### 1. Design an Architecture to store and process tweets

* Vari **Twitter Listeners (pub)** per garantire ascolto contemporaneo su più hashtag tramite **Twitter Streaming API**
* **Coda di messaggi Pub/Sub** su cui scrivono i vari listeners
* **Caching agents (sub)** che scrivono su **DataLake** per elaborazioni future
* **Real-Time processing (sub)** per individuare **tweet pericolosi** (da mandare a WebApp e SQL Server)
* **Batch CDC models** da Datalake su datalake stesso o su DB SQL
* **DB SQL** che serve poi per la **WebApp**

### 2. What is quality?

Qualità è **diretta esperienza** indipendente da e **prima di astrazioni intellettuali**. Non è sempre possibile definirla chiaramente, ma bisogna sempre cercare di raggiungerla. L’esperienza e la cultura influisce direttamente sulla definizione soggettiva di qualità.

### **M -** 3. What do we mean with Architectural Styles, Materials, and Elements?

Uno stile architetturale incapsula **importanti decisioni** sugli **elementi** tramite **vincoli** su essi e sulle **relazioni reciproche**. In questo modo, è possibile **vincolare** alcune parti dell’architettura e nello stesso momento **coordinare**, dare una via da seguire agli architetti.

Uno stile permette di mettere **in risalto** quelle **decisioni** che soffrono da:  
- **Erosione** 🡪 L’**implementazione** è **diversa** dall’architettura originale; con l’andare avanti nel tempo, diventa sempre più difficile risalire all’architettura originale e si aumentano i possibili errori. E’ necessario un processo di continuo monitoraggio e controllo.  
- **Drift** 🡪 Problema più leggero, quando si implementa qualcosa che **non viola** alcuna decisione di design dell’architettura, ma che **non** era **richiesta**. Porta a problemi simili all’erosione.

Gli elementi possono essere:

* **Data Elements** 🡪 Componenti che **contengono l’informazione** usata e transformata
* **Connecting Elements** 🡪 Componenti che fanno da **collante** tra diversi pezzi dell’architettura; possono essere data/processing elements, o entrambi
* **Processing Elements** 🡪 Componenti che **applicano le transformazioni** sui data elements.

Le **proprietà** servono per **vincolare la scelta degli elementi**; definiscono vincoli minimali, altrimenti **ciò che non è vincolato** dall’architetto può prendere **qualsiasi forma** desiderata dal designer/implementatore.  
Le **relazioni** permettono di definire vincoli sul **piazzamento** degli elementi, su **come interagiscono** e **come sono organizzati** tra loro.  
“**Rationale**”, ossia la **base logica che sta sotto** (ma è parte fondante) all’architettura; rappresenta le **motivazioni** per la scelta dello stile architetturale, la scelta degli elementi e il soddisfacimento dei vincoli di sistema.

Esempi di stili architetturali sono CORBA, SOA e MOM.

I **materiali** (ossia i framework/tecnologie/sotto-sistemi) possiedono alcune **proprietà** che possono essere sfruttate da un certo stile architetturale. Gli aspetti materiali degli elementi di design prendono in considerazione **sia** la parte **estetica che pratica/ingegneristica** per un’architettura. Una **stessa funzione** può essere ottenuta da **diversi sotto-sistemi**, ognuno dei quali però può essere più adatto in un certo contesto.

### 4. Why do we need multiple POV when we present an Architecture? Provide an example

E’ importante considerare più punti di vista nella realizzazione di un’architettura, in modo da avere una **visione completa del problema** e avere una modellazione completa per tutte quelle figure coinvolte nella realizzazione di un certo progetto. Ogni POV, infatti, mette in risalto **diverse caratteristiche (e vincoli)** che l’architettura può o deve avere, avendo alla fine una visione completa di essa.

Nell’architettura classica abbiamo prospettive di **diversi stakeholders**:

* L’**owner** sa **perché** qualcosa deve essere fatto, ma non sa bene come.
* L’**architetto** deve progettare qualcosa che sia coerente con i bisogni dell’owner, pianifica il **cosa**.
* Il **muratore** deve progettare il **come**, rispettando le leggi naturali e i vincoli tecnologici.

Nell’architettura software abbiamo **diversi tipi di utenti** che useranno l’architettura, di conseguenza ognuno di essi avrà bisogno di un differente punto di vista.

* **Process model** 🡪 descrizione del prodotto con funzionalità, apparenza e interazione con utenti
* **Data model** 🡪 descrizione di data in ogni blocco
* **Network model** 🡪 Flusso tra ogni componente
* **Security model** 🡪 SQL
* **Data scientist**
* **Cost model (CFO)**

### 5. What do we mean we “Abstraction”? Provide a couple of examples seen during lessons and explain them

Si intende il **riconoscimento di un pattern**, il **nominarlo** e il definirlo, **analizzarlo**, trovare il modo per **specificarlo** e trovare un modo per **invocarlo tramite il suo nome senza intervento manuale** che possa generare errori.  
In questo modo, si **sopprimono** i dettagli del**l’implementazione** del pattern, riducendo la possibilità di errore e semplificando la comprensione del risultato.

Un’astrazione cattura proprietà essenziali dei sotto-sistemi più importanti e i modi in cui interagiscono.  
Inizialmente, i problemi vengono risolti ad hoc; più si accumula esperienza, più si notano soluzioni migliori altre, con un passaggio di folklore che viene passato informalmente da persona a persona. Pian piano, le soluzioni migliori vengono capite meglio e vengono codificate ed analizzate.

Alcuni esempi sono:

* **Pipe & Filter – UNIX** 🡪 Ogni modulo accetta uno **stream di input** ed emette uno **stream di output**. Concatenazione di comandi.
* **Programmazione orientata agli oggetti** 🡪 Oggetti indipendenti invocano le operazioni di altri oggetti, senza conoscere l’implementazione sottostante.
* **Sistemi a livelli – OSI layers** 🡪 Sistema organizzato a gerarchia, ogni livello fornisce un servizio al livello superiore senza esporre la propria organizzazione.
* **Sistemi basati su regole** 🡪 Computazione organizzata in una **collezione non ordinata di regole**, ognuna delle quali fornisce una condizione sotto cui la regola si applica e un’azione da eseguire (Triggers).

**Sistemi** possono essere costruiti combinando **sotto-sistemi**, ognuno dei quali con una **struttura interna**; può essere utile progettarne una parte dell’architettura prima di implementarla (**diversa granularità di astrazione**).  
Ogni sotto-sistema può riferirsi ad una specifica funzione o una funzione più comune (storage/comunicazione).

### **M -** 6. Why do we need to build an Architecture? What pillars are behind an Architecture?

Ci serve per risolvere un problema specifico (tenendo in conto anche i **vincoli umani e operativi**, come che conoscenze hanno i miei developer e che legacy systems sono già presenti, è un **Framework per l’astrazione**) e poterne **orientare il processo di sviluppo**, imponendo **vincoli** sugli elementi e le relazioni tra essi. Inoltre, tenendo conto di tutti i possibili punti di vista, permette di facilizzare l’implementazione di essa e l’**individuazione di possibili problemi** (quali drift, erosione e lock-in).

Le colonne portanti di un’architettura sono:

* Essere il **framework per la soddisfazione dei requisiti** 🡪 **Funzionali, tecnici e di sicurezza (GDPR)**
* Essere la **base tecnica per il design** 🡪 Indica la strada da seguire per le interfacce dettagliate degli elementi di progettazione, algoritmi, procedure e tipi di dato
* Essere la **base manageriale per una stima dei costi e della gestione dei processi**
* Abilitare il **riuso di componenti** 🡪 Stessi componenti, ma con logiche diverse
* Porre **focus sulla centralizzazione** 🡪 Singola fonte di verità, per evitare caos (e nascondendola dai Data Scientist ad esempio)
* **Aumentare la produttività e la sicurezza** 🡪 Individuare punti deboli, porte da aprire, costruire un gateway, nascondere il Data Lake…
* Abilitare la possibilità di **integrare sistemi aziendali**
* Abilitare una **scalabilità ordinata**
* **Controllare l’esecuzione dei processi software** (dove)
* **Evitare handover e lock-in**

### 7. What do we mean with Vendor Lock-in? How can we protect from him?

Quando un **cliente** è estremamente **dipendente da un fornitore** per prodotti e servizi; il **costo di cambiare** fornitore è **più alto che tenerlo** (senza quel costo, altri fornitori sarebbero migliori del fornitore attuale).

Possiamo **proteggerci** applicando un’**architettura** aziendale, ad esempio usando l’Adapter Pattern.  
Oltre al Vendor Lock-in, un’architettura software può **mitigare** gli effetti **anche** del **knowledge lock-in**, ossia il costo di trasmettere conoscenza è più alto che rimuovere dall’impiego una persona/team.

### 8. What is a Design Pattern?

Sono **best practices formalizzate** e **condivise** che risolvono **problemi comuni**.  
E’ un template che può essere usato in contesti e situazioni diverse.

### **M -** 9. What is a Data Lake? How it evolved the ETL pattern?

E’ uno **spazio di storage** praticamente immenso, che permette:

* La **memorizzazione di dati non processati, eterogenei e da qualsiasi fonte (serializzati in oggetti)**
* Dati **salvati per sempre**
* Buone performance di reading/writing
* **Schema** disponibile **on-read**

E’ basato sull’**Object Storage Model**, architettura di data storage che gestisce i dati come oggetti. **Ogni oggetto** contiene **dati**, **metadata** (informazioni contestuali) e **informazioni tecniche** (header).  
E’ basato su una **convenzione di nomenclatura**, la quale genera un **ID univoco** per ogni oggetto, ed una **strategia di serializzazione condivisa** (es JSON). Permette di distribuire dati su più nodi.

**Ha cambiato il pattern ETL in ELT**, Extract – Load – Transform; infatti, i legacy sistems sono abbastanza costosi e complessi, e l’**informazione prodotta e non salvata è persa per sempre**. Non c’è ragione dunque per non salvare tutta l’informazione possibile, per **effettuare a posteriori** qualsiasi **trasformazione** si voglia (lasciando anche **più spazio alle possibilità** di analisi effettuabili). Da qui, lo schema on-read.  
  
**ETL** 🡪 **Singolo processo**, prende dati non-strutturati, fa una qualche sorta di processing e li salva in un luogo strutturato. **Informazione scartata non estraibile** in futuro.  
**ELT** 🡪 **Un primo processo** prende e carica i dati così come sono, applicando solamente una **serializzazione standard** (es. Json). Un **secondo processo** prende queste strutture dati e esegue le **trasformazioni richieste**.

### 10. Explain the strategy and the limit of a given Traditional CDC

**Insieme di design patterns** per **determinare i dati che sono cambiati** (freschi) **e le azioni** da intraprendere sui dati cambiati. Riferiti a Log o Registry/Master tables (e sono **batch**).

**Invasivi lato Database**:

* **Timestamp** su righe 🡪 Ok log, critico per registry perché rallenta SELECT, INSERT, UPDATE, DELETE e se non è indicizzato la ricerca è molto lenta.
* **Numero di versione** su righe 🡪 Come timestamp, ancora peggio in quanto **richiede una query interna** per trovare il numero massimo di versione.
* **Indicatore di stato** su righe 🡪 Gravissimi problemi di concorrenza (chi resetta?)
* **Trigger** su tabelle 🡪 Problemi di performance, possono diventare ingestibili e hanno problemi di concorrenza (inserire in una tabella lockata pausa l’esecuzione)

**Invasivi lato Applicativo** 🡪 **Programmazione ad eventi,** la **responsabilità** di propagare i cambiamenti è **dell’applicazione**. Ogni volta che committa qualcosa, esso deve essere mandato anche al CDC.  
Problema soprattutto perché l’app potrebbe essere **basata su legacy system** e potrebbe non garantire proprio questa possibilità.

**Invasivi lato CPU Database**:

* **Transaction Log scanners** 🡪 Leggere e processare alcuni log tecnici del database, individuare i cambiamenti e propagarli al sistema CDC. **Non sono generici** e **non sono aperti**.
* **Log Shipping** 🡪 Sfruttare un **processo automatico di backup del DB** per propagare i cambiamenti (in modo incrementale e con tecniche interne non accessibili). Usato soprattutto nelle soluzioni di **Disaster Recovery**.

In generale, il problema principale è quello di **non sovraccaricare** il Database principale, che normalmente **non è possibile modificare** (aggiungendo la colonna timestamp) e **dipendono dal contesto**, non sono generici..

Adesso la **migliore soluzione** sarebbe avere uno **stream di dati su una coda**, in modo da avere vari subscriber che eseguono lavori diversi.

### 11. Given a source, describe how to apply the Diff&Where pattern

Se **log**, tenere **traccia dell’ultimo timestamp salvato** e fare una select direttamente sulla log table per trovare le righe successive (**where**).

Se **registry**, fare **kHash** e **vHash** per ogni riga del DB, controllarla con il **sync.json** per trovare **insert (kHash\_1 not exists)**, **update (kHash\_0 = kHash1 && vHash0 != vHash1) o delete (kHash0 not exists).**

### 12. Explain the Move and Rename pattern and when/how to use it

Il pattern Move and Rename (**Store and Forward**) permette di **garantire la transazionalità di un CDC**, o più in generale lo **spostamento di dati da una sorgente ad una destinazione**, che di per sé è **stateless ma non transazionale**. I file vengono inizialmente spostati con un’**estensione temporanea** (.tmp); una volta finiti tutti gli spostamenti, tutti i file .tmp vengono **rinominati** in .json.  
Il rinominare file è infatti un’**operazione** pressoché **atomica**; se si dovesse interrompere il processo di CDC, all’iterazione successiva nella **fase iniziale** verrebbero **puliti tutti i file** .tmp (permettendo la **fase di rollback**).

### 13. Why Adapter pattern mitigate the Vendor Lock-in problem

L’adapter pattern permette di mitigare l’effetto di Vendor Lock-in in quanto le **applicazioni non dipendono direttamente da un servizio esterno**, ma **comunicano con** esso tramite **un’interfaccia interna (Wrapper)** che espone funzionalità del servizio esterno (e può decidere anche di **nascondere alcune funzionalità**). In questo modo, nel caso si scegliesse di cambiare fornitore del servizio esterno basterebbe **cambiare il codice** all’interno **dell’Adapter Pattern**, **senza** dover cambiare ogni singola **applicazione**.

La classe Wrapper permette di chiamare servizi esterni senza esporre direttamente le interfacce di essi, ma bensì e interfacce del Wrapper.

### 14. Show how to build an Architecture to support a Data Scientist using the correct patterns

I Data Scientist non hanno bisogno di sapere la logica per avere dati freschi (CDC) o dove/come i dati vengono salvati; tutto ciò di cui hanno bisogno è **avere i dati freschi** per poterli analizzare, ed un modo per salvare queste analisi a posteriori. Di conseguenza, verrano specificate delle **astrazioni** che permettono di chiamare metodi come **readData(), writeResult()**… che permettono al Data Scientist di avere i dati richiesti senza dover capire da dove prenderli e come prenderli, lasciandoli concentrare solamente sulla fase di analisi in sé.

### **M -** 15. Explain what is a job, all the components needed to schedule a Job, and what does Job Impersonation mean

Un **job** è un **pezzo atomico di lavoro**; ogni job è **composto da più task/passi**, e può essere identificabile come un **singolo processo**.  
Può essere **lanciato interattivamente** (command line) **o schedulato per una esecuzione non interattiva**.  
Ogni job è **descritto da un nome** e una volta iniziato ha un **id univoco** tramite il quale si può **risalire allo stato**.   
Un job può essere **finito** (può completare, fallire o essere terminato) o **Online** (stoppabile solo quando terminato).

Il **job scheduler** è un’**applicazione** che **controlla l’esecuzione dei job in maniera non-interattiva**.  
Può lavorare usando uno **scheduling temporale** (batch scheduling) o dopo l’occorrenza di un certo evento (**event-oriented**).  
Il Job Scheduler tiene d’occhio lo **stato dei processori** e **distribuisce job** nella coda di job **ai processori in attesa**. Se si tratta di un sistema distribuito, può decidere a quale server e quando mandare un certo job.

La **Job Queue** è una **struttura dati** che mantiene i **job da eseguire**, e può essere consumata in **varie strategie**: priorità job, disponibilità delle risorse, stima del tempo di esecuzione, tempo di esecuzione trascorso, occorrenza di eventi prescritti, dipendenza tra job.

L’**impersonificazione** è l’**abilita di un thread di eseguire un job** usando **informazioni di sicurezza diverse** rispetto **al processo** che contiene il thread. Ad esempio, un thread in un’applicazione server **impersonifica un client**, e ogni operazione viene svolta **usando le credenziali del client**. In questo modo, il thread esegue per conto del client.

Diversi job possono essere svolti **sequenzialmente o in parallelo**; questo dipende dalla tecnologia usata. Se il Job Scheduler può mandare **job ad altre macchine (Broker Pattern)**, altre strategie più avanzate possono essere pianificate.

### **C -** 16. What Components are used by Hadoop?

**Hadoop** è un **Framework** usato per facilitare lo sviluppo di servizi multi-nodo, basato sull’assunzione che l’hardware può fallire.

I componenti principali sono:

* **Storage** 🡪 **Hadoop** **Distributed** **File** **System** **Storage** (**HDFS**)
* **Resource** **Management** 🡪 Hadoop **YARN** (**Global** **ResourceManager**, **NodeManager** & **ApplicationMaster**)
* **Computazione** **parallela** 🡪 Hadoop **MapReduce**

Ogni componente di Hadoop dovrebbe essere **location** **aware** e dovrebbe condividerlo con il sistema per poter distribuire dati e computazione.

Architettura:

* **Job** **Tracker** 🡪 Servizio che decide dove un certo task dovrebbe essere seguito
* **Task** **Tracker** 🡪 Nodo nel cluster che **accetta** **task** dal Job Tracker. Espone un **numero** **finito** di **slot** in cui **eseguire** **task** in parallelo (ognuno su una **JVM** **spawnata**), e manda **battiti** al **Job** **Tracker** **ogni** **minuto**.
* **NameNode** 🡪 Contiene tutte le informazioni e i percorsi su **dove** i **dati** sono **memorizzati** nel cluster.
* **DataNode** 🡪 **Memorizza** **dati**. Lavorano tra loro per replicare i dati, e il DataNode con i dati da processare dovrebbe essere sulla stessa macchina dove il TaskTracker sta eseguendo.

**Un’app** manda un **job** **asincrono** al **Job** **Tracker**, dopodichè si mette in **polling** per ricevere lo stato del job. Il **Job** **Tracker** trovare la **locazione** dei dati **dal** **NameNode** e identifica i **Task** **Tracker** **con** **slot** disponibili **più** **vicini** **ai** **Data** **Nodes**.   
Il **Job** **Tracker** **manda** **i task ai Task Tracker selezionati** e inizia a **monitorarne** **il** **battito**, per controllare possibili fallimenti. Una volta completato (o fallito) un task, il **Task** **Tracker** **notifica** il **Job** **Tracker** con lo **stato**; in caso di fallimento, il Job Tracker può mandare il job da qualche altra parte, marcarlo come to-skip o blacklistare il Task Tracker come non affidabile.

### 17. Comment the statement “Moving Computation is Cheaper than Moving Data”

Una **computazione** è molto più **veloce** ed efficiente se è eseguita **vicina** **ai** **dati** su cui opera, soprattutto se il Data Set è molto grande; in questo modo, si **minimizza** la **congestione** della **rete** e si **aumenta** la **banda** generale del sistema.  
Infatti, muovere la **computazione** significa spostare un **paio** **di** **righe** di codice, mentre un **dataset** su cui opera potrebbe essere potenzialmente **gigante**.

### **C -** 18. How Hadoop components enable HDFS?

Il **NameNode** prende tutte le **decisioni** **riguardanti** la **replicazione** dei blocchi e **riceve** periodicamente il **battito** e il **BlockReport** (lista di tutti i blocchi) **da** **ogni** **DataNode**.  
Ogni **file** è memorizzato come una **sequenza** di **blocchi** della **stessa** **grandezza**, ciascuno di essi **replicato**.  
I **file** in **HDFS** sono **write**-**once** (eccezion fatta per aggiunte alla fine o troncature) e hanno **uno** **scrittore** **solo** in ogni momento.

HDFS ha una **struttura master/slaves** (1 NameNode, n DataNodes). Il **NameNode** esegue tutte le **operazioni** **riguardanti** il **namespace** del file system (**aprire**, **chiudere** e **rinominare** **file**/**cartelle**), mentre i **DataNodes** **soddisfano** le **richieste** **di** **lettura** e scrittura dal client. I DataNodes inoltre **creano**, **cancellano** e **replicano** **blocchi** in base alle **istruzioni** **ricevute** dal NameNode.  
I dati non sono mai immagazzinati nel NameNode.

### **M -** 19. What are the HDFS Replication Policies?

La ripartizione deve essere **rack-aware**.  
**Policy** **1** 🡪 Replicare **l’intero** **rack**, in modo da **ottimizzare** l’uso della **rete** (**routing** delle **richieste** di read) e facilitare il **re-load balancing in caso di fallimenti** (senza perdita dei dati). In compenso, il **costo** delle **scrittura** **aumenta** in quanto è necessario trasferire blocchi a rack multipli.

**Policy 2** 🡪 **Two rack factor 3 replication**; si mette **una** **replica** su un **DataNode** nello **stesso** **rack** dove lo scrittore è e altre **due** **repliche** su **due** **DataNode** **distinti** di uno **stesso** **rack** **esterno**. In questo modo, si migliorano le performance di scrittura (2 rack invece che 3), si ha una **lettura** **sub**-**ottimale** (un blocco è su due rack distinti invece che 3) e si ha una distribuzione dei file sub-ottimale.

I **DataNodes** sono **affidabili** (**Heartbeat** al **NameNode**, replicazione se uno fallisce), i **dati** di un **DataNode** sono **affidabili** (controllo tramite **checksum**) e il **NameNode** potrebbe diventare affidabile usando la tecnica del **Quorum** **Journal** **Manager**. In questo caso, ci sarebbero **più** **NameNode** lettori in standby ed un solo NameNode attivo che funge da scrittore del log usando **demoni** chiamati “**JournalNodes**”; i DataNodes mandano il **battito** **a** **tutti** quanti i NameNodes.

### 20. Why we must ensure a reduction function is commutative?

**L’ordine** **di** **esecuzione** potrebbe **cambiare**, e di conseguenza il risultato potrebbe essere diverso da iterazione ad iterazione (con rischio anche di veri e propri errori).

### 21. When we need to call a repartition after a read?

Quando il numero di chiavi è minore delle partizioni, oppure quando si ha uno **scenario** **non** **bilanciato** (ogni **worker** **legge** un **sotto**-**insieme** di file, ma un **file** può essere **letto** da **un** **solo** **worker**); in questo modo si forza una **re**-**distribuzione** **equa** (nel caso **n° chiavi < n° worker**, crea un **nuova** **chiave** con **randint** per ottimizzare i primi passi che non richiedono le stesse chiavi sugli stessi nodi).  
Non si possono avere più partizioni di chiavi distinte.

Un’altra possibilità d’uso della repartition si ha nel caso di **join tra due RDD (o sé stesso)**, per distribuire le **stesse** **chiavi** dei due RDD sugli **stessi** **nodi**.

### **C -** 22. What is the difference between Driver, Master, Worker, and Executor?

* **Driver** 🡪 **Processo** responsabile per l’**esecuzione dell’applicazione spark**, **converte** l’applicazione utente **in** unità di esecuzione più piccole chiamate **task**, e le **schedula**. Contiene lo SparkContext e **mantiene i metadati** (come il NameNode) **di tutti i Resilient Distributed Databases**.  
  **Riceve i battiti dai Worker**.
* **SparkContext** 🡪 **Singleton** (per JVM), è l’oggetto Core di Apache Spark. **Permette al Driver di usare il cluster** e **mandare richieste** tramite il **Cluster Resource Manager**. E’ l’unico modo per creare tipi specifici di Spark (RDD, broadcast variables, accumulators…). Riceve heartbeat dagli executors.
* **Master** 🡪 Sulla stessa macchina del Driver, è un’**entità specifica** del framework **unica per applicazione**; **negozia le risorse** con il ResourceManager e **lavora con i NodeManagers per eseguire e monitorare i task**.
* **Worker** 🡪 **Nodo computazionale di Spark**, contiene gli executors (JVM, processo separato startato alla creazione di uno SparkContext) che eseguono i task serializzati ricevuti in thread pools. Usano il **Local Block Manager** per **comunicare con altri Workers**.
* **Executor** 🡪 Agente distribuito, responsabile per l’**esecuzione dei task**. Abilitano la **resilienza** degli RDD; se un executor dovesse fallire, i suoi task verrebbero dirottati e re-eseguiti su un altro nodo. I **task** sono tracciati con un **id univoco** e gli executor mettono a disposizione della **memoria principale** per RDDs usata per le chiamate persist() o cache().

### **C -** 23/25. Describe a Spark Job from a Spark Submit to the execution? Is it a Syncronous or Asyncronous job?

E’ un job **asincrono**.

1. Applicazione Spark inviata tramite **spark-submit**
2. Il **Cluster Resource Manager** inizializza l’**Application Master** e alloca le risorse richieste.
3. L’Application Master **si registra al Resource Manager** per garantire una comunicazione.
4. Il **Driver esegue il codice** comunicando (e mandandolo al) con il Master
5. Il Driver **converte il codice** utente da un piano logico ad un piano di esecuzione/fisico, passando dal **DAG e il DAGScheduler**.
6. Il Driver **negozia con il Cluster Manager le risorse**; in questo modo, gli **stages** sono **creati** dal DAGScheduler.
7. Gli **Executors** sono **avviati** sui Workers; quando si avviano, si **registrano al Driver**.
8. Il Driver tiene **monitorato lo stato degli executors**, in modo da re-distribuire gli executors non attivi ad altri Workers.
9. Il Driver invia **tasks al Cluster Manager in base al posizionamento dei dati**.
10. Il **Master lancia il container** dando al **Node Manager** una configurazione.
11. Il **primo RDD** è creato leggendo **dati dal HDFS in differenti partizioni** su nodi diversi in parallelo. Ogni nodo ha un certo sottoinsieme di dati.
12. Durante l’esecuzione, il **Driver comunica con il Master** per ottenere lo **stato dell’applicazione**.
13. Una volta che l’applicazione è completata, il **Master si de-registra dal Resource Manager e de-alloca tutte le risorse**.

### 24. What does lazy mean for Spark?

Non **computa** i risultati immediatamente, ma solamente **quando il driver richiede** di mostrare un qualche **risultato** (trasformazioni non computano, azioni sì).  
Ogni **trasformazione** è **ricomputata** ogni volta che viene richiesto un risultato; questo si può evitare, persistendo il risultato delle trasformazioni con il metodo **persist()**. Questo metodo richiede comunque memoria, quindi è consigliabile usarlo solo quando si svolgono più operazioni sullo stesso RDD.

**Solo la parte** di computazione **richiesta** per mostrare i risultati **è fatta** (nessuno scan dell’intera tabella se viene richiesta solo la prima riga), e sapendo il DataNode dove sono tutti i dati i risultati dovrebbero essere sempre li stessi.

Si cerca di evitare di avere operazioni che richiedono righe distribuite su differenti nodi; questa operazione viene chiamata Shuffle.

### 26. What is a transaction?

I **processi di Big Data** richiedono **molto tempo** (3/15 minuti) prima di concludere; è dunque necessario assicurare una consistenza dei dati e un modo di comunicare tra i processi.

Una transazione è una **sequenza di operazioni** che **soddisfa le proprietà ACID**, e che **permetta il Rollback** (operazione di ritorno di un sistema al suo **stato originale** prima che la transazione iniziasse).

Il **controllo della concorrenza** permette di assicurare il corretto risultato di operazioni concorrenti, mitigando il problema di **lost update** (primo update perso), **dirty read** (valore letto modificato da una transazione poi abortita) e la **somma incorretta**.

* **Atomicità** 🡪 Ogni asserzione di una transazione deve essere considerata come una singola operazione che va a buon fine completamente o fallisce; se anche solo una fallisce, l’intera transazione fallisce e nessun cambiamento viene applicato.
* **Consistenza** 🡪 Le transazioni portano il sistema solo da uno stato valido ad un altro.
* **Isolamento** 🡪 Transazioni in parallelo devono portare a un sistema nello stesso stato di quello che avrebbe nel caso in cui fossero svolte sequenzialmente.
* **Durabilità** 🡪 Una transazione committata non può essere rollbackata se qualsiasi transazione consecutiva fallisce.

### 27. What are Transaction Logs and Log Shipping?

- **Transaction Logs** 🡪 **Storia delle transazioni eseguite** da un database che può essere usato per garantire le proprietà ACID in caso di crash o fallimento di hardware.  
Può essere usato nel caso in cui il sistema si trovasse in uno stato inconsistente **per garantire atomicità e durabilità**; roll-back di transazioni non committate e re-applicazione di transazioni già committate i cui cambi non erano stati ancora materializzati a sistema.  
Possono creare **file molto grandi**, e possono essere la **causa di errori** (finiscono lo spazio di memorizzazione del DB).

- **Log Shipping** 🡪 Tecnica per **aumentare l’affidabilità del DB** e garantire una **strategia di disaster recovery**. Essi sono mandati su uno **slave server** per creare una **copia** esatta del DB originale.  
Sono **HW intensive** perchè il master server deve scrivere tutte le transazioni per ogni tabella; potrebbe essere **attivato** su un **sotto-insieme di tabelle**.

### 28. How can we evolve CDC using Log Shipping?

**Operare** direttamente **sulla** tabella **secondaria**, oppure **inserire una coda in mezzo di tipo Pub/Sub**, in cui un subscriber manda al server secondario i cambiamenti e un altro opera da vero e proprio CDC.

Ogni database mette a disposizione un suo servizio di streaming specifico.

### 29. Provide an example of a deadlock

Si ha un deadlock quando **un gruppo di processi si incastrano tra loro**, in quanto nessuno può ottenere tutte le **risorse** di cui ha bisogno essendo **bloccate da altri processi** dello stesso pool (**combinazione di lock bloccante**).  
Per evitare ciò, un **algoritmo** di prevenzione di deadlock garantisce che almeno **un processo sia in grado di ottenere tutte le risorse di cui necessita**.

Processo 1: read A, write B = A + 5   
Processo 2: read B, write A = B + 1

### **M -** 30. What is a Delta Lake?

E’ uno **spazio di archiviazione a forma di tabelle su Cloud Objects** che **garantisce** le proprietà **ACID** su un **HDFS**. Mentre Data Lake è più visto come storage, Delta Lake si avvicina ad un vero e proprio DBMS.  
Usa un **transaction log** compattato nel **formato di Apache Parquet**, ogni tabella (file) è un set di oggetti (Parquet Files) in base alla colonna, per garantire una maggiore efficienza ed ottimizzazione (compressione più efficiente, la compressione può essere in grado riconoscere il tipo della colonna).

Normalmente **update non atomici** (oggetti divisi in più file, vista la colonna spezzata) e **rollback non possibile** (se si corrompe un file, l’intera tabella si rompe). **Tiene traccia** di quali **oggetti** fanno **parte di una Delta Table in maniera ACID** usando un **Write Ahead Log**, anch’esso memorizzato come un cloud object.  
Il log **contiene** inoltre **metadata come statistiche di minimo/massimo** per ogni data file (velocizza ricerche di metadati), e le transazioni usano un **protocollo di concorrenza ottimistico**.

Le proprietà di un Delta Lake sono:

* **Possibilità di spostarsi nel tempo** 🡪 Query snapshots in un certo momento
* **UPSERT, DELETE e MERGE** 🡪 Riscrivono efficientemente gli aggiornamenti sui dati archiviati
* **Streaming I/O efficiente** 🡪 Scrittura di **oggetti piccoli** nelle tabelle (scrittura molto veloce), per poi essere uniti in un secondo momento in oggetti più grandi (lettura efficiente).
* **Caching** 🡪 I nodi del cluster possono cachare gli oggetti sulla memoria locale.
* **Evoluzione dello schema** 🡪 Possibilità di **leggere vecchi file Parquet** anche se lo schema della tabella dovesse cambiare.
* **Forte lock-in**.

### 31. What does XXX can be used for (XXX is one of the docker command)?

* **FROM** 🡪 Inizializza una **nuova fase di build** e specifica la **Base Image** (o **Parent Image**) per le istruzioni seguenti. Alla fine deve puntare ad un’immagine di sistema, e può puntare anche ad un repository condiviso online.
* **ARG** 🡪 Unica istruzione che è possibile inserire prima del FROM, dichiara una **variabile** usata nel FROM e ha scope **globale**.
* **RUN** 🡪 **Esegue i comandi (bash)** specificati durante la fase di build
* **COPY** 🡪 Copia un **file dalla stessa cartella** del Dockerfile **in una locazione specificata** nel container.
* **CMD** 🡪 Usabile **una sola volta** (solo l’ultimo viene considerato), **esegue** l’istruzione quando si **avvia l’immagine/container**.
* **ADD** 🡪 **Come COPY**, ma permette di copiare **da un URL**
* **EXPOSE** 🡪 **Espone un protocollo o una porta**.
* **ENV** 🡪 Definisce **variabili di ambiente** locali (key=value).
* **WORKDIR** 🡪 Definisce la **cartella** di lavoro **corrente**.
* **USER** 🡪 Definisce l’**utente che esegue i comandi successivi**.

### 32. What import can be used for inside a Dockerfile?

Per **ereditare da un’altro dockerfile** o specificare un’immagine base di distribuzione Linux.  
Oppure crea un’immagine di filesystem da un .tar.

### 33. Dockerfile, docker images, docker container: what are the difference?

**Dockerfile** contiene **tutti i passi** da eseguire per poter **costruire un’immagine** (solo uno per cartella); l’immagine è l’**esecuzione di tutti i passi** del Dockerfile, ed è **statica** (anche più image da stesso file). Il **container** è un’**istanza** in sé di un’immagine.

### 34. Create a container: step by step activities needed.

1. **Dockerfile scritto** con tutti i comandi richiesti
2. Docker Image creata dal Dockerfile tramite **docker build** (-t name relative\_path)
3. **Docker images** per trovare tutte le immagini
4. Docker Containter creato e avviato da una certa Docker Image con **docker run** (**-p porta\_SO:porta\_docker -v mount external folders** to the container)
5. **Docker ps** per avere tutti i **container attivi**.
6. Gestire i **container Docker** come **servizi** con **docker start/restart/stop/kill**
7. Tutti i container possono essere mostrati tramite **docker ps -a**

Possibile orchestrare più Containers tramite un **file di configurazione yml**, usando **collegamenti** per renderli visibili gli uni agli altri (**docker-compose up/stop/start**).

### 35. What is an Index? Why the table is locked when we write inside an indexed table?

Un indice è una **struttura dati** che permette di **velocizzare il recupero dati su read**, andando a chiedere **più** operazioni di **scrittura e** più **storage**.  
In questo modo, si riesce a localizzare velocemente l’informazione cercata senza dover cercare ogni riga della tabella del DB, e possono essere creati su una o più colonne della tabella del DB.

La tabella è **lockata** perché è **necessario ri-strutturare l’indice** con i nuovi insert/update, dunque richiede una **transazione lockata**.

### 36. What MERGE statement can be used for?

Permette di **fondere due tabelle diverse** insieme **specificando la condizione** su cui dovrebbero corrispondere e le azioni da intraprendere nei vari casi di **Match, NotMatch** e **NotMatchBySource**.  
E’ possibile usarlo per **spostare dati da Spark a un server SQL**, usando la tecnica BCP & Merge; in questo modo non si rallenta il DB in produzione, come succederebbe invece con gli altri metodi.

### **M -** 37. How can I write from Spark to SQL?

* **Connessione diretta** 🡪 Troppo **overhead**, necessario stabilire connessione per ogni operazione.
* **Multiplo Pooling diretto dai Worker** 🡪 **Molte Dirty Pages** (copie dei dati da aggiornare, per garantire proprietà ACID) e la **tabella è lockata**. **Ogni Worker** deve **aspettare** gli altri **per poter chiudere** la connessione.
* **Singolo Pooling singolo da Driver** 🡪 **Global collect** per ricevere dati da ogni worker, che **richiede tempo** e può risultare in **problemi di spazio** (distribuzione su disco o Out of Memory). Sempre molte **Dirty Pages**, **tabella lockata** e si ha un **bottleneck** (100 VM ma poi 1 to 1 VM alla fine).
* **Data Lake Queue** 🡪 Ogni worker scrive direttamente sulla **coda su Data Lake**, garantendo **massima velocità**, ed un **Agente C#** userà un **client nativo** per scrivere sul server. Garantisce maggiore velocità e non è presente alcun bottleneck, ma si hanno ancora molte **Dirty Pages** e la **tabella** rimane **lockata**.
* **Double Queues & Merge** 🡪 Ogni Worker scriverà su una Data Lake queue; l’**agente C# userà un Bulk Copy Program** (importa molte righe/filesystem su tabelle SQL o contrario) per scrivere su una **tabella temporanea**. Non si hanno Dirty Pages o la tabella lockata, in quanto l’**operazione di Merge finale** sulla tabella di destinazione sarà unica e la tabella temporanea non avrà neanche il bisogno di essere indicizzata.

### **C -** 38. What is a page? And an Extent? How an Index is written? What is btree?

DataFiles sono collezioni di Extent.  
Una pagina contiene le righe vere e proprie di una tabella, più alcune informazioni nell’header, il puntatore alla pagina successiva e dello spazio libero. Ogni pagina è al massimo di 8KB, e si tende a tenere dello spazio libero per velocizzare gli insert.  
Un Extent contiene una concatenazione di pagine (8, 64 KB) e può essere uniforme o mixato (con anche pagine di indice).

Un B-Tree+ è un albero in cui i dati sono memorizzati solo nelle foglie, la distanza tra root e foglie è sempre la stessa ed è possibile navigarlo in modo ordinato, seguendo i puntatori dei nodi.

Un **indice** è costruito inizialmente **ordinando i dati in base alla chiave**, in ordine **crescente**; una **pagina vuota** viene creata come **root**, e si inserisce un **puntatore alla prima pagina.** Quando il **Root è pieno**, si **splitta** e si crea una nuova pagina di root.  
Per aggiornare un indice, si effettua una ricerca per capire a quale bucket corrisponda il nuovo dato. Se il bucket non è pieno, aggiungere il dato, altrimenti splittare ricorsivamente il bucket fino al root.

### 39. Spark Exercises (see Spark Exercises Section)

### 40. Which kind of strategies do we have to mix two different rdds? Which one is better?

* **Join**
* **Cartesian**
* **Union**
* **FullOuterJoin, Right/LeftOuterJoin**
* **Direct get** da dizionario (solo se dizionario è registry rdd, **collectAsMap()**)
* **Broadcast di rdd** registry as dict, poi get.

Dipende dal contesto e dagli rdd (la get da dizionario funziona solo se rdd2 è di tipo registry, cartesian e union funzionano anche se i due RDD non sono PairedRDD).  
In generale, **Join** funziona meglio con **big/huge registry** (evita errori di memoria, normalmente è **meglio** fare **una repartition** prima per avere gli stessi dati sugli stessi nodi ma non quando i dati sono immensi); una direct get è quasi sempre peggio che una broadcast, e una **broadcast** può essere molto efficace quando si ha uno **small registry** come dizionario (evita soprattutto errori di memoria e tempi di trasporto ai vari worker).

### 41. What is the role of an Abstract Class in a Software Architecture?

Una classe astratta permette di definire una **via da seguire**, **implementando** direttamente quei **metodi** che dovrebbero essere **vincolati** ad un certo standard e lasciando **astratti metodi** che possono essere implementati in modo diverso in base al contesto o allo sviluppatore.

### **M -** 42. What are the main components a CDC Architectural Abstraction should have?

* **ABC Source Adapter** 🡪 Astratte la **connessione** e la **submit della query**, in quanto dipendono dall’implementazione in sé (astrae chi è la sorgente, di caso in caso). **Gestisce l’Extract**.
* **CDC Process 🡪** ExecutionTrace(), run(), commit(), rollback()…
* **ABC Data Manager** 🡪 Gestisce i **dati** in sé e la **trasformazione**, ha astratto il **getFreshData()** e il **dataToDictParser()**, mentre di **concreto** può avere la **serializzazione** (sia dei dati che della struttura di controllo come il sync.json, che deve essere lo stesso e vincolato).
* **ABC Destination Adapter** 🡪 **Gestisce il Load**, non deve essere scelta la destinazione e deve contenere in modo astratto **tutti i metodi di fileSystem**, mentre può avere come **concreto la scrittura dei dati freschi**.
* **ConfigurationObject** 🡪 E’ il punto fondamentale per qualunque architettura che voglia essere astratta e riusabile. Il CDC potrebbe essere lo stesso, ma su tabelle diverse ad esempio dello stesso DB; come faccio a **chiamarlo come servizio con configurazioni diverse**? Tramite questo oggetto e la sua combinazione con **MessageIn**. Contiene il **codice SQL** che specifica cos’è il **FreshData**, la stringa di **connessione SQL**, il **nome unico del processo** del CDC e la **logica del parsing**.

### 43. What kind of distance metric you should propose to compute the top-n similar texts to a given one? Why?

Prima di tutto, si rappresentano i documenti con **bag of words** (vettori con n° occorrenze di una parola in un documento).

Meglio usare la **cosine similarity** rispetto alla **distanza euclideana**, in quanto la seconda fa una **differenza tra i due vettori** (quindi è necessario mantenere in memoria anche tutti gli zeri), mentre la prima opera tramite **moltiplicazioni**, di conseguenza è possibile **non tenere in memoria tutti gli zeri** (in **Spark** gli **zeri non** verranno **caricati**, dunque un certo termine mancherà in un documento, e tramite il **join** si avranno solo i **termini in comune**).  
Tenendo conto che si opera con una **matrice estremamente sparsa** (40 parole per ogni testo rispetto alle 22.000 totali), caricarsi solo le parole effettivamente presenti e non anche le assenti (ossia gli 0) permette di risparmiare TB di memoria.

La cosine similarity con vettori di conteggio però non tiene in conto del contesto, stemming ecc… Si usa dunque la **tecnica di TF-iFD** per dare uno score ad ogni parola.

### 44. Why SOA has been created?

Permette di collegare team e organizzazioni tra loro, ponendo un’**importanza** cruciale sull’**accordo su termini importanti**, ma soprattutto pone l’**attenzione** sul **lato business** e non sul lato implementativo; i servizi dunque devono rappresentare e soddisfare richieste del business.  
Ogni **servizio** è infatti l’espressione di un **bisogno lato business**.

### **M -** 45. What is a service? What are the pillar for a Service and a SOA?

Un **servizio** è una **risorsa scopribile** che esegue un **task ripetuto** ed è descritto da una **specificazione esterna** del servizio (dunque senza entrare nei dettagli).

I pilastri di un servizio sono:

* **Allineamento lato business** 🡪 Non sono basati sulle possibilità lato IT, ma su cosa serve lato business. Sono infatti la rappresentazione logica di un’attività di business ripetuta con un certo risultato.
* **Specificazioni** 🡪 Sono **self-contained** (chiari sia per IT che per business) e **descritti** in termini di **interfacce**, operazioni, semantiche, polizze, comportamenti dinamici e QoS.
* **Re-usabili**
* **Accordi** 🡪 Basati sulle specifiche e non sull’implementazione, il provider fornisce qualcosa in cambio di qualcosa da parte del consumatore. Si ha uno **scambio di valore** tra partecipanti indipendenti tra loro.
* **Hostati e trovabili**
* **Aggregabili** 🡪 I servizi possono essere formati o formare altri servizi.
* **Black-box** 🡪 I consumatori del servizio non sanno nulla di come siano davvero implementati i servizi.
* **Orientato ai messaggi** 🡪 Definito come **scambio di messaggi tra** l’agente **fornitore** e l’agente **consumatore**, astraendo via la struttura in sé, l’implementazione e le proprietà degli agenti.

SOA è basato sul modo di pensare a servizi, per garantire un collegamento più stretto tra IT e business; essi infatti rappresentano **bisogni e attività del mondo del business**. Persone, organizzazioni e sistemi offrono e usufruiscono l’uno dei servizi dell’altro, in cambio di qualche valore; può essere vista come una forma di architettura di sistema distribuito, in cui agenti software devono lavorare tra loro comunicando tramite protocolli hardware/software (tenendo conto dell’imprevedibilità dell’accesso remoto, problemi di concorrenza e fallimenti parziali) e meccanismi di broker request.  
Ogni servizio è identificato e ciò che fa viene specificato chiaramente sotto forma di **contratto**, e viene **offerto tramite un’interfaccia ben definita** al pubblico.

Il risultato di un servizio porta normalmente ad un **cambio di stato del consumatore**, ma potrebbe portare anche ad un cambio di stato **del fornitore** **o** di **entrambi**.

### **M -** 46. Represent SOA layers

1. **Sistemi operativi** 🡪 Le applicazioni già presenti nel sistema (**legacy systems**)
2. **Componenti aziendali/dei servizi** 🡪 Che realizzano funzionalità e **garantiscono il QoS dei servizi esposti**, ad esempio per **gestire il workload**, garantire una buona affidabilità e load balancing. **Permettono di chiamare le risorse del livello aziendale** dai servizi.
3. **Servizi** 🡪 su cui la parte di **business** ha deciso di **investire e** di **esporre**; possono essere scopribili o vincolati staticamente ed invocati in un servizio composto.
4. **Processi di business** 🡪 **Composizione di servizi** del livello precedente per garantire una **applicazione completa per il business**.
5. **Layer di accesso/presentazione/clienti**
6. **Integrazione** 🡪 Per **gestire meglio le richieste**, come **routing** intelligente, mediazione di protocolli… Chiamato **ESB**
7. **QoS** 🡪 Monitora, gestisce e mantiene la sicurezza, le performance e l’affidabilità.

### 47. What is a

* **Service** 🡪 Identificato da un **nome univoco**, una **versione**, un **tipo di esecuzione** (sync/async), un **tipo di logging** (before/after), un **messageIn** (configurazione e struttura del data In), **messageOut** (configurazione e struttura del data Out) ed un **cono di visibilità** (ACL).
* **Catalogo dei servizi** 🡪 **Lista di tutti i servizi disponibili** e contiene la lista dei **messageIns** usabili per configurare l’esecuzione di un servizio.
* **Coda di servizi** 🡪 **Lista di tutte le esecuzioni di servizi** con la traccia di ogni esecuzione. **Punto singolo da cui monitorare lo stato e la salute** di qualunque processo. Solitamente si tratta di una tabella SQL.
* **Enterprise Service Bus** 🡪 Sistema di comunicazione che permette ai servizi di **interagire con i sistemi aziendali**, in modo da evitare un’integrazione sistema-a-sistema.
* **Service Broker** 🡪 E’ in carica di **spedire i servizi all’application server** giusto, **gestisce la coda di servizi e aggiorna lo stato** dei servizi e messageIn/messageOut. Fornisce inoltre la descrizione di un servizio quando richiesto dal client, logga e controlla se il client ha i requisiti di accesso richiesti.

### **M -** 48. How a XXX works:

* **Syncronous** 🡪 Il client **compila il MessageIn**, ESB guida la richiesta, il **Service Broker** la **invia** all’Application **Server** specifico che **esegue** la logica e **ritorna un Message Out**.   
  **Ogni esecuzione è collegata ad uno stato** che cambia durante l’esecuzione ed un trace log. L’esecuzione tecnica è tracciata anche all’interno alla tabella di Service Queue per analisi future e per controllare lo stato.
* **Asyncronous** 🡪 Il Client **compila il Message In**; il **Service Broker** **ritorna immediatamente un ID di comunicazione** e aggiunge il **servizio alla Service Queue**. Quando arriva il tempo di esecuzione, il Service Broker la invia al giusto Application Server.   
  Il client può **controllare lo stato dell’esecuzione usando l’ID di comunicazione**, e può **ottenere il risultato dell’esecuzione** quando lo stato è **DONE** usando l’ID di comunicazione e facendo una **query sulla Service Queue** per estrarre il **messageOut**.
* **Stream Outbound File Reference** 🡪 Estensione del modello asincrono, può essere usato quando un **output di grosse dimensioni** potrebbe essere generato (messageOut non basta). **Oltre all’ID di comunicazione** per controllare lo stato viene ritornato al Client anche un **Reference ID** che permette al client di **aprire uno stream e iniziare a processare** i file creati.
* **Pub/Sub** 🡪 Permette ad un client di **iscriversi ad un certo topic** e **ricevere un Call-back** quando un nuovo evento si verifica. L’ESB **nasconde la complessità del Bus di Messaggi** (Apache Kafka) per semplificare la vita al client.  
  Ogni volta che un nuovo evento si verifica, l’ESB chiama il call-back del client **ritornando un Object ID che può essere usato** per recuperare i dati freschi.
* **Batch Sync Data Process** 🡪 **Processo schedulato** per avere un’**estrazione incrementale di dati** da qualunque sistema aziendale; le righe fresche estratte sono salvate direttamente sul DataLake, e un **servizio di gateway** (**Reverse CDC job**) **invia** queste estrazioni incrementali **ai sistemi di destinazione**.

### 49. Spark Exercises (see Spark Exercises Section)

### 50. Difference between DAGS, Stages and Tasks

Il **DAG** è un **grafo** compilato dal **Driver** avente come **nodi** tutte le **trasformazioni** eseguite **su un RDD**; un **DAGScheduler** riceve questa esecuzione logica e la **trasforma in un’esecuzione fisica** (**nuovo DAG**, calcolando anche la quantità di memoria potenzialmente richiesta), avente come **nodi gli stage interni** che implementano la logica di trasformazione, nel momento in cui viene **richiesta un’azione**.

Uno **stage** eseguito **su un certo gruppo di righe** viene chiamato **task**; il **TaskScheduler invia i task** ai worker **tramite il Cluster Manager**, ed il numero di task inviati dipende dal numero di partizioni in cui è separato l’RDD. I Worker eseguono i task.

### 51. Why Spark skips sometime some activities?

Perchè è stato chiamato persist(), che **persiste in memoria** le **trasformazioni** effettuate **in precedenza** senza bisogno di re-computarle.

### **M -** 52. What is new in Spark 1.3+?

Sono stati introdotti dei **nuovi tipi di dato basati sull’idea di Spark SQL**, un modulo per processare **dati strutturati**. E’ necessario **specificare** informazioni sulla **struttura sia dei dati che della computazione** da eseguire, in modo da sfruttare delle **ottimizzazioni extra**.  
**Spark SQL** inoltre mette a disposizioni un’**interfaccia per eseguire query SQL**.

I nuovi tipi di dato sono **Datasets** (collezione distribuita di dati, combina i benefici degli RDD e i benefici dell’ottimizzazione del motore di esecuzione di SparkSQL) e **DataFrame** (**Dataset organizzato in colonne**). I DataFrame sono concettualmente equivalenti ad una tabella in un database relazionale, ma con maggiori ottimizzazioni.

### **M -** 53. Why DF reduce the Spark expressivity?

**RDD** offrono funzionalità di **basso livello e controllo**, garantendo una **maggiore flessibilità**, mentre i **DF** garantiscono una **vista personalizzata e strutturata**. Inoltre, gli **RDD** possono essere **nestati e profondi**, mentre i **DF** devono essere **appiattiti**.

I **DataFrame ottimizzano l’esecuzione**, mentre con gli **RDD** sono gli **utenti** a poter **ottimizzare** l’esecuzione; se con i **DF** si ha un’espressività simile a SQL (**distribuzione di funzioni lambda su righe**), con gli **RDD** si **distribuiscono funzioni lambda su oggetti**.

In compenso, i **DF** hanno incredibili librerie di **Machine Learning**.

### 54. How the submit-to-execution process changes with Dataframes?

**Dopo** la separazione in **stages** da parte **del DAGScheduler e prima dell’invio dei task**, si ha un **DAG Scheduler ottimizzato** (**Tungsten**, in C) che, sapendo a priori lo schema, può eseguire **ottimizzazioni** avanzate sull’esecuzione.

### 55. Return on investment and Business over cost analysis: what do they mean?

E’ più facile monetizzare i costi che i benefici. Al business non interessa nulla di quanto sia incredibile un’idea, ma quanto **guadagno** porta **rispetto al costo** da intraprendere.  
L’**ICT** ha **controllo sulla parte dei costi**, ossia **come** una cosa chiesta dal business deve essere fatta, **chi** la deve fare, **quando** e **dove**; **non** ha alcun **controllo** (o poco) sulla parte di **guadagno/marketing**, gestibile dal **business** (**cosa** deve essere fatto **e perchè**).

Lo scopo del **ROI** è avere una **metrica** in grado di identificare il ritorno economico di un investimento (quanto riprendo dopo averti dato come investimento), per capire se farlo o meno; quanto renderebbe il capitale investito in quell'azienda. Se il ROI è una stima, il BOC è un’analisi effettuata su dati veri (quanto guadagnerò rispetto a quanto spenderò).

### 56. What is the difference between a CAPEX and an OPEX cost?

**CAPEX** (**capital expenditure**) sono i soldi spesi per comprare **asset fissi**, come edifici, veicoli, software, startup… Sono capitali che **rimangono fisse** **e di proprietà** dell’azienda nel tempo, e possono essere **detratti dalle tasse**.

**OPEX (operating expense)** è il costo per **mantenere un prodotto non di proprietà** dell’azienda.

I costi possono essere **One Shot o Running**; pagare di più **One Shot** potrebbe far risparmiare alla lunga rispetto al Running, ed è anche **meno rischioso**.  
I costi possono essere inoltre **variabili** (in base all’output della produzione) **o fissi**.

### 57. As a Service vs Bare Metal: pro and cons

* **X As A Service** (**Infrastructure, Software, Platform**)🡪 Qualcosa offerto **nascondendone la complessità interna** (no necessità di manutenzione/aggiornamento o processi di configurazione e gestione complessi). **CAPEX bassi e OPEX alti**, ma **senza costi nascosti e prezzi generici** che rimangono abbastanza **bassi**; la **scalabilità** è data **di design**.
* **Bare Metal (On premise)** 🡪 Il contrario, **si è proprietari e si controlla tutto**. **Maggiore CAPEX e minore OPEX**. Ci sono i **costi nascosti di manutenzione e di operazioni**, in più è necessario **pianificare la scalabilità**. Inizialmente può essere un **costo maggiore**, se pianificato bene a lungo termine può restare abbastanza basso.

### 58. Describe a Waterfall approach and the GANTT tool

L’approccio **Waterfall** per guidare un team è un **approccio molto statico**, in cui si hanno **fasi ben distinte e sequenziali** tra loro; ciò che si progetta all’inizio non può essere cambiato, e si rischia di arrivare alla fine con un progetto sbagliato.  
I passi sono: **Raccolta requisiti**, **design** della soluzione, **implementazione**, **testing e mantenimento**. **Ogni fase necessita dei deliverables** delle fasi precedenti.

Il **grafico Gantt** rappresenta le **attività necessarie per portare a termine** un certo progetto, con riferimento al **lasso temporale**, alle **dipendenze** e allo **sforzo necessario** (che rappresenta quante persone devono lavorare ad ogni attività, **FWT**). Permette la rappresentazione grafica dello stato di avanzamento del progetto.

### 59. How an Agile project is structured?

Un Progetto Agile è **basato sul Deming Cycle**, ossia **Pianifica – Fai – Controlla – Agisci** per migliorare; questo è un ciclo continuo di miglioramento che permette di essere molto **più flessibili**.  
Ogni ciclo è inserito in un **lasso temporale pre-stabilito** chiamato **Sprint**, e **alla fine** di ogni sprint un **MVP** viene generato, per poter raccogliere feedback dagli stakeholders.

**Prima dell’inizio** del progetto, tutti i **bisogni** sono raccolti **nel product backlog**.  
I **ruoli** individuabili sono: **Product Owner** (prioritizza il Product Backlog), **Scrum Master** (controlla che le regole Agile vengano applicate), il **Team Agile** e gli **Stakeholders** (chiunque sia minimamente interessato/coinvolto dall’applicazione finale).

Ci sono diversi tipi di **incontri**: **Standup Meeting**, **Sprint Review**, **Backlog Review** e **Sprint Retrospective**.  
Le **necessità di alto livello** sono studiate e **divise in piccole entità dal team Agile**; quando ogni bisogno è piccolo abbastanza da poter essere sviluppato in uno sprint, diventa una **User Story** (divisa successivamente in task atomici). Vengono decisi un certo insieme di User Stories per ogni sprint, e solo quelle devono essere fatte.  
Molto importante il focus sulla documentazione e il versioning del codice.

### **M -** 60. What should I do if my business ask me “I need to know if an invoice is going to be paid over time or not (overdued invoices).”

Devo subito capire **cosa** devo **sviluppare e perché**, **più altri dettagli** da parte **del business**.

Capire **come**:

* **Integrare con i sistemi già presenti** usati per registrare le fatture (**quanti** sistemi, che **tecnologie** hanno, che tipi di **permessi** ci sono)
* **Parte di Machine Learning** 🡪 **Quando** bisogna comunicarlo? In **real-time** a tempo di creazione della fattura o a **batch** time, ogni giorno? Quante persone si hanno nel team (per **bilanciare accuracy, recall e precision** usando la **matrice di confusione**)?
* Come si vuole visualizzare il risultato? **Presentazione**
* **Dettagli tecnici** 🡪 Si è liberi di **usare** qualsiasi **Cloud**? Che **Authentication Providers** sono presenti e che **Access Control List**?

Capire i **requisiti tecnici** (estrarre tramite **job CDC da legacy system**, avere un **layer di analisi avanzata**, **allenare algoritmi di Machine Learning**…) e costruire la soluzione logica; dopodiché **selezionare il software**, **modellare i componenti e i moduli (astratti e concreti)** e **stimare il tempo di realizzazione e i costi** (con **tipo di costo** e **confronto** tra **As a Service e On Premise**).

Fare una presentazione per mostrare lato business i dati e stupirli, per poi alla fine preparare gli sprint Agile con Azure Devops.

### 61. Differences between Query Context and Filter Context

Nel **query context**, l’intenzione è di **assegnare comunque uno score** al documento, per poi **ordinare l’output** dei documenti in base allo score ricevuto (**must, must not e should**).  
Nel filter context, non si assegna alcuno score ma semplicemente si ignorano documenti dal risultato (molto veloci da computare).

### **M -** 62. What is a Shard? Replica logics to ensure reliability and scalability

E’ **simile ad un DataNode** (basato su **Storage Object Pattern**), si ha un **Master node eletto automaticamente** che è incaricato di **operazioni generiche** quali la **creazione dell’indice** o la **gestione delle risorse**. Il **Name Node non è unico**, **ogni nodo sa dove qualunque documento è** presente in modo da mandare le richieste in modo diretto.

Possono essere **primari o repliche**, e **più shard** possono essere inseriti **in ogni nodo**.  
Uno **shard** è un vero e proprio **motore di ricerca a sé stante**, capace di usare tutte le risorse di un singolo nodo, e contiene i documenti.  
Per avere **maggiore affidabilità e scalabilità**, è conveniente avere **primary e repliche su diversi nodi** (**no**n c’è **single point-of-failure** e si hanno più punti di accesso per read che permettono una **velocità maggiore**). **Più repliche** ci sono, **maggiori** saranno le **performance delle query** (ma **minori quelle delle write**, che possono essere ricevute solo dal primary che le propagherà alle repliche).

### 63. Explain what an index query is and how to deal with typical NLP issues

ElastichSearch è basato su vari **inverted index** (le righe sono sulle parole, le colonne sui documenti), composto da **documenti JSON** dello **stesso argomento** e la **rilevanza** è calcolata tramite il **TF-IDF**. L’**analyzer applicato** ad un certo valore deve essere applicato nello stesso modo **alla query** (a meno che non si usi term/terms).

I problemi di **spelling e root** della parola sono risolti quasi totalmente tramite l’**Analyzer** (che **normalizza**, **tokenizza** e procede al **rimuovere** *stopword*, **cambia** *lowercasing* e **aggiunge** *sinonimi*), con tecniche di **lemming e stemming** (**Porter**). Rimane **più difficoltoso** i casi in cui si abbiano **parole diverse con lo stesso significato**, o **parole identiche con significato differente in base al contesto**.

L’**Analyzer arricchisce i dati ricevuti**, parsati e normalizzati prima di indicizzare; si tratta dunque di un modello **Extract, Enrich & Post (ETL)**.