Enterprise Architecture

1. Describe the link between building architecture and software architecture? What are common concepts between these two subjects?

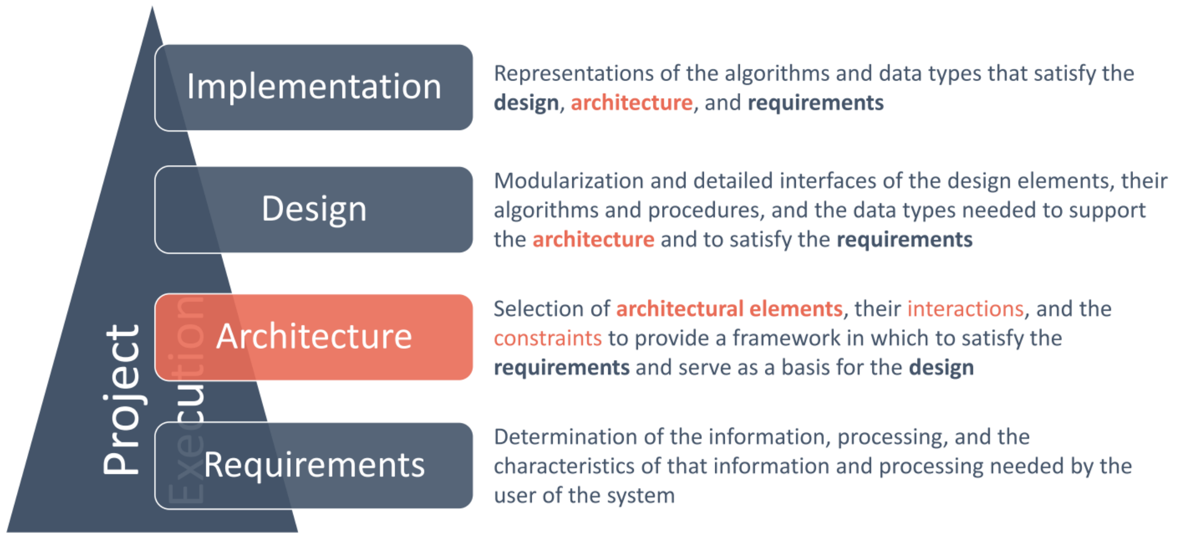
Software architecture principles can be inherited by appealing to several well-established architectural disciplines. While the main view of each one is different, there are a number of interesting architectural points in building architecture that are suggestive for software architecture:

- Multiple views: each aspect of a building require different consideration. For example, the stability needs some consideration different from price aspect! There are different stakeholders which each one has different accountability. In the software there is similar condition. What a full stack care and is focused on is different from what a data scientist focuses on.
- Architectural style: each building may have different architecture style in comparison to other ones. The purpose of each building may be different from the others. One building may be built for being huge and big but the other may be built for purpose of beauty. Based on the purpose each one has a different architecture style. In software we also have something similar. In software we have lots of architecture styles. For example, microservice or service oriented, message oriented, etc.
- Style and materials: in building materials are obvious: woods, stone, cement, etc. and in software, the technology, programing language, subsystem, etc.

2. Define what "Software Architecture" mean

A software architecture is a set of architectural elements that have a particular form. The architectural form consists of weighted properties and relationships. An underlying, but integral, part of an architecture is the rationale for the various choices made in defining an architecture. The software architecture requires new kinds of abstractions that capture essential properties of major subsystems and the ways they interact.

3. The Project Execution pyramid (slide 23 - Lesson 002 – Architecture 101)



Project Execution
Implementation
Representations of the algorithms and data types that satisfy the design, architecture, and requirements
Design
Modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements
Architecture
Selection of architectural elements, their interactions, and the constraints to provide a framework in which to satisfy the requirements and serve as a basis for the design
Requirements
Determination of the information, processing, and the characteristics of that information and processing needed by the user of the system

4. What is an abstraction? How an abstraction is created? How can we use this concept when we try to design an Architecture?

The abstraction is between architecture designer and developer. Developer use it to implement the architecture.
We create abstract classes to force the other to follow the way we designed. By abstract classes the developer must implement all methods that are specified so we are sure he would follow the architecture.
"The development of individual abstractions often follows a common pattern:
First, problems are solved ad hoc.
Then as experience accumulates, some solutions turn out to work better than others, and a sort of folklore is passed informally from person to person. Eventually the useful solutions are understood more systematically, and they are codified and analyzed. This in turn enables a more sophisticated level of practice and allows us to tackle harder problems."

5. *** What are the main pillars of any Software Architectures? Describe each of them with practical examples to show the added value**

practical examples to show the added value.
The main pillars of software architecture are:

1. **Being the framework for satisfying requirements**: given that it is based on the Requirements in the Project Execution pyramid the architecture should be able to satisfy them or to make them satisfiable in the design and implementation. For example, if in the requirements there is the need to process streaming of data, the architecture must contain a component that processes them.
2. **Being the technical basis for design**: create the right modularization and design of element's interfaces, algorithms, procedures and data types needed to satisfy the requirements. Kafka
3. **Being the managerial basis for cost estimation and process management**: looking at the design of the architecture each component must be made explicit and the decision over which implementation is chosen for each component is well represented. From these decisions it is possible to obtain, for each component, a forecast over its cost (the cost of implementing it and for using and maintaining it).
4. **Enabling component reuse**: instead of creating a single purpose function to read from a database, it is better to create a module that permits access to data in the correct way. This component will be helpful to a lot of other parts, not only for the already designed ones.
5. **Focus on centralization**: single data saving point so that data are not inconsistent, there is a single source of truth and all the components can access data in the same way, it is not needed more than one component to correctly access data. Use a data lake (Azure, AWS...) to save raw data and then elaborate them, to save versions etc....
6. **Enhancing productivity and security**: the architecture must be designed also according to the security perspective adding security components (DMZ) avoiding well known attacks to all, for example trying to hide the logical structure of the architecture, all without compromising the efficiency of the entire system.
7. **Enabling enterprise systems integration (Enterprise Application Integration):**
8. **Allowing a tidy scalability:**
9. **Controlling software processes execution:**
10. **Avoiding handover and people lock-in:** a good architecture must ensure a clear vision over the project and allow the team to easily transmit the knowledge about it. The knowledge over certain parts cannot be maintained only by a single/some individual(s), otherwise the company is forced to maintain who is able to handle the obscure component.

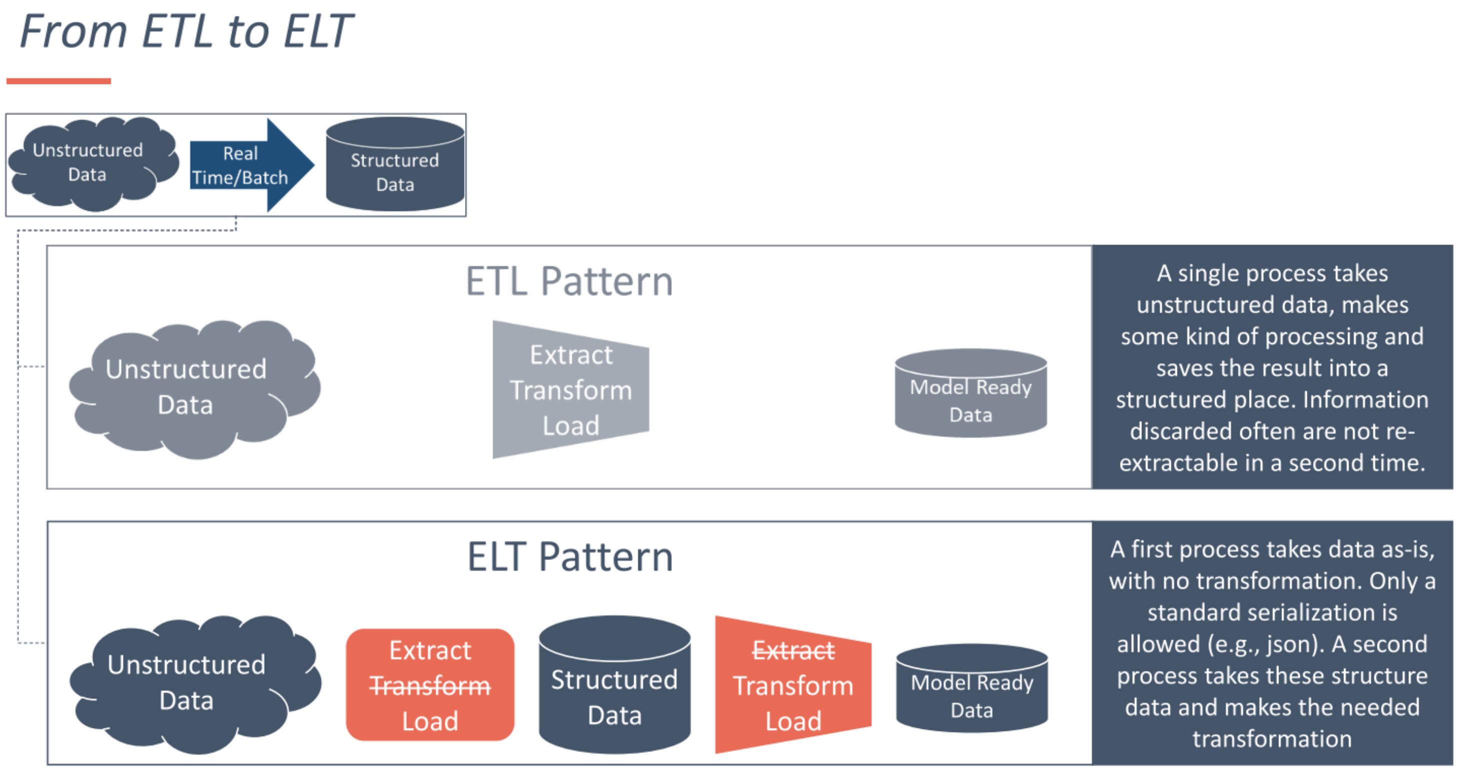
execution pyramid → based on requirements and basis for design
managerial → basis for estimating costs & avoid lock-in & EAI
security & control process execution
architectural components → enable component reuse & focus on centralization & scalability

6. What is a design pattern?
   Design pattern is like a piece of architecture. You solve a common problem once and then; everyone can reuse your solution. In the other word, design patterns are formalized best practices found to solve common problems.

Design Patterns are formalized best practices found to solve common problems. They are divided in classes by the kind of problem that they try to solve, for example:

- Creational: Creating an object rather than instantiating it directly. They solve problems related to the creation of objects. Example: Singletons
- Structural: Using inheritance to compose new objects. They solve problems related to the structure of objects. Example: Decorator (it allows adding methods to pre existing classes at run time).
- Behavioral: Defining how objects can communicate. Example: Mediator that acts like man-in-the-middle between the objects' communications

7. ETL pattern and evolutions based on Data Lake



A single process takes unstructured data, makes some kind of processing and saves the result into a structured place. Information discarded often are not re-extractable in a second time.

A first process takes data as-is, with no transformation. Only a standard serialization is allowed (e.g., json). A second process takes these structure data and makes the needed transformation

In the basic ETL pattern data are taken from the various sources so they are heterogeneous and unstructured data, after the transformation where data are structured to be loaded into the data warehouse. The evolution with the Data Lake is the ELT where there is a first step in which data are extracted and only a standard serialization is allowed (e.g., json) then saved into the Data Lake. Then data are structured and loaded into a data warehouse.

8. **How can we implement the CDC pattern? Pro(s) and Con(s) of each approach**

   Change Data Capture (CDC) is a set of design patterns used to determine (and track) the data that has changed so that action can be taken using the changed data.
   Traditional Change Data Capture patterns works in this way that at the first time it scans all of the table. Then for subsequent times it stores a file let's call it changed file which just contains what has been changed. But how we can capture what data has been changed and store it in changed file?
   There are several traditional approaches, like:

1) Timestamps on rows: for log table is ok because it already has. But for registry table it is not easily possible. Because adding a column to table is not that easy specially if we are not DB admin. Further more it can easily be bypassed be developer (if we rely on developer to insert it not using trigger which in large scale triggers are not suitable). Also, it can slow down the CRUD operations.

2) Version numbers on rows based: we don't use version number for log table. So, it is ok. But for registry tables it requires complex and nested query to find out the last version of each row. Beside adding new column is not easily possible in industry. Also, it can be easily bypassed by developer

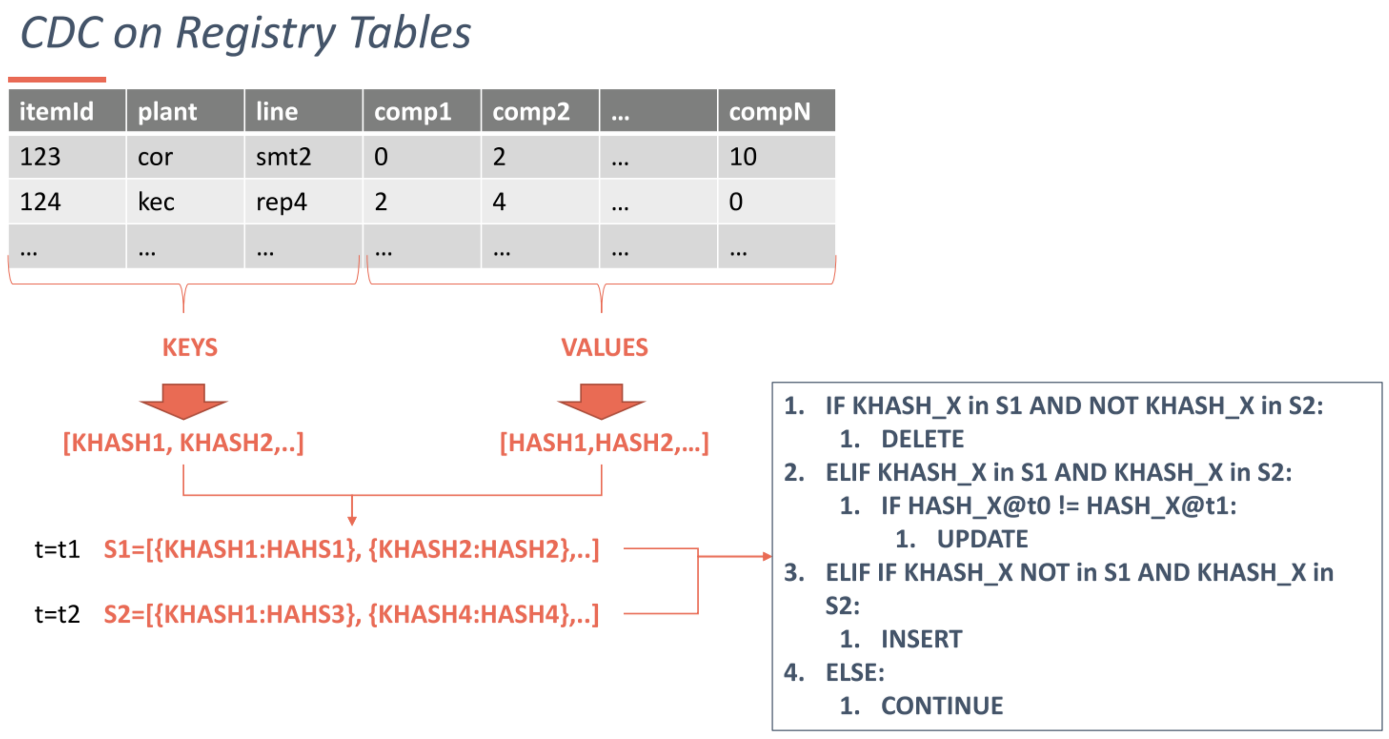
3) Status indicators on rows: this method is almost not used at all because:
what if the ETL is interrupted in the middle?
who is in charge of resetting the Updated status?
what if multiple jobs want to apply CDC on that table?
Again, adding a new column is not that easy if not DB admin! And also, it could be avoided.

4) Triggers on tables: honestly from theorical aspect triggers can solve all problems but in practice: we can't add trigger unless be DB admin. The triggers have performance issues and also, they are not easy to control.

Also, there are some other approaches which are better than previous ones:
Invasive Application-side:
• Event programming Give to the application the responsibility to propagate changes. Each time it commits something, the same operation must be sent also to the CDC system

Invasive Database CPU-side
• Transaction log scanners Read and process some Database technical logs (i.e., transaction logs) to imply changes and to propagate them on CDC system
• Log Shipping Use an automatic DB backup process to propagate changes. This tool is mainly used as a Disaster Recovery solution.

Today solution we use:



CDC on Registry Tables

itemId	plant	line	comp1	comp2	...	compN
123	cor	smt2	0	2	...	10
124	kec	rep4	2	4	...	0
...	...	...	...	...	...	...

t=t1 S1=[{KHASH1:HAHS1}, {KHASH2:HASH2},..]

t=t2 S2=[{KHASH1:HAHS3}, {KHASH4:HASH4},..]

```
1. IF KHASH_X in S1 AND NOT KHASH_X in S2:
    1. DELETE
2. ELIF KHASH_X in S1 AND KHASH_X in S2:
    1. IF HASH_X@t0 != HASH_X@t1:
        1. UPDATE
3. ELIF IF KHASH_X NOT in S1 AND KHASH_X in S2:
    1. INSERT
4. ELSE:
    1. CONTINUE
```

9. CDC, stateful/stateless and transactionality

Change Data Capture (CDC) is a set of design patterns used to determine and keep track of data that has changed, so that actions can be performed using the modified data.
By stateful we mean the ability of a system to remember previous user events or interactions. While, stateless refers to the ability of a system to always respond in the same way regardless of any previous state.

CDC can be both stateful and stateless but it better be stateless.
The CDC pattern we have seen is stateless as it is being read externally and each time it is run, the CDC takes the sync.json file and updates it. There is therefore no history of "past states".

Can CDC be transactional? Transactional means that it respects the ACID properties (Atomicity, Consistency, Isolation and Durability), in other words that it can be committed and rolled back. The incremental pattern can fail, since inserting the rows into the database and updating the sync.json file is done in parallel.

If the insertion of the lines fails (common for the insertion of large quantities of lines), however, the update of the sync.json file is done anyway. To solve this type of problem, you need to make CDC become transactional. To do this, when we connect to SQL we clean all the ".tmp" files of the Data Lake and
subsequently, after taking the new lines with "Diff & Where", we sequentially insert the lines into the Data Lake and update the sync.json file. This pattern is called Move and Rename and is based
on the use of temporary names, on the renaming of files in one go at the end of the writing process and on a final cleaning step to be performed at the beginning of every "job".

10. Differences between log and registry when we try to implement the CDC pattern

A Log Data table is a table that records events • Because of the past cannot be undone, a log table allows INSERT only transactions. A Registry Data Table describe entities could change over time. Because of this nature, all the 4 main verbs (SELECT, INSERT, UPDATE, DELETE) can be used on it. E.g., the full list of material needed to produce something – called Bill of Material (BOM). In registry table we don't have history just status of the given moment.

11. What could be used the wrapper pattern for? Why?

In wrapper pattern we have an object which is inside of another object. No one access to insider object and rest of the architecture can only reach the external one. For example, the inside object can be a data lake provided by Microsoft. Now instead of everyone be able to access to this data lake directly, we create a data lake wrapper and give them access to this object. Now if for any reason we change the inside object the rest of architecture remains as it was. The wrapper can also hide some methods to avoid application use them!

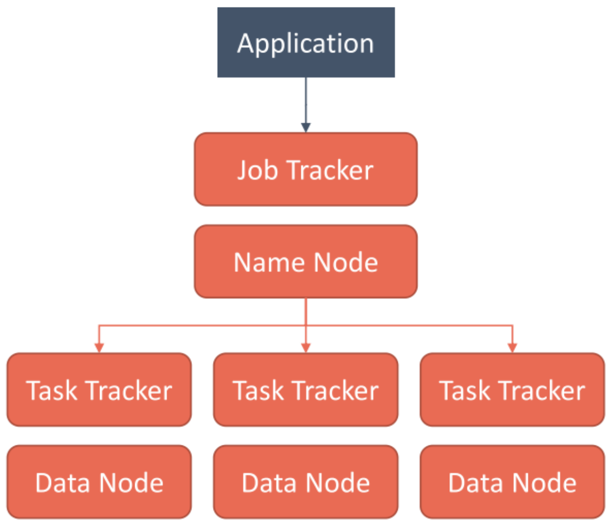
Distribute Computation

12. What is Map Reduce? When was it first formalised?

MapReduce is a programming model proposed in a Google paper. The idea of map reduce is that moving computation is easier than moving the data. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key
Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines
Inputs and operations over inputs are processed in parallel by different machines using a partitioning function

13. Hadoop Architecture

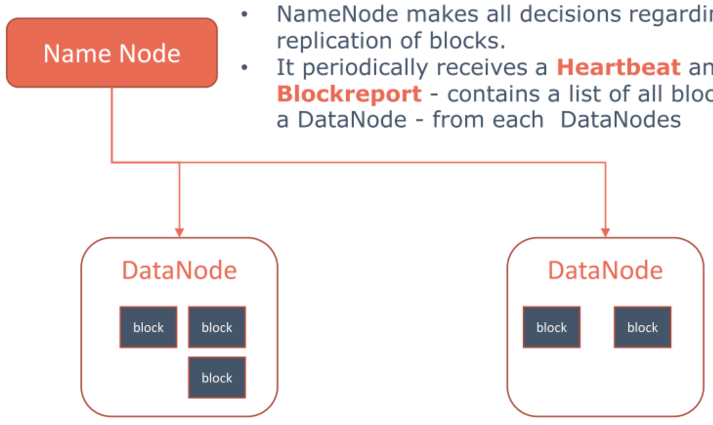
Hadoop Architecture - Sequence



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.

14. HDFS Architecture

Hadoop Architecture – HDFS



- NameNode makes all decisions regarding replication of blocks.
- It periodically receives a **Heartbeat** and a **Blockreport** - contains a list of all blocks on a DataNode - from each DataNodes

- Each file is stored as a **sequence of same sized blocks**, each block is replicated for fault tolerance.
- Files in HDFS are write-once (except for appends and truncates) and have **strictly one writer at any time**.
- DataNode **actively send information** to Name Node about blocks

- HDFS has **a master/slaves** architecture: 1NameNode per n Datanodes (1 per cluster).
- HDFS **exposes a logical unique file system namespace** and allows user data to be stored in files.
- Each file is **split into one or more blocks** and distributed in a set of DataNodes.
- The **NameNode executes file system namespace operations** like opening, closing, and renaming files and directories.
- The **DataNodes are responsible for serving** read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.
- **Data never flows** through the NameNode.

HDFS is a distributed file system designed to run on basic hardware.

• HDFS is highly fault tolerant and is designed to be deployed on low-cost hardware.

• HDFS provides high-throughput access to application data and is suitable for applications with large data sets.

Architecture:

• HDFS has a master / slave architecture: 1 NameNode per n Datanode (1 per cluster).

• HDFS exposes a unique logical namespace of the file system and allows you to store user data in files.

• Each file is divided into one or more blocks and distributed in a whole
by DataNode.

• The NameNode performs file system namespace operations such as opening, closing and renaming files and directories.

• DataNodes are responsible for handling read and write requests from file system clients. The DataNodes also perform the creation, deletion and replication of blocks on the instruction of the NameNode.

• The data never passes through the NameNode.

15. Apache Spark main data types, laziness, and Shared Variables

The first Data Abstraction of Apache Spark is the Resilient Distributed Dataset (RDD)
Spark is lazy: it does not compute their results right away. The computations happen only when the driver requires a result to be shown. Each transformation is recomputed each time a result is needed. However, transformation result may also be persisted in memory using . persist() method. Only the part of computation needed to show the required results are done (e.g., no tables can if the first element is needed).
By default, when Spark distribute a function, it ships a copy of each variable used in the function once per row. This could lead to performance issue if the variables are big size maps or other complex structures.
• Spark supports two types of shared variables: broadcast variables, which can be used to distribute a read-only values once on all nodes, and accumulators, which can be used to distribute a write-only variables.
• Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.
• Spark actions are executed through a set of stages, separated by distributed “shuffle” operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

16. **How does Apache Spark use Hadoop elements? Apache Spark Architecture**

Spark architecture is composed of the following elements:

- Driver: the driver is a process responsible for the execution of the Spark application as well as for converting a user application into smaller execution units called tasks and then scheduling them. It holds the SparkContext. Like NameNode, it stores metadata about all the Resilient Distributed Databases.
- SparkContext: it is a singleton object created inside the driver and it allows a Spark Driver to use the cluster and call it through a Resource Manager. It is the only way through which it is possible to create

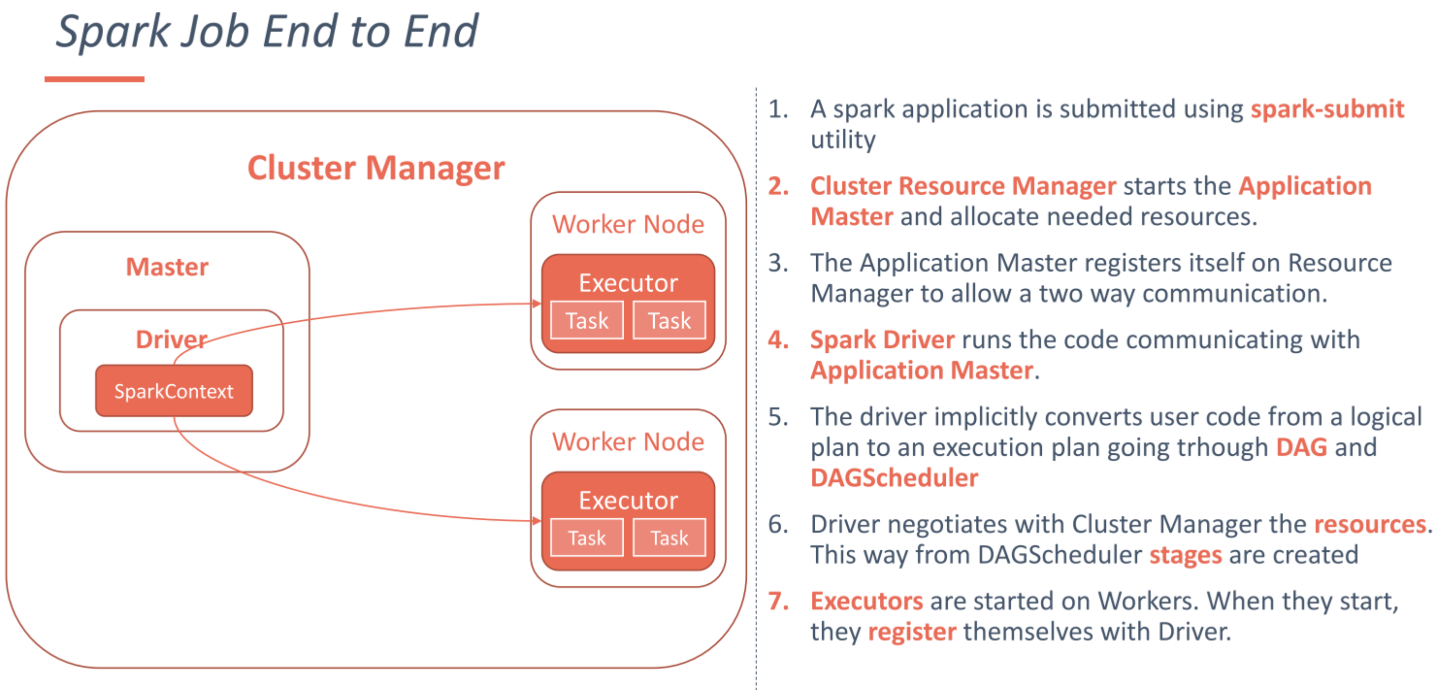
Spark types like RDDs, Broadcast and Accumulators. It is also in charge of keeping track of the executor status using heartbeat.

- Master: the master is a unique entity per application that is responsible for negotiating resources with the ResourceManager and works with the NodeManager to monitor the component tasks. It is created on the same node of the Driver.
- Worker: the worker is a Spark computation node where executors execute the tasks assigned. It uses a local Block Manager to communicate with other workers. When a SparkContext is created, each worker starts an executor as a separate process (JVM), and connects back to the driver program to receive serialized code.
- Executor: it is the agent responsible for the execution of tasks. If any of the executors fail the DAG, its tasks are moved to another executor, ensuring the resilience of the RDDs. Executors are also used to provide in-memory storage for RDDs when the user calls the persist() or cache() method.

17. Assumption(s) on a transformation distributed over Apache Spark

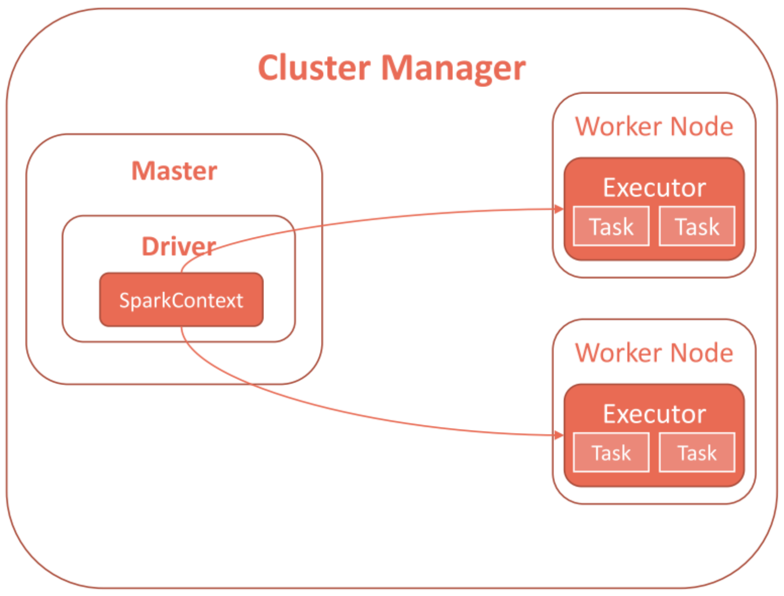
A transformation creates a new dataset from an existing one No processing is done after a transformation is called.
A transformation is an operation of Spark that, given its lazy behavior, does not return data but another RDD and adds on the DAG a new node corresponding to the transformation itself. One does not know if the transformation generates any exceptions.

18. Describe a Spark Job end to end



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.

Spark Job End to End



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**

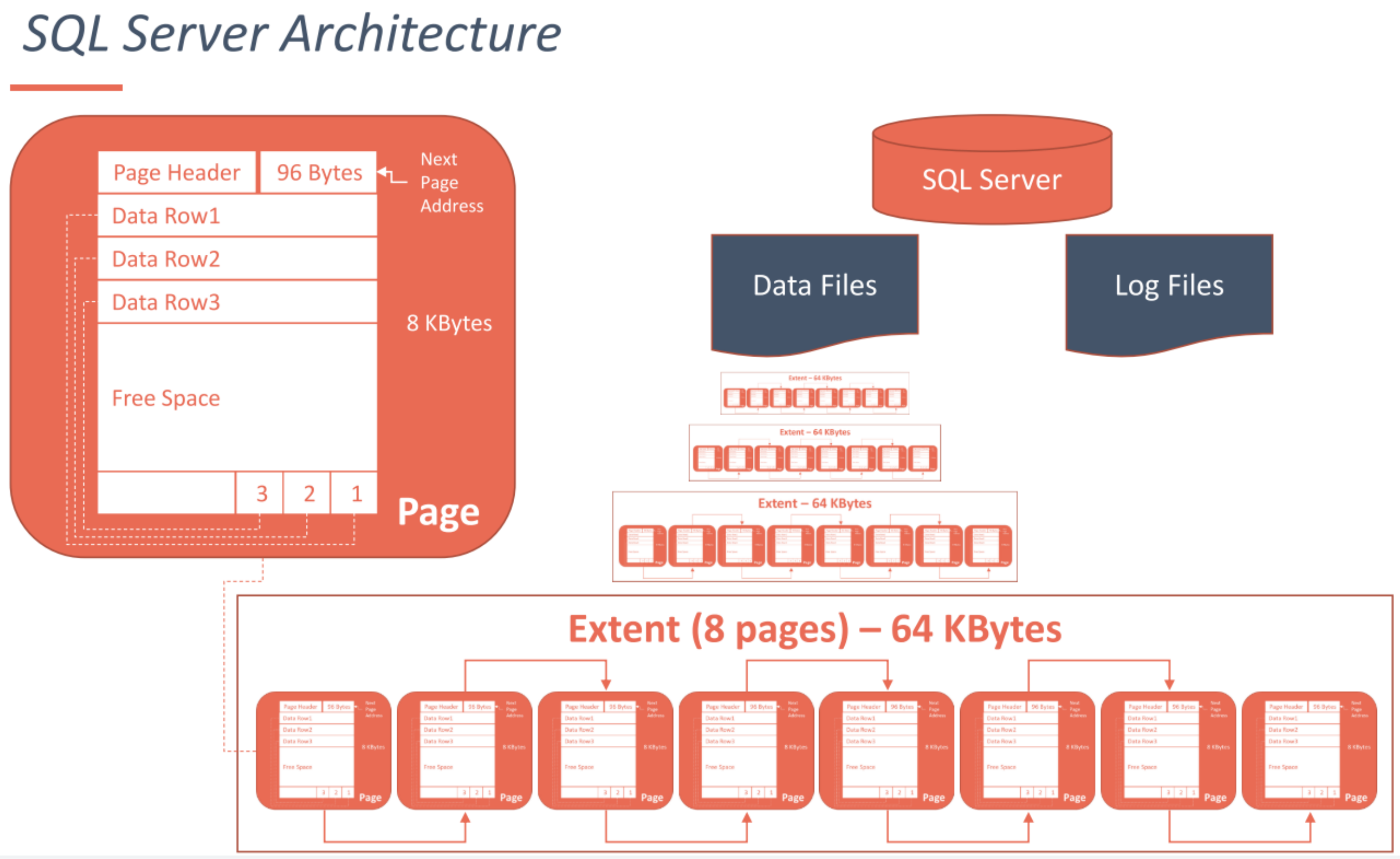
Set up phase: A spark application is submitted and the resource manager starts the Master allocating its resources; the latter one registers itself to the resource manager so two-way communication is now allowed.
Driver phase: The Driver runs the code transforming the logical plan into the execution plan (DAG → DAG scheduler), negotiating resources with the resource manager so that stages (sub component of the node defined in the DAG) can be created.
Execution phase: workers create executors and communicate with the Driver so that it can keep track of their status. The Driver sends tasks to the Cluster Manager based on the data placement and the Application Master launches the container by providing the Node Manager with a container configuration.
The first RDD is created by reading the data from HDFS.
During the execution, the Driver communicates with the Master to obtain the status of the application.
Closing phase: when the application has been completed, the Application Master deregisters from the Resource Manager and frees all the resources.

19. What is a transaction? What is an Index? How physically is SQL organised?

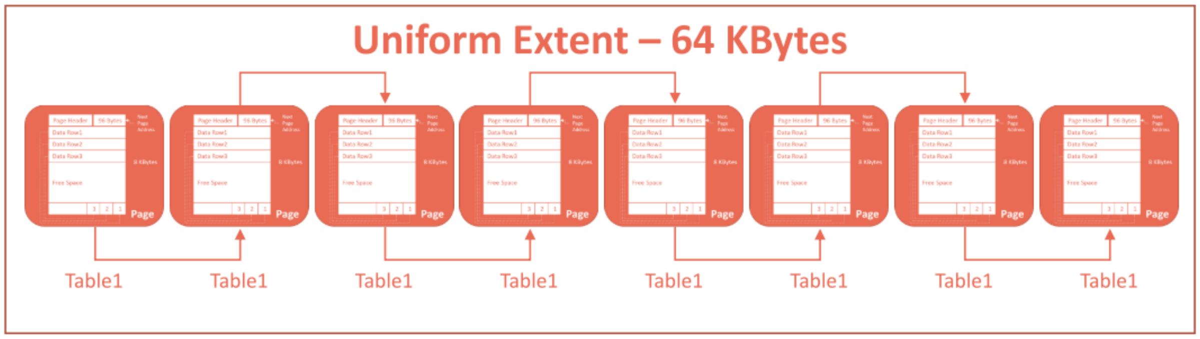
A transaction is an operation on a database that respects the ACID properties (Atomicity, Consistency, Isolation and Durability).

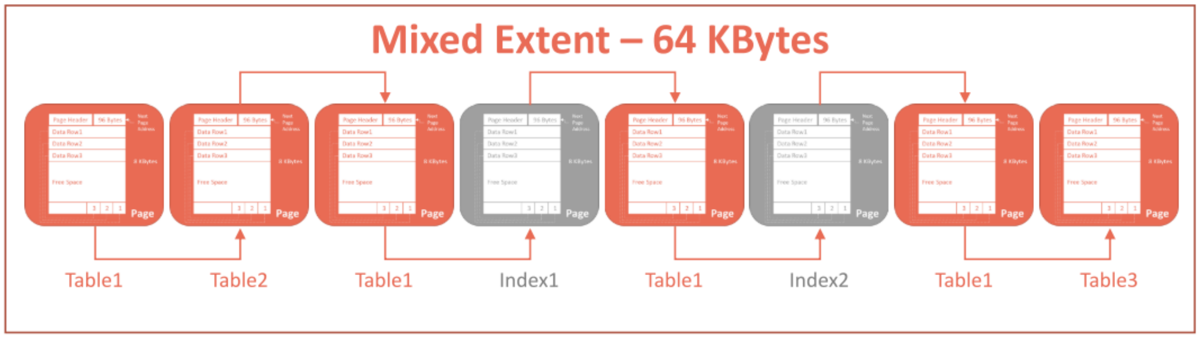
A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.
• Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed.
• Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Physically, SQL is organized into Data files and Log Files. The Data Files are
made up of pages, each page has a header, the next page address
, the actual data (organized in data rows), free space and
pointers to individual data rows. These pages are organized in Extent,
or groups of 8 pages for a total of 64 kBytes. The Extents
they can be of two types: Uniform if they contain only data pages or Mixed
if in addition to the data they also have tables with indices.



SQL Server Architecture





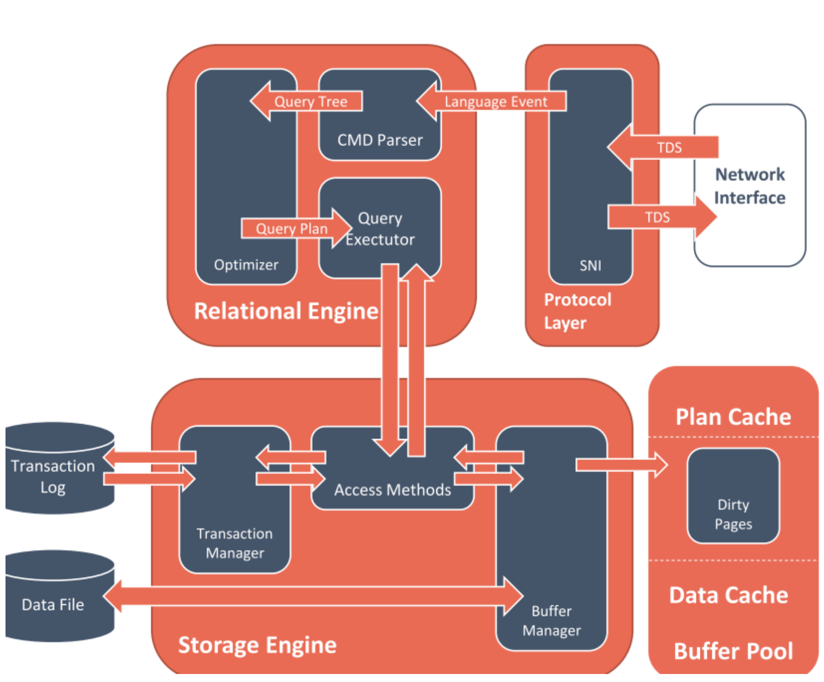
A transaction can be described as an atomic unit of work composed of several operations that respects the ACID properties:

- Atomicity: this property ensures that execution of a transaction does not leave the system in an intermediate state so the operations inside of it are executed as a whole.
- Consistency: this property ensures that the transaction does not violate any constraint defined over the data.
- Isolation: this property ensures the independence of the transactions during their parallel execution so that each transaction is invisible to the others.
- Durability: if the transaction ends well the changes carried must be made permanent in the database.

An index is a structure used to enhance the performance in accessing data: RDBMs normally generate indexes using the PK of the table but it is also possible to generate custom ones. Usually indexes are B trees or B+ trees.

20. Describe what happens when you try to execute a SQL Statements

SQL Statement Execution



SQL Statement Execution

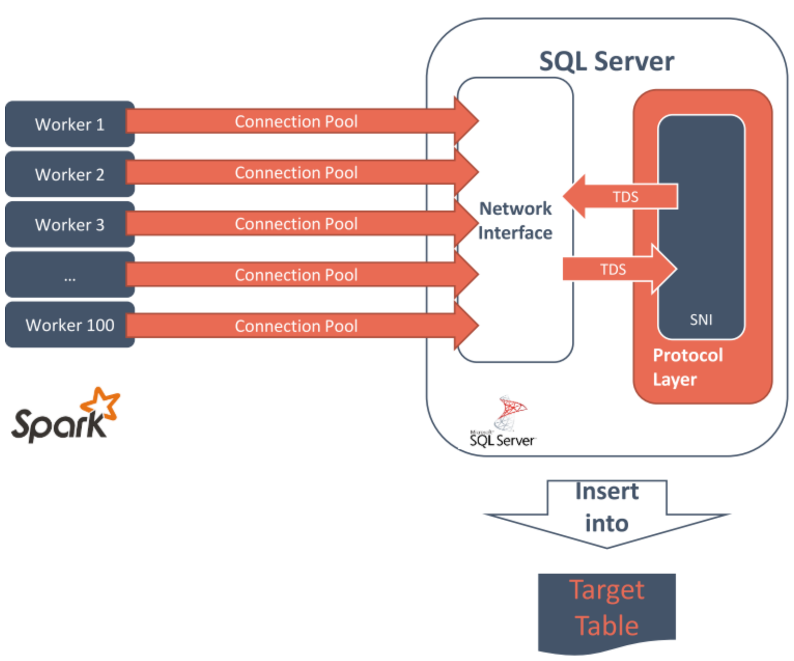
1. SQL query is converted into **TDS** (Tabular Data Stream) Packets using **ODBC** or **OLEDB** or **SNAC** (SQL Native Client) or **SNI** (SQL Server Network Interface).
2. Once TDS packets reach the server endpoint, the SNI **decapsulates** those packets into SQL commands and pass to **Query Parser**.
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

- Protocol layer: it receives the requests' packets from the network then unpacks and converts them into the tabular data stream.
- Relational engine: this part is in charge of parsing the queries (CMD parser), optimizing them (query optimizer) and executing them (query executor).
- Storage engine: the query executor asks for data to the access methods that retrieves them physically and passes them to the executor. If the query is not a select operation it is necessary to invoke the transaction manager that will manage and keep track of the transaction (log and lock managers).
  For performance reasons, a buffer is used to write temporary data; the buffer manager decides when to write them on disk.

The Log Manager is in charge of keeping track of the execution of each transaction. This is essential to the system in order to rollback transactions or recover from failures.
The lock manager ensures that DB's objects are accessed by only a transaction at a time, ensuring consistency and isolation.

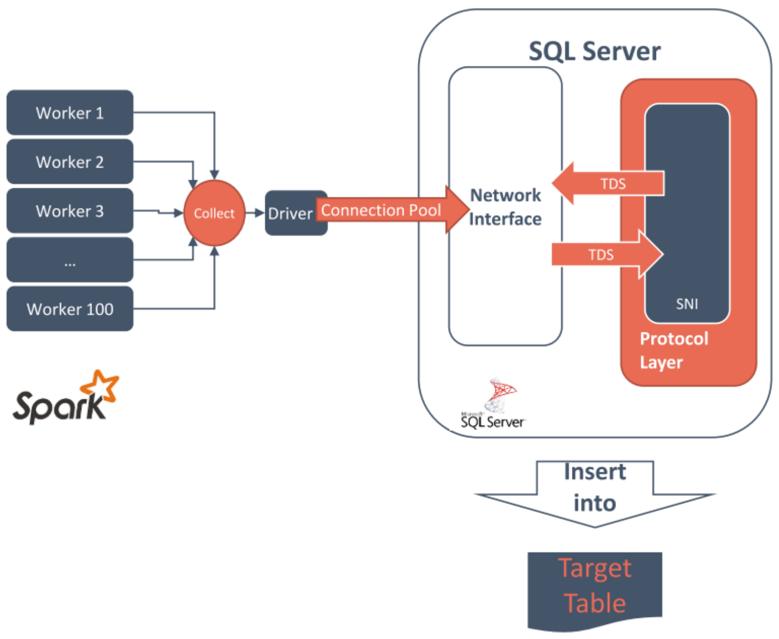
21. How to write from Apache Spark? Pro and Con(s)

*How **not** to write – Directly from Spark Workers*



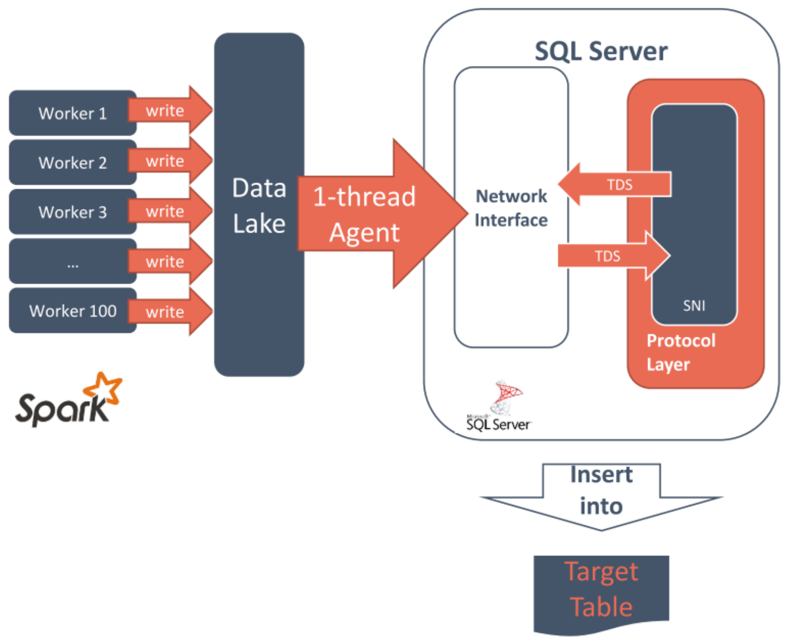
1. **Every Worker** will create a Connection Pool
2. Connection Pools will be **kept open** till all workers finish
3. A lof of **Dirty Pages** are created during the insert operation
4. Table is lock – inserting on an indexed table is a **lock transaction**
5. [*If you are using an elastic Spark Cluster*] you are paying 100 VM with a double 1 to 1 VM **bottleneck**

*How **not** to write – Directly from Driver*



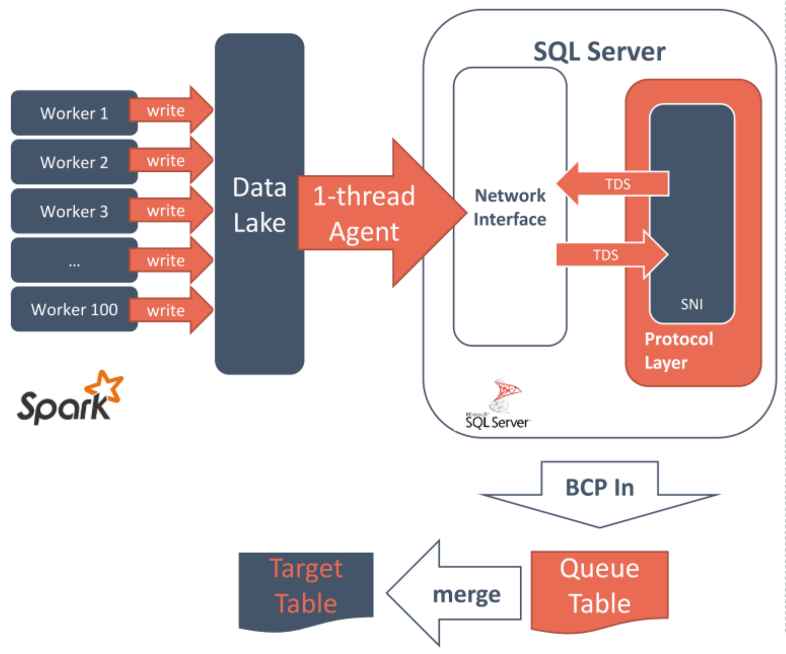
1. Driver makes a **global collect** to get data from each worker
   - Is **time consuming**
   - It must be able to keep in memory the whole dataset to avoid
     - **Disk spilling**
     - **Out of Memory**
2. **Driver opens** a single Connection Pool
3. A lof of **Dirty Pages** are created during 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**

How **not** to write – Data Lake Queue



1. Every Worker will write directly into **Data Lake queue – max speed**
2. Monothread 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**~~

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

22. Spark Dataframe

A Dataframe is a Dataset organized in columns with a name. It is conceptually equivalent to a table in a relational database or to a dataframe in R / Python, but has a greater wealth of optimizations.
Dataframes can be built from a wide variety of resources, including:
structured data files, Hive tables, external databases, or existing RDDs.
A dataset is a distributed collection of data that combines the RDD's benefits (strong typing, ability to use lambda functions), with the benefits of Spark SQL's optimized execution engine. A dataset can be constructed from objects and be manipulated by functional transformations (map, reduce...)

A dataframe is a dataset organized in columns (conceptually equivalent to a relational table) very optimized. Dataframes can be constructed from a lot of different structured data (structured data files, RDDs, tables…)
The pros of the dataframe structure are the custom view and structure, the execution is internally optimized (with RDD the optimization must be done by the programmer), it is possible use SQL-like expressions distributing lambda functions over rows (with RDD is possible distribute lambda functions over objects).
Dataframes offer a higher and facilitated interaction with the data, but they take away some freedom in their manipulation.

Distribute Processes

23. What is a SOA? How can you map SOA with Architectural Principles?

Service-Oriented Architecture (SOA) is an architectural style for creating an enterprise IT architecture that exploits the principles of service-orientation to achieve a tighter relationship between the business and the information systems that support the business.
Service Oriented Architecture is an architectural style that exploits the concept of "service" to create a structure suited to increase the interoperability among different systems and it allows the use of the single applications (services) as Blackbox components with specified outcomes.
A service:

- Is a logical representation of a repeatable activity that has a specified outcome
- Is self-contained: it means that it is fully independent
- May be composed of other services
- Is a "black box" to consumers of the service

24. During the class we discussed several times about differences between Services and uServices: can you give an example to show differences?

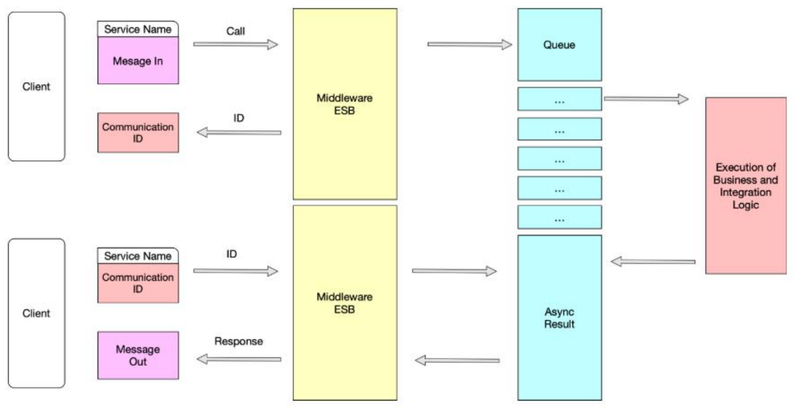
25. Service Definition, Service Queue, Service Broker

Services are obviously at the heart of Service-oriented architecture, and the term service is widely used. "A service is a discoverable resource that executes a repeatable task, and is described by an externalized service specification. "It is based on: • Business alignment: Services are not based on IT capabilities, but on what the business needs. • Specifications: Services are self-contained and described in terms of interfaces, operations, semantics, dynamic behaviors, policies, and qualities of service.
• Reusability: Services reusability is supported by services granularity design decisions. • Agreements: Services agreements are between entities, namely services providers and consumers. These agreements are based on services specification and not implementation.
• Hosting and discoverability: As they go through their life cycle, services are hosted and discoverable, as supported by services metadata, registries and repositories.
• Aggregation: Loosely-coupled services are aggregated into intra- or inter-enterprise business processes or composite applications.

The Service Queue is the list of all running Services. It is the only point where you can monitor the greeting of each individual process and it is technically an SQL table.
The Service Broker, implements the service broker pattern (having something that takes services from service queue and submit to the right application service to allow the execution of the service itself):
1. Sends the services to the correct application server
2. Manage the Service Queue
3. Update the Service status and MessageIn / MessageOut

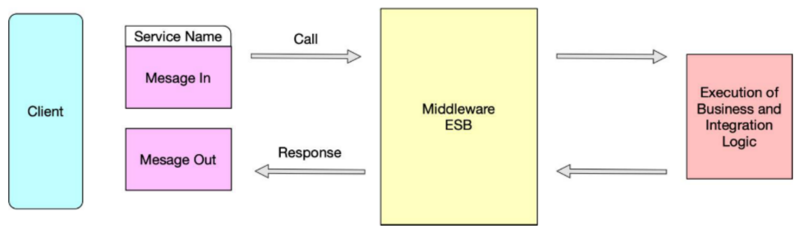
26. Describe all Communication Models we have seen during the class

Communication Model – Asyncronous Service Call



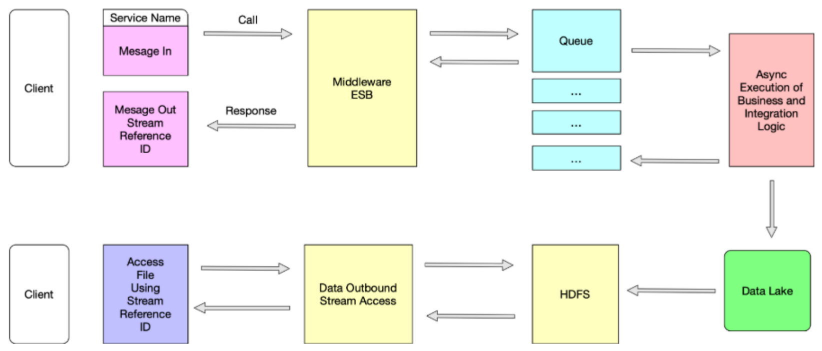
1. Clients **compiles** the Message In
2. Service Broker returns immediately the **Communication ID** and add the service to the Service Queue
3. When the **time** of execution **comes**, the **Service Broker submits** the execution to the right Application Server
4. The client can **check the execution 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 – 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

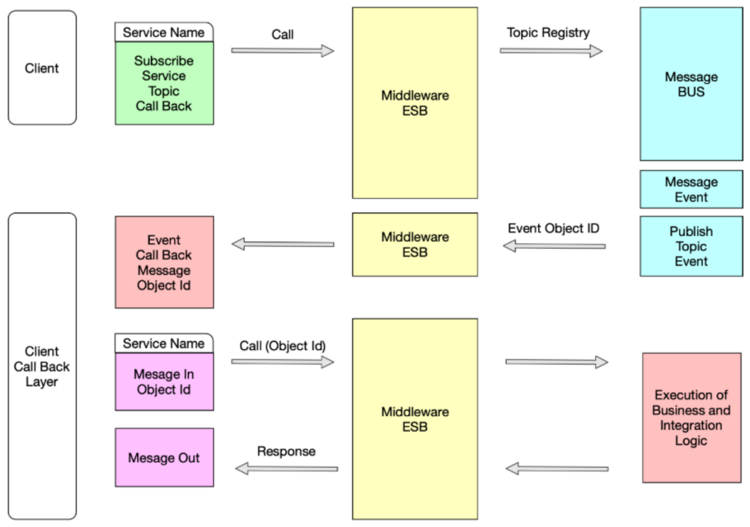
Communication Model – Stream Outbound File Reference



Client
Service Name
Mesage In
Mesage Out Stream Reference ID
Call
Response
Middleware ESB
Queue
...
...
...
Async Execution of Business and Integration Logic
Client
Access File Using Stream Reference ID
Data Outbound Stream Access
HDFS
Data Lake

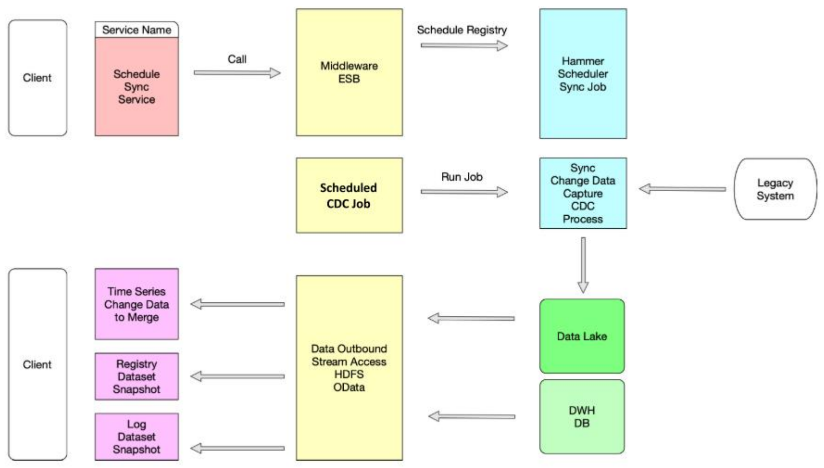
1. Stream Outbound File Reference model is an extesion of Async services
2. It can be used when large dimension output is going to be generated
3. The same **Communication Id** strategy is used to check the status
4. Furthemore a **Reference Id** allows client to open the stream towards the crated file

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



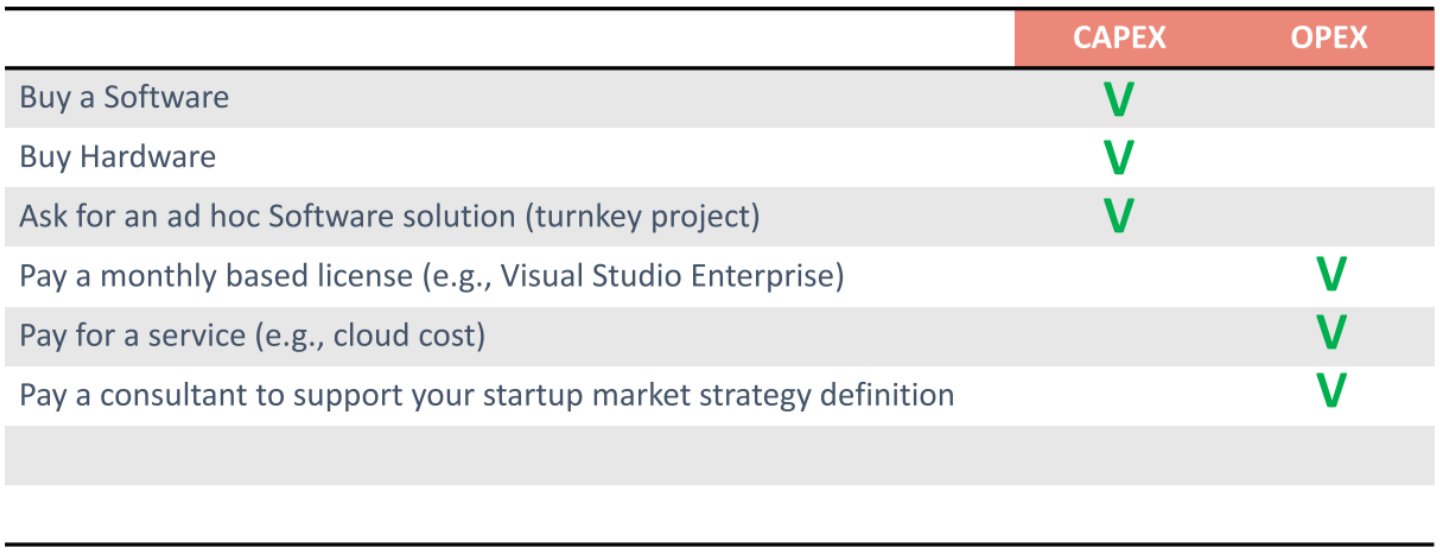
Client
Service Name
Schedule Sync Service
Call
Middleware ESB
Schedule Registry
Hammer Scheduler Sync Job
Scheduled CDC Job
Run Job
Sync Change Data Capture CDC Process
Legacy System
Client
Time Series Change Data to Merge
Registry Dataset Snapshot
Log Dataset Snapshot
Data Outbound Stream Access HDFS OData
Data Lake
DWH DB

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, …)

27. CAPEX vs OPEX

CAPEX: Capital expenditure (capex or CAPEX) is the money spent to buy or improve a fixed asset, such as buildings, vehicles, equipment. Software is considered an asset.

OPEX: An operating expense is an ongoing cost for running a product, business, or system. Software Maintenance and Licenses fees are OPEX costs.



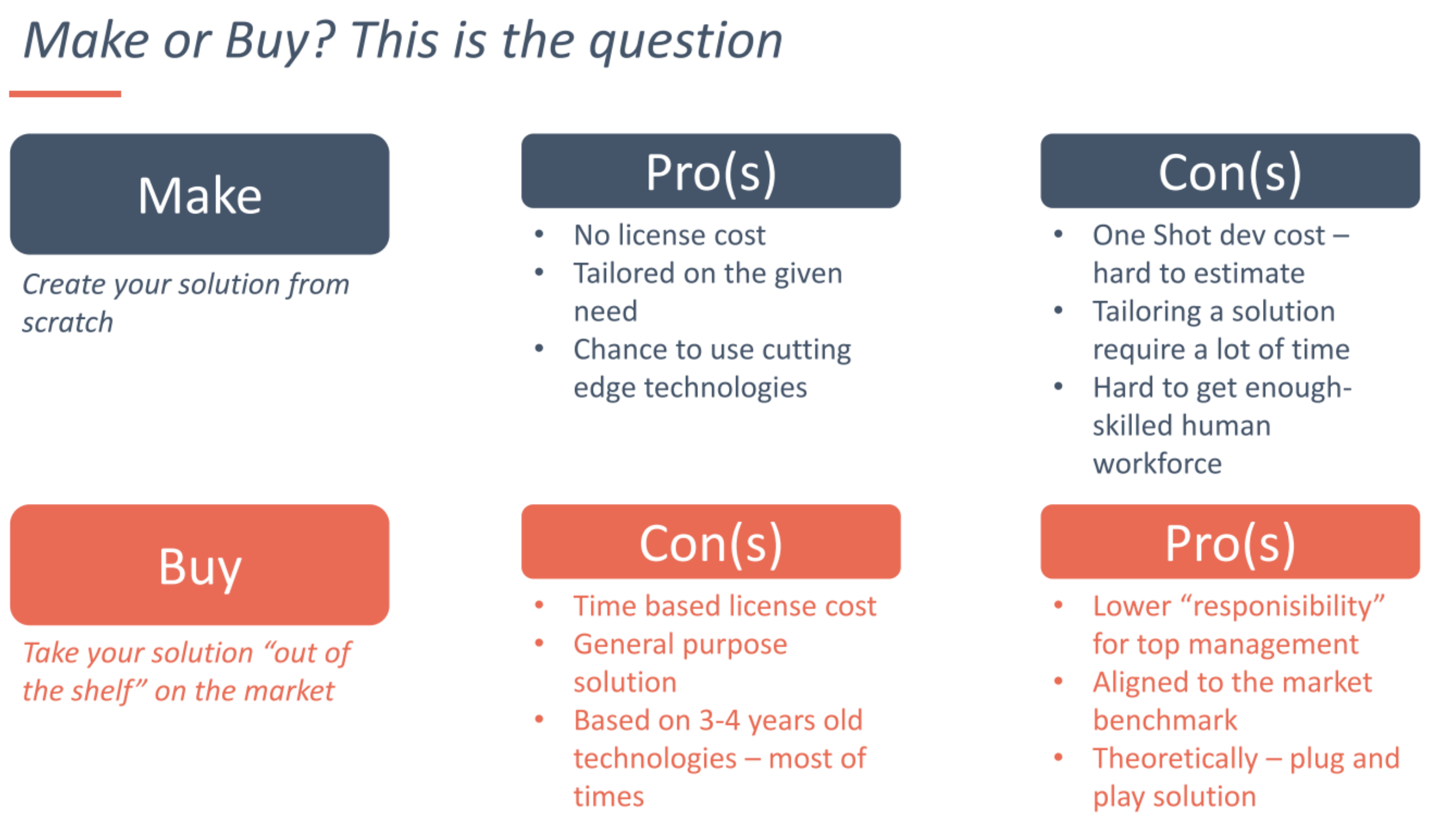
	CAPEX	OPEX
Buy a Software	V	
Buy Hardware	V	
Ask for an ad hoc Software solution (turnkey project)	V	
Pay a monthly based license (e.g., Visual Studio Enterprise)		V
Pay for a service (e.g., cloud cost)		V
Pay a consultant to support your startup market strategy definition		V

28. As a Service pattern

X As A Service refers to something offered to someone hiding all the internal complexity • Infrastructure As A Service allow users to use HW as if they own them • Software As A Service (e.g., Gmail, Salesforce, …) give users the possibility to use a software without the need of installing and maintaining them
• Platform As A Service (e.g., Databricks) allow users to use complex platform to build your own application without worrying about complex configuration and management
• Cloud provides X As A Service services

The opposite of “As A Service” is owning and controlling everything • This could mean an higher CAPEX cost but a lower OPEX one • Bare Metal is opposite to Cloud: on premise private data center offers such a possibility

29. Make or Buy



Make or Buy? This is the question

Make

Create your solution from scratch

Pro(s)

- No license cost
- Tailored on the given need
- Chance to use cutting edge technologies

Con(s)

- One Shot dev cost – hard to estimate
- Tailoring a solution require a lot of time
- Hard to get enough-skilled human workforce

Buy

Take your solution "out of the shelf" on the market

Con(s)

- Time based license cost
- General purpose solution
- Based on 3-4 years old technologies – most of times

Pro(s)

- Lower "responisibility" for top management
- Aligned to the market benchmark
- Theoretically – plug and play solution

30. GANTT vs Agile

GANTT is a particular chart used to plan the schedule of the activities, taking into account the dependencies among them and their duration. This planning is spread along various steps of time (usually the size of a week). From this kind of schedule has been generated the waterfall model in which the project is splitted in different logical phases that are sequential and permit to focus on a specific aspect of the project that is the base of the subsequent phase. This approach is flexible, but not iterative, once the cycle is completed, it is completed.
Agile is born from the so-called Deming cycle, a Japanese technique in which the development of the project is divided into cycles of equal phases. Each cycle aims to do a step forward in the project, permitting a good evolution strategy and an incremental approach.
In agile each cycle is called **sprint,** in each sprint the team must release a usable version of the project with increased value. Before the project starts there is a **pre-sprint** phase in which all the requirements and information are collected.
These phases are:

- Requirement definition
- Design
- Implementation

- Testing
- Maintain

Main roles in agile:

- The Product Owner is in charge of prioritizing Product Backlog
- The Scrum Master is in charge of ensuring Agile Rules are not broken
- The Agile Team develop the solution
- Stakeholders Evaluate the product

The priorities are discussed by the agile teams and divided in small items, when a requirement is small enough to be handled in a single sprint it becomes a use case and it will be implemented.