sensor with Temperature greater than 50 :

```
select a.sensor from pattern
[every (
    a = SmokeEvent(smoke=true)
    ->
    TempEvent(val>50, sensor=a.sensor)
    where timer:within(2 sec)
)]
```

Let's talk about the function of the "every" keyword in the previous query, its function it's to restart the pattern matching process every time an event with the specified characteristics is found. This means that if we didn't put that keyword in the query only the first SmokeEvent would have been considered.
The pattern matching behaviour can be modified in many ways by changing the way we use the every keyword.
Let's consider the following sequence of the events and let's see how each different usage modifies the query output:
SEQUENCE: A1 B1 B2 A2 A3 B3 A4 B4

- every (A $\rightarrow$ B): Detect an event A followed by an event B: at the time when B occurs, the pattern matches and restarts looking for the next A event
  OUTPUT : {A1, B1}, {A2, B3}, {A4, B4}
  (A3 isn't considered since the program was still searching for a match for A2)
- every A $\rightarrow$ B: The pattern fires for every A followed by a B event
  OUTPUT : {A1, B1}, {A2, B3}, {A3, B3}, {A4, B4}
- A $\rightarrow$ every B: The pattern fires for an A event followed by every B event
  OUTPUT : {A1, B1}, {A1, B2}, {A1, B3}, {A1, B4}
- every A $\rightarrow$ every B: The pattern fires for every A event followed by every B event
  OUTPUT : {A1, B1}, {A1, B2}, {A1, B3}, {A2, B3}, {A3, B3}, {A1, B4}, {A2, B4}, {A3, B4}, {A4, B4}

The every keyword can be combined with logical operators and windows to create more expressive queries:
SEQUENCE: A1 A2 B1

- every A $\rightarrow$ B: Same as before OUTPUT : {A1, B1}, {A2, B1}
- every A $\rightarrow$ (B and not A): Searches every A followed by a B without any other A in between
  OUTPUT : {A2, B1}

SEQUENCE(the number after the "@" denotes the time of the event in seconds):
A1@1 A2@3 B1@4

- every A $\rightarrow$ B: Same as before OUTPUT : {A1, B1}, {A2, B1}
- every A $\rightarrow$ (B where timer:within(2 sec)): Searches every A followed by a B within 2 seconds
  OUTPUT : {A2, B1}

Finally, the insert into clause forwards events to other streams for further downstream processing.
By combining all of these features we can write the final query for our fire detection system:

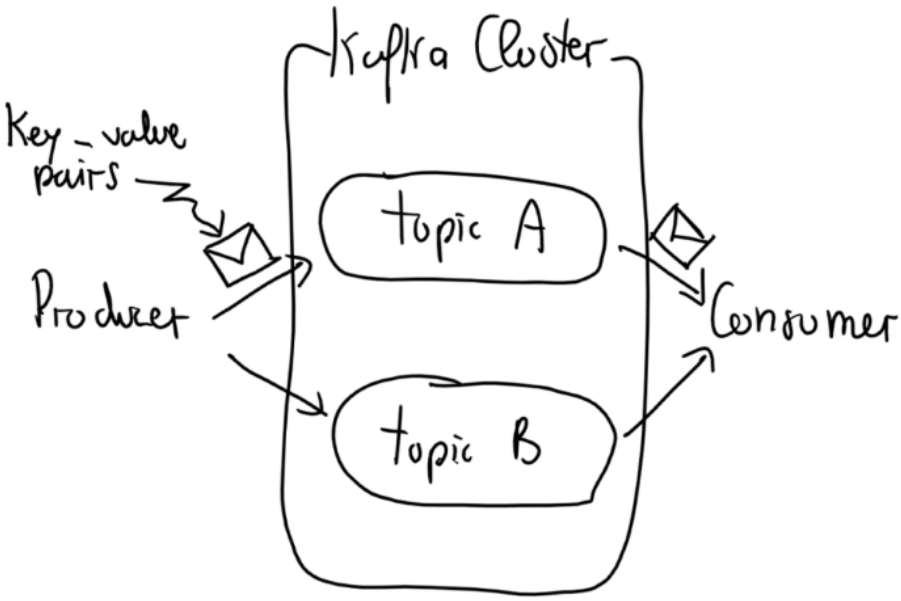
```
insert into FireComplexEvent

select          a.sensor as sensor,
                a.smoke as smoke,
                b.temperature as temperature
from pattern    [every a=SmokeSensorEvent(smoke=true)
                ->
                b=TemperatureSensorEvent(
                sensor=a.sensor, temperature>50)];

select          count(*)
from            FireComplexEvent.win:time(10 min);
```

3.4 Kafka

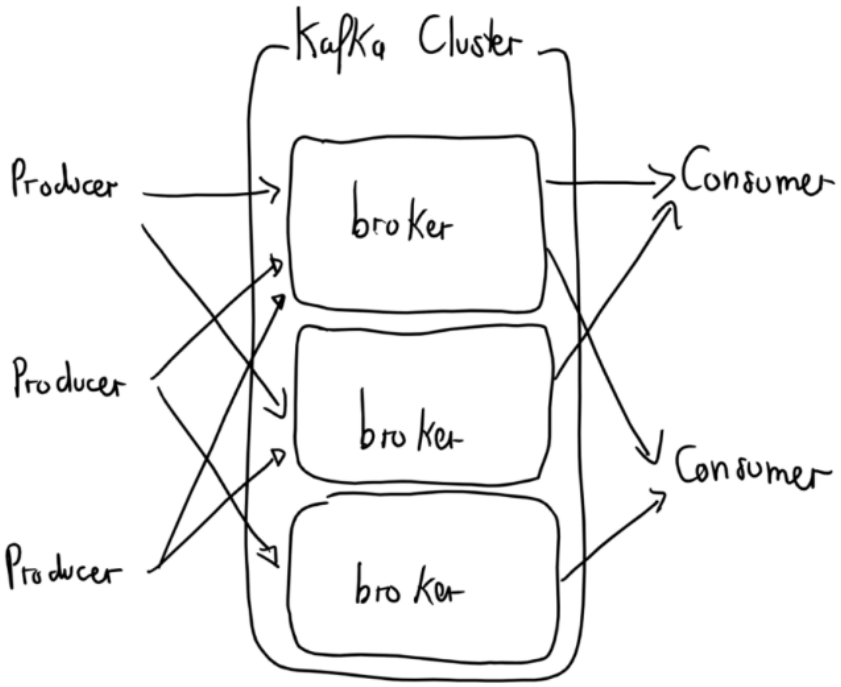
3.4.1 Logical View

Kafka is probably the most significant data stream processing system around at the moment.
While EPL and Esper are mainly a vertically scalable system, Kafka is completely based on horizontal scaling. This type of architecture allows to process a big quantity of data at very low latencies. There are some main concept/components that make up the Kafka architecture, which is based on the publish/subscribe model:

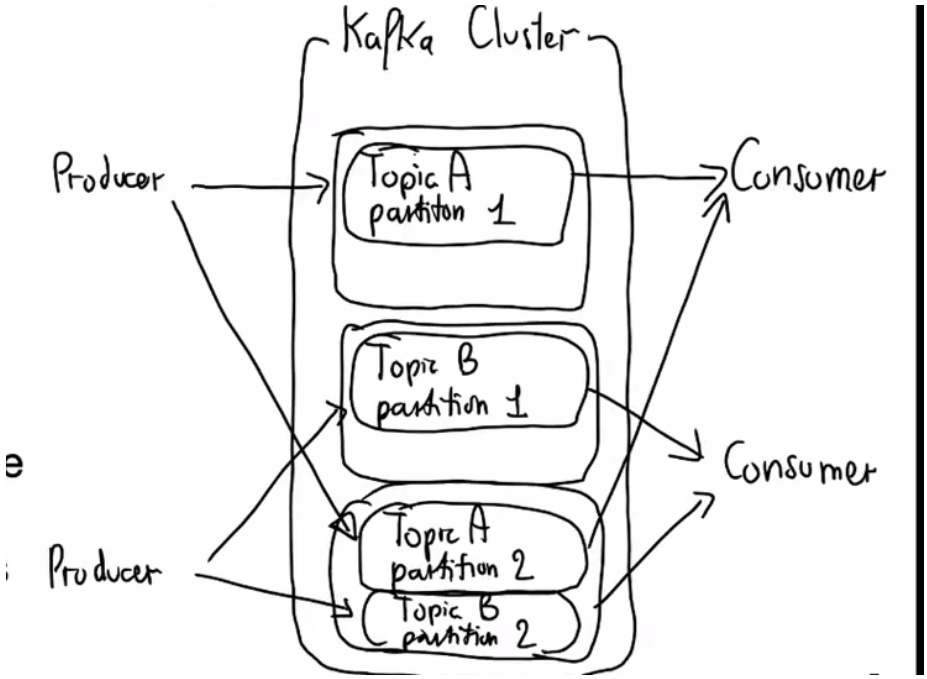
- Messages, simple key-value pairs
- Topics, stream of messages
- Producers, who produce the messages and send them to the Topics
- Consumer, who read the messages from the Topics
- Cluster, who organizes the Topics



From a logical point of view, a Kafka cluster is divided into Brokers, who are the main storage and messaging components.
It is important to note that Topics are just an abstract concept built on top of Brokers.

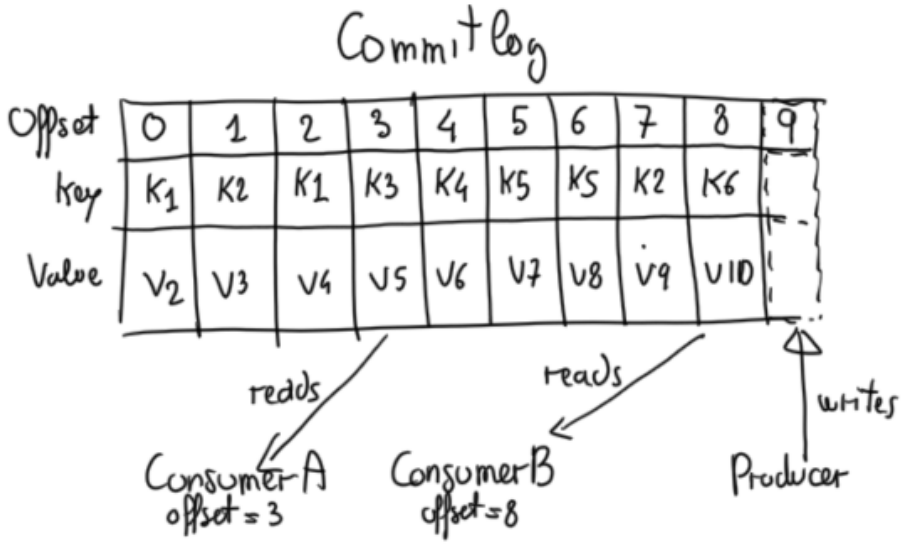


Topics are implemented in a distributed way: a single Topic is split in many partitions across different brokers, that may reside on different machines of the cluster.
This means that each producer sends a message to the broker containing the it's message key partition (the partitioning policy is usually implemented with an Round Robin or by an hash function to keep partitions balanced). Consumers work in the same way but from a read perspective.
Each Kafka cluster usually has at least 3 Brokers, all of which are single threaded, thus solving any possible concurrency related issue.
With Brokers we have covered how to scale the storage, but can we also scale the consumers? In fact we can, and that is done by dividing the consumers into groups and each item in the consumer group is assigned a certain number of partitions from which he can read. The consumer groups keep track of how many partitions each of its components has and rebalances them if needed.

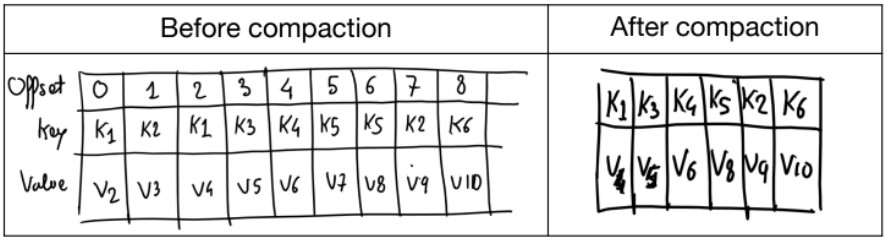


3.4.2 Physical View

Each partition is stored on the Broker's disk as one or more log files. Each message in the log is identified by its offset number.



New messages are appendend at the end of the log file, and Consumers can consume from different offsets. Buffers can be use to store in memory pieces of partitions and further improve performance.
Obviously log files can't keep growing forever, so Retention Policies need to be applied:messages are by default deleted after 7 days or after the log reaches a certain size (but this parameters can be customized, and Retention Policies can even be overridden by Topics).
During cleanup, the log can be either completely deleted or just compacted, keeping only the latest version of each key value.



In order to guarantee fault tolerance log files are replicated across Brokers, with a configurable number of replicas.
Replication is implemented with a master-follower approach, a Broker is the leader of the partition and all reads and writes have to go through him before being replicated to followers.
Producers can control durability by requiring the leader a number of acknowledgments the before considering the request complete.

- acks=0 Producer will not wait for any acknowledgment from the broker
- acks=1 Producer will wait until the leader has written the record to its local log
- acks=all Producer will wait until all insync replicas have acknowledged receipt of the record

The last important component of a Kafka cluster is Zookeeper, a centralized server that stores configuration for distributed applications.
Kafka Brokers use ZooKeeper for a number of important internal features:Cluster management, failure detection and recovery, access Control List (ACL) storage...

To summarize, here's a list of relation multiplicities between Kafka components:

- Broker to Partition → 1:N
- Key to Partition → N:1
- Producer to Topic → N:N
- Consumer Group to Topic → N:N
- Consumer in a Consumer Group to Partition → 1:N

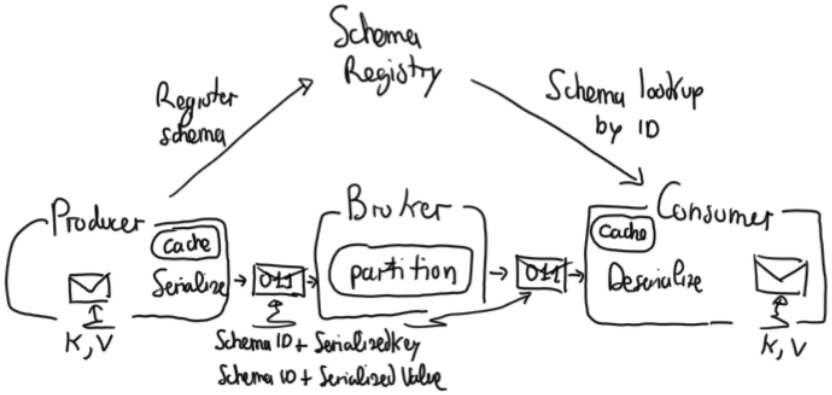
3.4.3 Avro and Schema Registry

It is possible to add a data manager layer on top of the key-value messages in order to treat them like rich data structures.
That's what Apache Avro does, it provides data serialization into a binary format, allowing for automatic generation of serializers and deserializers for any language and write time type checking.
Avro works by defining JSON schemas that can be applied to the Kafka messages.
This is the typical structure of an Avro schema:

```
{"namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "favorite_number", "type": ["int", "null"]},
    {"name": "favorite_color", "type": ["string", "null"]}
   ]
 }
```

Namespace defines the Avro environment, type and name are identifiers of the schema and fields define the actual schema structure.
Since sending the schema with each message would be inefficient, schemas are stored in the Avro Registry, a component that provides centralized management and identification for schemas, and this way tight coupling is avoided in Producers and Consumers.
The register stores all the schema ids, and the id-schema mapping is stored in a special Kafka Topic.

Here's a graph representing an Avro workflow in Kafka

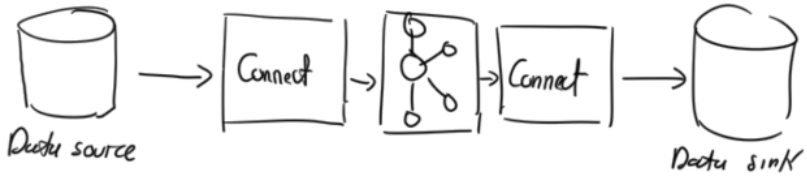


Producers register the new schema to the Schema Registry and send the serialized message alongside the schema id to the partition in the Broker, then when Consumers want to read that message they need to deserialize it by looking up the associated schema in the registry.
Another important feature of Avro is the support for Schema Evolution:

- Backward compatibility: Code with a new version of the schema can read data written in the old schema, and in that case will assume default values if fields are not provided
- Forward compatibility: Code with previous versions of the schema can read data written in a new schema, and in that case the new fields will be ignored
- Full compatibility: Forward and Backward

3.4.4 Kafka Connect

One of the most basic applications of Kafka is moving data from a data source to a data sink, ì.
Kafka Connect is a framework that allows you to do just that, providing you with Connectors to read and store data from and to many differents Data Systems in a scalable and reliable way.



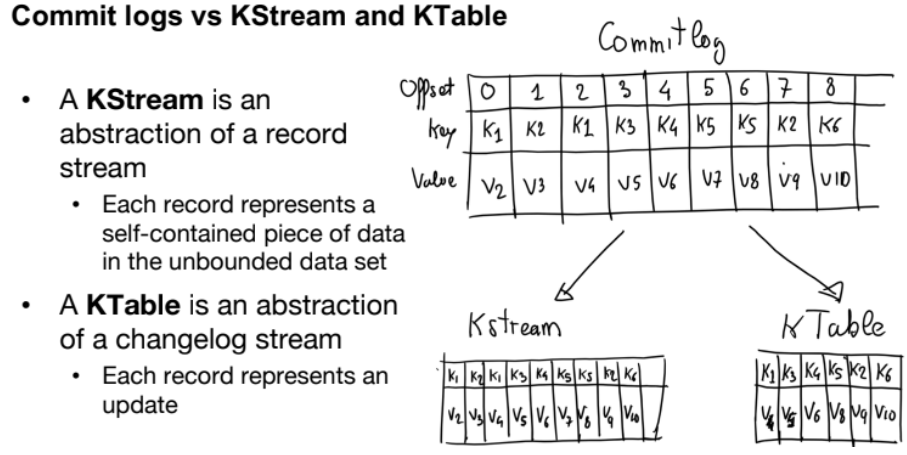
In a nutshell, Kafka Connect works by splitting the Connector's job into many task handled by different threads in parallel, and each of them is a Producer

that sends data from the Data Source to the Topic in the Kafka Cluster Data is serialized from the data source format into Avro compatible schemas with the use of Transformers, and in the same way Transformers deserialize data from Avro to the Sinks formats.
Kafka Connect has support, either in a built-in way or via plugins, for almost all of the most popular data technologies.

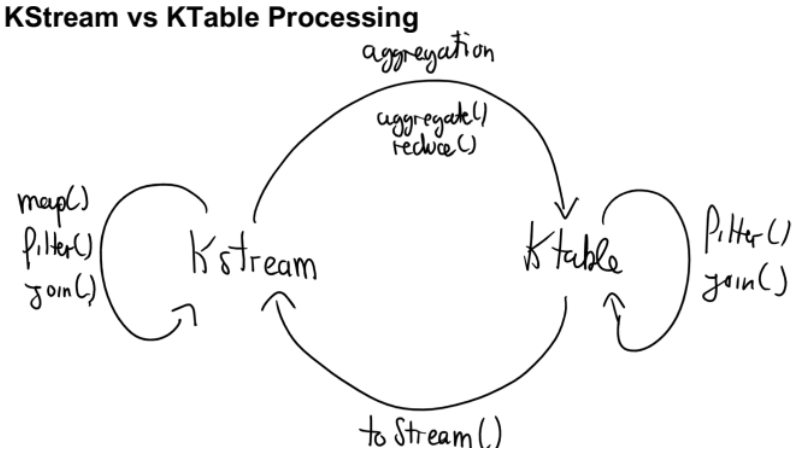
3.4.5 Kafka Stream Processing

Now that we have discussed the data streams system (the "message passing" part), we need to talk about how we can process the data of those streams. Kafka does this by stacking distributed applications on top of the Brokers. This can be done at many different abstraction levels, and Kafka offers APIs for most of them.
For example, there are Producers and Consumers APIs for all the most popular programming languages that allows you to write distributed application code on top of the Kafka Cluster.
There are also declarative libraries like Kafka Streams API that provides functional style commands to operate on data streams.
Finally at the top of the abstraction ladder there is KSQL that allows you to write SQL-like queries on streams.
There are two main data structures used to interface with streams with the Kafka Streams API:

- A KStream is an abstraction of a record stream:each record represents a self-contained piece of data in the unbounded data set
- A KTable is an abstraction of a changelog stream:each record represents an update

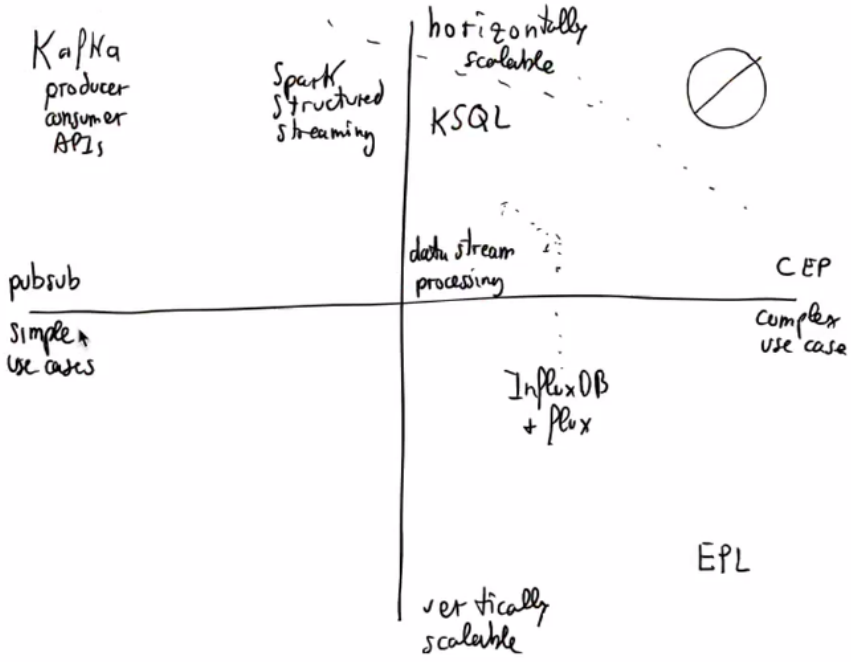


There's a language that allows you to perform operations on KStreams and KTables and to move between them.

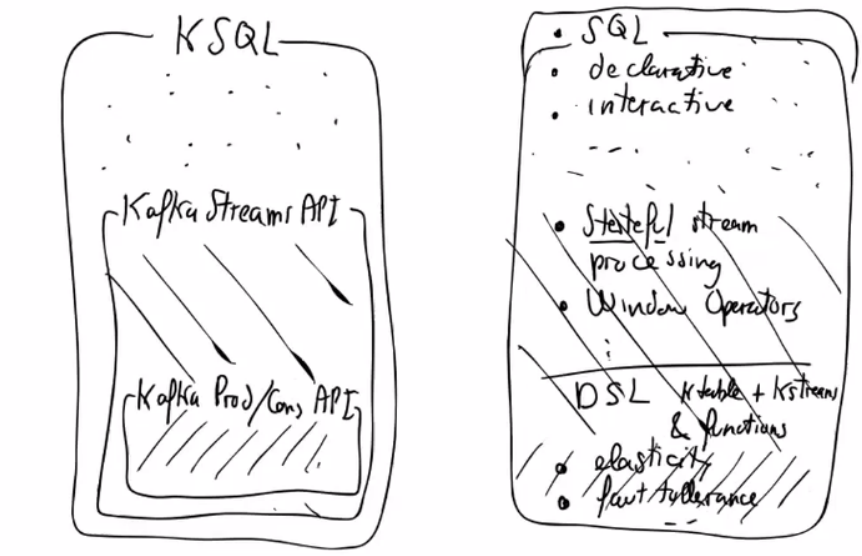


3.5 KSQL

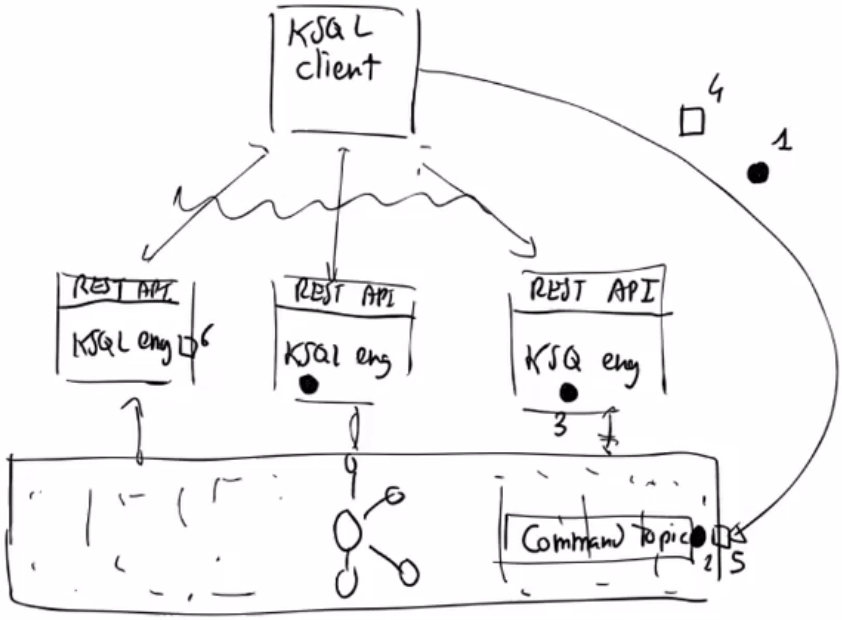
Before going forward let's talk a bit about how different data streaming technologies sit in the scalability-features graph:



The general guideline for this kind of technologies is "the more features you want in your system, the less scalable it will be". This means that we can have technologies capable of handling complex event relationships like EPL who as a drawback can only scale vertically, or we can have a system that can scale horizontally pretty much indefinitely but that only offers basic stream processing features.
In the middle between these 2 extreme use cases there are other technologies who like Spark Structured Streaming or KSQL, which is the focus of this section.
KSQL is an almost completely declarative stream processing language built on top of Kafka stream APIs. So basically it works by translating SQL queries into KStreams and KTables. This gives us the advantage of being able to use all of the features of Kafka stream APIs (like stateful stream processing and window operators) without having to write code but just queries.

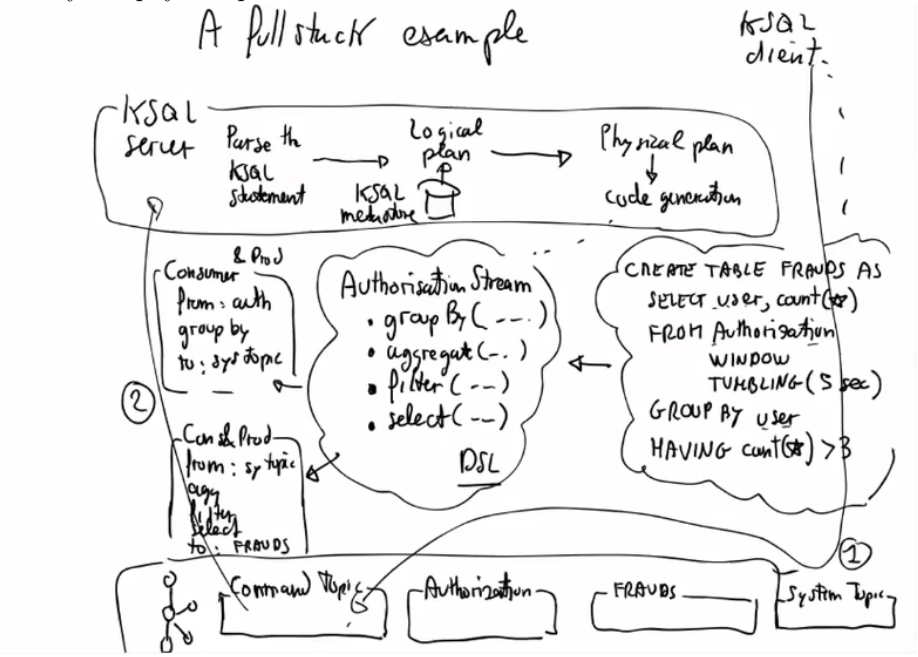


From a physical point of view, KSQL works by stacking up more machines, called KSQL engines, on top of an already existing Kafka Cluster, each oh which exposes a REST API



You can talk to the system by connecting to the REST APIs from a client in order to obtain some general information (number of machines in the cluster, network structure, metadata...), but in order to use KSQL you need to send the

query directly to a specific Command Topic in Kafka, who will in turn send it to the engines for processing. This architecture has the advantage of being able to run the commands in parallel, both in the topic partitions and in the KSQL engines. Moreover you also get Fault Tolerance this way.
The query processing process is similar to the one in relational databases: when the query reaches the KSQL server it's first parsed into a logical plan, and then into a physical one.
Finally the physical plan is translated into Kafka streams API code.



KSQL queries are run upon Streams or Tables connected to a Kafka Topic, which are the corresponding versions of KStreams and KTables, and their schemas can be described with Avro.
Examples:

```
CREATE STREAM Ratings WITH (KAFKA_TOPIC = 'ratings', VALUE_FORMAT = 'AVRO')
```

For the DML part, KSQL has very similar queries to SQL with the addition of window operators. Of course unlike in normal databases queries have an unbounded output. Examples:

```
SELECT * FROM Ratings WHERE Stars < 3;
```

The output of the queries can be persisted and used as input for other processes (this operation creates an underlying Kafka Topic):

```
CREATE STREAM POOR_RATINGS AS
SELECT * FROM ratings
WHERE STARS <3 AND CHANNEL='iOS';
```

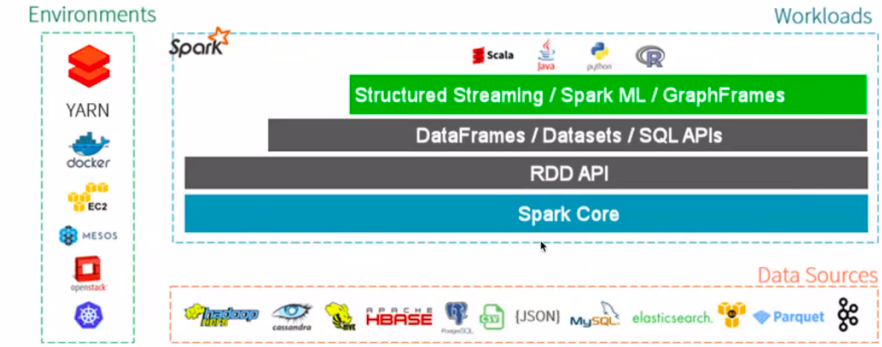
One thing to keep in mind is that if we want to write aggregation in our queries we need to specify a window and move from streams to tables, since it's impossible to do aggregation on unbounded data.
This is a final example of a complex KSQL query using aggregates, windows and tables:

```
CREATE TABLE RATINGS_BY_CLUB_STATUS_AND_CHANNEL_1M AS
SELECT WindowStart() AS TS,

CLUB_STATUS, CHANNEL,
CAST(COUNT(*) AS INT) AS CNT,
SUM(STARS) AS TOTS

FROM RATINGS_WITH_CUSTOMER_DATA

WINDOW TUMBLING (SIZE 60 SECONDS)

GROUP BY CLUB_STATUS, CHANNEL;
```

3.6 Spark

3.6.1 Introduction

Apache Spark is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing
It's similar to Hadoop but it has the purpose of being many times faster, and it achieves that goal by performing all computation in memory rather that on disk. While in Hadoop each partial output of a data processing job is saved in disk, Spark keeps everything he can in RAM and only stores in disk the final output of its jobs.
Spark has a layered architecture in which each layer has APIs that are used by the upper one. The higher the layer, the more abstract are the data structtures.
Spark has connectors for almost any existing data storage solution and it can run on a variety of environments (Yarn, Docker, Kubernetes, Mesos...).



These are its main data structures, from the most barebone to the most abstract:

- RDD:a distributed collection of JVM objects with functional operators
- Dataframe: relational db like collection of rows that support expression based operators and UDFs. Operations are very efficient thanks to logical plans and optimizers, that take adavantages its well defined internal representations.
- Dataset:abstraction that allows you to threat Dataframes like JVM objects. Operations aren't as fast as on pure Dataframes but they are still quite fast, typesafe and integrate well into traditional OOP programming. However they aren't as popular as DFs because they are not as convenient for data analysis with is what Spark is mainly used for.

3.6.2 Spark APIs

Let's go through each Spark API in detail:
RDD stands for: Resilient (Fault Tolerant), Distributed (split across many nodes and computed in parallel), Dataset. RDDs are immutable, that means that once they are initialized they can't change in any way.This allows to take advantage of many caching and locality strategies that greatly improve performance.
Since RDDs are distributed it's necessary to keep track of how each partition it's connected to the others and where each piece of data resides and which job generated it.
Here's an example of a wordCount job with RDDs written in Python:

```
# Open textFile for Spark Context RDD
text_file = sc.textFile("hdfs://...")
# Declare word count
res = (text_file.flatMap(lambda line: line.split())
.map(lambda word: (word, 1))
.reduceByKey(lambda a, b: a+b))

# Execute word count
res.collect()
```

On a RDDs it's possible to execute transformations (such as map, filter and group) or actions (such as count, collect and save).Transformations are lazily evaluated which means that they won't actually be executed even after their declaration until they are required by an action(which are eager instead).
Lazy execution has many advantages:

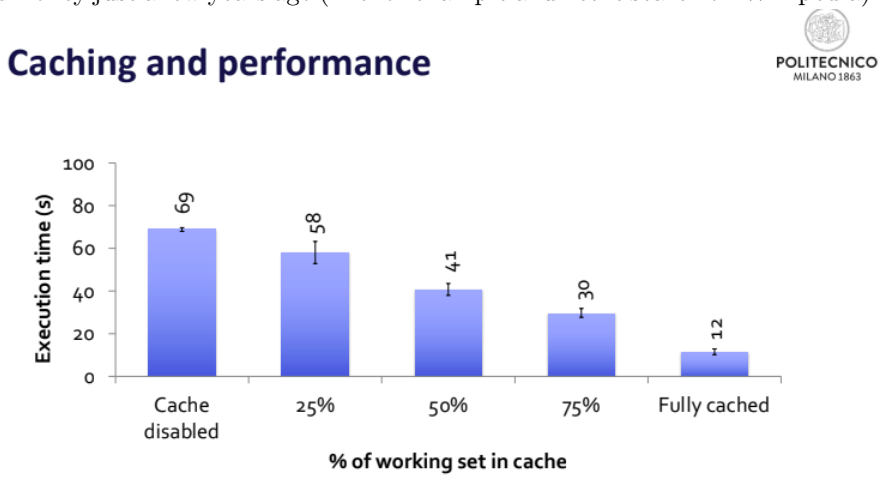
- Not forced to load all data at 1st step, technically impossible with large datasets
- Easier to parallelize operations: N different transformations can be processed on a single data element, on a single thread, on a single machine
- Allows optimizations

It's important to note that since RDDs are immutable each transformation doesn't modify the original but creates a new object.
Transformations can be either narrow if they can be executed within a single partition, or wide if they require data distribution between nodes (this operation is called shuffling). A Job is orchestrated by a Driver node and all the processing work on the partitions is done by Worker nodes.
Actions are operations that either return a value or write something to disk.
Another huge contributor to Spark's speed is caching, which basically works by keeping in main memory RDD blocks that can be reused in the same job instead of loading them from disk each time they're needed. Example:

```
lines = spark.textFile(\hdfs://...")
errors = lines.filter(lambda s: s.startswith(\ERROR"))
```

```
messages = errors.map(lambda s: s.split(\\t")[2])

//with this action the message RDD is kept in memory and reused in the next 2 lines
messages.cache()

messages.filter(lambda s: \mysql" in s).count()
messages.filter(lambda s: \php" in s).count()
```

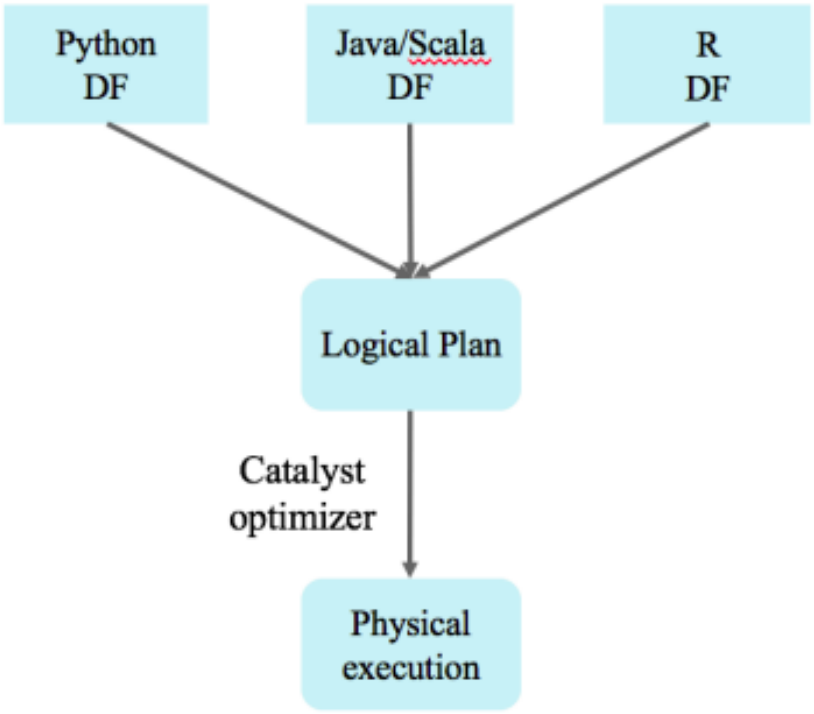
Thanks to all this optimizations with Spark is now possible to make with few resources and in a small time tasks that would have taken huge amounts of time and money just a few years ago (like for example a full text search on Wikipedia).



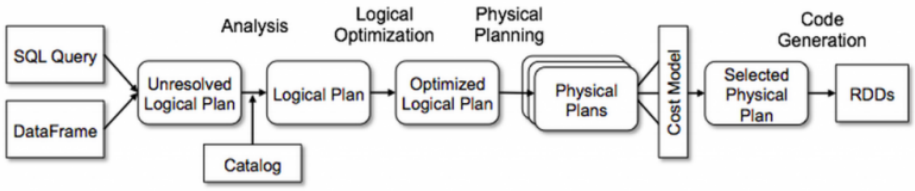
While RDDs are the building blocks of Spark, they are relatively low level abstractions and the APIs that Data Engineers usually use are the Dataframes. Dataframes are a collection of immutable data with named columns built on top of RDDs.They offer a user-friendy, cross-language API and support many performance optimizations. Here's an example of Dataframe Python code which utilizes SQL to query and join data coming from different sources.

```
# Read a CSV
userDF = spark.read.csv(" .../userData.csv")
# ...or Use DataFrame APIs and register a temp view
middleageSmokers = userDF.filter(col("smoker")=="N").filter(col("age")>40)
middleageSmokers.createOrReplaceTempView("middleageSmokers")
# ...read a CSV and register a temp view
spark.read.json(" .../part-00000.json.gz").createOrReplaceTempView("iot_stream")
# ...execute SQL query or ...
spark.sql("SELECT avg(calories_burnt) FROM iot_stream JOIN middleageSmokers ON ...")
```

One very important feature of dataframes is that every operations has the same performance across every programming language.This is possible because the language APIs are just declarative interfaces calling a low level process that formulates the best logical plan for that operation and execute in by generating RDD code.



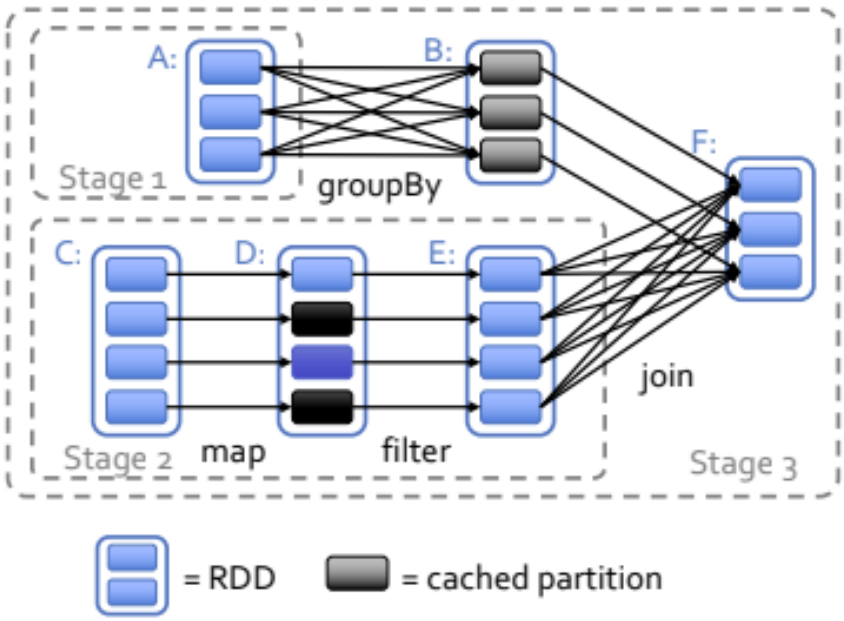
Here's the query manager architecture for Dataframes:



The last Spark API is the Dataset, which offers strong static typing over Dataframes. This feature (which is only useful in languages with static types like Java and Scala) allows for more compile time error checks compared to the loosely typed Dataframes.

3.6.3 Spark Scheduler

The DAG Scheduler is the scheduling layer of Spark, it transforms a logical execution plan into a physical one (a Job) considering data locality and partitioning (to avoid shuffles).
Each Job is a DAG of Stages which are broken down into Tasks



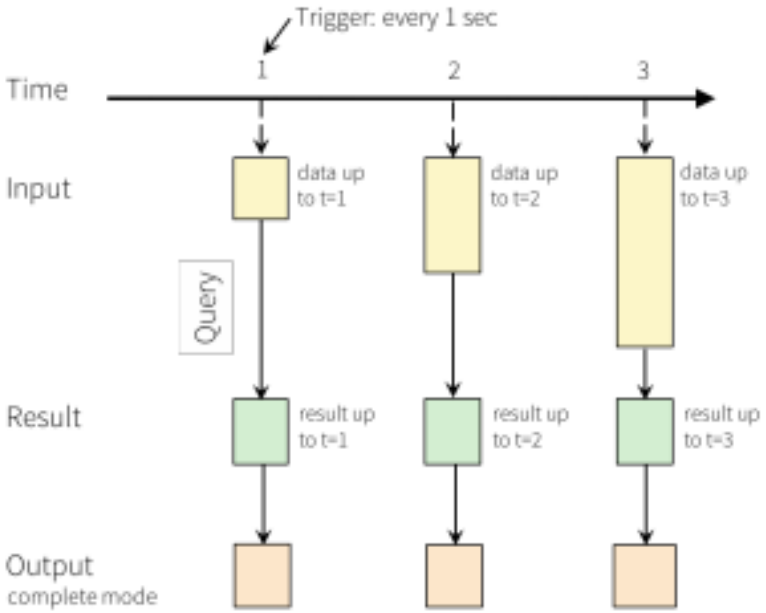
3.6.4 Spark Structured Streaming

Spark Structured Streaming(SSS) provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.This means that Spark allows you to operate on streams by writing Dataframe and the developer could even forget he is working on a stream.
There are 2 possible operational modes in SSS

- Micro-batch processing engine:the default mode, it offers end-to-end latencies as low as 100 milliseconds and exactly-once fault-tolerance guarantees
- Continuous Processing:Added from Spark 2.3, it offers end-to-end latencies as low as 1 millisecond and at-least-once guarantees

The key idea behind this technology is treating a stream like a table that is being continuously appended, this way the user can write static queries over a Dataframe and Spark runs it as an incremental query over the unbounded table.

The programming model works by executing a new computation periodically, and the state calculated in the current computation is passed on to the next one so each computation works with all existing data by integrating past state and the newly received records.

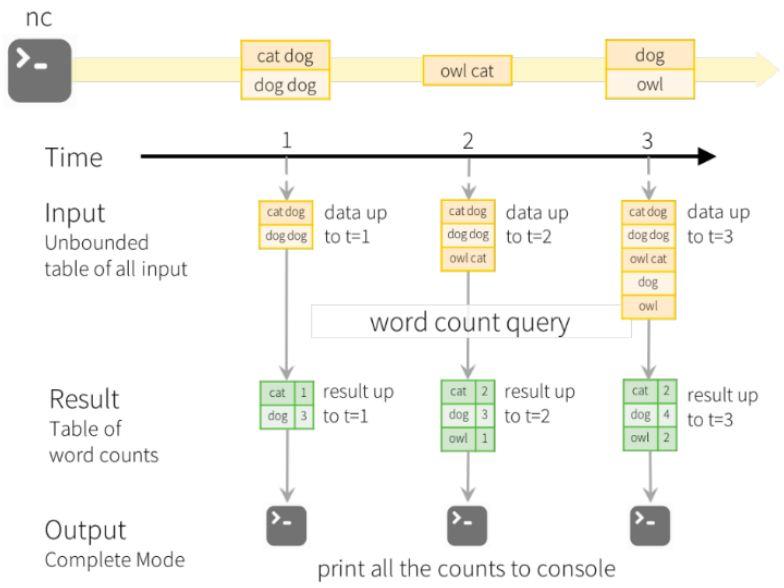


There are 3 possible output modes for stream processing jobs:

- Complete Mode: The entire updated result is written
- Append Mode: Only the new rows appended in the result are written, applicable only when existing rows in the result are not expected to change
- Update Mode: Only outputs the rows that have changed since the last trigger.If the query doesn't contain aggregations, it will be equivalent to Append mode.

Each mode is compatible only with some queries and data sinks and so the Data Engineer needs to evaluate the best mode for each use case.

Here's an example of a word count process in a streaming environment that shows how Spark computes new state from old state and current record (old records are discarded since only the state derived from them is necessary to the following computations).



Model of the Quick Example

The SSS Dataframes API works by calling the SparkSession.readStream() function which returns a stream based Dataframe from a specified source, which could be a file(supported formats include CSV, JSON, ORC, Parquet), a Kafka Topic or even a Socket(even though this way you don't get Fault Tolerance).
Python Examples:

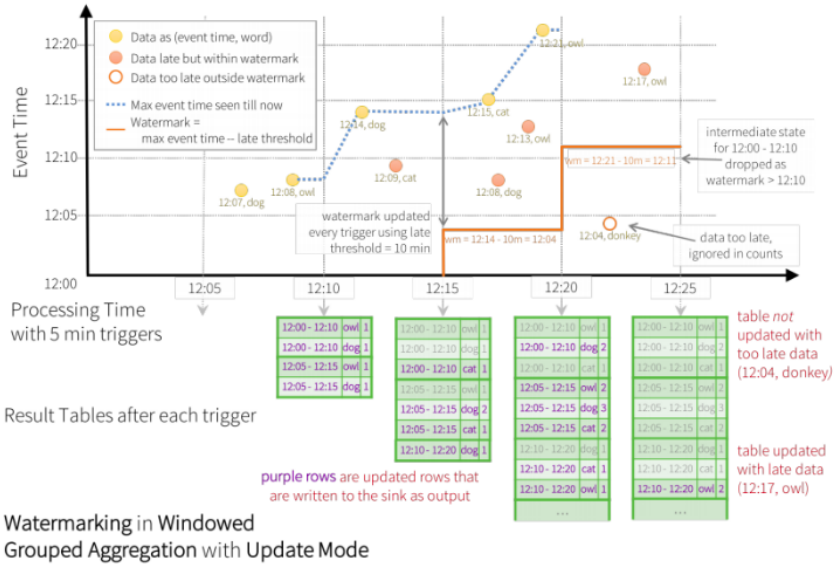
```
spark = SparkSession....
# Read text from socket
socketDF = spark
              .readStream
              .format("socket")
              .option("host", "localhost")
              .option("port", 9999)
               .load()
socketDF.isStreaming() # Returns True for DataFrames that have streaming sources
socketDF.printSchema()
```

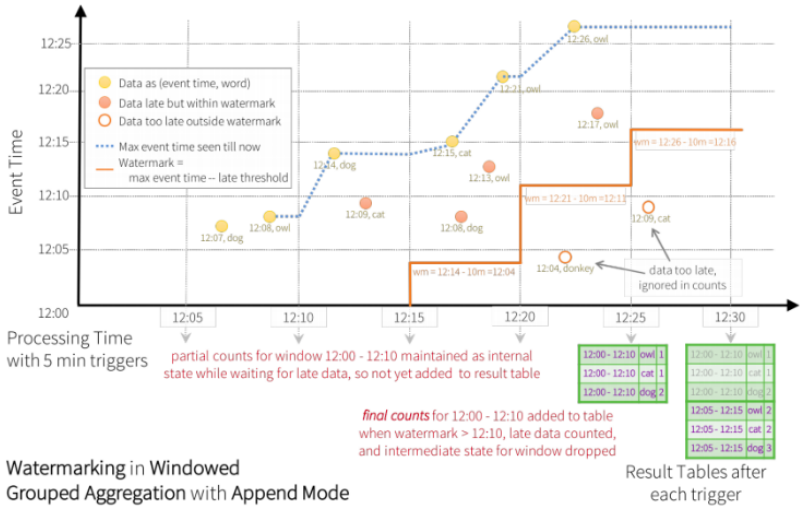
```
# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark
          .readStream
          .option("sep", ";")
          .schema(userSchema)
          .csv("/path/to/directory")



# Input: streaming DataFrame with IOT device data with schema
{ device: string, deviceType: string, signal: double, time: DateType }

df = ...
# Select the devices which have signal more than 10
df.select("device").where("signal > 10")
# Running count of the number of updates for each device type
df.groupBy("deviceType").count()

# Alternatively, register a streaming DataFrame/Dataset as a temporary
view and apply SQL on it

df.createOrReplaceTempView("updates")
spark.sql("select count(*) from updates")
```

Windows operations are implemented through an additional timestamp column, and are viewed as grouping operations based on that column.
Example

```
words = ...# streaming DataFrame of schema { timestamp: Timestamp, word: String }
# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```

An important topic in SSS is how late records (which are record with a different generation time and ingestion time) are managed.
The solution is to use watermarking, which means that in update mode the dataframes can be retroactively updated with late records if the delay is within a certain time limit, while in append mode the processing output is delayed for some time in order to wait for possible late entries.
This way you at least have the guarantee that data with a delay inferior to a certain time frame won't be ignored.Data with greater delays could either be ignored or considered, but the greater the delay the more likely it is that the record won't be taken into account.





Spark Structured Streaming supports both stream-to-stream and stream-to-table joins.
Of course native stream joins isn't possible, as it isn't possible to compare each row of 2 unbounded tables, so Spark performs this kind of joins by using watermarks that tells you how long should you keep on buffering before performing a static join with all the buffered data.

Example:

```
from pyspark.sql.functions import expr
impressions = spark.readStream....
clicks = spark.readStream....
# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")
# Join with event-time constraints
impressionsWithWatermark.join(
    clicksWithWatermark,
    expr("""
        clickAdId = impressionAdId AND
        clickTime >= impressionTime AND
        clickTime <= impressionTime + interval 1 hour
        """)
)
```

Here's the join compatibility list:

- Inner: supported, optionally specify watermark on both sides + time constraints for state cleanup
- Right/Left Outer: conditionally supported, must specify watermarks on left/right + time constraints for correct results, optionally specify watermark on left for all state cleanup
- Full Outer: not supported.

The only topic left to discuss is the stream configuration.When creating a query on a stream in fact you need to specify the following:

- Output sink: Data format, location, etc.
- Output mode: what gets written
- Query name: Optionally, a unique name of the query
- Trigger interval: Optionally, the trigger interval
- Checkpoint location: for the end-to-end fault-tolerance

The output sink can be one of the following:

- File sink - a directory
- Kafka sink - one or more topics in Kafka
- Foreach sink - Runs arbitrary computation
- Console sink (for debugging)
- Memory sink (for debugging)

3.7 Flux

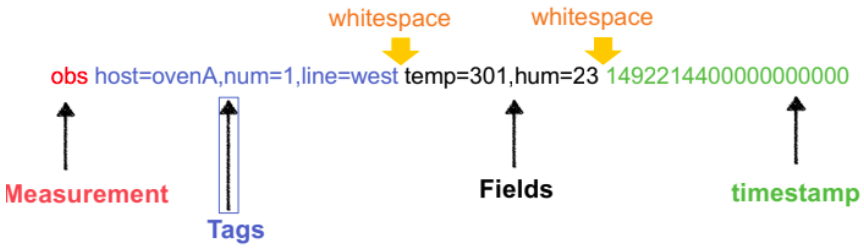
3.7.1 Introduction

Flux is, as stated by its creator, a platform where any kind of event and metric can be processed.It's made up of a open source core which contains an UI, an ingestion system, a steaming events processor and a purpose-built time-series DB, and of a commercial offering that adds some more features like clustering, security and manageability.
the db is called InfluxDB, it's optimized for storing time-series data, which is data that can be interpreted with a graph that has time on the x axis and values on the y axis.

The logical structure of InfluxDB is composed of:

- Measurement: name to group data at high level
- Tag set: A set of key-value pairs to group data at low level (values are strings)
- Field set: A set of key-value pairs to represent data (values are numerical & strings)
- Timestamp:Time of the data with nanosecond precision
- Series:A unique combination of measurement+tags

At a physical level, an event is represented as a line with a specific format:



Measurements are stored in buckets, which are a set of time-series and could contain many different event types.
Buckets built with a columnar structure, similar to the one of the DBs we talked about in the previous sections:

_time	_m	host	num	line	temp	humidity
1492...1	obs	ovenB	1	west	301	23
1492...0	obs	ovenA	1	west	125	75
...	...	...	...	...	...	...

Flux's data collection agent is called Telegraf, it has a plugplay architecture with a variety of plugins. Flux is a data scripting language built with the purpose in mind of putting together in one tool the advantages of both query processing and programming languages.

3.7.2 Flux language

A typical Flux query is made of :

- Data Source, which is usually a bucket
- Time Range
- Data Filters, which are functional operators that can be applied to the data and chained with the pipe operator

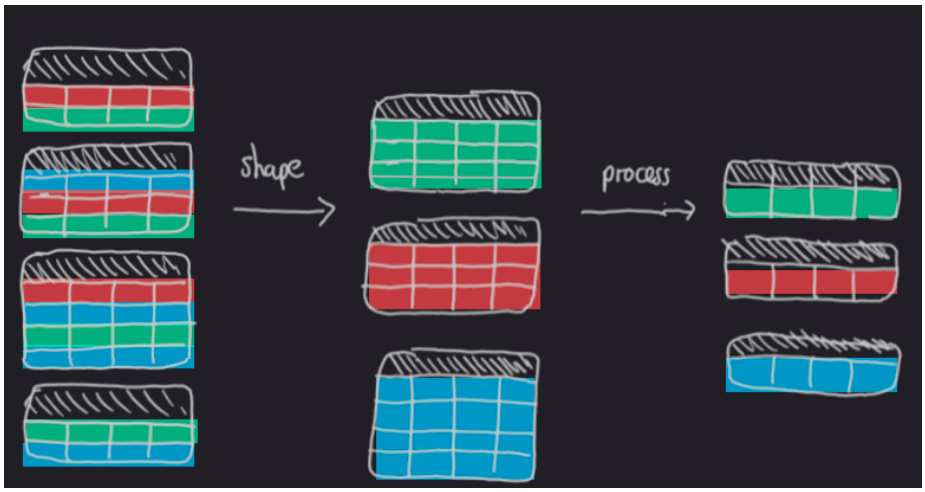
Example Query:

```
from(bucket:"telegraf/autogen")          // data source
    |> range(start:-1h)                  // time range
    |> filter(fn: (r) =>                 // filtering
        r._measurement == "oven" and
        r._field == "temp" and
        r._value > 300.0)
```

The data model is an infinite stream of finite tables(which are created by parsing bucket data) identified by the GroupKey(which is an identifier of a group of measurements and tag sets).

Each incoming row in a stream is processed and from the inputs an output table is created.
Every query undergoes a set of logical and physical optimizations which are done both in client and server side.
In addition to filtering, there are other three powerful operations available in flux:shaping, processing and selecting:

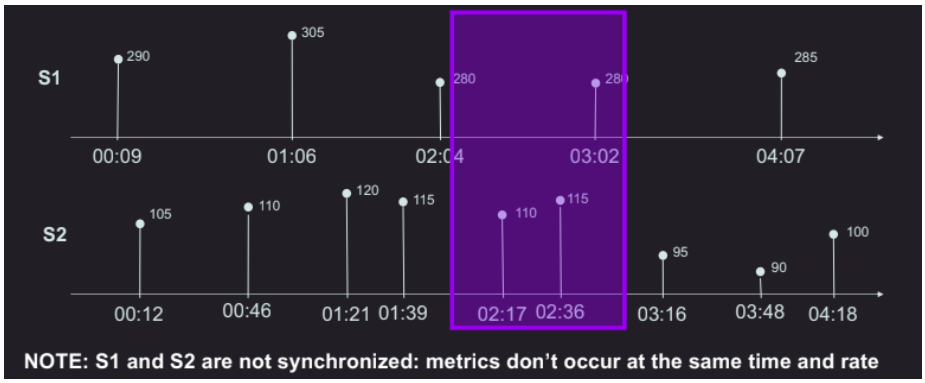
```
from(bucket:"telegraf/autogen")
    |> range(start:-1h)
    |> filter(fn: (r) =>
        r._measurement == "oven" and
        r._field == "temp" and
        r._value > 300.0)
    |> group(columns: ["_field"])    //grouping
    |> mean()                        //processing
    |> last()                        //selecting
```



Tumbling windows are supported as a way to specify the behaviour of a processing operation.
Example of a tumbling window applied to the previous query:

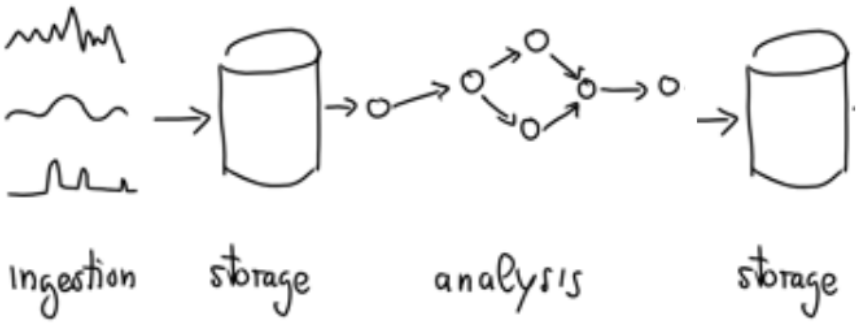
```
|> aggregateWindow(every: 60m, fn: mean)
```

Joins are also supported of course, and if the 2 time series to be joined are synchronized, the operations works in the same way as any traditional database.
However this is rarely the case and so a tumbling window is needed to synchronize them.




```
temp = from(bucket: "training")
|> range(start: 2020-04-30T10:00:00Z, stop: 2020-04-30T10:05:00Z)
|> filter(fn: (r) => r._measurement == "iot-oven")
|> filter(fn: (r) => r._field == "temperature")
tempS1 = temp |> filter(fn: (r) => r.sensor == "S1")
|> aggregateWindow(every: 60s, fn: mean)
tempS2 = temp |> filter(fn: (r) => r.sensor == "S2")
|> aggregateWindow(every: 60s, fn: mean)

temp_join = join( tables: {s1:tempS1, s2: tempS2}, on: ["_time"] )
temp_join |> map(fn: (r) => ({ _time: r._time,

_value: r._value_s1 - r._value_s2 }))
```

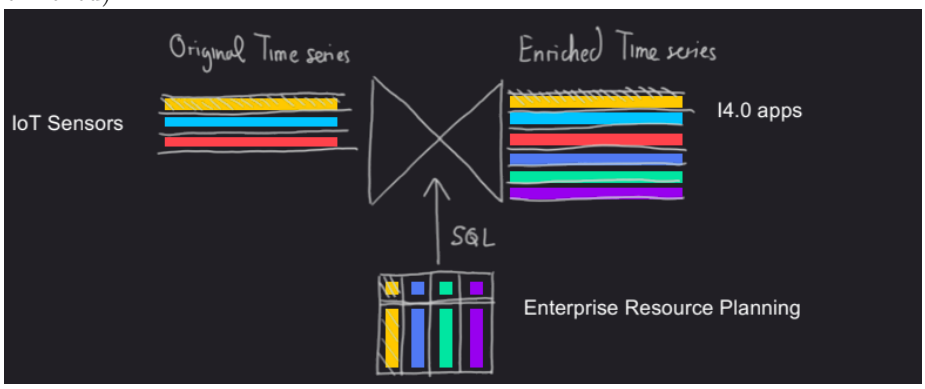
Flux also has support for continuous queries, called Tasks, and they work by continuously writing back to the storage the results of the analysis



Task syntax is the same of standard queries plus an options configuration which includes a name and a cron command for the query.

3.7.3 Time Series Enrichment

It's also possible to do stream to table joins in flux, also called time-series enrichment.
This offers you the feature to have a normal time series flow but in which every record has also access to data from a different data source(that's why it's called enriched).




```
import "sql"
staticSrc = sql.from(driverName: "...",
dataSourceName: "...",
query:"SELECT ...FROM ...WHERE ...")

streamingSrc = from(bucket: "...")
    |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
    |> ...

join(tables: {R: staticSrc, S: streamingSrc}, on: ["..." ])
    |> ...
```

Lastly Flux also has support for more advanced mathematical and analytics functions, like mapreduce, derivatives, integrals, regex, holt-winter and many more.

4 Data Pipelines

4.1 Data Ingestion

4.1.1 Introduction

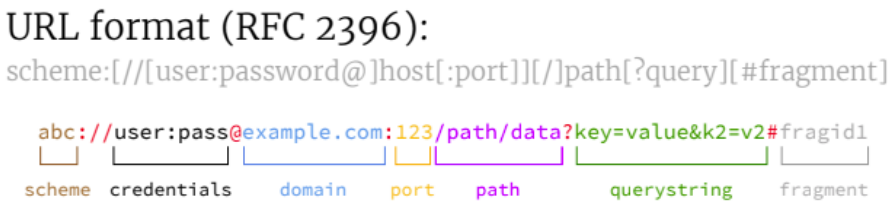
Data ingestion is the first and fundamental step of any Data Analysis Pipeline. The focus of this section is on how is it possible to collect data from publicly available sources over the web.
It is in fact a common practice nowadays to integrate the proprietary data coming from OLTP databases with data coming from public web resources in order to also have a data source which isn't coming from the Company domain and (and could as such be biased in many ways) . We'll now dive deep into the two main ways of collecting data from the web:APIs and Scraping.

4.1.2 Web APIs

An API (Application Program Interface) is a set of routines, protocols and tools for building software applications. A Web API is a API which is based on the HTTP protocol, and it's usually used to give programmatic access to data and platforms.
The main advantage of Web APIs are:

- Separation between model and presentation
- Regulate access to the data (traceable accounts, enable access to only a portion of the available data)
- Avoid direct access to the platform website
- Request throttling
- Avoid server congestion
- Avoid DOS/DDOS
- Provide paid access to full (or higher volume) data

To access a resource via Web API, you need to specify a method and an URL:



There are many commands (GET, POST, PUT, DELETE...) each with its own purpose. A request can be tweaked by adding parameters to end of the url;Example:

```
GET https://api.twitter.com/.../search/tweets.json
?q=%23expo2015milano
&lang=it
&result_type=recent
&count=100
&token=1235abcd
```

To give a more standardized structure to the APIs, most developers follow the REST paradigm.
The RESTful approach is:

- Client-Server: a uniform interface separates clients from servers.
- Stateless: the client-server communication is constrained by no client context being stored on the server.
- Cacheable: clients and intermediaries can cache responses.
- Layered system: clients cannot tell whether are connected directly to the server.
- Uniform interface: simplifies and decouples the architecture.
- Resource centric

A REST API in comparison to a generic one has the advantage of having a standard/predictable URL format:

```
METHOD http://domain/collection/item
GET http://example.com/user/1234
```

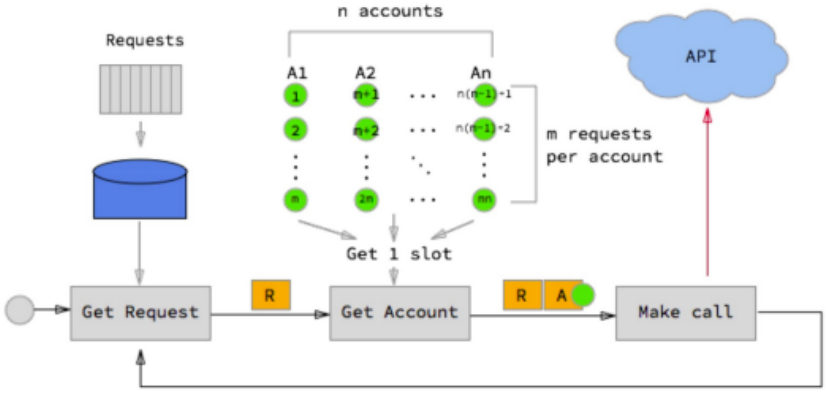
REST also standardizes the behaviour of a Method call based on the context (collection/item)

VERB	Collection	Item
POST	Create a new item.	Not used
GET	Get list of elements.	Get the selected item.
PUT	Not used	Update the selected item.
DELETE	Not used	Delete the selected item.

Almost all the APIs require a kind of user authentication.The user must register to the developer of the provider to obtain the keys to access the API. Most of the main API providers use the OAuth protocol to authenticate a user.

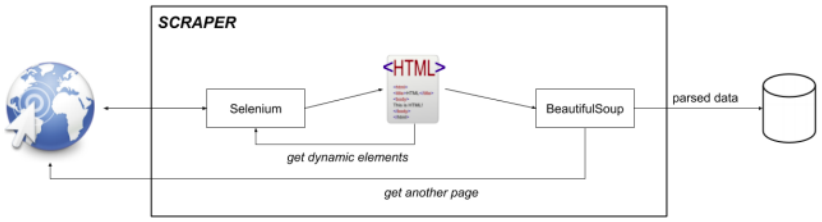
The user registers to the developer portal to obtain a key and a secret. Platform authenticates the user via key/secret pair and supply a token that can be

used to perform HTTP(S) requests to the desired API endpoint. The de facto standard format for exchanging data with REST APIs is JSON, who has almost completely replaced XML thanks to its simplicity and Javascript integration.
Since the HTTP calls are stateless, state need to be stored either in the client application or in the resources
Now that we have seen a high level overview of Web APIs, how can we actually use them to collect data? If we need to get a big amount of data from an API it isn't possible to do it in just one call, we need to develop a Crawler, which is a software that methodically interacts with a WebAPI to download data or to take some actions at predefined time intervals."
Like SQL queries and Web search engines, most APIs support data pagination to split huge chunks of data into smaller set of data.
Smaller chunks of data are easier to create, transfer (avoid long response time), cache and require less server computation time. Moreover many websites, like Facebook, base their pagination system on the concept of timeline, adding more challenges to the crawlers, that need to use cursor to keep track of their API request and keep making calls increasing the pagination index until they have collected all the necessary data.
When dealing with big amounts of data to collect, crawlers scale by using parallelization and splitting the collecting task into more nodes and accounts(often API providers put a limit on the number of requests a single account can make so more accounts are needed in order to collect large quantities of data).
This allows for great performance boost but also adds the complexity needed to coordinated all the crawlers working in parallel.These complexity is dealed with by extablishing task division policies.
In the following example a Round Robin policy is used in order to assign requests from a stack to the nodes of a crawler:



4.1.3 Web Scraping

If there's no API available to get data from a website, the next best thing is web scraping.
Web scraping, web data mining, web harvesting are synonyms to one single concept: "Build an agent which can extract, parse, download and organize useful information directly from HTML source code". To get meaningful data with web scraping you need to download the HTML code of the page you want to scrape and then analyze its structure and extract the information you need.
To navigate the HTML structure we can use a query language called XPath, which combined with regular expressions allows you to extract content from web pages in a very precise way.
If we need to scrape a dynamic page the process becomes more complex, as we may need to overcome authorization forms or perform some more complicated action in order to get the content we need.
Python is a very good language for building web scraper since it offers many good libraries and tools to help you. One very popular toolset is the combination of Selenium (a headless browser with support for dynamic behaviour) and BeautifulSoup (a HTML navigator)



It's important to keep in mind that scraping shuld be used just as a last resort when there's no API available, since scraper are very difficult to mantain (if the website changes structure the scraper doesn't work anymore) and many website providers don't allow scraping, so you could run into legal problems if you do.

4.2 Data Wrangling

4.2.1 The need for clean data

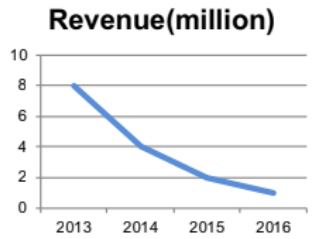
The step after Data Acquisition and before Analysis in the data management flow is Data Wrangling, also called Data Cleaning or Data Preparation.
In this step data is cleaned, tested and prepared to be the best input possible for the analysis process.In other words, we need to ensure that our data has high quality, which is a property defined by this metrics:

- 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.

Unfortunately it isn't easy to measure and define these metrics (e.g. how do you know that you're missing data? How do you know that the data is accurate). In fact, these metrics could be:

- Unmeasurable: Accuracy and completeness are extremely difficult, perhaps impossible to measure.
- Context independent: No accounting for what is important. E.g., if you are computing aggregates, you can tolerate a lot of inaccuracy.
- Incomplete:What about interpretability, accessibility, metadata, analysis, etc.
- Vague:The conventional definitions provide no guidance towards practical improvements of the data.

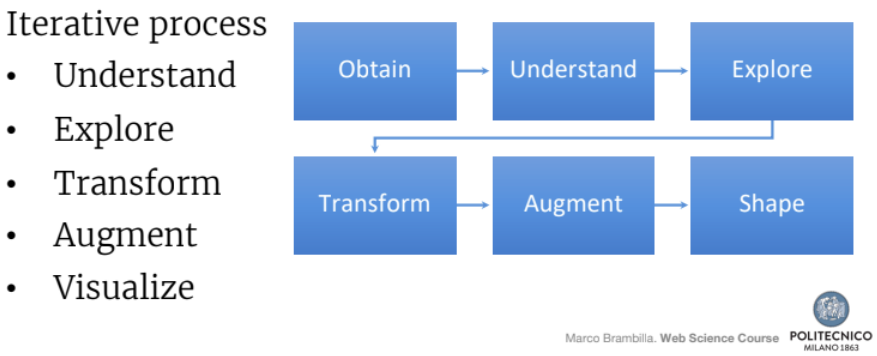
For these reasons data wrangling is often the most crucial, difficult and predominant(statistics say that in average 80% of the time of a data scientist is spent cleaning data) task of a data scientist/engineer.
If the data wrangling step is done poorly, it may lead to analysis being run on bad data which could in turn lead to bad decision making in company and bad business.
Research says that this is a very common issue even in big companies, who often can't even trust their own internally generated data.



Data Quality is an issue…

4.2.2 Data cleansing

So how can we transform raw unreliable data into refined high quality material? The common solution that has been spreading in the last few years is to improve the traditional ETL(Extract-Transform-Load) approach by adding more steps that test and improve the quality raw data.

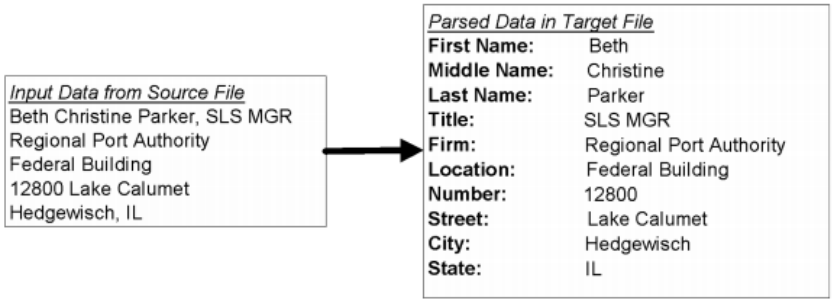


The first of this steps is the so-called Data cleansing or Data Scrubbing, which is the act of detecting and correcting (or removing) corrupt or inaccurate records from a data set.
The term refers to identifying incomplete, incorrect, inaccurate, partial or irrelevant parts of the data and then replacing, modifying, filling in or deleting this dirty data (when it's just too noisy to be fixed).

But what does it mean to have dirty data? to give you an idea, here are some examples:

- 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

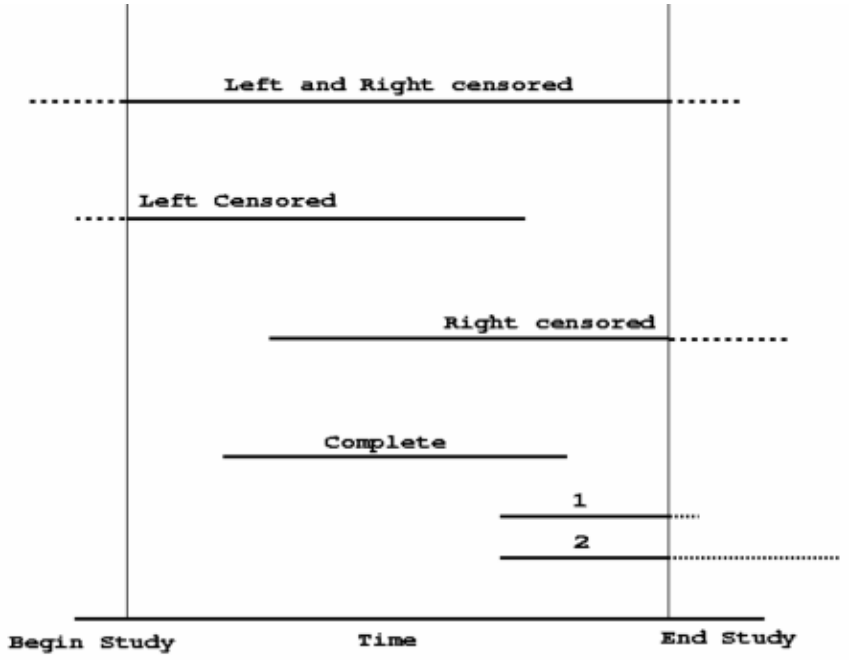
In practice, data cleansing consist of a few actions that is common to perform on raw data: for example data may need to be parsed, which means to be converted from some kind of raw format (often plain text) to a more formal structure.
Example:



Even after been parsed, data may still have missing fields or wrong values, which need to be corrected by using sophisticated data algorithms and secondary data sources.
Then, data may need to be standardized, by applying conversion routines to transform data into its preferred (and consistent) format using both standard and custom business rules, as well as coherent measurement units,
The next step is analyzing and identifying relationships between matched records and consolidating/merging them into ONE representation, which is usually done by pattern matching on historical data sources. If after the pattern matching two pieces of data appear to be referring to the same record, one of them is discarded.
Only after all this steps data can be considered reliable.
Sometimes data can come in an completely unstructured shape (like PDFs or free text), and complex algorithms need to be applied in order to extract some kind of structure from it.
Sometime it may be necessary to perform an operation called Data Munging, which are potentially lossy transformations applied to a piece of data or a file, or vague data transformation steps that are not yet completely clear (E.g., re-

moving punctuation or html tags, data parsing, filtering, and transformation)
Let's focus now a bit on maybe the most problematic issue when dealing with raw data:missing data. We may miss entire files or just some fields, or the data could be present but damaged.
There can be many reason for missing data:source system faults, ingestion errors(e.g. wrong schema), data who is partial by nature (like empty forms).
We can deal with these kind of situations in a number of ways:

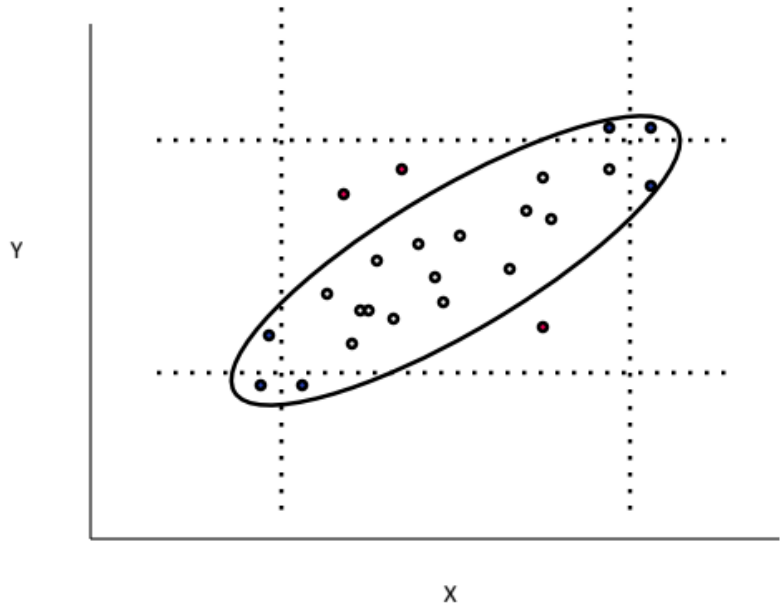
- When there's a field missing, delete the entire record. This solution is very risky as can lead to losing big quantities of data and should therefore not be used if there's an alternative.
- Fill missing fields with estimators like the average. This method is convenient and simple but it assumes that the data distribution between records is uniform and ignores possible bias which could influence the result.
- Fill missing fields with more accurate algorithms based on attribute relationships.
- Use more complex methods like Markov Chain Monte Carlo who calculate the field by analyzing pattern in other records.



Censored time intervals

This picture shows another significant cause for missing data: interval based

data ingestion. In these cases it's very likely that the values ingested at the start and end of the process will be truncated.
Another possible cause for anomalies in data are outliers, records who have values completely outside of the data distribution which need to be eliminated.
There are many techniques for detecting outliers, the simples of which is plotting with control charts.

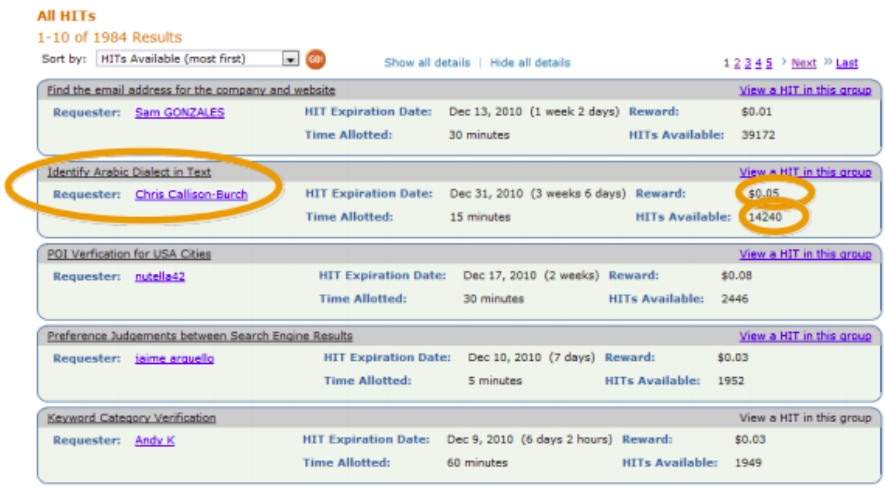


In the previous picture the points outside the black circle are outliers.
Another significant issue which needs to be taken into account is the "matching key problem", which can occur when we need to identify the same resource coming from different sources.
There are many tools which help dealing with this issues, ranging from programming libraries (e.g. Numpy, Pandas and SkLearn for Python) to full fledged applications like Trifacta Wrangler.

4.3 Crowdsourcing

4.3.1 Human Computation

The last topic of the course is a method that has gained popularity only in the recent years:crowdsourcing;
This methodology relies heavily on the concept of human computation, which means to collect or process data by assigning tasks not to computers but to humans.
This concept isn't new: many companies did it on an industrial scale before computers were invented. Nowadays this practice makes sense only for those tasks who can't be performed by computers alone (the so called AI-complete problems).
Human Computing is a form of Crowdsourcing, which is a practise in computer science where humans are directly used as a source of information.
One of the most useful applications of crowdsourcing is microtasks, systems where you can employee people to do small operations in exchange for a reward. This is done in practice through a platform, where a owner can publish the tasks he wants to be performed alongside with a pay per task, then performers can connect to this platform, choose a set of tasks to execute and get payed for it.
This differs from traditional worker employer relationships in 2 ways: the scale of the job (which is made up of little standalone tasks that can be performed even in minutes) and the conception of the workers as an automated data collection and analysis system.
Examples of microtask job offerings from the Crowdsourcing platform Amazon Mechanical Turk:



Working with flawed, biased humans instead of deterministic computers has many risks and problems of course, the biggest of which is the quality of the generated data that needs to be checked constantly.

Here are some ways to do that:

- Quality through redundancy: Combining votes (Majority votes, Quality adjusted vote, Managing dependencies...)
- Quality through gold data
- Estimating worker quality (Redundancy + Gold)
- Joint estimation of worker quality and difficulty
- Active data collection

Majority rules and worker quality estimation are necessary since a malevolent worker could try to perform the task randomly in order to do them faster and get more money. If you ask many people the same question chances are that the majority of them will give the correct answer, and this way you can keep track of users correct answer rates and assess the quality of a user.
Determine the quality of a user however isn't that easy as checking answer rates, since spammer could be randomly right more often the honest users and personal biases could also influence results.
An effective way to test workers is to send them questions to which the owner already knows the answer of, and checking if the users answer correctly.
One of the typical use cases of crowdsourcing is labeling dataset items for supervised machine learning models training.
Let's talk now about optimization in crowdsourcing, our three main goals are:

- Reduce Costs (cheap)
- Maximixe Quality (good)
- Minimize Completion Time (fast)

There's a tradeoff between Time and Quality, increasing one means decreasing the other. Surprisingly however, research shows that cost doesn't affect quality: quality stays the same no matter ho much you pay the workers.
Increasing payment actually increases completion time, because an higher reward will attract more workers (sometimes paying a low amount of money per task is even worse than not paying at all, and paying too much is counterproductive because it attracts spammers).
There are also alternatives to money when it comes to giving rewards to workers, for example you could use a leaderboard/points system as an incentive, or demonstrate that the completing the tasks has a moral or didactic value.
Last but not least, proposing fun task is another significant way of attracting workers.
Other factors that could influence participation or the results of the computation is the way the requests are formulated: since people are human and biased, the could misinterpret the question or be influenced by the way the request is written.

4.3.2 Gamification

Gamification is the process of game-thinking and game mechanics to engage users and solve problems. Gamification turns user experience into a game in order to push people to use an application.
Gamification can be implemented from an individual point of view:

- Intrigue:The enterprise needs to develop relevant content to keep users engaged.
- Challenge: Users must earn a sense of accomplishment to remain engaged, and challenges will tie back to reward and intrigue over time.
- Reward: Both non-monetary and monetary incentives, that match the level of difficulty so users gain a sense of accomplishment.

or from a social one:

- Status: Leaderboards codify status in gamification as well as a way to tier users.
- Community: Social is a key part of gamification: connect, share, and reach out.

Non monetary incentives have tree levels:

- Recognition(badges, placements, awards...)
- Access(additional features, exclusive events)
- Impact(influence product and business direction)

An example of a successful gamification campaign is the one made by Lego where users could propose ideas for new products that had a chance of actually being built and released.
Another effective way of using gamification is using GWAPS, which uses games as a way to perform useful tasks. For example, there are games where the users have to recognize the content of images or unfold 3D proteins. The answers to these problems gathered while the users where playing are then collected, stored and become a source of data for the company.