Appunti

Introduction

The course tries to give methodologies to get ready for any kind of project involving big data, in particular Apache Spark is the framework used.

Oral interview topics:

- Distribute an algorithm with PySpark
- Big data theory
- Data workflows
- Architecture Design

Main goals, we will learn:

- How to distribute computation over clusters using map reduce model
- How to write Spark code, basically use the map reduce pattern in Spark
- Understand Apache Hadoop
- Understanding what a software architecture is
- How to design batch (time based scheduling) architectures to manage data workflows
- Several design patterns for distributed environments
- The limit of traditional SQL with Big Data

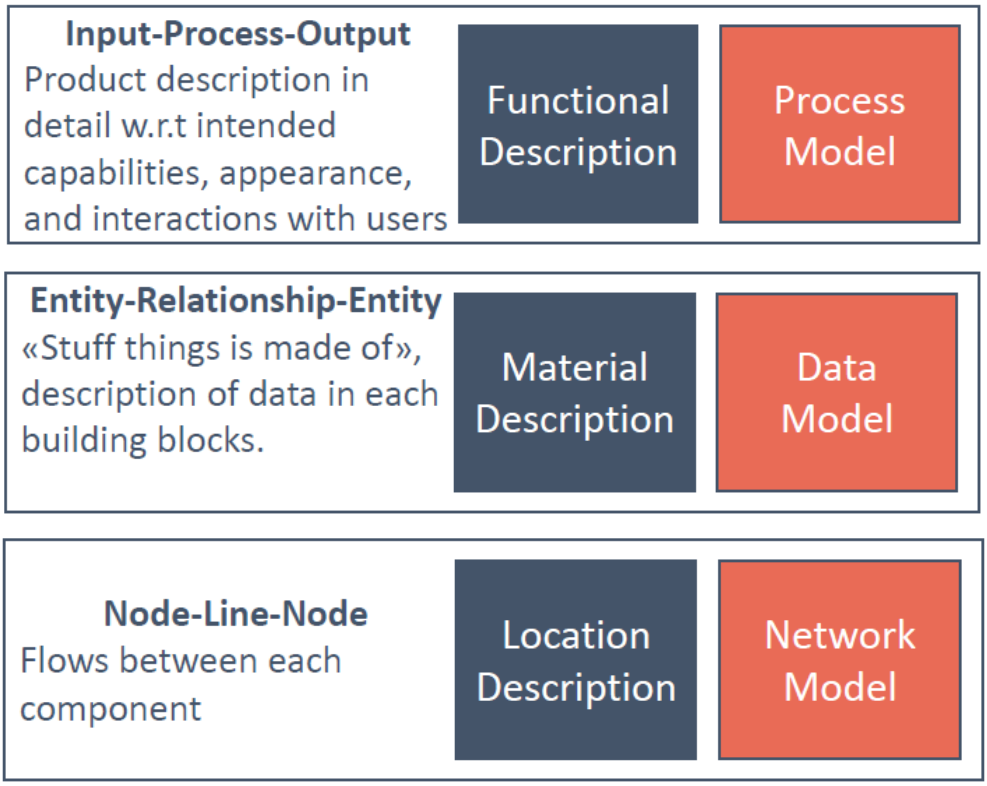
Technologies, we will use:

- Python
- Apache Spark 1.0 - Resilient Distributed Dataset
- ELK Stack: Elastic Search, Logstash, Kibana
- Docker

Architecture 101

What is an architecture? By taking principles directly from building architecture, we can list multiple aspects of software architectures:

- Multiple views: a software architecture must take into account multiple views needed by all the stakeholders which will interact with the system and contemporarily it must match all the constraints imposed by laws and reality. Examples of multiple views are:



- Architectural styles encapsulate decisions about elements and constraints on them, even relationships among elements are described this way. Besides, erosion, so the problems coming over time for a particular architecture, and drift, so the changing of aims over time, are limited.
  Examples of architectural styles are:
  - CORBA,
  - SOA → most used, Service Oriented Architecture,
  - MOM.

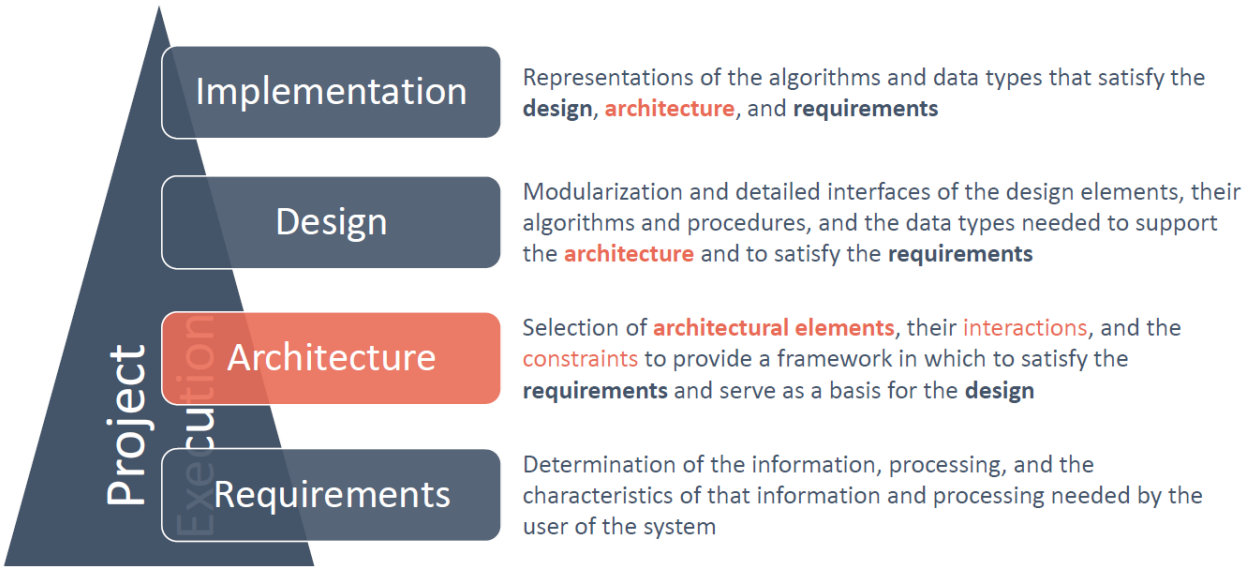
  There are different elements inside a style, we start from data elements containing information, connecting elements which connect the architectural elements and processing elements which deliver the transformation on data.
- Materials used: they actually are our subsystems (which could be composed by other subsystems), for example python to create and train a neural network and FPGA when it comes to deploy it.

In general, after distributing computation in remote locations, architecture is the key to maintain order inside a decentralized system.

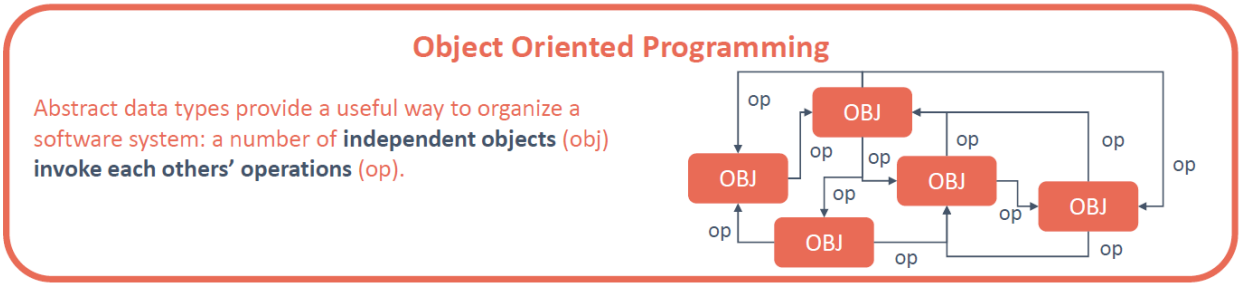
More formally, an architecture is a set of architectural elements that have a particular form, consisting of weighted properties and relationships. An underlying, but integral,

part of an architecture is the rationale (why every choice is made) for the various choices made in defining an architecture.

Project execution and design place the architecture in the second phase of the work as in the slide below:



The architecture is also seen as a method of abstraction, with patterns and conventions we can avoid error-prone manual intervention and suppress the pattern's implementation details to simplify the understanding of the result.
An example of abstraction is the OOP pattern:



This abstraction and designing process terminates when we have clearly in mind which steps have to be taken and how they are taken by the system, so that coding them is a kindly linear task, if we do not have a clear view of all the subsystems and the steps they take to accomplish a task, we cannot stop the architecturing process.

Architecture 102 / 103

state

A state in the ICT world is the set of properties which define the internal status of an element at precise point in time, there are two kind of systems:

- Stateful: which has the capability to remember events (ex. The authentication of a user)
- Stateless: which do not have this capability (ex. REST API, they answer always in the same way).

Apache Kafka

Kafka is a distributed system based on pub/sub pattern, it consists of servers and clients with a high performance TCP protocol.
It implements three capabilities to perform end-to-end event streaming:

- Publish and Subscribe stream of events (basically, reading and writing events);
- Store durably and reliably streams of events (similar to long term caching, although not reliable as a Database);
- Process events streams in real time or in a retrospective way.

The pub/sub model is based on a grouping “Key” called topic, which separates and groups events based on a particular subject.
The unit of work is the event, namely a record that something has happened, clients which write them in kafka are called producers, whereas the ones who read them are called subscribers.
Producers and subscribers are totally decoupled for performance and scalability reasons.

Another reason to add Kafka (or any other message bus) is the one that aims to render the system futureproof. If we need to add features and create new kinds of subscribers to achieve new computation goals, there is no need to perform any actions but implement the new subscribers. This lets external systems connect to your system in order to consume events as any of your personal subscribers.

ETL - Extract Transformation Load

The ETL process is a general procedure of copying data from one or more sources, cleaning them and loading the cleaned ones on a destination system; the extraction happens from heterogeneous sources.

It is used to increase data quality and consistency, normalize a dataset or prepare them for particular tasks.

Object Storage Model

It is a storage architecture which manages data as objects, each of them contains data, contextual information (metadata) and technical information (for example an header). It is also possible to distribute objects and serialize/deserialize them to ease communications.

Data Lake

The data lake is a model which aims to solve a big problem about data storage: once you decide to discard a piece of data as it is useless, it cannot be re-acquired later if it becomes useful.
The data lake is based on the object storage model as wrapping data as is and saving it in the data lake is the simplest way to avoid the problem above and to have all data coming also from legacy systems.
The four pillars of the data lake are:

- The saving of unprocessed data (in form of serialized objects);
- The saving of data forever;
- Good read/write performances;
- The availability of the schema during read operations.

Lock-in

The lock-in concept is based on the fact that a vendor makes a customer dependent on him for products and services, the architect should aim to avoid a lock-in.

Software architecture pillars

1. Assure the satisfaction of the requirements: there are various types of requirements as functional one (features of the application), technical requirements (ex. Can I process all in time?), security requirements (ex. It is compliant with GDPR?);
2. Support the technical design of the system: which means all things that help the development (datatype choice, interfaces. etc);
3. Support the establishment of a cost estimation and process management (cost of each component, cost of technology used, etc);
4. Enabling component reuse for a higher reliability;

5. Centralize operations and processes to avoid the need of modification in various parts of the system and maintain a tidy system (ex. Data store in the same place);
6. Enhance productivity and security;
7. Enabling integration with other enterprise systems;
8. Enabling scalability in a tidy way;
9. Enabling a finer control over the processes execution;
10. Avoiding hand-over and people lock-in.

Design patterns

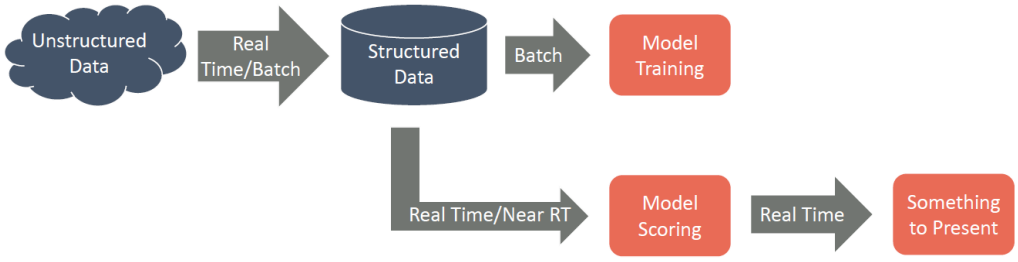
Design patterns, formally, are best practices to solve common problems.
Examples are:

- Singleton,
- MVC,
- SOA,
- Etc..

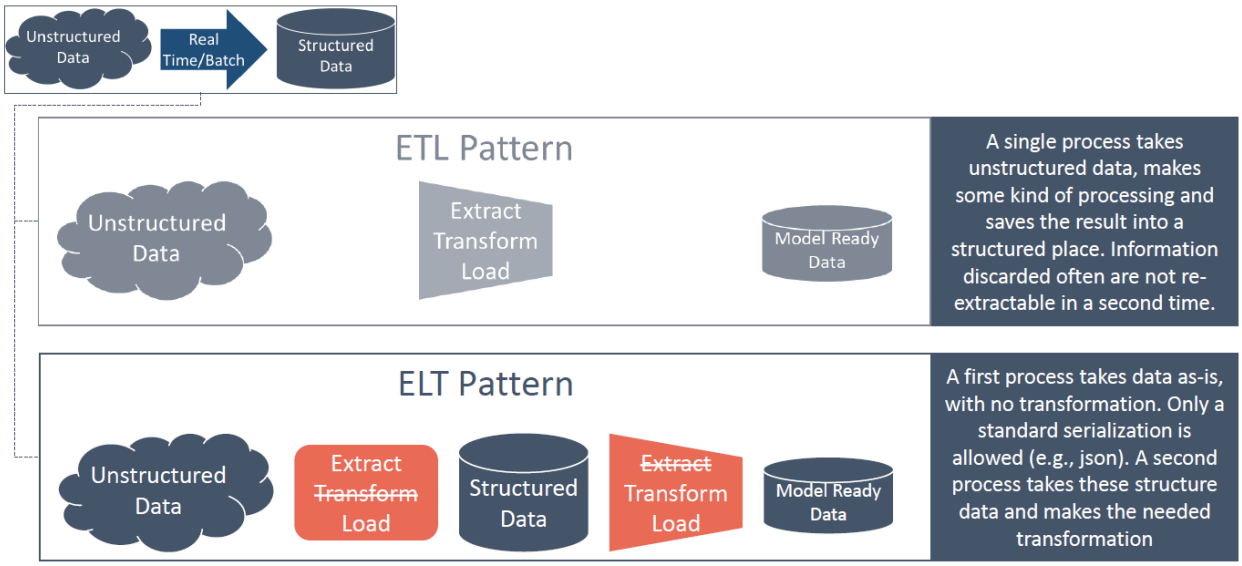
Three big families are found:

- Creational: Creates an object rather than instantiating it,
- Structural: using inheritance new objects are composed,
- Behavioral: define how objects communicate.

Regarding these, a classic big data challenge is to have unstructured data to be transformed into something presentable.



In the subsequent slide, there are pros and cons of 2 different solution:



Why is the second solution for big data better than the first? We avoid losing some kind of information which could not be re-extracted later, which is the key point of a data-lake. ALL DATA MUST BE SAVED BEFORE PROCESSING.

Change Data Capture (CDC)

The CDC is a set of design patterns used to determine and track changes in data so that action can be taken using the changed data, although this problem is mitigated in a pub/sub architecture, for normal SQL DBs based systems this is a big issue.

Traditional CDC patterns are:

- Invasive Database-side:
  - Timestamps on rows
  - Version Numbers on rows
  - Status on rows (ex. MODIFIED, CLEAN, etc)
  - Triggers on tables
- Invasive Application-side:
  - Event programming
- Invasive Database CPU-side:
  - Transaction log scanners, basically the DBs logs are scanned regularly to see which data has changed, sometimes logs are not standard and not open, so it could be a hard task.
  - Log shipping, basically it involves building a copy of the DB to perform the transaction log scan, it is also used as a backup method of DBs.

The CDC patterns explained just now, although functional, are not good solutions, this is due to some proprietary technologies like SAP, which does not give the possibility to actuate them. Besides, users can behave in unexpected ways and each pattern is system dependent, so no generalization is possible.
Last but not least, invasive database side CDCs could destroy the application performances.

We could have several different CDC patterns in a single table, whereas not useful, it could happen.

The Diff&Where method

A useful CDC pattern is the **Diff&Where**, it assumes that data exists only in 2 forms:

- Log data: any kind of data where there is the idea of chrono-sequence, insert operations are permitted,
- Registry data: any other kind of data, every kind of operation is permitted.

Log data

- Timestamp on rows → granted by design, a column with a timestamp is added
- Version number of rows → Not used
- Status indicator → a simple boolean not used in practice
- Triggers on table → not desiderable for performance issues

Registry data

- Timestamp on rows → Critical problems of slowing down selects and other actions types, besides it is not safe (the timestamp column could be forgotten and not updated) and impossible in case the developer is not a DBA.
- Version numbers on rows → has the same drawbacks of the timestamp on rows method due to an inner query to retrieve the maximum value of the version number
- Status indicator → Not desiderable due to the same drawbacks as before, besides if the process fails in the middle we are not in a reliable state
- Triggers on table → not desiderable for performance issues

This pattern is used also to solve the problem of incremental extraction, it is basically based on two strategies:

- Where-like: which deals with log-like sources;
- Diff-like: which deals with registry-like sources.

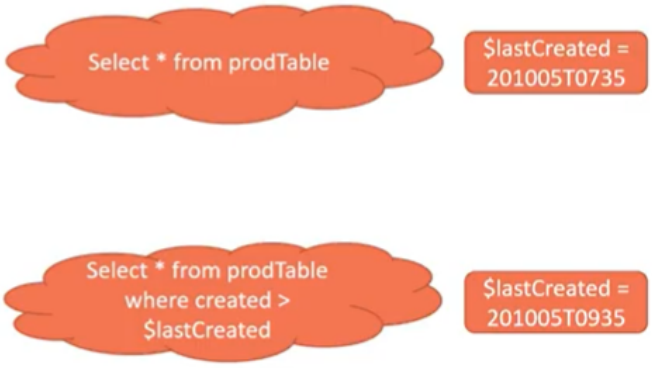
Where like

t=t0

NIP	product ID	#errors	created	cycleT
A01	123	0	201005T0630	126
0B2	123	2	201005T0735	128
...	...	...	...	...

t=t1

NIP	product ID	#errors	created	cycleT
0C1	123	4	201005T0830	124
11M	123	1	201005T0935	126
...	...	...	...	...

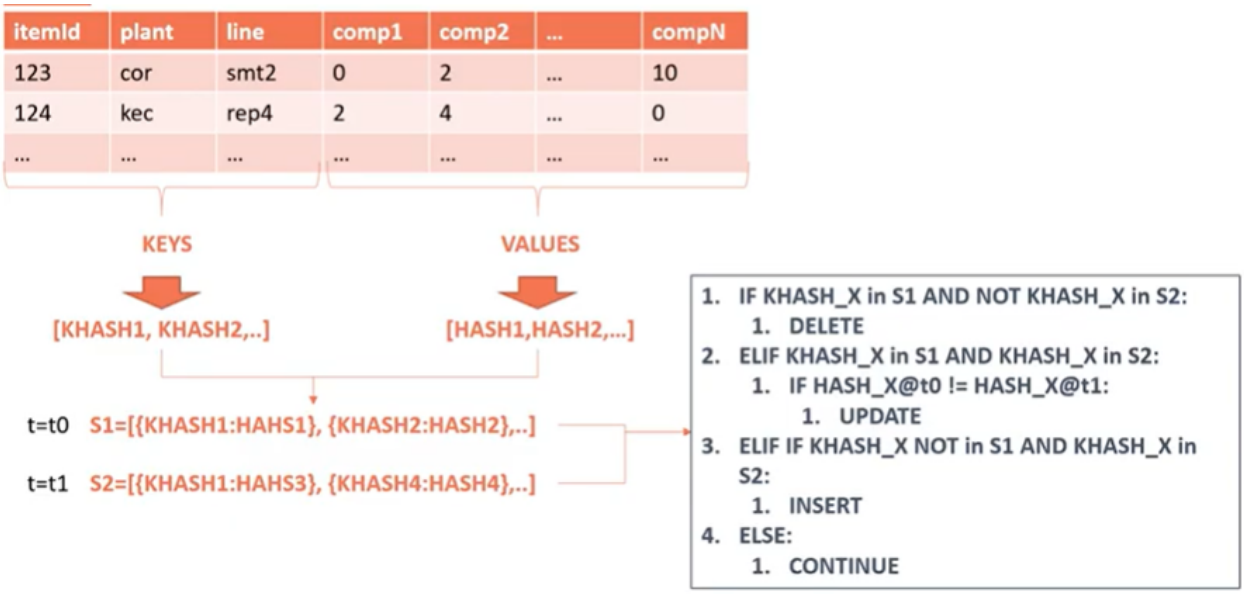


We basically save the last saving moment and take everytime only the objects saved after the last saving time, except for the first time, when all the data is taken.

Diff like

We aim to extract the “keys” of data and “values” of data and then apply a hash function, we will have tuples of hashed “key - value”.
At a future moment t1, we then have 4 possibilities:

- The tuple is no more present → the record got deleted
- The hashes of the keys are equal, whereas the values hashes are not → the record got modified
- The key hash in t1 was not present in t0 → the record got inserted
- The keys and values hashes are equal → no action performed on the record.



An example is:

t=t0

itemId	plant	line	res	cap
123	cor	smt2	0	2
124	kec	rep4	2	4
124	cor	smt3	2	1
126	kec	rep4	1	1
129	kec	rep5	5	5

t=t1

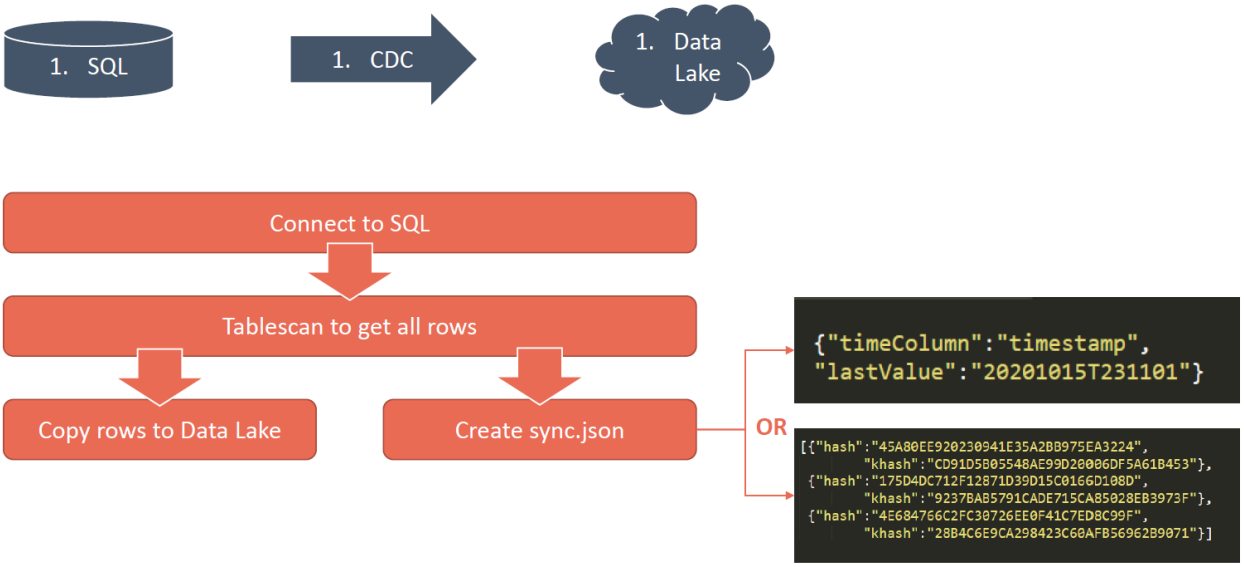
itemId	plant	line	res	cap
123	cor	smt2	2	2
124	cor	smt3	2	1
126	kec	rep4	1	1
129	kec	rep5	5	5
129	cor	rep1	4	4

?

1. KHASH_0(123,cor,smt2) == KHASH_1(123,cor,smt2) AND HASH(0,2) != HASH(2,2)
    1. UPDATE (123,cor,smt2)
2. KHASH_1 (124,kec,rep4) NOT EXISTS
    1. DELETE (124,kec,rep4)
3. KHASH_0 (129,cor,rep1) NOT EXISTS
    1. INSERT (129,cor,rep1)

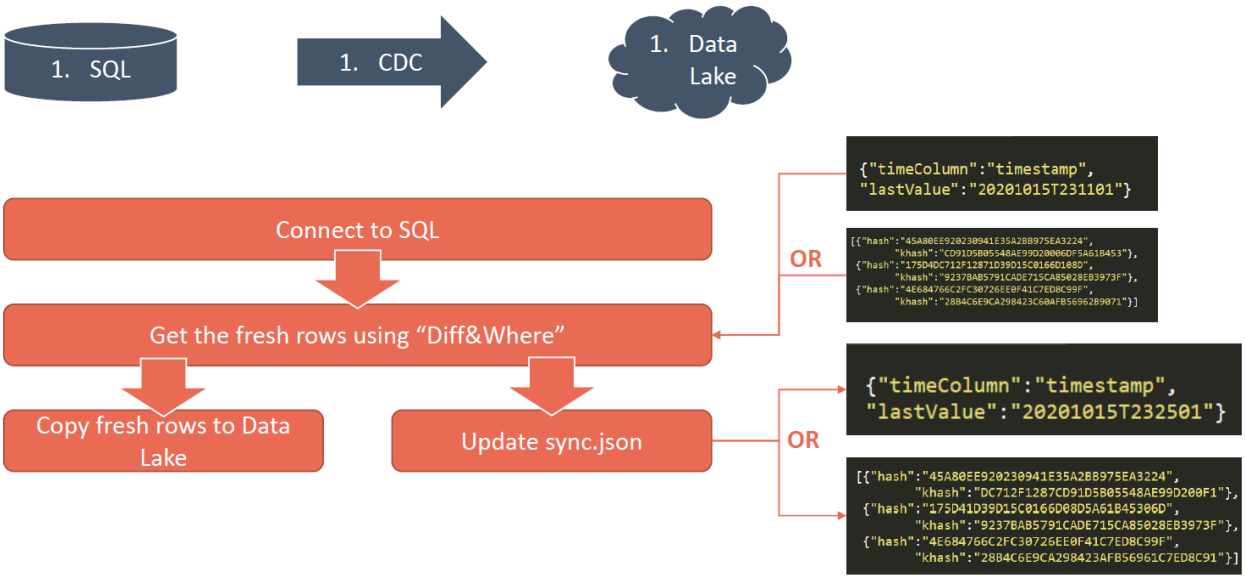
The main point of this method is that it is data agnostic, so it could be applied to any kind of data and any kind of system.

Bulk in phase



This is the structure of the first time the method is employed, the whole dataset is taken and a starting point with information useful for next employment is written (sync.json).

Incremental phase

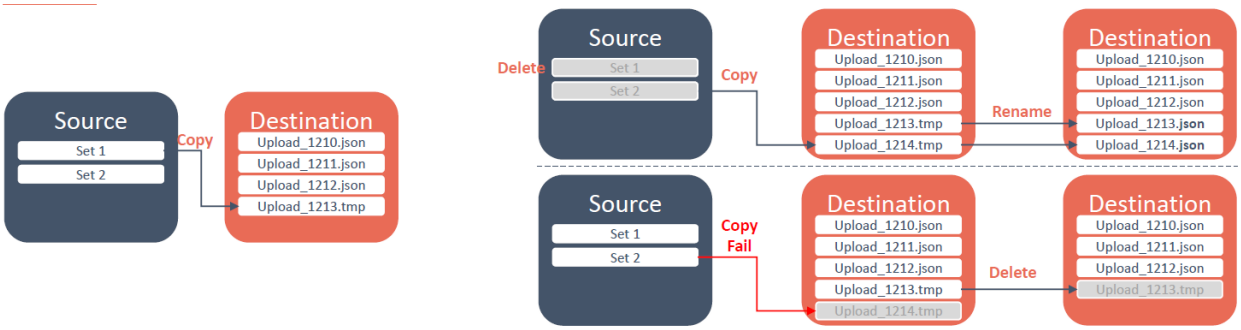


From the second use onward, we will use an incremental manner to check for data changes as in the chart above, the sync.json is used to get these changes and is updated for the next iteration.

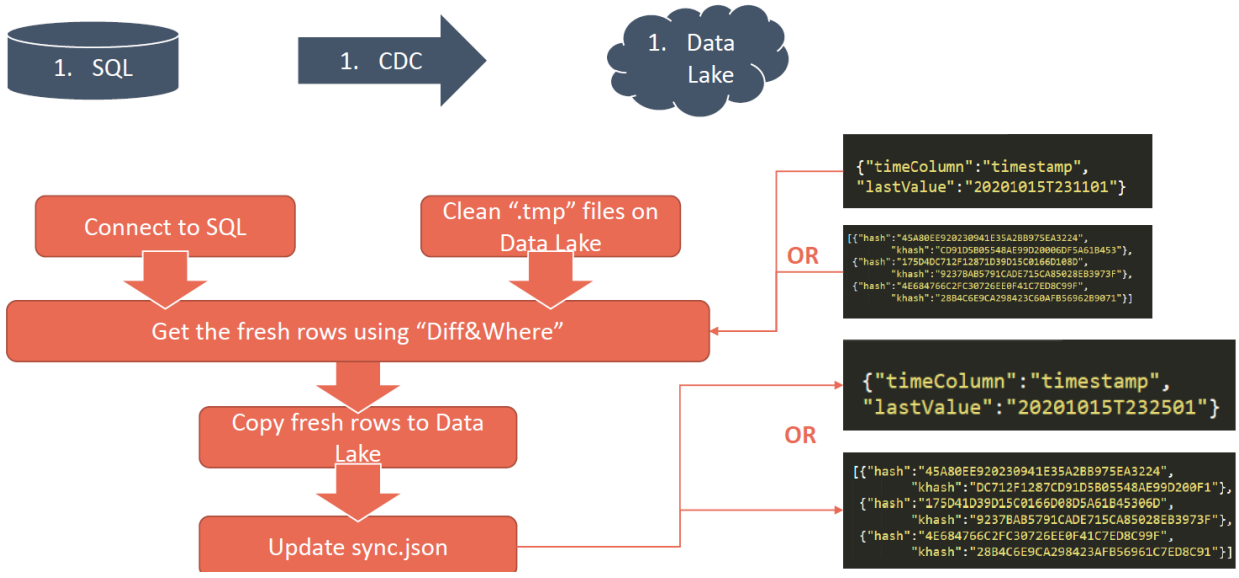
What if the incremental step fails?
We are in a state where some rows have already been written and some not inside the sync.json, so not in a reliable state.

As the CDC is stateless (the sync.json is outside the CDC itself) and should remain as is, we can add a transactional aspect to the CDC:

- We use a different naming convention using temporary files to set a difference between already present files and the one in creation, so we can roll back the process and restart in case of failure while maintaining a reliable state of the system. Infact, if a fail occurs, we just delete the temporary files to actuate the rollback itself.



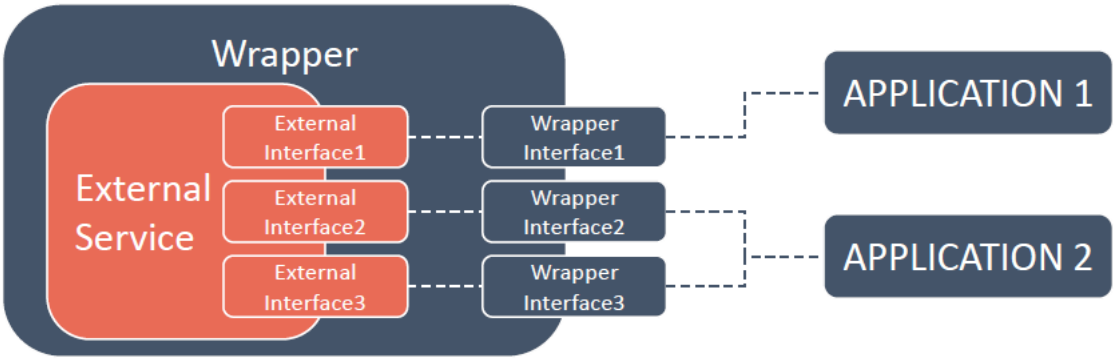
We use a renaming operation for the fact that it is a more or less consistent operation in every kind of system, so there should not be problems in renaming a file.
Applied to the CDC, the method becomes:



The first operation of the CDC procedure will be "delete tmp files" from now on, in order to apply the "transactionality" we need to make the procedure consistent.

Wrapper pattern

But what if the data lake type changes abruptly?
The adapter pattern (wrapper) comes handy:



The adapter pattern consist on wrapping external services inside a “wrapper” exposing some interfaces which will be used by our application to communicate with the external services inside that particular wrapper:

- In case of changing only the adapter should be manipulated and not the whole application,
- This gives a higher reliability to the system,
- This gives a higher security level due to the fact that applications cannot communicate directly with external services,
- This reduce the possibility of lock-in,
- This facilitates the tracing strategies for errors and bugs.

This pattern also can hide some methods from external services in order to not let applications use them.

There is a side effect with this CDC procedure: registry records (master data table) are transformed into log type records, as for a change we do not make an “update” operation in the data lake, we create a new record for each change. The same stands for all the CRUD verbs, in making a join this could be a disadvantage, whereas it is not in case we have a join over time (ex. Compute the industrial cost of a product over 3 years).

Join over time

For the example subsequent tables will be used:

Tables

▸ (26) Spark Jobs

STATUSNAME

	HammerGW.ts_load	StatusID	StatusName	StatusType
1	2013-01-14T22:58:31.579Z	14	2013_14-MancanzaMaterDaFornit	1
2	2014-01-14T22:58:31.579Z	14	2014_14-MancanzaMaterDaFornit	1
3	2015-01-14T22:58:31.579Z	14	2015_14-MancanzaMaterDaFornit	1
4	2016-01-14T22:58:31.579Z	14	2016_14-MancanzaMaterDaFornit	1
5	2017-01-14T22:58:31.579Z	14	2017_14-MancanzaMaterDaFornit	1
6	2013-01-14T22:58:32.032Z	21	2013_21-Guasto Etic/LaserMark	1
7	2014-01-14T22:58:32.032Z	21	2014_21-Guasto Etic/LaserMark	1
8	2015-01-14T22:58:32.032Z	21	2015_21-Guasto Etic/LaserMark	1

Showing all 590 rows.

▸ (20) Spark Jobs

STATUS

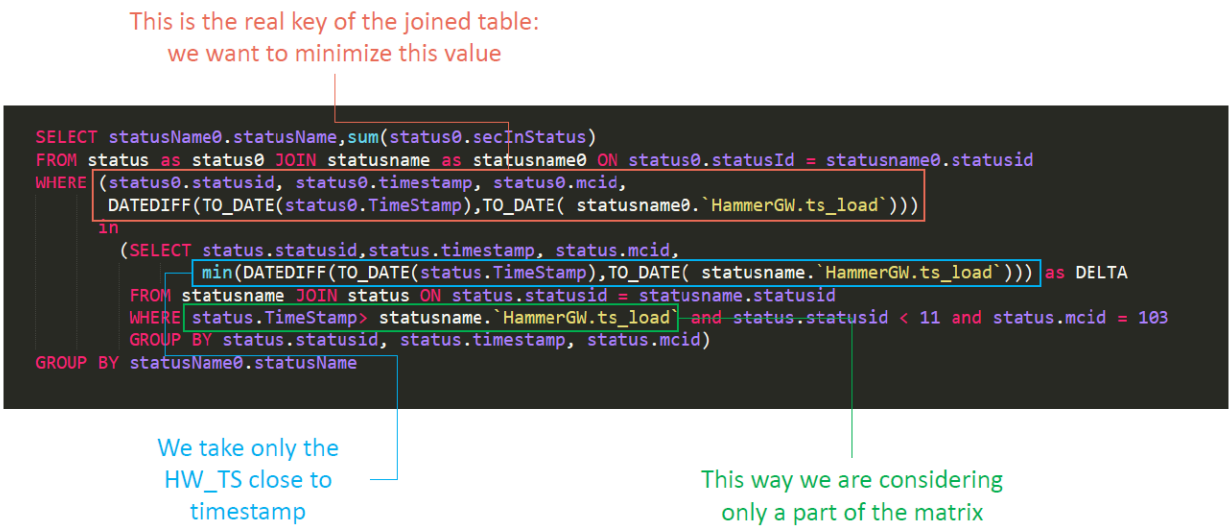
	McID	SecInStatus	StatusID	TimeStamp
1	242	4	7	2015-11-06T11:40:22.520Z
2	123	333	1	2015-11-06T11:40:24.640Z
3	123	0	0	2015-11-06T11:40:24.647Z
4	241	236	1	2015-11-06T11:40:25.040Z
5	241	0	0	2015-11-06T11:40:25.047Z
6	241	20	1	2015-11-06T11:40:27.077Z
7	241	2	10	2015-11-06T11:40:27.267Z
8	241	18	7	2015-11-06T11:40:29.073Z

Showing all 100 rows.

A bit of explanation:

- STATUSNAME is a master data table
    - HammerGW.ts_load : is the moment in which the row has been updated
    - StatusName : changes over time
    - StatusID : changes over time along with the StatusName
- STATUS is a log table
    - StatusID : object connecting the two tables
    - SecInStatus : seconds in that particular status
    - McID : is the machine ID which at TimeStamp time stayed in particular status and for how many seconds

An example of a query regarding the join over time pattern is illustrated below



In this query we want to find how many seconds every machine has been in each status.

Now the question becomes: how to scale this to billions of rows?
The Map-reduce model comes handy.

Data Integration patterns

Apache Spark

It evolves the Hadoop architecture to enhance performances.

Databricks

Databricks is a company which adds enterprise-grade functionality to innovations such as Spark, it is actually a fully managed cloud service.

Functional Programming

It is a programming paradigm where software is constructed by applying and composing functions; they are trees of expressions that each return a value. Rather than a sequence of imperative statements which change the state of the program.

Jupyter notebook

It is an open source web application that allows to create and share documents that contain live code, equations, visualizations and narrative text. It is used for data cleaning, simulation, data visualization, machine learning, etc.

Map-reduce

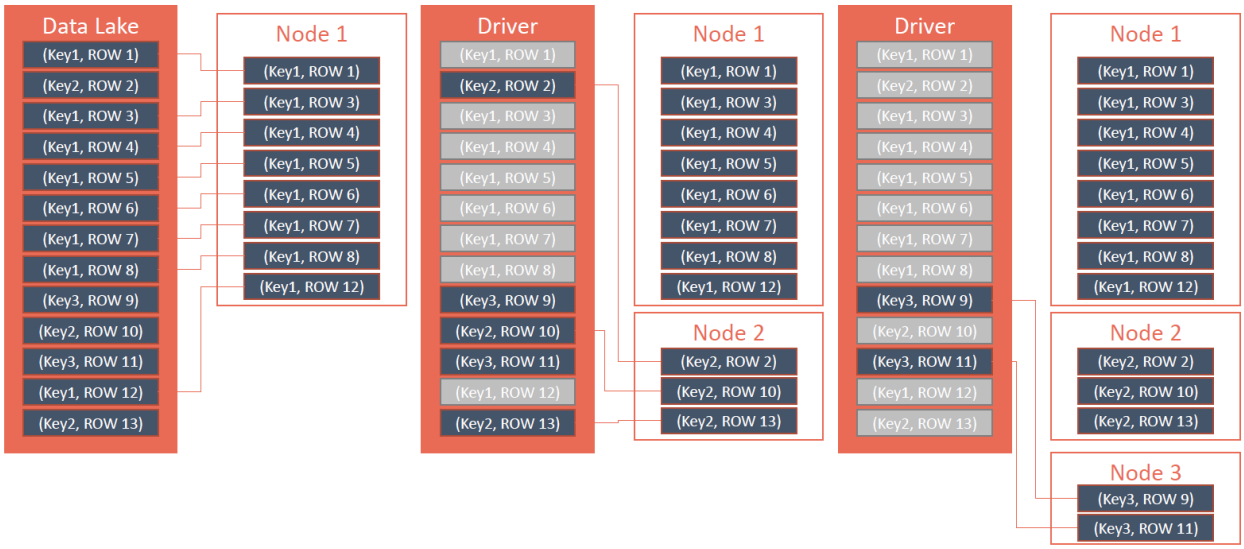
The map reduce pattern is in charge of this challenge, it is a programming model proposed in Google to ease multi-node process parallelization.
Hadoop is a framework that implements this model and takes care of the scheduling of tasks, it monitors them and re-launch in case of failure.

Basically, to ease multi-node process parallelization:

- Users specify a Map function that processes a key/value pair to generate a set of intermediate values
- Users specify a reduce function that merge all intermediate values associated with the same intermediate key
- Inputs and operations over inputs are processed in parallel by different machines using a partitioning function (ex. hash(key) mod R)

The whole structure stands on the functional programming paradigm, each of the functions is executed on a component of a large cluster facility.

We can see a general example below:



Where keyX is a value, rowX could be anything.
How it works:

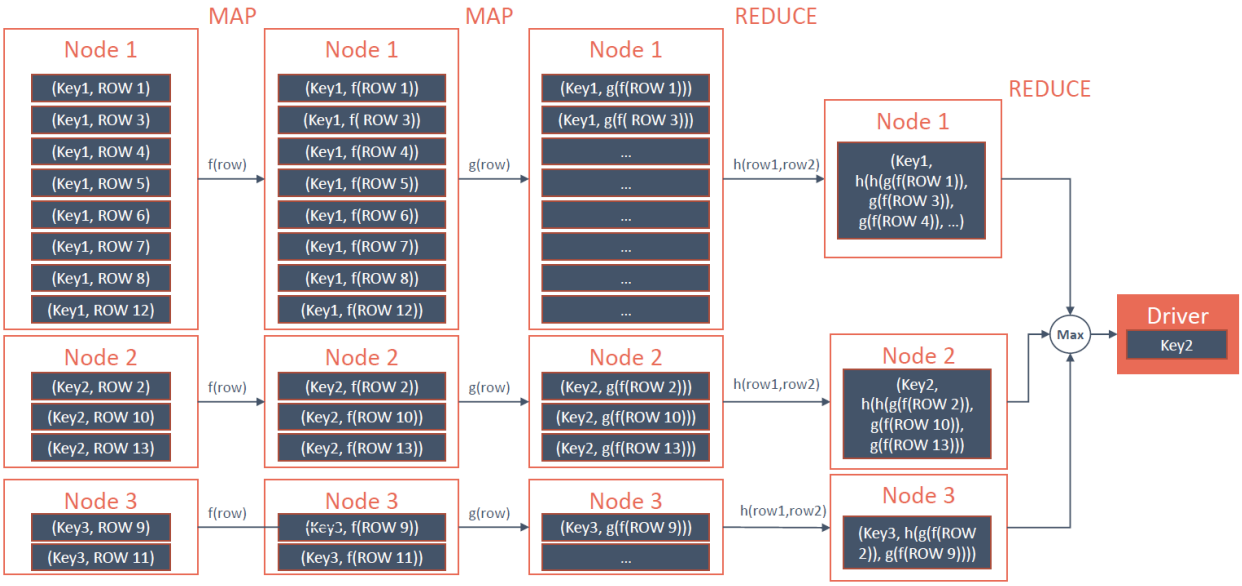
- We start to split the data set among the nodes at disposal, from the Data Lake we send to Node 1 all tuples with the Key1

- Then we send the tuples with the key2 to the Node2
- The same stands for the rows with key3 and the Node3

Basically we are mapping to which node every row will be processed:

- Each node will have a single or a subset of the keys
- We are sure that each key is present in only one node.

Obviously the communication among nodes should be reduced to the necessary and not more.



We then start to apply a Map function to each row as in the figure (f(row) in the figure), Map functions could be applied again if it is required (g(row) in the figure).
We then apply a reduce function:

- It takes a couple of rows with the same key and applies the reduce function (ex. An hash function) until couple after couple there are no more lines with the same key
- After all we can compute a value (ex. The max) and then return it

A practical example could be:

- We divide the dataset containing information about plants using the name of the plant as the key, value of rows will be the number of pieces produced in that implant
- We map the rows to be multiplied by a constant for each implant
- We reduce to have the sum of the pieces produced in each implant in total
- We reduce to have the sum of pieces produced in all implants.

Job Scheduler

Job

A job is **an atomic piece of work**. Each job is composed by a set of task (or steps) and could be identified with a **single process**.
Jobs can be **started interactively**, such as from a command line, or **scheduled** for non-interactive execution by a job scheduler, and then controlled via automatic or manual job control.
Jobs are described with a **name**, once started have an **unique id**, which can be used to get the **status**.
Jobs could be

- **Finite**: they can complete, fail, or be terminated
- **Online**: they can only be stopped when terminated

Job Scheduler

A job scheduler is an application **for controlling background program execution** of jobs: it manages the execution in a **non-interactive** way.
If it works using a temporal-based scheduling, is commonly called **batch scheduling**. Another common strategy is to schedule a job after a given event occurs: this is called **event-oriented** scheduler.
From an **Operating System** perspective, the *"Job Scheduler gathers status of **processors**, idle or busy, and dispatches a job in the shared job queue to idle processors."*.
From a **Distributed System** perspective, it gathers status of a given remote resource or **application server** to decide where and when submit the job

Job Queue

The data structure of jobs to run is known as the **job queue**.
Different strategies exist to **consume** the job queue:

- Job priority
- Compute resource availability
- Estimated execution time
- Elapsed execution time
- Occurrence of prescribed events
- Job dependency

Job Impersonation

Impersonation* is the ability of a **thread** to execute a job using different security information than the process that owns the thread.
Typically, a thread **in a server application impersonates a client**.
When the thread is impersonating the client, any operations performed by the thread are performed by using the client's (impersonating user's) credentials.
This allows the server thread to **act on behalf of that client** to access objects on the server or validate access to the client's own objects.

Job Concurrency

Jobs could be executed:

- **Sequentialy:** one completing before the next starts
- **Concurrently:** during overlapping time periods

The jobs concurrency relies on the hosting application server and the technology used.

If the Job Scheduler is able to submit jobs to other machines (**Broker Pattern**), even more advanced strategies could be put in place.

Finally Big Data

Hadoop

It is a framework of mostly open source components built to facilitate the development of multi-node services, the core idea is that hardware can fail, this brings hadoop to enhance the reliability of the cluster.

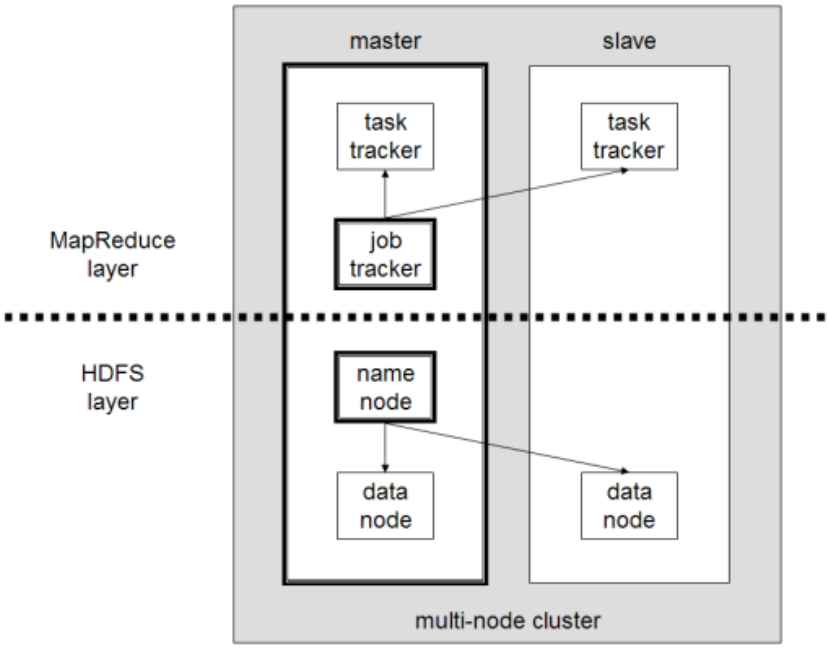
Architecture

There are three main components in the hadoop architecture:

- Storage: Hadoop distributed file system Storage(HDFS);
- Resource Management: Hadoop YARN;
- Parallel computation: Hadoop MapReduce, which is the implementation of MapReduce pattern.

Any of the Hadoop components should be location aware: it should know where it is and be able to share this position with the system.

Hadoop is then made by various components visible in the following picture



Job Tracker

It is a service that decides where a given task should be executed, ideally this place should be the node where the data are or a close one, this is the key point: reducing the need to take data for computation from other machines.

Task Tracker

A node in the cluster that accepts tasks from a job tracker is called Task Tracker, it exposes a finite number of slots, the sum of these slots represents the capability of a node to run multiple tasks in parallel.
Each task is executed on spawned JVM, this process also sends a “heartbeat” to the job tracker each minute to assure it is still alive.
Hadoop and Spark are so based on java due to the fact that spawning JVMs is the easiest way to make a multithreaded computation.

NameNode

This component keeps the directory tree of all files in the system (namely a list), besides it tracks where the files are kept across the whole cluster. It does not store the data of these files itself.
It is actually the single point of failure of an Hadoop application and it routes requests between the jobs and DataNode, this is due to the fact that if the NameNode is lost, the job tracker does not know where to send the request.

Backup NameNode

It is a component which is still under development and aims to give Hadoop a higher reliability.

DataNode

It is the component which stores the data, they work with each other to replicate data. DataNode with data to process should be deployed on the same machine where TaskTracker is running.

Master Node

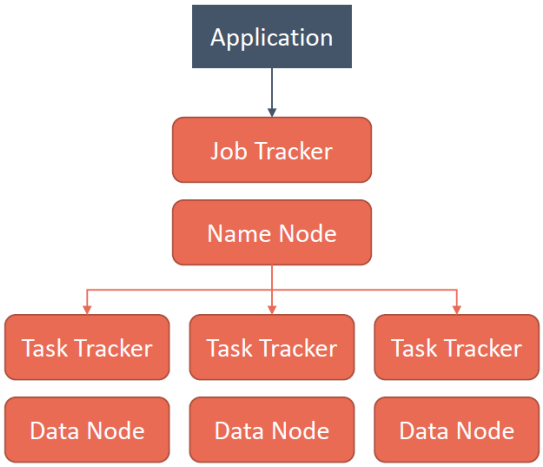
This node is composed of Job Tracker, Task Tracker, NameNode, DataNode.

Slave Node

A slave is only composed of DataNode and TaskTracker.

Sequence Activity

Let’s now see what happens in the case of an application submitted to a Hadoop Cluster.



1. Application **submits an asyncronous job** to the Job Tracker, then it can start **polling** the Job Tracker for job status
2. Job Tracker gets **data locations** in the Name Node registry
3. Job Tracker identifies the set of Task Trackers with **available slots** nearest to Data Nodes
4. The JobTracker **submits the work** to the chosen Task Trackes
5. Job Tracker starts to monitor **Task Trackers heartbeat**: If they do not submit heartbeat signals often enough, it **presumes they failed** and work is scheduled on a different Task Tracker.
6. When each Task Tracker completes the task, it notify the Job Tracker the **final status**. If it failed, the Job Tracker can:
    1. **resubmit** the job elsewhere
    2. mark that specific record as **to-skip**
    3. **blacklist** the Task Tracker as unreliable.

Hadoop is based on the idea that moving computation is cheaper than moving data, obviously if a computation runs near the data it needs, it is much less costly, HDFS provides interfaces to achieve this situation.
The pros are less network congestion and a higher throughput of the system.

Hadoop Distributed File System

HDFS is a distributed file system which runs on commodity hardware which may be low-cost and is designed to be highly fault tolerant.
Besides, it provides a high throughput to access application data.

Hadoop YARN

YARN is the component based on the idea to split up the functionalities of resource management and job/scheduling into separate daemons and it is composed by:

- Global Resource Manager (RM), which is the ultimate authority that arbitrates resources among all the applications in the system,
- NodeManager (per machine) which is in charge of monitoring the resource usage and reporting it to the RM/Scheduler;
- Application Master (AM, per machine) which negotiates resources from the RM and works with the NodeManagers to execute and monitor the tasks.

Hadoop Map Reduce

Is the framework which implements the MapReduce model taking care of scheduling tasks, it also has the task of monitoring them and re-executes the ones that fail.
It consists of:

- A single master ResourceManager;
- One worker NodeManager per cluster-node;
- One AppMaster per application.

The Hadoop job client submits the job and configuration to the ResourceManager which then assumes the responsibility of distributing it to the workers.

Apache Sqoop

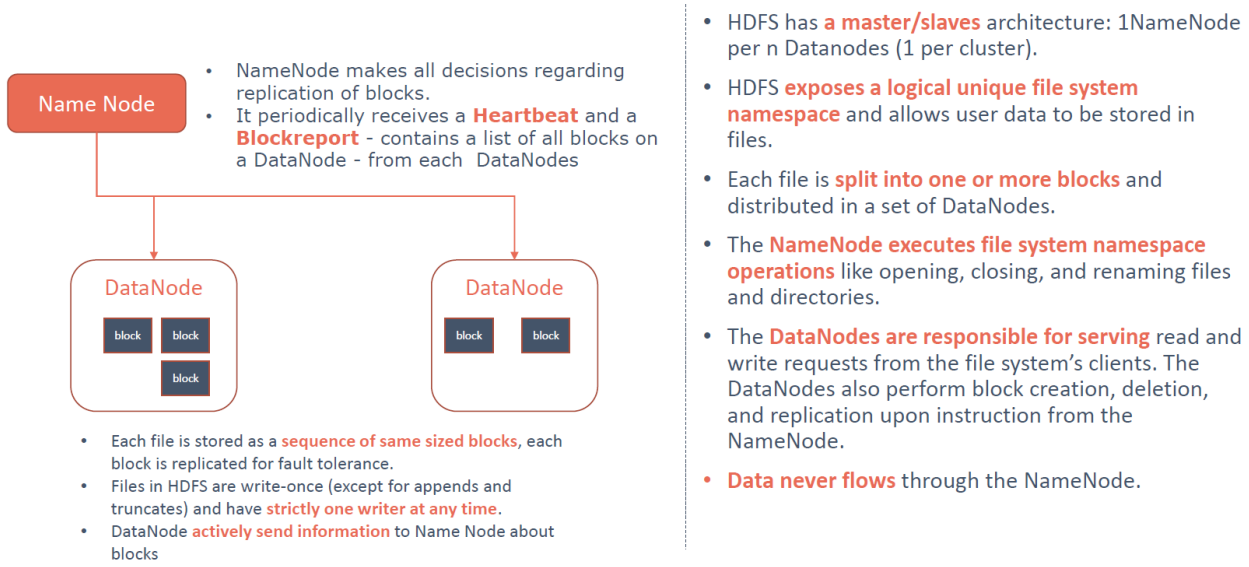
Tool built on top of Hadoop, it is CLI for transferring data between relational DBs and Hadoop HDFS, it supports both free SQL queries or incremental loads of a single table, they are also runnable multiple times to import updates to the DB.

Apache Hive

It is a data warehouse software built on top of Hadoop to provide data querying and analysis, it provides a SQL-like interface and a specific dialect (HiveQL) that is translated in Hadoop MapReduce jobs to query data.

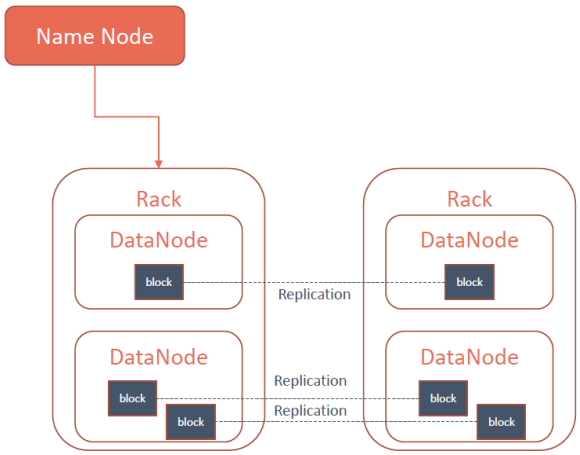
Data-Lake over HDFS

HDFS is the most common file system used to create a data-lake, we can see its architecture in the following slide:



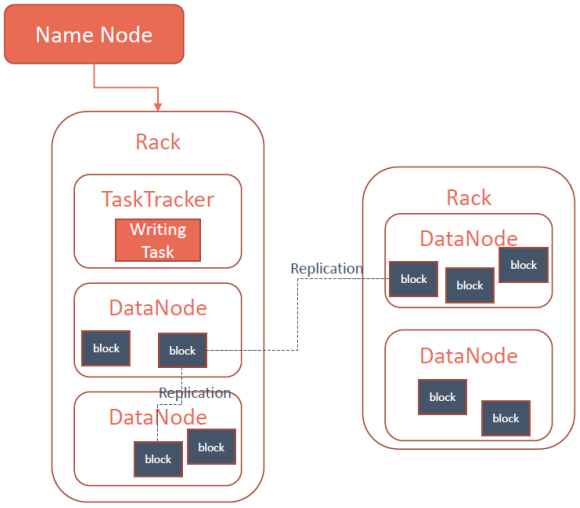
A keypoint of this structure is the dataNode actively sending information towards nameNode and not viceversa, this is made to achieve a scalability standard in which if I add a new DataNode there is no other actions to take as the NameNode will discover the new DataNode on its own when receiving for the first time some information from it.

HDFS could assure a certain reliability QoS thanks to some replication policies it has:



- The NameNode determines the **rack id** each DataNode
- Placement of replicas are **rack-aware** to:
  - optimize **network bandwidth**
  - increase **fault tollerance**
  - increase **I/O performance**

Policy 1 – **Replicate the entire rack**
Placement policy is to replicate the entire rack
- prevents losing data if an entire rack fails
- allows **routing read requests** over different racks optimizing network usage
- Make **easier re-load-balancing** on component failure.
- increases the **cost of writes** because a write needs to transfer blocks to multiple racks.



Policy 2 – **Two rack factor 3 replication Placement policy** is to put one replica on a DataNode where the writer task is, another replica on a node in a different (remote) rack, and the last on a different node in the same remote rack.
- **Improved write performance**: cuts the inter-rack write traffic
- **Sub-optimal reading**: a block is placed in only two unique racks rather than three.
- **Sub-optimal file distribution**:
  - one third of replicas are on one node,
  - two thirds of replicas are on one rack,
  - and the other third are evenly distributed across the remaining racks

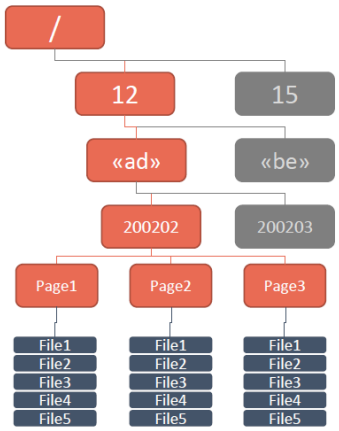
HDFS Reliability

DataNodes inside HDFS are considered reliable: they continuously send a heartbeat to the NameNode in order to say “I am alive”, they are considered dead only if the heartbeat is not sent anymore and a timeout occurs (about ten minutes).
In case of death, the blocks of a DataNode are copied onto another one, this leads to the necessity of keeping track of which blocks need to be replicated by the NameNode, which also initiates the replication whenever is necessary.

The same reliability is given to the data coming from a DataNode, in case a block of data arrives corrupted, HDFS employs checksum to assure the corruption.

Less reliable are the NameNodes, they could be made more reliable by using backup copies of metadata and registry content. This is due to the fact that in case of NameNode loss, all the content is lost as well.
A method to assure a high reliability to NameNode is the **Quorum Journal Manager** (QJM): from multiple NameNodes in different machines, only one is in an Active state, the other are in a Standby state, the active one is the only one who writes its log using daemons called JournalNodes (JNs). Regarding the standby nodes, they are able to read from the logs taken by JNs and are constantly watching for edits.
To make this more consistent, DataNodes are configured to know the existence and send the heartbeat to all NameNodes, either the active one or not.
JNs allow only one NameNode to be a writer to avoid the “split-brain scenario”, in case of failure the writer NameNode is simply substituted with the one which will become active.

DataLake tidiness



B-Tree+ could be used also when you design Data Lake objects structure

1. Try to keep the **same number of levels** between root and data folders
2. **Enrich each** line with the information about the path
   {"keys":{"lvl1":12,"lvl2":"ad","lvl3":"200202"}}
3. **Keep the balance** adding at the last level before leaves a new page if needed
4. Each file should have **same number of lines/object**
5. Each file should have a **maximum size** that depends on who is going to read them
6. At each level, you could add a **info.json** file to describe some statistics about the rest of the path
7. Even if you will always have a 1 page last layer, **always add that layer**: you will have a rule that does not depend on key values to end path recursion
8. [Optional] Use the **file name** as the most fine key to read only needed files as a smart **CDC approach**

Introduction to Apache Spark

Singleton

Singleton is a pattern of software design that restricts the instantiation of a class to one single instance; this is useful when exactly one object is needed and to coordinate actions across the system.
It is also referred to as the “get or create” pattern.

Apache Spark

Resilient Distributed Dataset (RDD)

RDD is a collection of arbitrary elements partitioned across the nodes with a “hashPartitioner” function, this leads to a lack of constraints regarding the structure inside the RDD → liberty is left to the contents inside them.

RDDs are mainly created by reading objects from HDFS file system using the primitive called .textFile(). It could be also created by parallelizing objects in memory with the .parallelize() function.

The default HashPartitioner function is applied to:

- A unique id built from files read, in case of HDFS reading,
- The row value casted to string after the transformation if a re-shuffle (the action to redistribute the element of an RDD across the nodes) is needed.

Users can set the number of partitions (with the primitive .repartition(n)) to tune the parallelization.

Paired RDDs

A variant RDD made by a two element tuple is called pairedRDD: in this case the hash function is applied to the first element of the tuple after any shuffle operation.
In particular it is a (key, value) version of RDD where:

- Key can be everything that could be serialized by your programming language;
- The same stands for the value.

The HashPartition function is applied to the value of “key”.

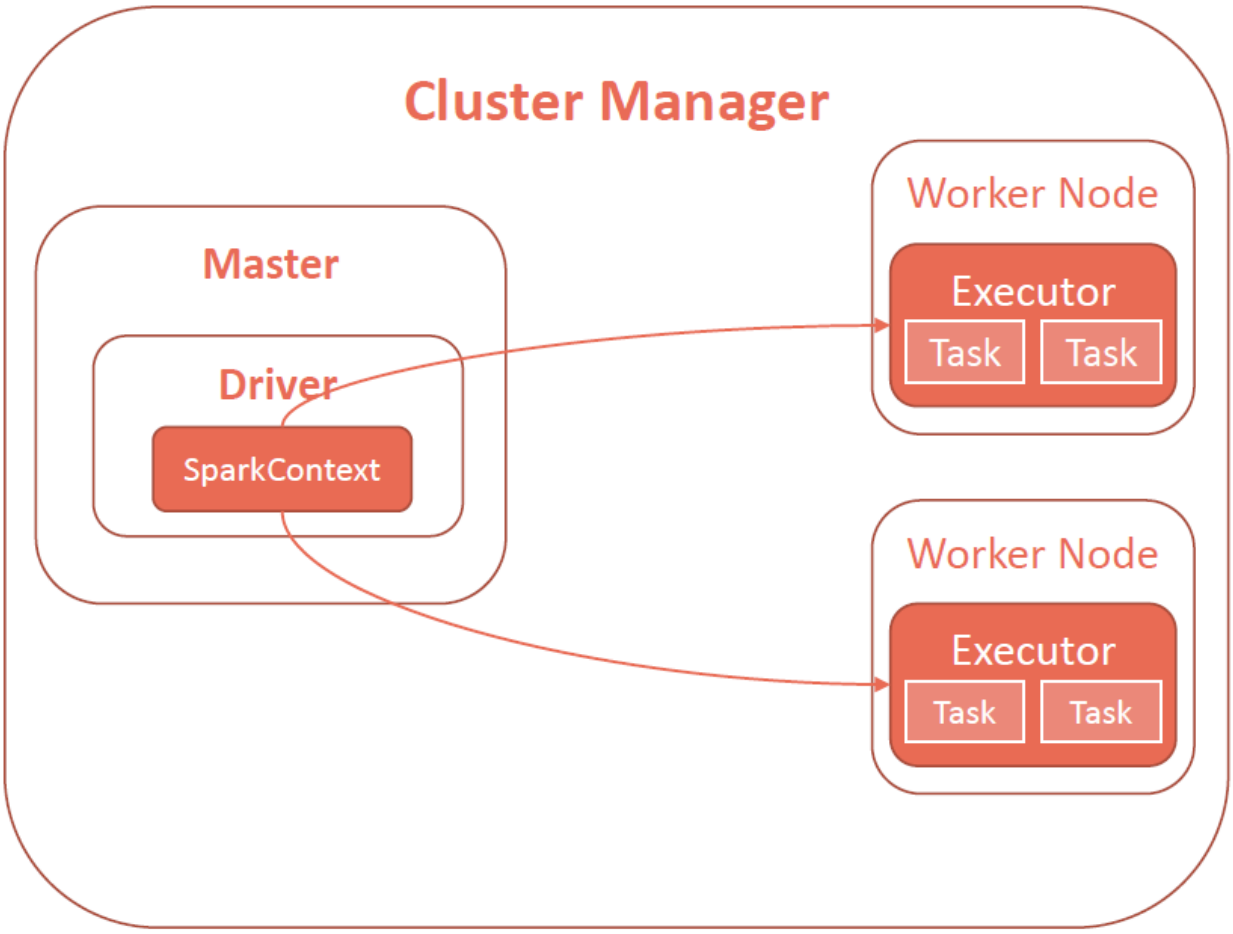
The PairRDDs are the only way to apply key-based transformations such as join or ReduceByKey.
Basically, we can “fuse” two different RDDs.

Thanks to the key management of RDD we can “play” with the hashPartitioner function to improve the distribution of data across the cluster to improve performances.
For example:

- File 1: 1k lines,
- File 2: 10k lines,
- File 3: 1M lines.

We can define a Map function and call a re-shuffle to balance the nodes in the cluster.

Architecture of Spark



Driver

It is a process responsible for the execution of a spark application, it is basically responsible to convert a user application into a smaller execution unit called tasks and to schedule them, this along with the Master.
It holds the SparkContext, a singleton that allows to use the context information of spark, besides it stores the metadata about all the RDDs (like a NameNode).

Master

It is a framework-specific entity unique per application and responsible for negotiating resources with the ResourceManager, it is created on the same node of the Driver and could be backed up to improve the reliability of the system.
It works with the NodeManagers to execute and monitor the component tasks, the workers will then send the heartbeat to the master to show their status.

SparkContext

It is the core object of Spark, it allows the spark driver to use the cluster and call it through a cluster resource manager with its primitives. It is the only entity allowed to create spark specific types (such as an RDD, an accumulator, etc).
It receives a heartbeat from executors in order to keep track of their status and it is spawned by the spark JVM.
It must be a singleton, so 1 SparkContext for each JVM, this leads to the necessity to stop and restart the sparkContext in order to change the JVM configuration.

Worker

It is a spark computational node where executors perform the serialized tasks received in the thread pool; it uses a local Block Manager to communicate with other workers.
When a sparkContext is created, each worker starts an executor (or more than one, this number is communicated to the master) as a separate process (a JVM in particular) and connects back to the Driver program to receive serialized code.

Executors

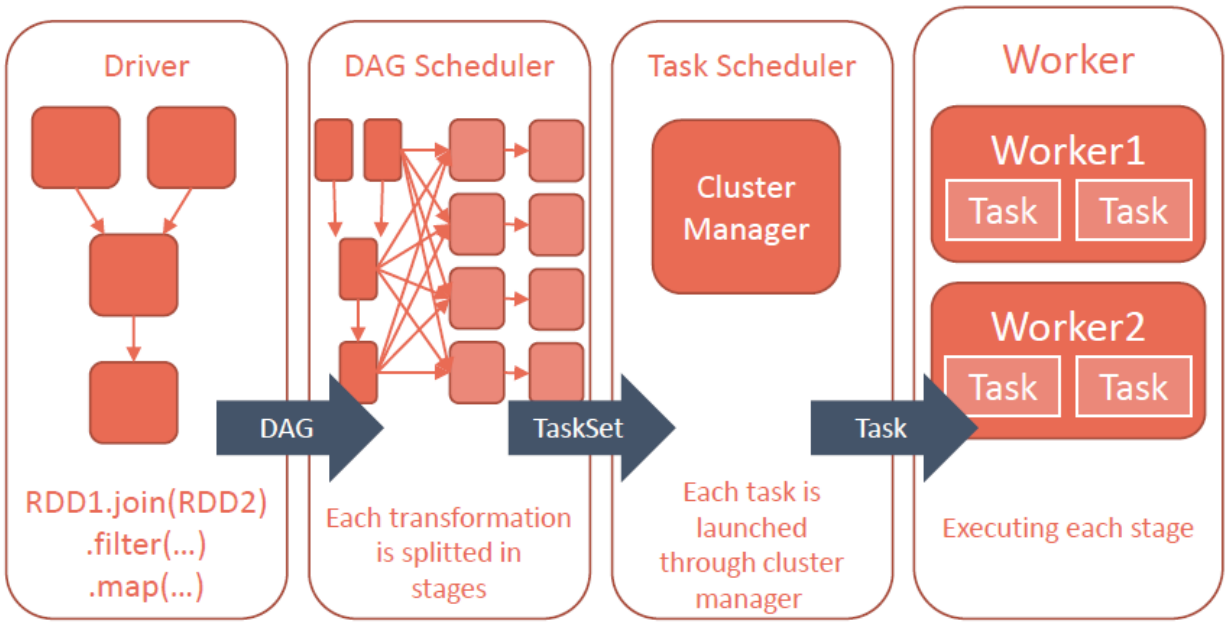
They are distributed agents that are responsible for executing tasks, they actually enable the resilient part of an RDD definition, as if one fails the computation is moved to another and re-played.
They are also used to provide in-memory storage for RDDs, in particular when .persist() or .cache() methods are called.
Tasks are tracked with a unique ID.

Spark “laziness”

Spark does not compute the result immediately, but only when it’s needed, so it happens after a Driver request for results, only the part needed for the result is computed.
Each transformation is recomputed each time a result is needed, we can in any case persist a result in memory with the homonymous primitive.

Spark offers several Map functions, each of them works line to rows-over-rows applying functions, as well as offers several reduce functions applied to a couple of rows given a key until there are no more couples with the same key.
This leads to the necessity of reducing functions to be commutative as no one knows the exact order in which the rows will be processed.

From RDD to Workers



Inside spark there two kinds of functions:

- Transformations: which are all the mapReduce functions available and no processing is done;
- Actions: they trigger the computations (ex. Give me the first result, etc.) and return the results to the driver program.

Examples of transformations are:

- .map();
- .flatMap();
- .filter();
- .join();
- .repartition();
- …

Examples of actions are:

- .first();
- .take();
- .top(n, key);
- …

Let's now see how spark work:

1. Any transformation made over an RDD just adds nodes to a DAG,
2. A DAG scheduler receives the logical execution plan and transform it into physical execution plan: Every execution plan is composed by a DAG of stages

that implement the transformation logic, a stage executed over a group of rows is called task

3. The task scheduler then launches tasks via cluster manager, the number of tasks submitted depends on the number of partitions in which the RDD is split in,
4. Finally, workers execute the tasks.

Shared variables

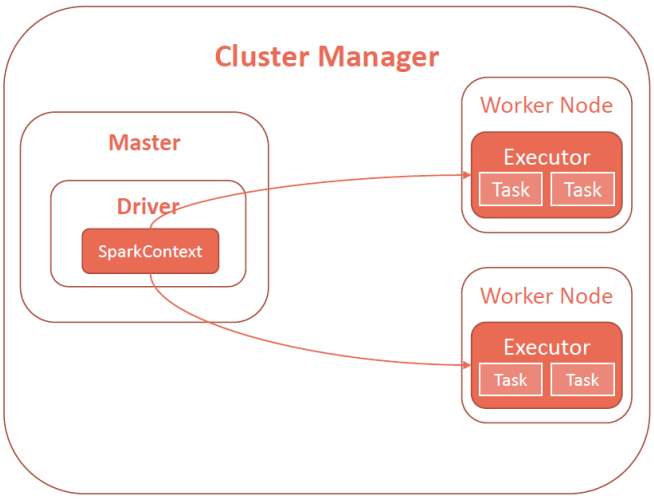
RDDs are not the only method to store data inside Spark, infact, by default, when Spark distributes a function it also ships a copy of each variable used in the function once per row. This could obviously lead to performance issues if the variables are big or complex.

Two types of shared variables are supported:

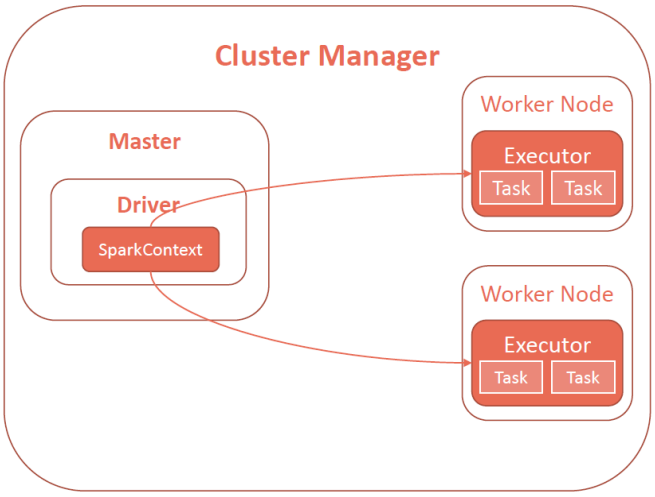
- Broadcast variables: used to distribute read-only values once on all nodes, it is permitted to cache them in a node in order not to ship a copy of them for each task sent to a worker;
- Accumulators: used to distribute write-only variables, they are often used to implement counters or sums for example.

Spark actions are executed through a set of stages, separated by “shuffle” operations, it broadcasts automatically and in the best way possible all the shared variables needed by tasks within each stage. Serialization and Deserialization are employed before and after a task to cache data.
This leads to the fact that explicit variables broadcasting is useful only when tasks across multiple stages need the same data.

Spark Job Execution: what happens under the hood



1. A spark application is submitted using **spark-submit** utility
2. **Cluster Resource Manager** starts the **Application Master** and allocate needed resources.
3. The Application Master registers itself on Resource Manager to allow a two way communication.
4. **Spark Driver** runs the code communicating with **Application Master**.
5. The driver implicitly converts user code from a logical plan to an execution plan going trhough **DAG** and **DAGScheduler**
6. Driver negotiates with Cluster Manager the **resources**. This way from DAGScheduler **stages** are created
7. **Executors** are started on Workers. When they start, they **register** themselves with Driver.



8. Driver keeps trace of **executor status** to be able to redistribute dead executors to other workers.
9. Driver sends tasks to the **Cluster Manager** based on data placement.
9. Application Master launches the container by providing the Node Manager with a container configuration.
10. The **first RDD** is created by reading the data from HDFS into different partitions on different nodes in parallel: each node has a subset of the data.
11. During application execution, **Driver communicates with the Application Master** to obtain the status of the application.
12. When the application has been completed, the Application Master deregisters from Resource Manager and **free all the resources**

Advices

- Even if you are working with **a small subset** you should never consider this when you write your code
- Remember, you cannot count on **rows sorting** when you write your code
- Avoid **.collect()** action always, is very dangerous
- If you are putting a **for iterator inside your lambda function**, probably there is something wrong with your code
- If you are putting a **for iterator outside your lambda function**, there is something wrong with your code for sure

Delta Lake

It is a way to obtain transactionality over HDFS, even if it is not a good idea.

Transactionality

Transactionality is the property of a process which can support transactions, sequence of operations that satisfies ACID properties, and consequent Rollback operations, an operation to restore the system in its original state, so as before the transaction started.

Transactionality has a big complexity in terms of time and resources, there 2 big challenges to face:

- Ensuring data consistency: all the problems consequently present for concurrency (especially among writing and reading processes);
- Allowing processes to communicate between them: so a way to have an inter process communication to send output or information between processes.

Concurrency Control

All of this leads to concurrency control problems, it ensures that concurrent operations produce correct results as quickly as possible, possible scenarios are:

- Lost update problem: two transaction update the same data and the first update is lost;
- Dirty read problem: a transaction read a value written by a later aborted one;
- Incorrect summary problem: an aggregation is done by a transaction while another is updating the source.

Locks and Deadlocks

Concurrency control could be faced with locks, a kind of functionality which assigns the control over chunks of data to a single transaction and prevent others from managing that particular chunk.
In particular we observe locks and mutex as sync mechanisms, the most simple one is the binary semaphore, which can assign exclusive access to resources.
A more complex one could be the resource-operation semaphore which can grant a shared lock permitting only a subset of operation over a resource.

The wrong management of concurrency could also lead to a deadlock situation, in which we have some circular waiting dependency among some transactions, for example transaction A needs the resources locked by transaction B which needs the resources blocked by the first transaction.
To prevent these situations there exist deadlock prevention algorithms.

There also exist concurrency control strategies:

- Optimistic: systems with multiple transactions with low data containment, no locks are used and a check is performed to avoid read of updated data (so only freshest are read), in case fresher data are present, the transaction is restarted.
- Pessimistic: systems with multiple transactions with high data containment, locks are widely used and transactions block other processes to read in case of conflict.
- Semi-optimistic: systems with multiple transactions with uncertain level of data containment, operations are blocked in some situations and validate in certain cases.

To ensure transactionality pessimistic lock, two-phase locking, multiversion lock could be then used.

Logs

Every transaction logs the history of operations executed and the DB logs all the transactions performed in order to guarantee ACID properties in case of failures. Obviously these logs are huge files and could be a reason for DB crash due to low disk space.

Log shipping

It is a technique used to increase DB reliability and as a disaster recovery strategy, the logs are sent to a slave server to create a copy of the original DB. It is maybe the most secure disaster strategy but it is hardware intensive for the master for double writing problems. This problem is slightly softened by activating it only on a subset of tables.

Inter process communication (IPC)

IPCs are mechanisms to allow different processes to manage shared data, a common architecture is the client/server one. Each of them could:

- Synchronous;
- Asynchronous.

Some examples of IPCs strategies are:

- File: stored on disk and accessible by multiple processes;
- Socket: data are sent over network interfaces to different processes ot machines;
- Anonymous pipe: a one directed data channel using system I/O, data are written to an end and consumed in the other;
- Named pipe: it is a pipe treated like a file that replace the system I/O;
- Shared memory: Multiple processes are given access to the same block of memory which creates a shared buffer.

Delta Lake

Delta Lake is conceived as a high-performance ACID table storage over cloud object stores.
It uses a transaction log that is compacted into Apache parquet format, a format optimized for metadata operations over tabular datasets. The most common way to store relational datasets in cloud object stores is using columnar file formats such as

Parquet and ORC where each table is stored as a set of objects, all possibly clustered into partitions by some fields.

Parquet

Parquet is an open source file format available to any project in the Hadoop ecosystem, it is designed for efficient storage of columnar data (rather than row based data as CSV, TVS).
The **record shredding and assembling algorithm** is used to manage data, it also gives the possibility to compress and encode in different types efficiently.
It is obviously used for queries that need to read a certain column, minimizing the I/O.

The algorithm stores data in a column-wise and serialized way, as a consequence the compression is far more efficient and type-aware and can give to each field a custom compression technique.

Unfortunately, cons of parquet are not atomic updates, no isolation between queries and individual updates to each object. The major con is the absence of rollback, if a file is corrupted, all is lost.

Pills

- In delta lake we maintain information about which objects are part of a delta table in an ACID manner using a write-ahead log that is self-stored in the cloud object store.
- The objects themselves are encoded in parquet, connectors between engines that support this format are easier to implement.
- The log contains metadata statistics (min, max, etc) for each data file, enabling faster metadata searches.
- Transactions are achieved using optimistic concurrency protocols against the object store.

Features

- Time travel: let the user query a certain point in time and rollback erroneous updates to data;
- UPSERT, DELETE, MERGE: operations which efficiently rewrite the relevant objects to implement updates to archived data and compliance workflows;
- Efficient Streaming I/O: by letting streaming jobs write small objects into the table at low latency, then transactionally coalescing them into larger objects later for performance. Fast “tailing” reads of the new data added to a table are also supported, so that jobs can treat a Delta table as a message bus;

- Caching: due to data immutability, cluster nodes can safely cache them on local storage.
- Schema evolution: Delta is allowed to read old parquet files without rewriting them if a table's schema changes.
- Audit logging: it is based on transaction log.

Docker

Kernel

The kernel is a computer program at the core of a computer's OS with complete control over everything in the system, it is an integral part of it and it facilitates interactions between hardware and software

Virtualization

There are two kinds of virtualization:

- Hardware or platform one, which refers to the creation of a virtual machine that acts like a real computer, the software that creates a virtual machine on the host is called hypervisor;
- OS-level one is an operating system paradigm in which the kernel allows the existence of multiple isolated user space instances, a user space is a memory area where application and drivers executes.ù

Jupyter Notebook

The jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, etc..

Docker

Docker is a set of PaaS products that use OS-level virtualization to deliver software in packages called containers.
Each container is isolated from the others and bundles its own software, libraries and configuration files, although they can communicate through preset channels.
The isolation is obtained using a linux facility called **cgroups**, which isolates accounts and resources of a collection of processes.
Instead of a fully virtualized system, docker virtualizes a part of it to require less resources.

Dockerfile

A dockerfile (the name cannot be changed and it has to be only one per directory) is a text document that contains all the commands needed to assemble the image used by Docker for the virtualization.
Using the docker build command we create the relative docker image.

Once the build starts, the Docker daemon takes care of executing all the commands, so commands are not performed by the linux CLI → Each line of the docker file is executed as a specific and separate command. If a line is modified, in the next build only from that line on will be executed.

Each Dockerfile must begin with a **FROM** instruction, which states from which parent image the docker will be built. The only exception is the **ARG** instruction, which declares arguments that are used in FROM lines in the Dockerfile.
The FROM chained calls will at some point end with a FROM statement which employs an operating system (often Linux).
Other important commands are **RUN** ones, which performs every kind of operation that we can run onto a bash CLI. Every RUN is considered a step in the building process of a DOcker image, if a line starts to be too long it can be chopped into multiple lines with the usage of '\' character.

We then have the **COPY** command, which allows us to copy a file from a relative path starting at where the Dockerfile is into the container.

CMD allows you to perform actions at the start of the container when "docker start" is launched, it has to be used once, if there are multiple CMD commands only the last will be taken in charge.

ADD is like the COPY command but allows to import even web resources from URLs.

EXPOSE command can be used to expose components of the docker to external systems, ex we can expose a port to the external world.

ENV command is used to define local environment variables in the form of key=value tuples.

Jupyter Docker Stacks is a useful resource to start working on Big Data environments with docker.

Docker step by step

1. A Dockerfile is written and contains all the commands needed
2. A Docker image is created from the Dockerfile using the “docker build” command
3. A Docker container is created from a given docker image with the “docker run” command
4. Running containers can be listed with “docker ps” command
5. Running containers can be managed as services with “docker start/restart/stop/kill” commands
6. Available docker containers can be listed with “docker ps -a”

Docker compose

Docker allows to orchestrate several containers together using a yml configuration file, “links” are constructs which make it possible that containers see each other.

Commands are:

- docker-compose up - build the cluster
- docker-compose stop - kill the cluster
- docker-compose start - restart a cluster

Spark to SQL

Table index

A DB index is a data structure which speeds up data retrieval operations at the cost of additional writes and storage space; they are used to locate data without scanning row by row a table.
Indexes can be created by one or more columns.

Easy insert into DB

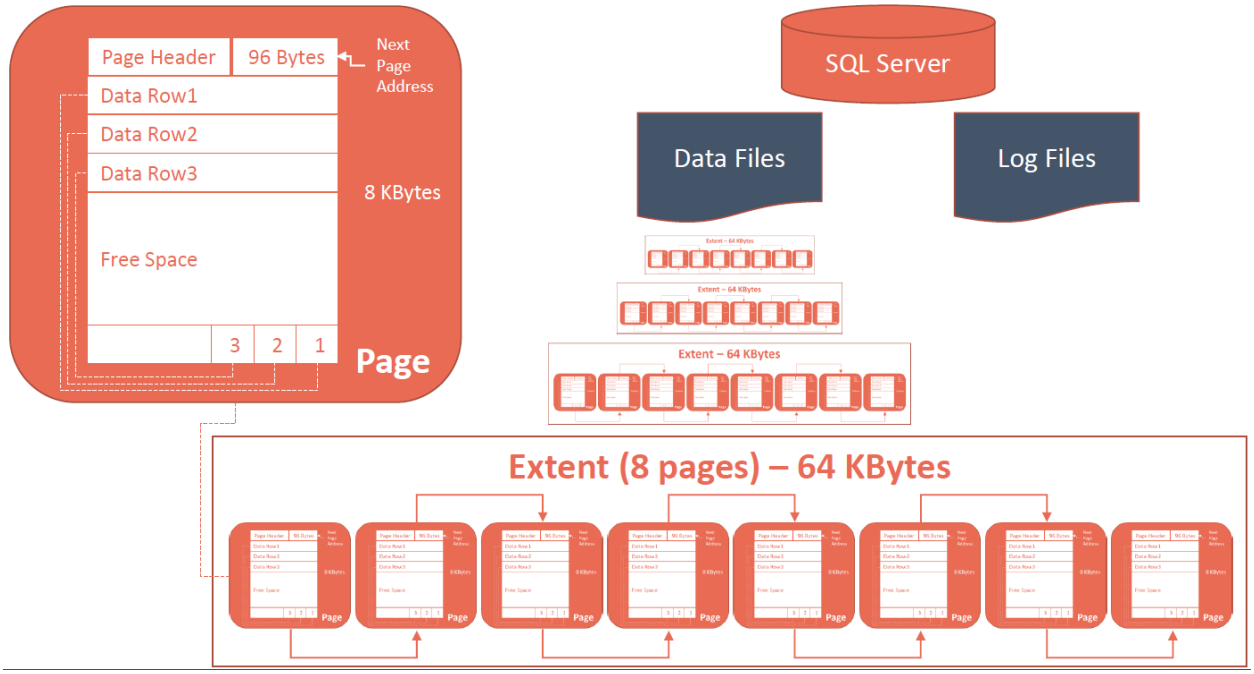
While inserting data into the DB the biggest problem we face is the concurrency, more in general:

- Write operations memory usage;
- Physical connections;
- Indexed tables need to be locked.

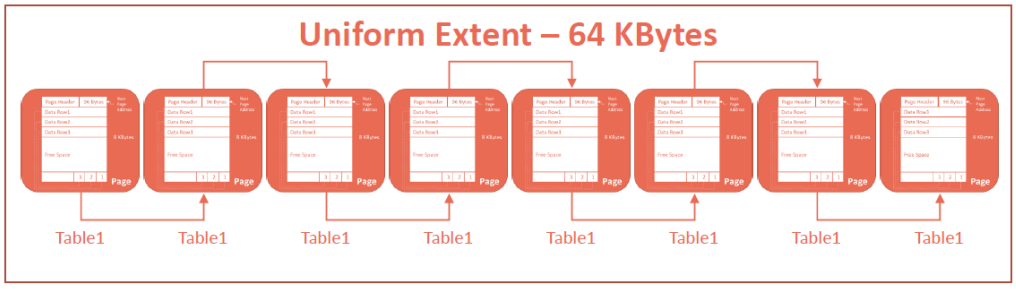
SQL Server Architecture

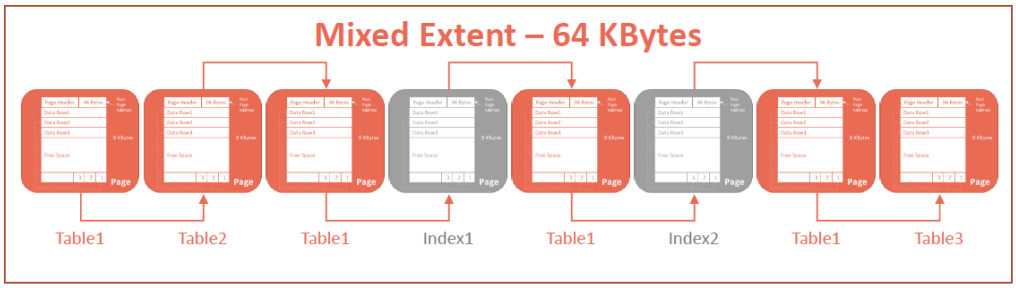
Before diving into these problems, we can see how a SQL server instance is physically structured:

- 2 main structures:
  - Data files: collections of extent which are collection of 8 pages of 64KBytes, inside each page there are the structures of the image in the top left corner;
  - Log Files.

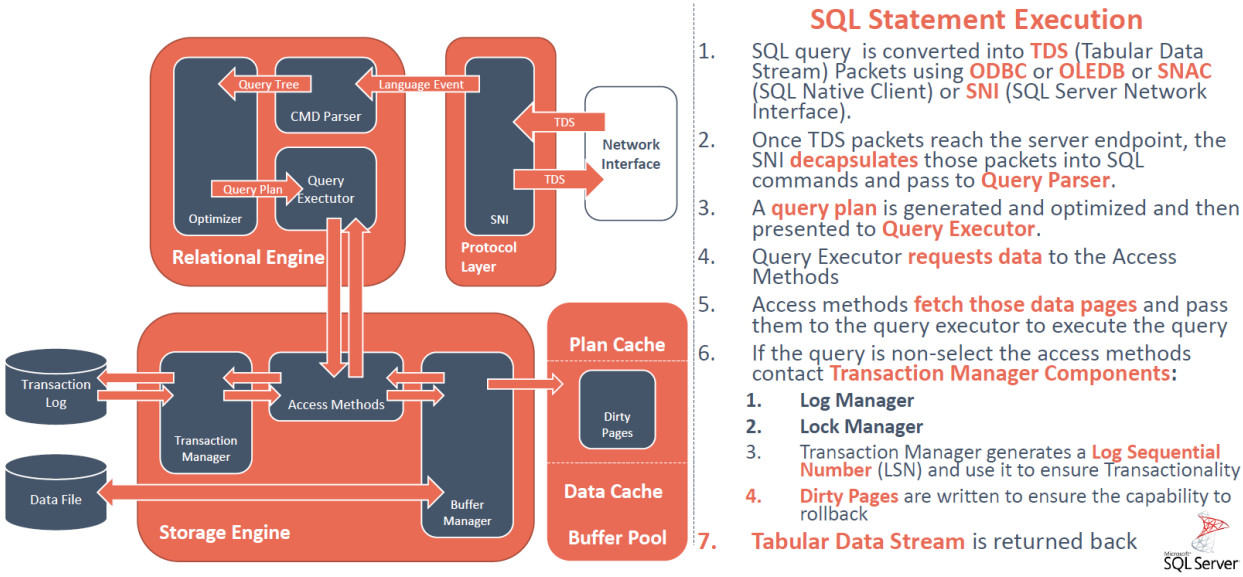


There are 2 kind of possible extents:





Now we can see how a query work and the first of three problem we face (memory usage):



SQL Statement Execution

1. SQL query is converted into **TDS** (Tabular Data Stream) Packets using **ODBC** or **OLEDB** or **SNAC** (SQL Native Client) or **SNI** (SQL Server Network Interface).
2. Once TDS packets reach the server endpoint, the SNI **decapsulates** those packets into SQL commands and pass to **Query Parser**.
3. A **query plan** is generated and optimized and then presented to **Query Executor**.
4. Query Executor **requests data** to the Access Methods
5. Access methods **fetch those data pages** and pass them to the query executor to execute the query
6. If the query is non-select the access methods contact **Transaction Manager Components**:
   1. **Log Manager**
   2. **Lock Manager**
   3. Transaction Manager generates a **Log Sequential Number** (LSN) and use it to ensure Transactionality
   4. **Dirty Pages** are written to ensure the capability to rollback
7. **Tabular Data Stream** is returned back

Microsoft SQL Server

Log Manager

- Log Manager keeps a track of all updates done in the system via logs in Transaction Logs.
- Logs have **Logs Sequence Number with the Transaction ID and Data Modification Record**.
- This is used for keeping track of **Transaction Committed and Transaction Rollback**.

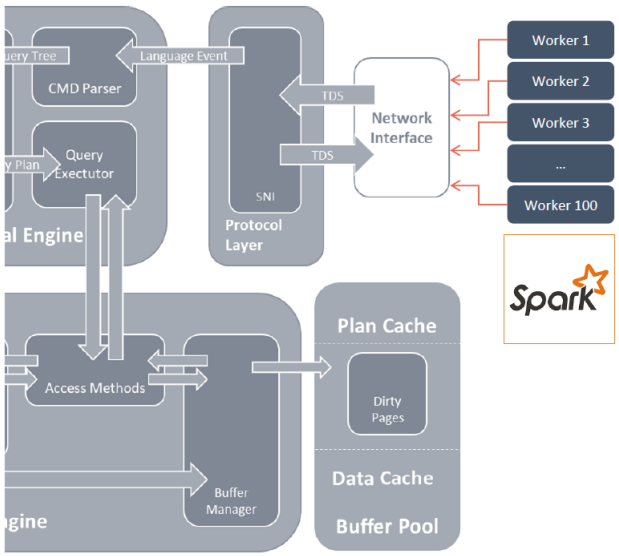
Lock Manager

- During Transaction, the associated data in Data Storage is in the Lock state. This process is handled by Lock Manager.
- This process ensures **data consistency and isolation**. Also known as part of ACID properties.

Execution Process

- Log Manager start logging and Lock Manager locks **the associated data.**
- **Data's copy** is maintained in the Buffer cache.
- **Copy of data supposed to be updated** is maintained in Log buffer and all the events updates data in Data buffer (**Dirty Pages**)

Now we face the problem of the external connections:



Direct Connection

1. A physical channel such as a **socket** must be established
2. the initial **handshake** with the server must occur
3. the **connection string** information must be parsed
4. the connection must be **authenticated** by the server
5. checks must be run for enlisting in the **current transaction**

Connection pooling

1. A pooler maintains **ownership of the physical connection**.
2. It manages connections by **keeping alive** a set of active connections.
3. Whenever a user calls **Open on a connection**, the pooler looks for an **available connection** in the pool and then **it returns** it to the caller.
4. When the application calls Close on the connection, **the pooler returns** it to the pooled set of active connections and is ready to be **reused**

Pool fragmentation

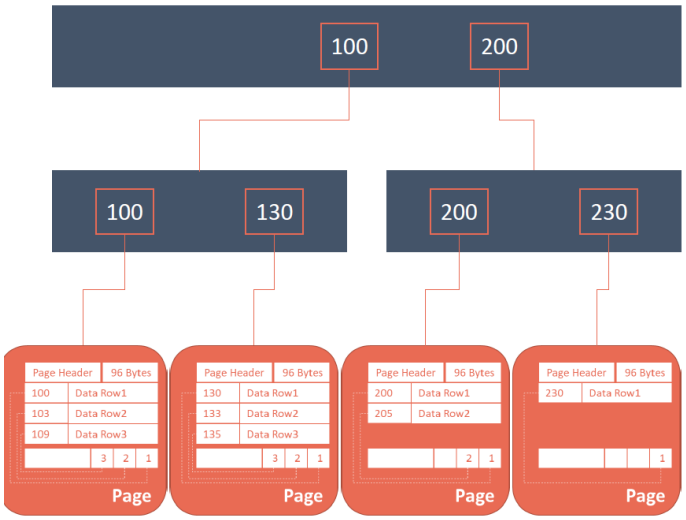
- application can create a **large number of pools** that are not freed until the process exits consuming memory.
- Connections are pooled according to the **connection string.**
- Therefore, you get **one pool per worker**: workers cannot take advantage of connections made by other workers.

SQL Server

Where pool fragmentation is brought by the connection pooling connection system.

Each of these situations are not optimal, as each of them will kill the database due to a high memory usage and an impracticability of connecting directly hundreds of workers to a DB.

Now we face the last problem, we take as example the B-Tree+ as index structure:



B-Tree+

1. Data are stored only in **leaf nodes**
2. **Distances** between root and leaves don't change

Bulk-Loading

1. Data is **sort by search** key in ascending order.
2. An **empty page is created** to serve as the root, and insert a pointer to the first page of entries into it.
3. When the **root is full**, we split the root, and create a new root page.

Update

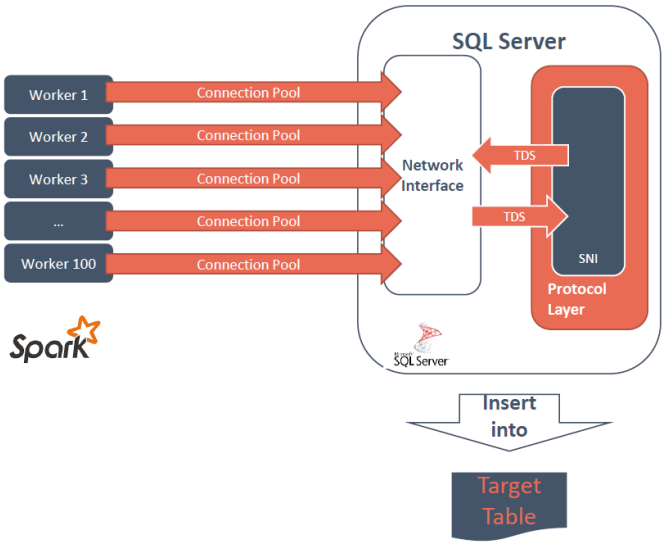
1. **Perform a search** to determine what bucket the new record should go into.
2. If the bucket is **not full** add the record.
3. Otherwise recursively **split the bucket up the the root**
4. If the **root splits**, treat it as if it has an empty parent and split it.

As we can see, inserting data, especially in case of hundreds of workers, is tricky and not easy, direct consequences are:

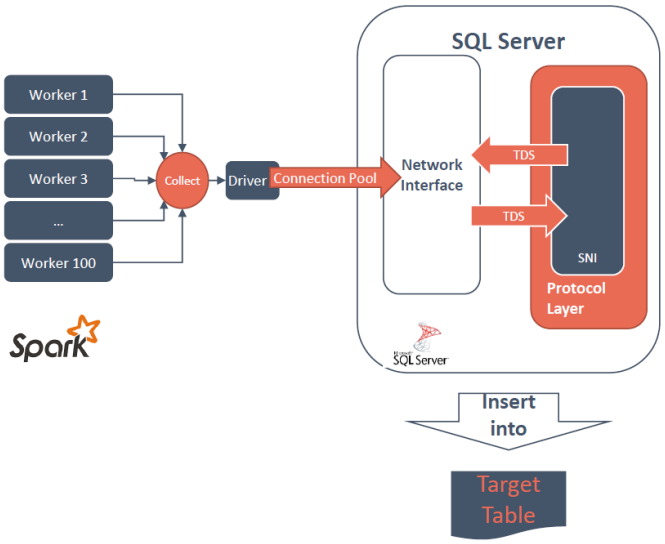
- Dirty pages created to enable the rollback in case of failure,
- Every worker is going to open a connection pool with the DB,

- Inserting in indexed tables is time consuming due to the necessity of updating the index itself and the lock on the table.

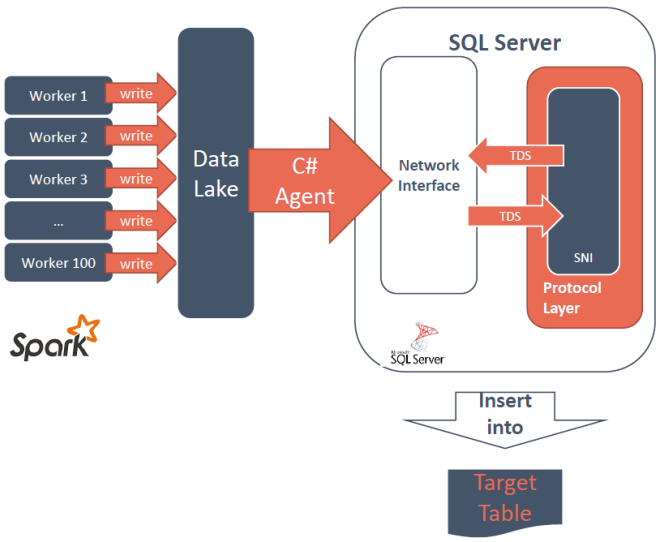
By starting with how **not** to write data from Spark to DB:



1. **Every Worker** will create a Connection Pool
2. Connection Pools will be **kept open** till all workers finish
3. A lof of **Dirty Pages** are created during the insert operation
4. Table is lock – inserting on an indexed table is a **lock transaction**



1. Driver makes a **global collect** to get data from each worker
   - Is **time consuming**
   - It must be able to keep in memory the whole dataset to avoid
     - **Disk spilling**
     - **Out of Memory**
2. **Driver opens** a single Connection Pool
3. A lof of **Dirty Pages** are created during the insert operation
4. Inserting on an indexed table is a **lock transaction**
5. [*If you are using an elastic Spark Cluster*] you are paying 100 VM with a double 1 to 1 VM **bottleneck**



1. Every Worker will write directly into **Data Lake queue – max speed**
2. C# Agent will use a **native client** to write into Sql Server
3. A lof of **Dirty Pages** are created during the insert operation
4. Table is lock – inserting on an indexed table is a **lock transaction**
5. ~~[*If you are using an elastic Spark Cluster*] you are paying 100 VM with a 1 VM **bottleneck**~~

We arrive to a solution of all problem:
With SQL Server we practically copy a datafile into the DB in a new table with BCP and then merge it with the already existing one.

```
bcp {database_name.] scheme.{table_name | view_name | "query"}
    {in data_file | out data_file | queryout data_file | format nul}

    [-a packet_size]
    [-b batch_size]
    [-c]
    [-C { ACP | OEM | RAW | code_page } ]
    [-d database_name]
    [-e err_file]
    [-E]
    [-f format_file]
    [-F first_row]
    [-G Azure Active Directory Authentication]
    [-h"hint [,...n]"]
    [-i input_file]
    [-k]
    [-K application_intent]
    [-L last_row]
    [-m max_errors]
    [-n]
    [-N]
```

Bulk Copy Program

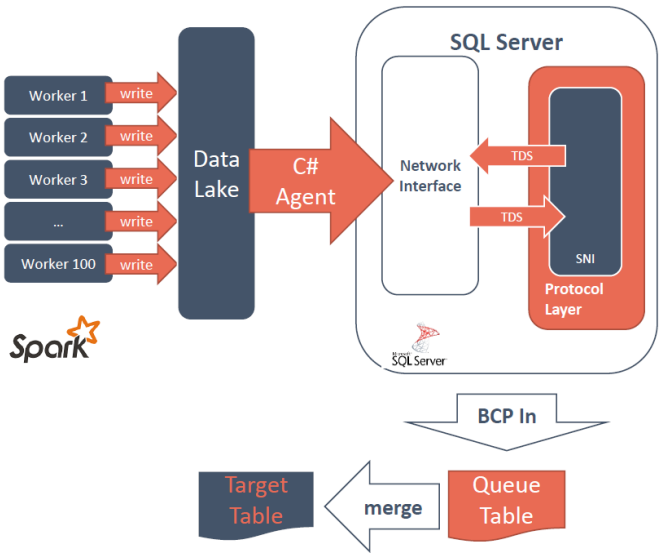
- bulk copies data between an instance of Microsoft SQL Server and a data file in a user-specified format.
- The bcp utility can be used to **import large numbers of new rows** into SQL Server tables or to export data out of tables into data files.
- BCP-In is the fasest way to import data inside SQL

Merge statement

- Runs insert, update, or delete operations on a target table from the results of a **join with a source table**.
- The conditional behavior described for the MERGE statement works best when the two tables have a **complex mixture of matching** characteristics.

```
1  MERGE target_table USING source_table
2  ON merge_condition
3  WHEN MATCHED
4     THEN update_statement
5  WHEN NOT MATCHED
6     THEN insert_statement
7  WHEN NOT MATCHED BY SOURCE
8     THEN DELETE;
```

How to write – Double Queues & Merge



1. Every Worker will write directly into **Data Lake queue – max speed**
2. C# Agent will use a **Bulk Copy Program** to write into a temp table **– max speed**
3. No **Dirty Pages** are created during the insert operation
4. ~~Table is lock – inserting on an indexed table is a lock transaction~~
5. ~~*[If you are using an elastic Spark Cluster]* you are paying 100 VM with a 1 VM bottleneck~~
6. **A merge** query is used to update target table

Spark Hands-on

Spark in pills

- Transformation: add a task to Spark DAG but does not start computation;
- Actions: force Spark to start executing tasks on DAG, partial results are sent to driver to collect them and serve to client, only needed tasks for that particular results are executed.
- Piped operations: Spark allows pipe transformations one after the other but only one action at the end.

Instantiate Spark context

- Pyspark library can easily instantiate the spark context (called sc) which is a singleton;
- The master address must be specified (ip address or local[n] where n is the number of executors created locally)
- While instantiating the sc, we can also create a sparkConf object which contains multiple information about the cluster (memory, executors, core per executors, etc)

Transformation - input

- rdd = sc.parallelize(list) → used to create the rdd by taking a list and distributing it among workers
- rdd = sc.textFile(path) → used as the above operation with “/n” as line separator for records

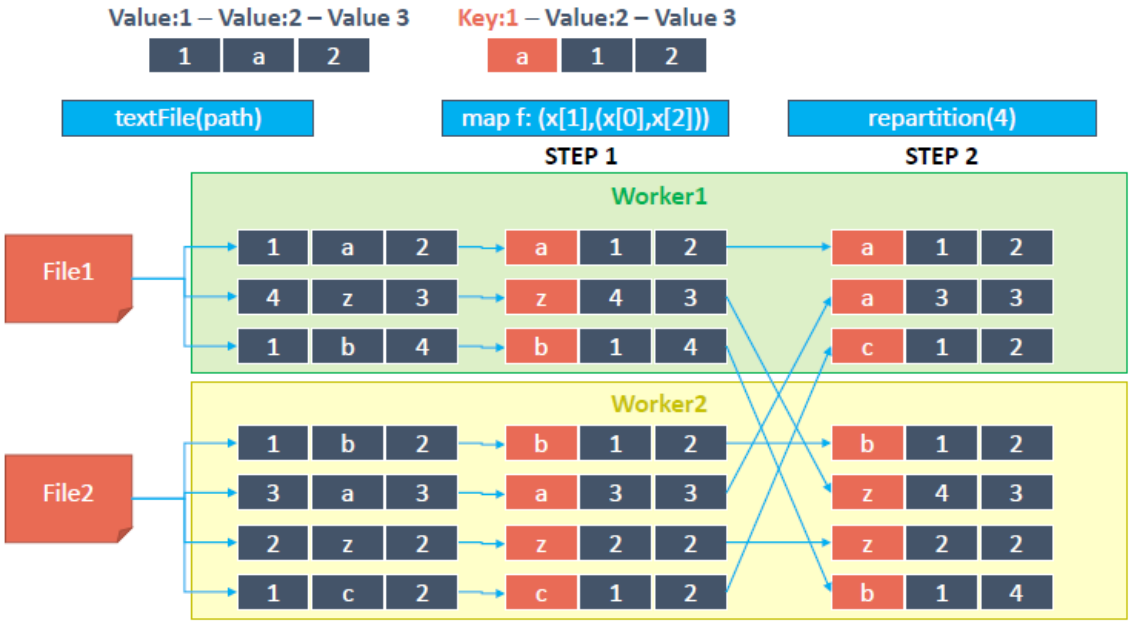
Transformation - T:rdd.map(lambda row: f(row))

- Map functions are used to apply the function ‘f’ to each row;
- ‘f’ must be commutative as we do not know the order of computation;
- Due to laziness if a row raise an exception, it will be seen only at runtime

Common pipeline - T:map, T:repartition, T:persist, A:count

- The map function is used to arrange the dataset in a key-value shape;
- rdd.repartition(n): forces a re-shuffle to distribute the rows of rdd in n partitions, if the number of the keys is less than the number n, spark will ignore it and use the maximum number at disposal (so the number of the keys);
- rdd.persist(n): forces executors to persist that specific DAG to speed up re-use;
- rdd.count(): execute all transformations and then count the number of lined inside the rdd, it is used to check if any line raise an exception and it is the most effective way to persist an rdd

Visual textFile, (map), repartition, persist, and count



Transformation - T:rdd.filter(lambda row: f(row))

- Filter is used to keep only lines where the value of the function ‘f’ is true;
- Consequently ‘f’ must return a boolean

Common Pipeline – T:textFile, T:map, T:filter

- Read all the lines from a given path, loads json to dict, and return only lines where the value of the dict for the key “status” is inside the list [“wip”,”ok”,”on hold”]

```
In [1]: #slide Transformation - example 001
        import sys
        sys.path.append("/home/jovyan/work/Architectures_for_Big_Data/")
        import pyspark
        sc = pyspark.SparkContext("local[3]")

In [5]: ## reading from path
        import json
        path = "./example.jsonl"
        sc.textFile(path).map(lambda x: json.loads(x)).filter(lambda x: x.get("status") in ["wip","ok","on hold"])
Out[5]: PythonRDD[9] at RDD at PythonRDD.scala:53

In [4]: ## from memory
        import json
        myList = {}
        sc.parallelize(myList).map(lambda x: json.loads(x)).filter(lambda x: x.get("status") in ["wip","ok","on hold"])
Out[4]: PythonRDD[6] at RDD at PythonRDD.scala:53
```

- Only with **“local” master** is possible to use a local path to read from

Transformation - rdd.flatMap(lambda row: f(row))

- The flatMap is used to create more rows from one;
- Consequently ‘f’ must return a List type

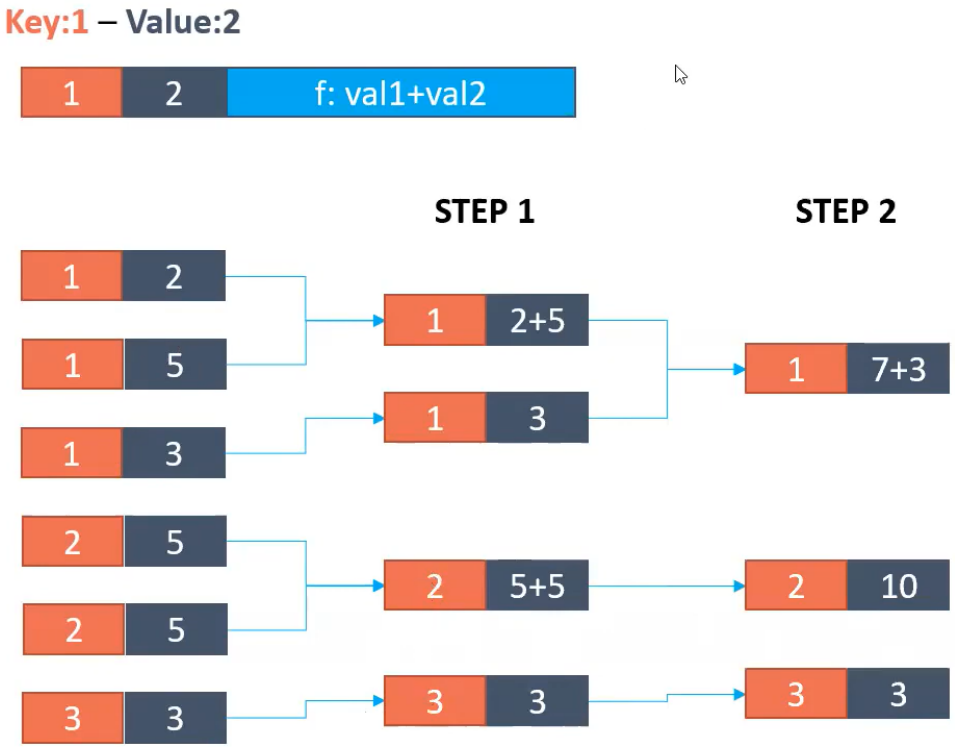
Transformation - rdd.reduceByKey(lambda val1, val2: f(val1, val2))

- reduceByKey apply the reduce pattern to an RDD
- rdd must be a pairRdd

```
In [6]: ## reading from path
        from random import randint
        path = "./example.jsonl"
        sc.parallelize([(randint(0,10),randint(0,10))])
Out[6]: ParallelCollectionRDD[10] at readRDDFromFile at PythonRDD.scala:262
```

- Each couple of row with the same key value is paired and the 'f' function is applied to their values
- This operation goes on iteratively until there are no more two rows with the same key

Visual reducebyKey



Action - Collecting as list

- **rdd.first()**
  execute all transformations and then move first element of an rdd to the driver
- **rdd.take(n)**
  execute all transformations and then move firsts n element of an rdd to the driver
- **rdd.top(n,f(row))**
  Apply f to each row, use the result to sort them and move the firsts n values - executing all transformations - to the driver
- **rdd.collect()**
  execute all transformations and then move all elements of an rdd to the driver

Other possible actions are:

- rdd.map().sum(): which sums all the elements of an rdd;

Transformation - rdd combinations

- **rdd.join(rdd2)**
  join 2 rdds. They must be pairedRdd to work and the key must be serializable
- **rdd.cartesian(rdd2)**
  combines all the lines of first rdd with the lines of second
- **rdd.union(rdd2)**
  Merge the 2 rdd appending the second to the first

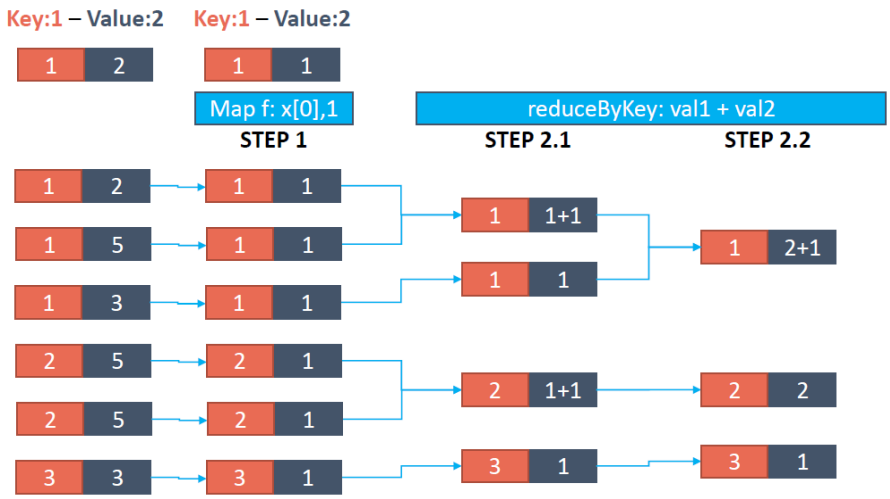
ASK WHY WE HAVE TO USE EVERY EXECUTOR WE HAVE BY TWEAKING THE CODE

Spark hands-on vol 3

Common spark pattern

Count Distribution pattern

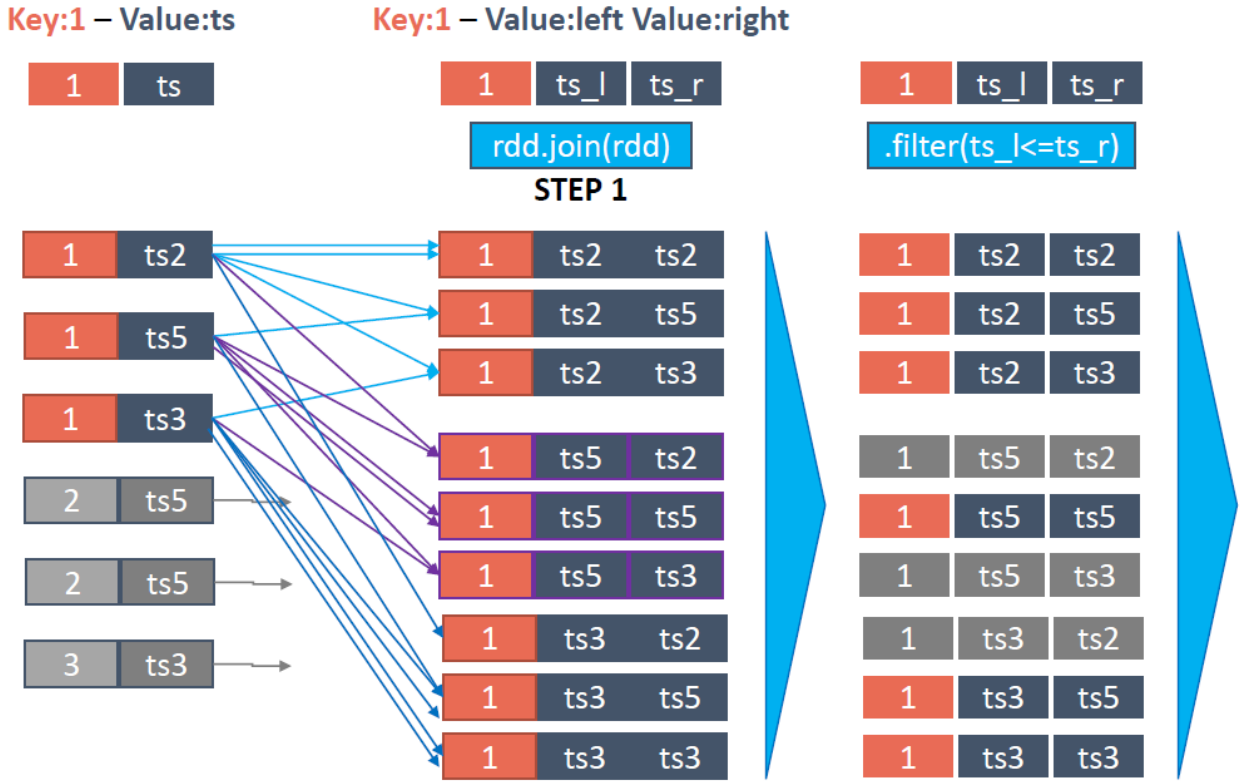
In the count distribution pattern we aim to count how many times a key appear in the dataset, we have an example in the picture below:

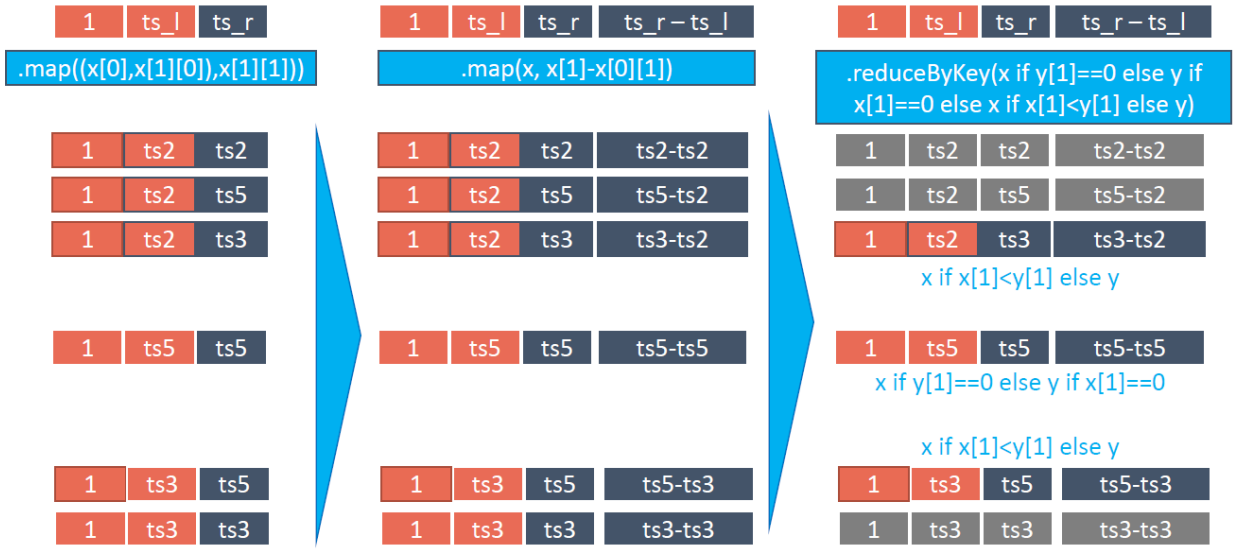


For the pattern we can employ these steps to resolve the problem:

- 1°: use a map function which associates a one for each row in which a key appears in, from this step we will have an RDD with as many lines as the original dataset, each of them will be a tuple in which the key resemble the original one and the value is a 1.
- In the second step a reduceByKey is used to sum up the number of appearances of the key.

Get Closest Next Element pattern





Imagine we have a list of elements with different keys and we would like to compute for each row the closest one, for example: for [1, ts2] the closest is the [1, ts3] (with an ascending direction of computation).

The first step is making a self join of the RDD, the RDD becomes now as in the figure 1, every element is coupled to all the other.

The second step consists in filtering out the elements that do not satisfy the direction chosen, for example if we chose the ascendant direction we will filter rows which do not satisfy this condition: ts_l <= ts_r. In this way we bring the older element to the left part of the RDD.
The equal in this case is needed to avoid the cancellation of the last element in the list, which actually has no closest element. Another way to achieve this, is to inject another element that we are sure is not duplicated and major than the last max one (ex. ts6).

The next step is to prepare the data for the reduceByKey operation, in particular we want to add to the key also the left part of the row, in order to have a tuple of the original key and the older ts. The value of the row will now be the candidate closest element.

We can then compute the delta between the two timestamps in order to have the difference between the two.

The last reduceByKey is now performed.

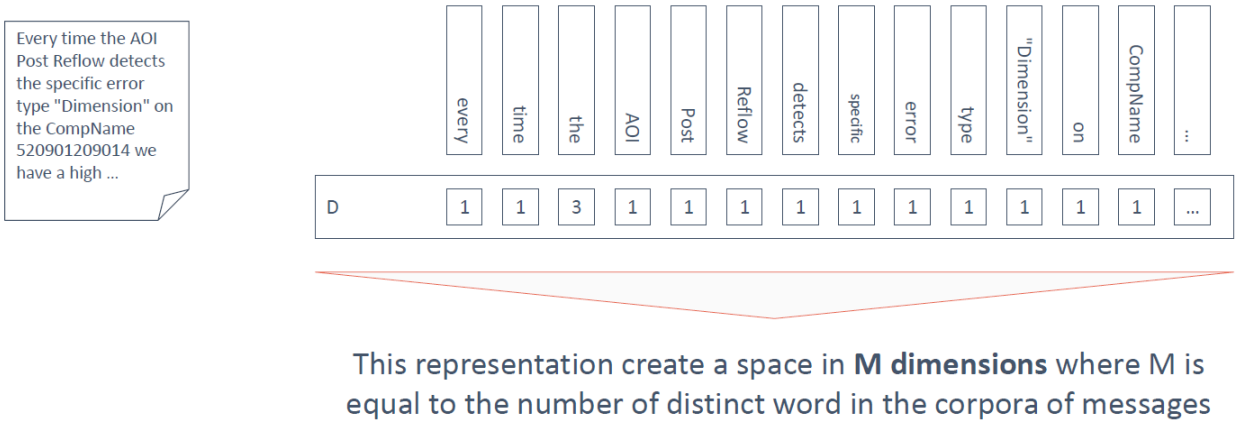
NLP - Natural Language Processing

The NLP scope is the one that aims to analyze natural language written or spoken phrases.

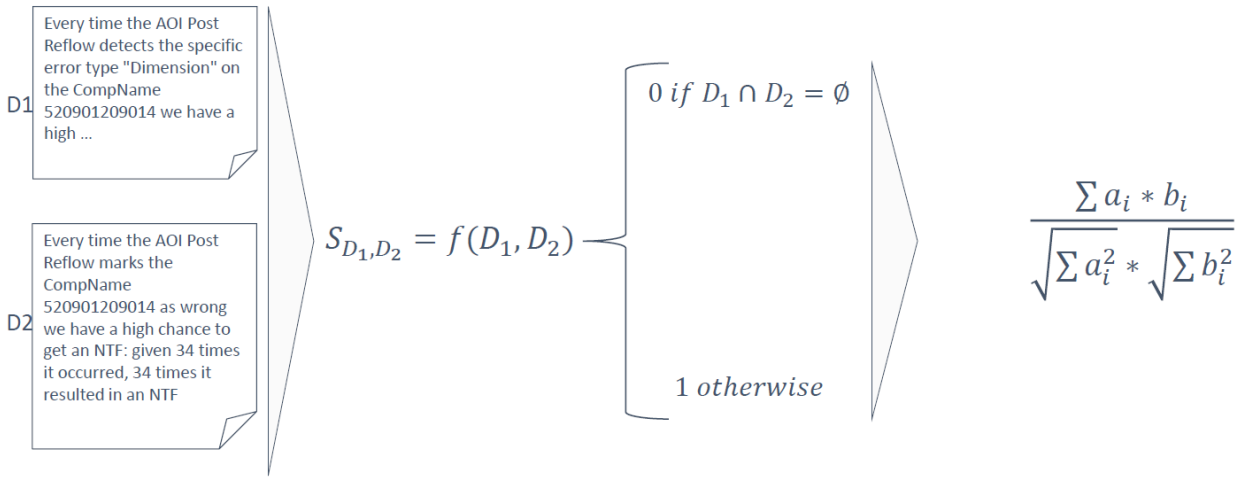
Main problems of this scope are:

1. Express documents in a quantitative way;
2. Create a metric to compute how similar they are.

The most effective way (for problem 1) to reformat text is by using the “bag of words”: it means representing a text using a vector in which we count how many times a token appears.



The most effective way (for problem 2) to create the metric we need, is to use the cosine similarity as distance metric: it calculates the distance between two vectors by using the cosine of the angle between two vectors.



The more the cosine similarity is close to one, the more two text are similar, the formula is actually “the sum of the frequency of each term in both text divided by the norm of the first text multiplied by the norm of the second text”

Why does cosine similarity outperform other distance metrics?
It is a matter of operation we need to do, in Euclidean distance we have to store even 0s as they add a contribution to the distance itself, with cosine similarity they are ignored.

$$euclidean\ distance(D_1, D_2)$$
$$\sqrt{\sum(a_i - b_i)^2}$$

$$c = 1000000^2 * 22000 = 2.2 * 10^{16}$$

22TB RAM

$$cosine\ similarity(D_1, D_2)$$
$$\frac{\sum a_i * b_i}{\sqrt{\sum a_i^2} * \sqrt{\sum b_i^2}}$$

#words/text = 40

$$c = 1000000^2 * 40 = 4 * 10^{13}$$

40GB RAM

An example of bag of words and cosine similarity is shown below:

		I	like	this	wolf	he	likes	ate	something	steak
D1	I like this wolf	1	1	1	1	0	0	0	0	0
D2	He likes this wolf	0	0	1	1	1	1	0	0	0
D3	I ate something like this wolf steak	1	1	1	1	0	0	1	1	1

$$cosine\ similarity(D_1, D_2) = \frac{2}{2 * 2} = 0.5$$
$$cosine\ similarity(D_1, D_3) = \frac{4}{2 * \sqrt{7}} = 0.75$$
$$cosine\ similarity(D_2, D_3) = \frac{2}{2 * \sqrt{7}} = 0.37$$

We will end by saying that the first and the second are similar texts, while they are not. This is because we face some problems while performing NLP analysis:

- Linguistic rules for stems: plural/singular and genres rules are specific for any given language;
- Polysemy: same symbol could be related with several concepts;
- Synonymous: more concepts can be expressed with several symbols.

Compute term frequency - Inverse Document Frequency

The Tf-idf algorithm is a weight computation algorithm used to assign weight to information present inside collections of documents and to evaluate how important a word is for a corpus.
Normally two terms are used:

- TF: Term frequency, which measures how frequently a term appears in a document (often the result is divided by the document length, due to the fact that in longer documents a term could appear more than in a shorter one);
- IDF: Inverse document frequency, which actually measures how important the term is. In this phase, certain terms like “is”, “this”, etc may appear a lot of times but have less importance for sure, so using the log_e(Total number of documents / Number of documents with the term in it) we weight down these terms in favor of rarer ones.

SOA - Service Oriented Architecture

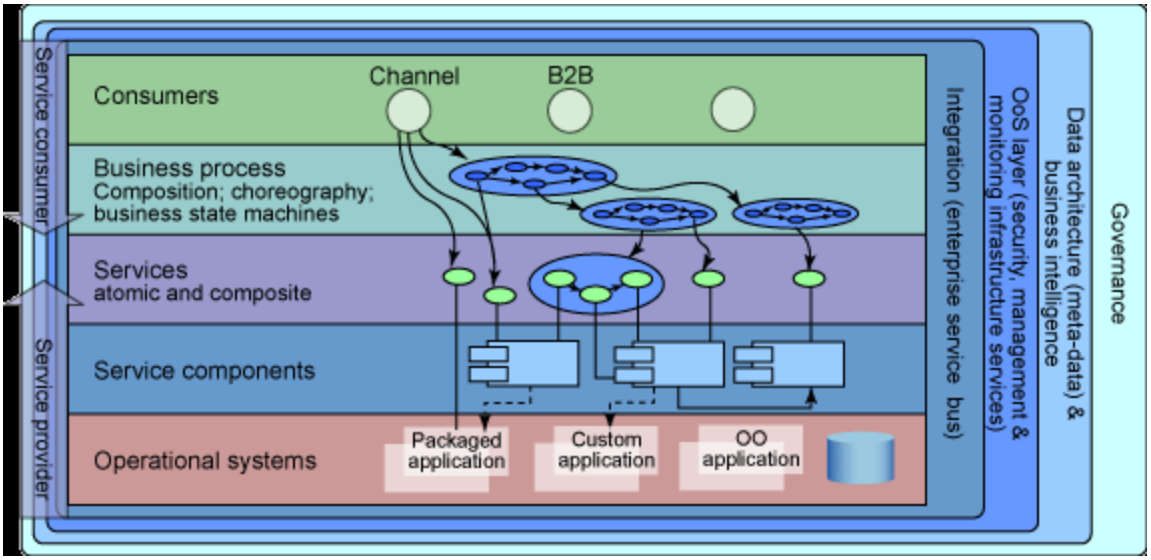
The service oriented architecture spans teams, as a consequence agreement upon relevant terms is crucial.

Among the different definitions (nine at this time at least) we cite for sure the IBM one:

- A service is a discoverable resource that executes a repeatable task, and is described by an externalized service specification, it is based on:
    - Business needs;
    - Specification which render them self-contained and described in terms of available interfaces;
    - Reusability;
    - Agreements among entities of different services to achieve a particular goal (ex. The communication format between producers and consumers);
    - Aggregation and loosely-coupled services which can be composed for a bigger end.

So for IBM a SOA is an architectural style for creating an enterprise IT architecture that exploits the principles of service-oriented to achieve a tighter relationship between the business and the information systems that support it.

The SOA stack is imagined as a 5 layered architecture:



Layers are:

1. Operational systems: existing IT assets
2. Service components: realize services using one or more operational systems layer applications
3. Services: services deployed to the environment which are discoverable
4. Business process: they are the operational artifacts that implements business processes
5. Consumers: represents the channels that are used to access business processes, services and applications.

Another definition of SOA comes from the Open Group:

- SOA is an architectural style that supports service-orientation;
- Service-orientation is a way of thinking in terms of services and service-based development.

SOA architectural style:

- Is based on the design of service, each of them mirror a real-world business activity or a business process;
- Service representation utilizes business description to provide context and implements services using service orchestration;
- It requires strong governance over service representation and implementation.

We can then define a service as:

- A logical representation of repeatable business activity with a specified outcome;
- A self-contained entity;
- A maybe composed entity (maybe composed by other services);
- A black box to consumers of the services.

In general, services define what they do in a clear way with a “contract” enabling service composition, discovery, message-based communication and model-driven implementation which give fast development of effective and flexible solutions.

OASIS goes beyond what we said till now and states that a SOA lives inside an ecosystem, so a system which can also notice the context in which a service (part of the ecosystem) is acting. As a consequence the relationships among elements are really important.

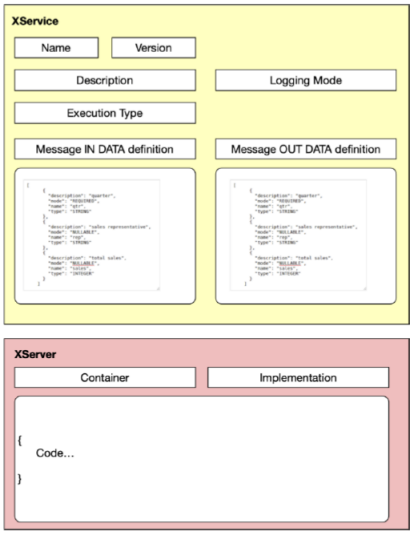
IBM SOMA - Description language

SOA Core Concepts

Integration Model

- Service
- Service Catalogue/Repository
- Service Queue
- Enterprise Service Bus
- Service Broker

Service Model



Each Service is defined – **outside the implementation** – by:

1. A unique **SERVICENAME**
2. A **version**
3. An **Execution Type** (Sync/Async/…)
4. **Logging** Mode (before, after, …)
5. A **messageIn** ➔ the configuration and the structure of **data In**
6. A **messageOut** ➔ the configuration and the structure of **data Out**
7. **Visibility** Cone

Each Service can be binded with one or more Server Object, defined by:

1. Application **Container**
2. Integration **Implementation**

Service Catalogue

- List of all available services
- It contains the list of “messageIns” to be used to configure a service execution

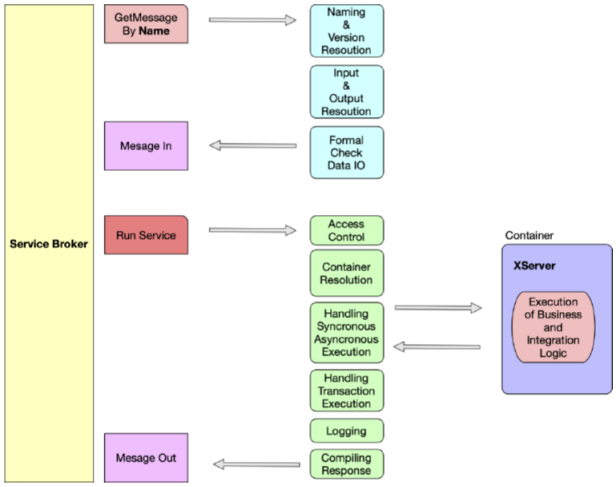
Service Queue

- List of all service executions
- Full trace of each execution
- Single point to monitor to get the health of any process
- Technically, a SQL Table

Enterprise Service Bus

- Communication System to allow services to interact with legacy systems
- It can be used to avoid the system-to-system integration
- Systems are represented as logical abstraction that can expose simple to complex functions

Integration Model – Service Broker



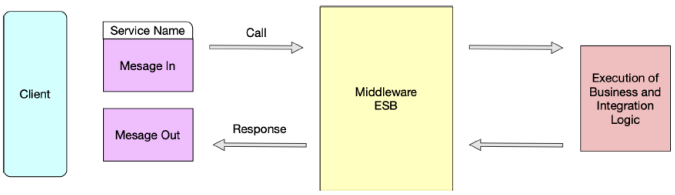
Service Broker implement the service broker pattern:

1. It is in charge of **submiting services** to the correct application server
2. It is in charge of **managing the Service Queue**
3. It is in charge of **updating** service **status** and messageIn/messageOut

Communication models

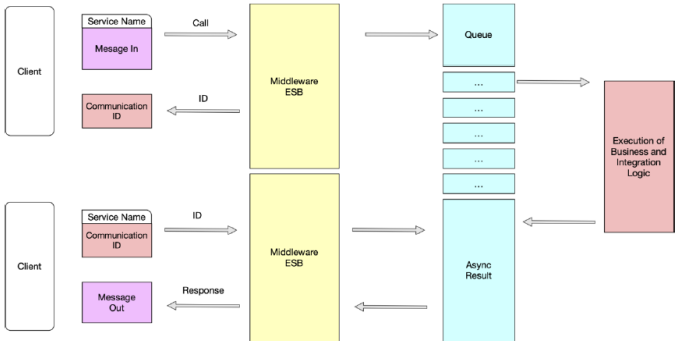
- Syncronous
- Asyncronous
- Stream Outbound File Reference
- Pub/Sub Service Call-back
- Batch Sync Data Process

Communication Model – Syncronous Service Call



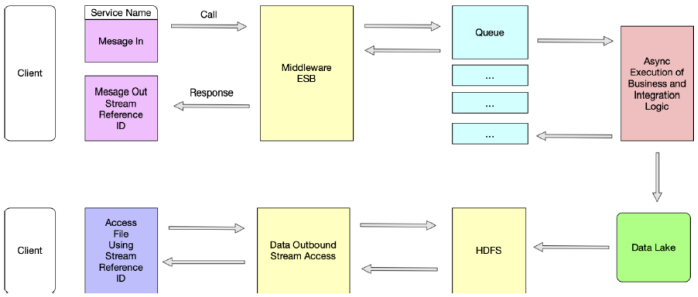
1. Clients **compile** the **Message In**
2. **ESB marshals** the request
3. The **Service Broker** submits to the specific Application Server
4. Application Server **executes the Logic** and return a Message Out
5. Each execution is corredated with
    1. A **status** that changes during the **execution** (To-Do – WIP – DONE|ERROR)
    2. A **trace log** (application and technical execution)
6. The Technical Execution is traced inside the **Service Queue Table** for further analysis and to check statuses

Communication Model – Asyncronous Service Call



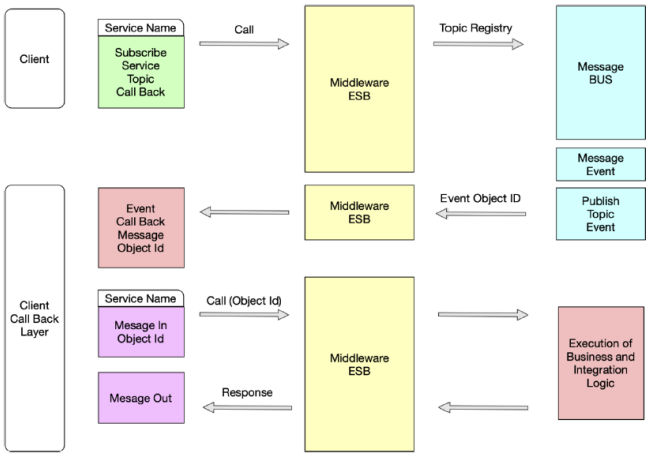
1. Clients **compiles** the Message In
2. Service Broker returns immediately the **Communication ID** and add the service to the Service Queue
3. When the **time** of execution **comes**, the **Service Broker submits** the execution to the right Application Server
4. The client can **check the execution status** using the Communication Id
5. The client can **get the results** of the execution when the status is DONE using the Communication ID and querying the Service Queue to extract the messageOut

Communication Model – Stream Outbound File Reference



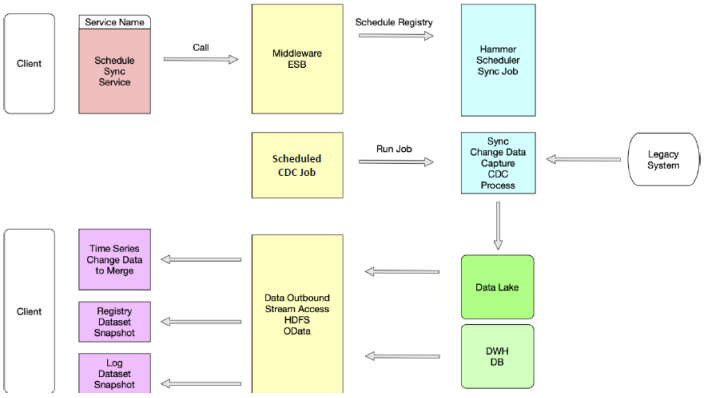
1. Stream Outbound File Reference model is an extesion of Async services
2. It can be used when large dimension output is going to be generated
3. The same **Communication Id** strategy is used to check the status
4. Furthemore a **Reference Id** allows client to open the stream towards the crated file

Integration Model – Pub/Sub Service Call-back



1. Pub/Sub allows clients to subscribe to a topic and to **get a Call-back** when **new event** happens on the subscribed topic
2. The ESB **hides the complexity of the Message Bus** (Apache Kafka) to make the client life easier
3. Every time a new event happen, the ESB **calls the Client call-back providing an Object Id**
4. The client can now **use the provided Object** Id to get fresh Data

Communication Model – Batch Sync Data Process



1. CDC Job allows **incremental Batch Data extraction** from any legacy system
2. Fresh rows extracted are saved on **Data Lake** (or DWH) on a shared naming convention path
3. A gateway service – **Reverse CDC Job** – exists to forword these incremental extractions on target systems (e.g., SQL DB via MERGE or via INSERT, APIs, …)

Spark Last Act

Shared Variables

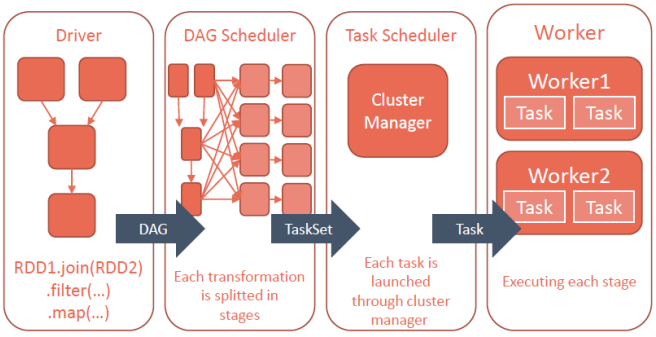
Spark, normally, ships a copy of the variables used by the distributed function once per row, this could obviously lead to performance issues in case of big and complex size. The answer to the problem is the shared variable:

1. Broadcast variable: used to distribute once on all nodes a read only value, they are used to ship for example a huge dataset in an efficient manner only once as Spark try to broadcast this variable keeping in mind to reduce communication costs;
2. Accumulators: used to distribute a write only variable, they remain efficient thanks to the fact that they can only be “added” with an associative and commutative operation. They are normally used to implement counters or sums.

Besides this, we may avoid trusting Spark caching strategies and use the broadcast variable instead to force this caching.

Inside Spark Engine

What happens when we start a park job?



- Any **transformation** made over an RDD just adds nodes to a Direct Acyclic Graph (DAG)
- **DAGScheduler** receives a logical execution plan (the DAG) and transform it into a physical execution plan.
- Execution plan is composed by a DAG of **stages** that implement the transformation logic.
- A Stage executed over a specific group of rows is called **task**
- The TaskScheduler launches tasks via the **cluster manager**
- The number of tasks submitted depends on the **number of partitions** in which actually is splitted the RDD
- Workers **execute task**

Dataframes

After Spark 1.3, new data types and features were added to the architecture, they are based on the library called Spark SQL, with respect to the standard RDD API, these APIs provide more information about the structure of data and computation performed. The Spark SQL library is basically an interface which executes SQL queries.

Besides, two other structures were introduced:

- Datasets: which are distributed collections of data, they combine the benefits of RDDs (lambda support and string typing) and Spark SQL's optimized execution engine. A dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, etc)
- Dataframes: which are datasets organized in named columns, they can be constructed from a wide array of sources as structured data files, hive tables, external DBs or existing RDDs.

RDD vs Dataframes

- While RDDs offer low-level functionality and control, dataframes and datasets allow custom view and structure.
- Rdd can be nested and deep whereas dataframes must be flattened.
- Dataframes optimizes execution for you while with RDD you can control the optimization.
- With dataframes you have a similar expressiveness to SQL, you are distributing lambda functions over rows.
- With RDD you have superpower, you are distributing lambda functions over objects.

From big data to big money

Costs and benefits

As in every field, the cost and benefit ratio is taken as the main point which could lead to a successful project or not.
As architects we should be able to monetize the cost of the system because we are in charge of deciding:

- How to do;
- When things will be ready;
- Where the system will be developed;
- Who is going to work on it?

This is the cost part of the system, then there will be a benefits part which will be monetized by the business itself due to the fact that it knows what needs to be done and why.

Inside cost scope, we can divide it into two big classes:

- CAPEX: called capital expenditure, this is the money spent to buy or improve fixed assets (buildings, vehicles, etc) → software is considered an asset.

- OPEX: called operating expense, this is the maintenance cost of a running product → software maintenance or licenses fees are opex costs.

From another point of view, more specific and nearer to a technical aspect, we can divide costs in 4 other classes:

- One shot: Paid only once to obtain the property of what you are buying, less dangerous than running costs and paying more one shots could reduce running costs;
- Running: They are a fee paid every now and then on a standard window time basis, more dangerous and less controllable;
- Variable: The variable part of these costs is an expense which change based on the production output size;
- Fixed: they are the fixed part of a cost overall which is paid independently of any other constraint or activity.

Infrastructure and licensing

Nowadays we can use the “as a service” definition to list product licensing and fees to help categorize costs and benefits.
There are three principal and particular definitions:

- Infrastructure as a service (IaaS): allow users to use hardware as they own it;
- Software as a service (SaaS): allow users to use a software without installation and maintenance needs;
- Platform as a service (PaaS): allow users to use complex platforms to build applications without worrying about configuration and management.

The other side of the coin is the bare metal, where you own and control everything, whereas it could lead to higher capex costs, it lowers the opex ones in favor of on premise configurations.

Project management

But how can we optimize a system's development? With good project management.

Project management is the process of leading the work of a team to achieve goals and meet success criteria at a specified time and by respecting project goals and given constraints.

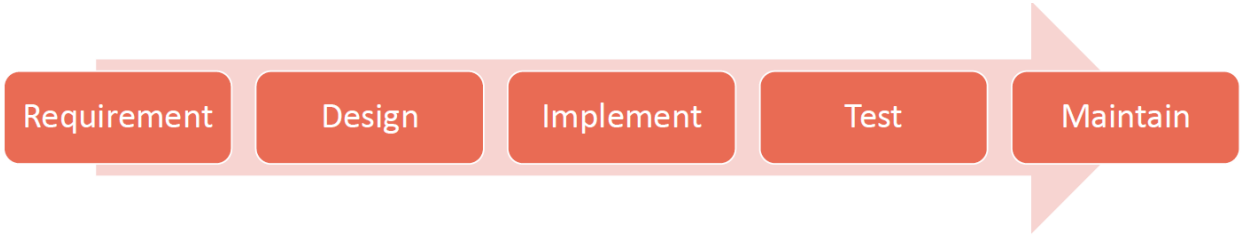
GANTT chart

A tool which can be used to achieve this goal is the GANTT chart, it can state how much the cost will be, how many hours are needed (or days) and the eventual team size.
An example of a GANTT chart could be seen below:

	Requires	Effort	Week1	Week2	Week3	Week4	Week5	Week6	Week7	Week8	Week9	Week10	Week11	Week12
Activity 1		5												
Activity 2	Activity 1	10												
Activity 3	Activity 2	5												
Activity 4	Activity 3	6												
Activity 5	Activity 2	...												
Activity 6	Activity 4	...												
Activity 7	Activity 5	...												
Activity 8	Activity 7	...												
Activity 9	Activity 8	...												
Activity 10	Act 4 & 7	...												

Waterfall chart

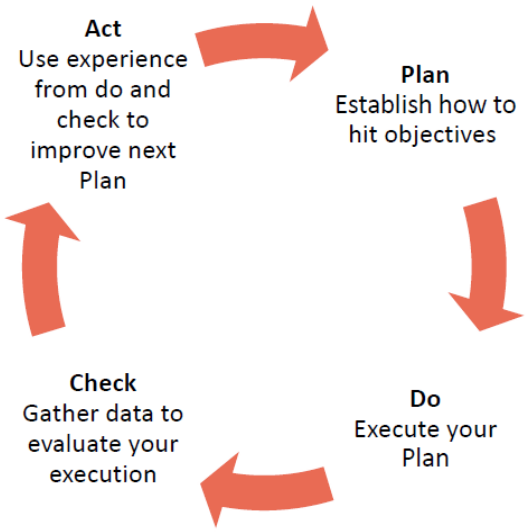
Another tool to manage a project is the waterfall chart:



It breaks down the project into linear and sequential phases where each one depends on the deliverables of the previous.
This dependence makes this chart less iterative and flexible with respect to the GANTT one.

Deming cycle

The last management method is the deming cycle:



Which brings to a methodology called “agile”.

Agile method

The agile method is divided into fixed periods of time called “sprints”, in each of them the team must release something that could be seen and used by customers.
Before the project starts, all the requirements are collected in a pre-sprints activity, the list of high level requirements is called “product backlog”.

There are various roles in an agile team:

- The product owner: in charge of prioritizing product backlog;
- The scrum master: in charge of ensuring agile rules are not broken;
- The agile team: in charge of developing the solution;
- The stakeholders: which evaluate the product.

Besides sprints, there are some other temporal window with a precise scope:

- Standup meetings: agile team, scrum master and product owner speak for about 10 minutes;
- Sprint review: at the end of each sprint summarize it with all the roles;
- Backlog review: the agile team, scrum master and product owner discuss about the backlog;
- Sprint retrospective: agile team and scrum master discuss the sprint just concluded.

Priorities are studied by the team and divided into smaller tasks if needed, when each is small enough to be developed in a sprint it becomes a “user story”.
Each component of the team commits for a set of stories each sprint and develops them.

CI / CD

Another way to approach the development of a system is the CI / CD method, the continuous integration - continuous delivery.
The CI focuses on producing small packages of software in a short time window, while the CD means releasing in production all the packages automatically.

Although the perfect GANTT would outperform a CI/CD approach, this in reality never happens and the CI/CD approach guarantees a better cost, time and risk management with more incremental updates on the software in production.

ELK Stack

The ELK stack, a tool for data ingestion, enrichment, storage and analysis consists of 3 technologies called:

- Elasticsearch;
- Logstash;
- Kibana.

In which case a read is needed?

- eCommerce: when we have to search in real time the availability of a product;
- Connected vehicles security: when we need to scan the logs to find anomalies.

There are millions of use cases in which a search is useful.

Introduction

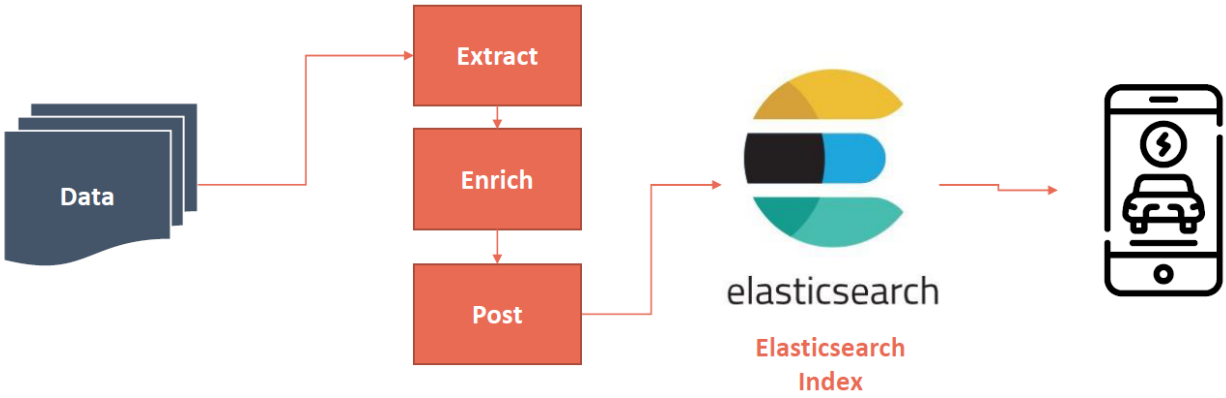
Elasticsearch aims to render these searches scalable and fast, with its abilities it can index many document types and it can be used for a large number of use cases:

- Application search,
- Website search,
- Logging and analytics,
- Geospatial data analysis,
- Etc..

But what is Elasticsearch?
It is a distributed, open source search and analytics engine for all types of data (structured and unstructured one).
The basis of this engine is Apache Lucene and it is known for the REST APIs it gives and it is the central component of the Elastic stack.

Raw data flows into Elasticsearch from a wide variety of sources, data is ingested by parser, normalizer and enricher tools before being indexed in Elasticsearch.
Once indexed, users can run complex queries against data using also aggregation to get summaries.
Kibana is the tool which lets the users create visualizations of their data and share dashboards.

General architecture



This high level overview demonstrates an architecture in which data is enriched at batch time and then indexed inside Elasticsearch.

Elasticsearch index

An Elasticsearch index is a collection of documents that are related to each other, data are stored in Json format and each document correlates a set of keys (names of fields or properties) with their corresponding values.

Elasticsearch index is a particular kind of data structure called “inverted index”, it is designed to allow fast full-text search.
An inverted index is a list of every unique word that appears in any document and identifies all of the documents each word occurs in. This index is served in a near real-time temporal window as soon as a document is uploaded through the index API.

Elasticsearch pills

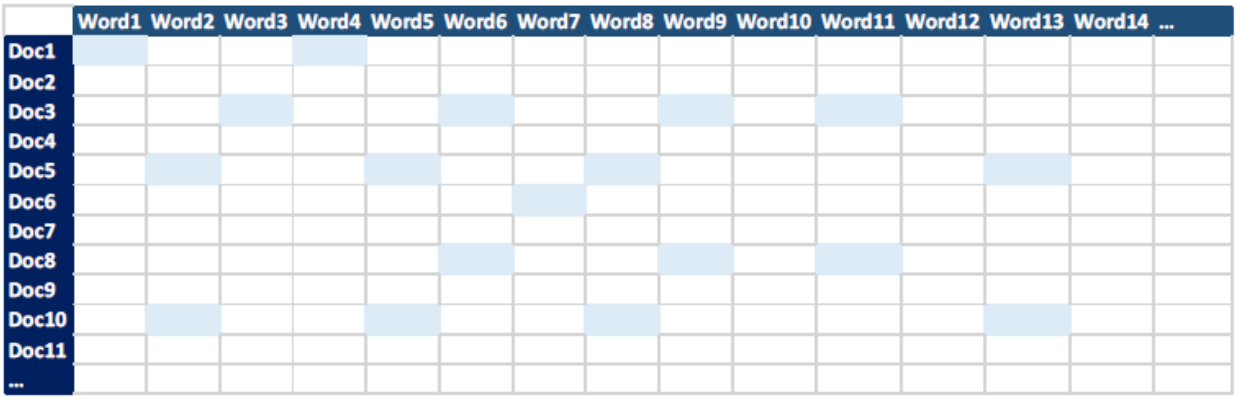
- It is built on top of Lucene, it excels at full-text search;
- It is a near real-time search platform, the latency between the document upload and indexing is typically one second;

- Documents are stored in containers called shards, these are duplicated to provide redundancy and increase affidability;
- The scalability and distributed nature of Elasticsearch allows it to manipulate and query petabytes of data in hundreds of shards.

Bag of words & Indexing

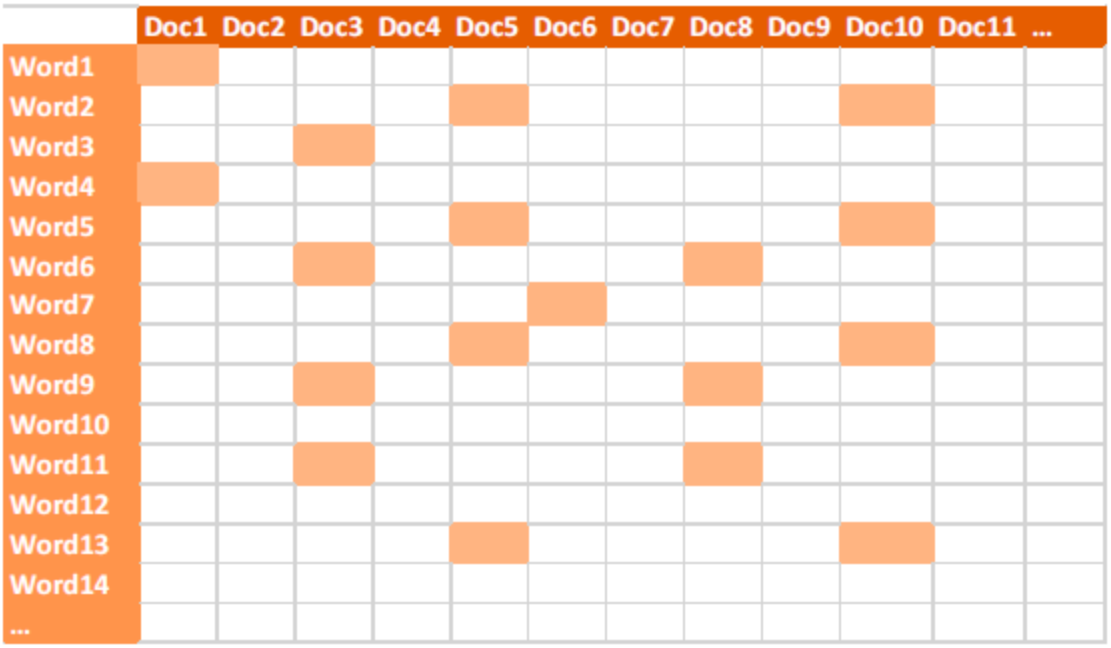
The inverted index is based on the bag of word concept.

In the bag of words a unique and ordered list (absolute dictionary) of all unique tokens inside a document, then each document is represented as a vector where each single value represents the amount of occurrences of that specific word inside the text. Normally we would have the whole structure represented as a matrix where the key is the document id, each row represents a document and each column the words occurred:



	Word1	Word2	Word3	Word4	Word5	Word6	Word7	Word8	Word9	Word10	Word11	Word12	Word13	Word14	...
Doc1															
Doc2															
Doc3															
Doc4															
Doc5															
Doc6															
Doc7															
Doc8															
Doc9															
Doc10															
Doc11															
...															

Whereas in the Elasticsearch inverted index we have the word id as a key, each word represents a row and the documents are now the columns. As a consequence I know in which documents a word occurs.



With this kind of index we can build what is called a “naive similarity” index, this index lets us analyze and check for phrases similarity in a “naive” way, what do we mean with naive? In case of synonyms or semantic differences among words, the result of the classification should change, this could be done with lemming and stemming and NLP analyzers, which are at disposal in Elasticsearch.

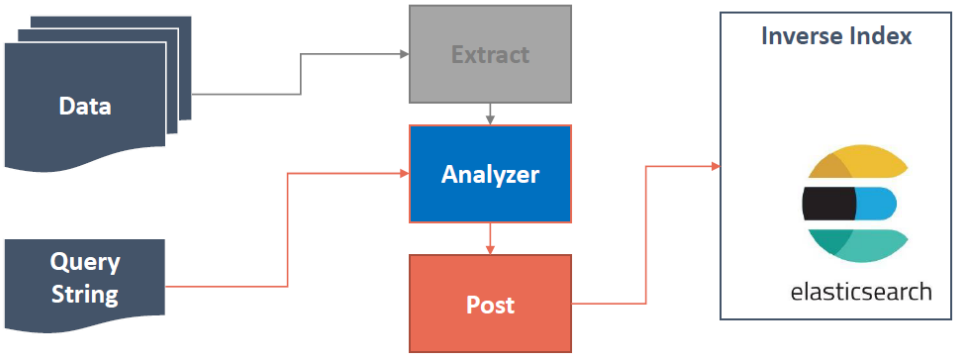
In particular, analyzers act this way:

- They first tokenize the block of text into individual terms suitable for the inverted index;
- Then they normalize these terms into a standard form.

The analysis is based on three strategies:

- Character filtering: tidy up the string before tokenization;
- Tokenizer: which split the text into tokens;
- Token filtering: which changes terms (lowercasing, etc), remove terms (stopwords, etc) and add terms (as synonyms etc).

The architecture of Elasticsearch now becomes:



Mapping

One of the Elasticsearch provided functions is the mapping: stated that everything inside Elasticsearch is an object, each of them is serialized in json format, this format doesn't actually allow strict typing (ex. A Date object is written as a String).

The mapping provides a way to attach information about type and additional data useful for type usage to a field.

elasticsearch

```
{
   "gb": {
      "mappings": {
         "tweet": {
            "properties": {
               "date": {
                  "type": "date",
                  "format": "strict_date_optional_time||epoch_millis"
               },
               "name": {
                  "type": "string"
               },
               "tweet": {
                  "type": "string"
               },
               "user_id": {
                  "type": "long"
               }
            }
         }
      }
   }
}
```

In order to exploit Elasticsearch capability each index has a custom type, every type has its own schema which defines the fields for a type and how it should be handled.
Elasticsearch tries to imply dynamically the json field type, for example String, Whole-Number, Float, Boolean, Date, etc. In case the type found in the JSON is not coherent with the schema, Elasticsearch will try a runtime conversion.
The type is defined using the “type” key.
For each field we can decide if Elasticsearch will analyze it or not, with index property set to “analyzed”, “not_analyzed”, “no”:

- In the first case are applied tokens, stopwords, etc and then the field is indexed,
- In the second case the field will be indexed with no analysis,
- In the third case the field will not be indexed.

Also, custom analyzer could be specified in order to process data put in Elasticsearch

Query

Querying implies the usage of a JSON as well:

```
GET /_search
{
    "query": {
        "match": {
            "tweet": "elasticsearch"
        }
    }
}
```

Inside a query we could have nested queries (called compound) as:

```
{
    "bool": {
        "must":     { "match": { "tweet": "elasticsearch" }},
        "must_not": { "match": { "name":  "mary" }},
        "should":   { "match": { "tweet": "full text" }},
        "filter":   { "range": { "age" : { "gt" : 30 }} }
    }
}
```

There are two types of DSL queries available:

- Query context (or scoring queries): aimed to assign to each document a score and to use that score to sort document as output;
- Filter context (or non-scoring queries): aimed to exclude some documents from the result without using a scoring strategy. These are the fastest as they are simple checks for inclusion/exclusions.

As a general rule, we should use the query clauses for full text search or for any condition that should affect scoring, use filters for everything else.

Examples of queries are:

- «match_all»: usually to display the result after a filter

```
{ "match_all": {}}
```

- «match»: a query against a specific field
  - On **analysed** fields same analyser is applied to the query as well
  - On **not_analysed** fields or **not string type** it will check for the exact value

```
{ "match": { "tweet": "About Search" }}
```

- «multi_match»: apply the same string search to multiple fields

```
{
    "multi_match": {
        "query":    "full text search",
        "fields":   [ "title", "body" ]
    }
}
```

- «range»: apply range search on some type
  - ‘gt’ == ‘>’
  - ‘gte’ == ‘>=’
  - …

```
{
    "range": {
        "age": {
            "gte":  20,
            "lt":   30
        }
    }
}
```

- «term» and «terms»: search for the exact value even in analysed fields

```
{ "term": { "age":    26           }}
{ "term": { "date":   "2014-09-01" }}
{ "term": { "public": true         }}
{ "term": { "tag":    "full_text"  }}
```

```
{ "terms": { "tag": [ "search", "full_text", "nosql" ] }}
```

- «existing» and «missing»: the same as **Is Not Null** and **Is Null**

Queries can be combined using several strategies:

1. **Must**
2. **Must not**
3. **Should** (increase the score only if match)
4. **Filter**

```
{
    "bool": {
        "must":     { "match": { "title": "how to make millions" }},
        "must_not": { "match": { "tag":   "spam" }},
        "should": [
            { "match": { "tag": "starred" }}
        ],
        "filter": {
          "range": { "date": { "gte": "2014-01-01" }} 1
        }
    }
}
```

- Moving “**date**” statement to “**filter**” will increase the Query execution performance
- This way the “**date**” statement will no more contribute on scoring (so result change as but the **score is not computed onnont-matching documents**)

- “**Sort**”: queries results can be sorted SQL like (field+asc/desc {…, “sort”:{“date”:{“order“:”asc”}}}

```
{
    "query" : {
        "bool" : {
            "must":   { "match": { "tweet": "manage text search" }},
            "filter" : { "term" : { "user_id" : 2 }}
        }
    },
    "sort": [
        { "date":   { "order": "desc" }},
        { "_score": { "order": "desc" }}
    ]
}
```

- Every time a query is executed, we can ask as well **to Lucene to provide details** about **how each field added value** to the final score
- This could be used both to **enhance user experience** and to **debug your query** (“Why I cannot see this document?” like complaints)

```
{
  "valid" :         true,
  "_shards" :       { ... },
  "explanations" : [ {
    "index" :       "us",
    "valid" :       true,
    "explanation" : "tweet:really tweet:powerful"
  }, {
    "index" :       "gb",
    "valid" :       true,
    "explanation" : "tweet:realli tweet:power"
  } ]
}
```

Inside the query, the TF-iDF is the algorithm used to state the relevance of a document for a search inside Elasticsearch, it tends to lower the score of common words in favour of rare ones which have a higher score.

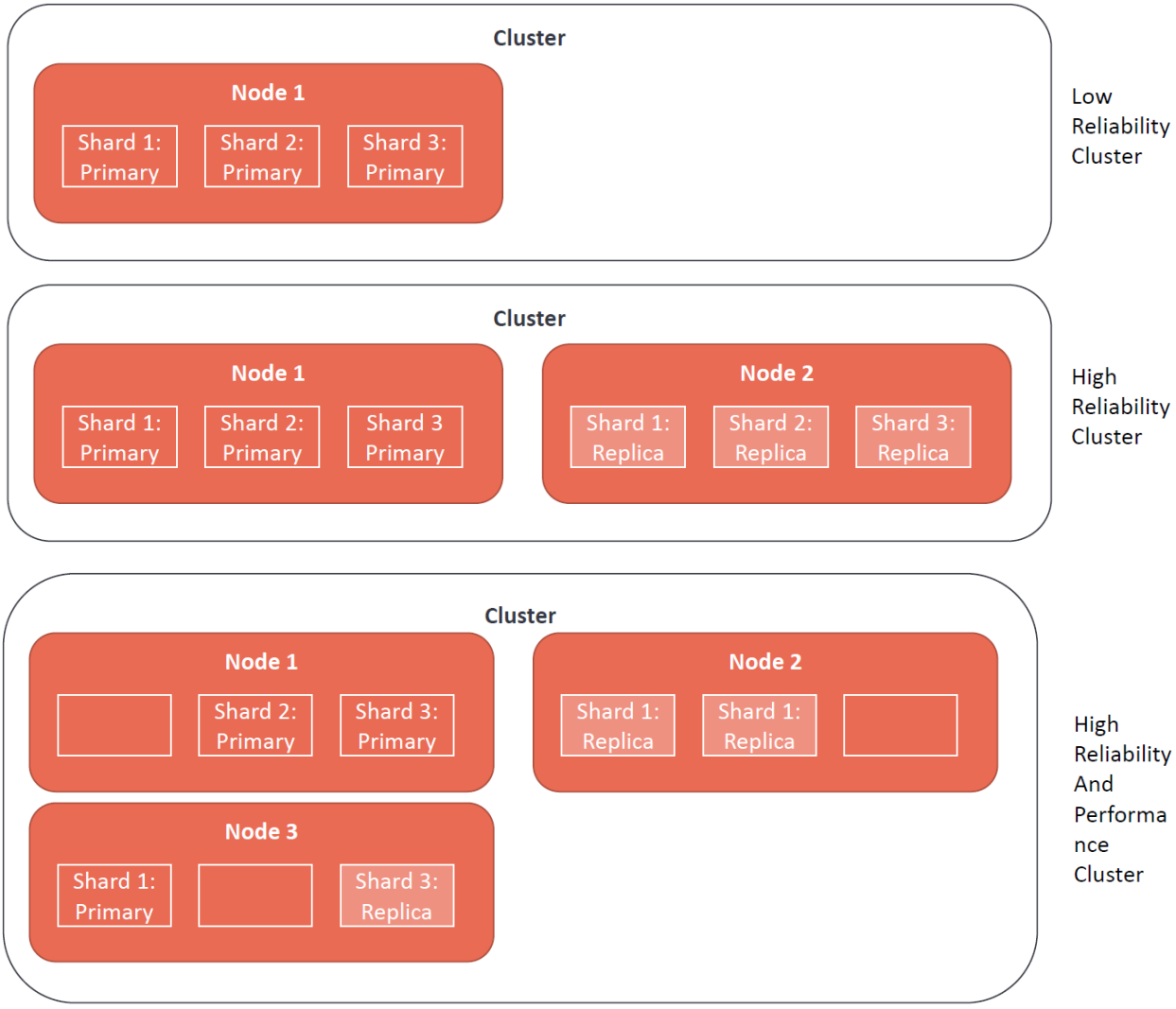
Scalability

Elasticsearch in based on storage object pattern and the horizontal scalability is granted by design, the architecture is similar to the Hadoop one:

- 1 master node elected automatically which is in charge of general level operations (creates index, manages resources, etc)
- Every node knows where each document lives and can forward requests directly to the nodes that hold the have needed data (name node is no more unique)
- Everything is arranged in shards (similar to data node)
- Shard can be primary or simple replicas
- Multiple shards are put inside each node.

A shard is a fully fledged search engine on its own and can use all the resources of a single node, replicas can also be used to improve querying performances and query computation.

In the subsequent pictures we can see three different architectures of Elasticsearch cluster:



Project

Contained in the Spark hands on 011 lesson.