Architectures for big data

Francesco Tomaselli

15 marzo 2021

Indice

1 Architetture

1.1 Introduzione

1.1.1 Stile architetturale

Uno stile architetturale definisce la lista di design elements utilizzabili, oltre alle relazioni che sussistono tra essi. Nel contesto di sviluppo software uno stile incapsula le scelte importanti prese dagli architetti e ne coordina le interazioni. Esistono due problemi, erosion e drift, il primo consiste nell'usare l'architettura impropriamente, il secondo si manifesta quanto un'architettura viene usata per scopi diversi da quelli originali.

1.1.2 Elementi

Per quanto riguarda gli elementi di un'architettura, essi si suddividono in:

1. *Data elements*: contengono informazione
2. *Connecting elements*: connettono diverse parti dell'architettura
3. *Processing elements*: trasformano i dati

1.1.3 Materiali

Un materiale ha qualche peculiare caratteristica, è necessario scegliere materiali adatti agli scopi. Nel software, si parla di framework, linguaggi di programmazione, etc ...

1.1.4 Point of views

È cruciale avere diversi punti di vista nella presentazione di un architettura, poichè esitono più attori con ruoli differenti. Ad ognuno interessano o meno alcuni dettagli o aspetti dell'architettura totale.
Un esempio potrebbe essere una vista ad alto livello data al cliente, e una molto specifica data allo sviluppatore. Si può pensare poi ad una vista riguardo alle spese, che molto probabilmente non importerà ad esempio al team di sviluppo.

1.1.5 Astrazione

Per astrazione si intende l'individuare un pattern, nominarlo, definirlo, analizzarlo e trovare un modo di invocarlo tramite il suo nome, così da evitare errori. L'idea di base è quella di omettere i giusti dettagli nel contesto opportuno, semplificando l'interpretazione del risultato.

1.2 Motivazioni

Esistono vari motivi per cui si architetta un'architettura software.

1. Framework per soddisfare richieste
2. Base tecnica per il design, ovvero, è la base per la definizione di un particolare design, che ha tra i suoi elementi un'architettura a supporto
3. Base per la stima dei costi di un sistema
4. Porta al riuso delle componenti
5. Porta alla centralizzazione dei dati
6. Aumenta produttività e sicurezza
7. Mitiga il rischio di lock-in

1.2.1 Vendor lock-in

Condizione in cui un cliente dipende da un certo vendor. Tale dipendenza si manifesta se il costo di cambiare vendor è maggiore rispetto al tenerlo. Per esempio, deciso un cloud o un servizio, un'azienda potrebbe avere adattato il suo sviluppo a quel particolare ambiente. Un cambio di cloud o tecnologia porterebbe a costi di adattamento troppo importanti.
Esiste anche il *knowledge lock-in*, ovvero quando il costo di spiegare o impartire una conoscenza in qualcuno costa più di mantenere la persona attuale. Per proteggersi da vendor lock-in si può ricorrere ad *Adapter Pattern*.

1.2.2 Design pattern

Un design pattern è la formalizzazione di una best practice per risolvere un problema comune. Ad esempio: singleton, SOA, REST, P2P, MapReduce ...

1.3 CDC e Datalake

1.3.1 Datalake

L'idea di un Datalake è quella di avere un luogo in cui salvare dati strutturati, ottenuti da una certa sorgente in forma non strutturata. Su un datalake si salva solamente, non si elimina nulla, alla modifica di un dato si inserisce la sua versione modificata.
Tale approccio cambia il patter *ETL* poichè, invece che estrarre, trasformare e caricare, si effettua una Extract e una Load per salvare su datalake, mentre in un secondo momento si trasformano i dati in base all'utilizzo.

1.3.2 Change data capture

L'idea del CDC è quella di salvare solo i dati che sono nuovi o aggiornati dall'ultima raccolta dati. Ad esempio, nella raccola giornaliera di una tabella SQL, è possibile, invece che salvare più volte la tabella intera, salvare solo le tuple nuove o modificate, riducendo di molto lo storage utilizzato e il costo di computazione seguente.
Esistono vari modi di implementare CDC:

- *Invasive database side*: timestamp su righe, numero di versione, indicatore di stato, trigger su tabelle
- *Invasive application side*: si basa su eventi, un'applicazione deve mandare informazioni sullo stato al cdc
- *CPU database*: si confrontano i log del database, oppure utilizzando log shipping, tipicamente coinvolto nel backup automatico

1.3.3 Diff and Where

Si differenziano due tipi di tabelle, log e registry table. Nella prima esiste un *chronoattribute*, utilizzabile per ottenere i dati aggiornati tramite query, nella seconda si applica un approccio basato su hash per capire se le tuple sono state modificate dall'ultima volta.
Nello specifico, considerata una tupla di esempio $(x, y, ...)$, possiamo definire una chiave e prendere il suo hash, $KHASH = hash(x)$, e calcolare l'hash del resto della tupla, $HASH = hash((x, y, ...) - x)$. Si salvano in un dizionario chiave e valore hashati, chiamato $SYNC$.
Nella ricerca di tuple nuove o modificate, si effettua il calcolo di chiave valore per ogni tupla della tabella, si confrontano tali dizionari:

- Una tupla è *nuova* se $KHASH \notin SYNC$
- Una tupla è *modificata* se $KHASH \in SYNC \land HASH_{new} \neq HASH_{sync}$
- Una tupla è *invariata* se $KHASH \in SYNC \land HASH_{new} = HASH_{sync}$

1.3.4 Move and rename

L'approccio consiste nel garantire transazionalità di un CDC, è necessario quindi rendere possibile un eventuale rollback e non lasciare mai il datalake in uno stato inconsistente.
L'idea è scrivere su datalake i file in formato tmp e rinominarli al termine del job cdc. All'iterazione successiva si eliminano eventuali file tmp, lasciati da un eventuale interruzione del CDC.

1.3.5 Adapter pattern

Per evitare vendor lock-in, si utilizzando classi wrapper per evitare di dipendere strettamente da metodi definiti da un vendor. Un esempio può essere un source o destination adapter, che espongono metodi read e write, implementati a seconda del contesto.

1.4 Jobs e scheduler

1.4.1 Job

Un job è un elemento atomico che identifica una sequenza di passi o task da compiere. Può essere lanciato interattivamente o schedulato. Si identifica con un singolo processo.
Un job è caratterizzato da un *id*, un *nome* e uno *stato*, può essere poi:

1. *Finito*: possono completare, essere terminati o fallire
2. *Online*: possono essere solo interrotti quando terminano

1.4.2 Scheduler

Governa l'esecuzione dei job, può essere basato sul tempo o su eventi. Può decidere a quale processore o server (sistema distribuito) mandare un certo job, scegliendo tra quelli liberi.

1.4.3 Job queue

Coda dei job che sono da eseguire, esiste un concetto di priorità, riconducibile alle classiche priorità dei processi nei sistemi operativi.

1.4.4 Job impersonation

Un thread che esegue un job può impersonare un determinato client. Ad esempio, dopo un login, un thread può agire impersonando l'utente loggato, in modo che il server possa rispondere nel modo opportuno.

1.4.5 Concurrency

Due job possono essere sequenziali o concorrenti. Nel primo caso il secondo attende la terminazione del primo, nel secondo caso no.

2 Apache Hadoop

Apache Hadoop è un framework open source con l'intento di aumentare l'affidabilità di un sistema, basandosi sul fatto che l'hardware puo fallire.
Ha tre componenti: *storage*, *resource management* e *computazione parallela*.

2.1 Elementi

2.1.1 Job Tracker

Servizio che decide dove una task deve essere eseguita.

2.1.2 Task Tracker

Nodo che accetta la task data da un job tracker. Espone slot che possono eseguire task parallele, infatti, all'arrivo di una nuova task, uno slot libero viene utilizzato per eseguire una JVM. Manda segnali di heartbeat.

2.1.3 Name node

Contiene informazioni su dove i dati sono scritti. Mantiene quindi una lista di risorse e i relativi nodi che le contengono. Rappresenta un sigle point of failure per Hadoop. Esiste un nodo di backup pronto in caso di fallimento.

2.1.4 Data node

Nodo che contiene i dati effettivi.

2.1.5 Master e worker node

Un nodo master è composto da un job tracker, un task tracker, un data node e un name node. Mentre un nodo worker è composto da data node e task tracker.

2.2 Processo

Moving computation is cheaper than moving data

Un esempio del processo seguito da Hadoop è il seguente.

1. Un'applicazione manda un job asincono al *Job Tracker*, aspettando in polling.

2. Il *Job Tracker* cerca i dati nel *Name node* e sceglie i Task Tracker più vicini con slot disponibili.

3. Il *Job Tracker* manda i job ai *Task Tracker*, e ne monitora lo stato attraverso i segnali di heartbeat.

4. Nel caso di fallimento il *Job Tracker* può scegliere di aggiungere il *Task Tracker* a una *blacklist*.

2.2.1 Confronto costi

Bisogna tenere presente che in un contesto come quello di Hadoop risulta più economico muovere la computazione piuttosto che i dati. Bisogna quindi scegliere i task tracer in modo opportuno, tenendoli vicini ai data node interessati, piuttosto che muovere i dati in sè ed eseguire le operazioni.
Nel primo caso infatti si parla di muovere magari GB di informazioni, nel secondo di muovere righe di codice.

2.3 HDFS

HDFS è uno storage distribuito pensato per essere utilizzato su macchine poco performanti.

2.3.1 Architettura

Un HDFS ha un architettura *master/slave*, dove esiste un *name node* per *N data node*. Il primo esegue operazioni sul file system, mentre i secondi si occupano di servire le richieste del primo e di gestire cancellazione, creazione e replica dei file.

2.3.2 Data node

I file sono suddivisi in blocchi, le scritture non sono concorrenti sullo stesso blocco. Mandano segnali di heartbeat al name node.

2.3.3 Quorum Journal Manager

I data node sono affidabili, viste le tecniche di replicazione, per estendere ai name node, essi potrebbero essere multipli, uno attivo e gli altri in standby. I data node manderebbero il battito a tutti questi nodi. Al fallimento del name node attivo, esso sarebbe rimpiazzato da un altro nodo.

3 Spark

Apache Spark è un framework open source per il calcolo distribuito.

3.1 Elementi

3.1.1 Resilient Distributed Dataset

È un *dataset distribuito* in memoria su cui si possono effettuare operazioni in parallelo.
Si può ottenere da un HDFS, oppure parallelizzando qualsiasi oggetto.
L'idea è quella di distribuire gli elementi tramite una funzione di hash tra i nodi che offrono computazione, tale distribuzione può essere forzata con un *repartition*.
L'operazione è necessaria poichè ci si potrebbe trovare in casi dove un nodo ha molti più elementi degli altri, oppure per raggruppare meglio le chiavi in seguito a una join.

3.1.2 Driver

Processo responsabile dell'esecuzione di Spark, divide l'applicazione in task piccole computabili dai nodi worker, che eseguiranno.
Racchiude lo *Spark Context*, che permette di utilizzare Spark, e i metadati per l'RDD.

3.1.3 Master

È un nodo che contiene il driver, si occupa di *orchestrare* il lavoro tra i *nodi worker*, e ne monitora lo stato. Può avere un nodo di backup pronto in caso di fallimento.

3.1.4 Spark Context

È il core di Spark, permette al driver di utilizzare il cluster. È singleton e manda segnali di *heartbit* agli *executor* per monitorarne lo stato.

3.1.5 Worker

Nodo che si occupa della *computazione*, contiene molti executors.

3.1.6 Executors

Sono quelli che effettuano le computazioni vere e proprie. Hanno un id, ed è garantito il backup se falliscono.

3.2 Funzionalità e caratteristiche

3.2.1 Pigrizia

Spark è pigro, non computa il risultato subito ma solo quando serve. È possibile quindi che un errore si verifichi solo all'esecuzione di una particolare riga dell'RDD.

3.2.2 Map e reduce

Spark offre alcune funzioni di map e reduce. La prima permette di applicare una funzione ad ogni entry dell'RDD, mentre la seconda permette di eseguire riduzioni. Le seconde devono essere commutative, poichè non è garantito l'ordine con cui si eseguono le operazioni.

3.2.3 DAG Scheduler

Ogni qualvolta che si effettua un'operazione su un RDD non si sta facendo altro che aggiugnere una task al DAG Scheduler. Il suo compito è quello di trasformare un *execution plan* logico in uno fisico, in modo da effettuare la computazione vera e propria. Ogni passo del DAG Scheduler è chiamato *Stage*.

3.2.4 Task Scheduler

Uno stage su un sottoinsieme di righe è chiamata Task. Una Task è lanciata dal Task Scheduler ed eseguita da uno Worker attraverso il Cluster manager. Il numero di Task è proporzionale al numero di partizioni considerate nell'RDD.

3.2.5 Shared Variables

All'esecuzione di una map o reduce, Spark distribuisce una copia delle variabili delle funzioni tra tutte le righe. Questo comportamento può portare a problemi, nel caso di variabili molto pesanti.
Si introducono le shared variables, che possono essere:

- *Broadcast variables*: read-only, per esempio, dizionario condiviso in memoria su cui fare lookup
- *Accumulators*: write-only

3.3 Processo

Definiti elementi e caratteristiche di Spark l'esecuzione puo essere riassunta in questi passi:

1. Submit di un'applicazione utilizzando spark-submit utility
2. Allocazione risorse necessarie dal Resource Manager
3. Application master si registra al Resource Manager
4. Spark driver manda il codice all'Application Master, convertendo il codice in un DAG
5. Il Driver negozia con il Cluster Manager sulle risorse, e si creano gli Stage del DAG Scheduler
6. Gli Executors sono istanziati dagli Worker
7. Il driver tiene traccia dello stato degli Executors e manda le task al Cluster manager in base alla distribuzione dei dati
8. L'application Master crea una Container configuration per il Node Manager
9. È creato il primo RDD
10. Durante l'esecuzione il driver parla con l'Application Master per monitorare lo stato, al termine si rilasciano le risorse

3.4 Spark to SQL

La scrittura di dati da Spark a un database SQL potrebbe risultare critica e rallentare di molto il sistema. Infatti, la presenza di molti worker che tentano di scrivere in parallelo non aiuta, rendendo il database un collo di bottiglia.

3.4.1 Table index

Struttura dati che aumenta le performance in lettura di una particolare tabella del database, non è necessario infatti effettuare una scan completa, ma si individua immediatamente la riga necessaria. In fase di inserimento, sono necessarie scritture addizionali per aggiornare l'indice.

3.4.2 Problemi

Esistono problemi nella scrittura da Sparkl a SQL. In particolare, le operazioni di write occupano memoria, bisogna stabilire una connessione fisica e le tabelle indexed vanno lockate.

3.4.3 Struttura interna database

Un database SQL è composto da data file e log file. I primi sono raggruppati in extents composti da pagine. Essi possono essere uniform, se ogni pagina è una data page, oppure mixed, se alcune pagine contengono informazioni sugli indici.

3.4.4 Esecuzione query

All'arrivo di una query, questa viene convertita in un TDS utilizzando uno tra ODBC, OLEDB, SNAC, o SNI. Successivamente, SNI sul server decapsula il TDS e passa i comandi al query parser. Il query executor esegue la query e richiede agli access methods le pagine che sono coinvolte dalla query. Se si parla di query non select, interviene la logica di log e lock. La prima logga le operazioni, la seconda effettua il lock sulle pagine per garantire ACID properties.

3.4.5 Tecniche di connessione

Per connettersi a un dataabase esistono tre metodi:

1. *Connessione diretta*: si apre una connessione con una socket, bisogna riaprirla ogni volta
2. *Connection pooling*: si mantengono connessioni aperte e vengono assegnate ai client che le richiedono
3. *Pool fragmentation*: si creano molti pool e ogni client può usare il proprio pool o no, differisce dal secondo perchè ogni client ha il proprio pool

3.4.6 Creazione e update di un indice

Se una tabella risulta indicizzata, un inserimento deve tenere conto di alcune cose. Utilizzando ad esempio un B-tree+, i dati sono solo nelle foglie, le distanze tra le foglie non cambiano.
Per creare un indice i dati sono inizialmente ordinati, si sceglie una root e si aggiungono dati ad essa, quando questa è piena, si aggiunge un'altra root figlia e si procede.
Per aggiornare un indice, prima si cerca la posizione dove mettere l'informazione, se la pagina trovata è piena, si splitta la pagina ricorsivamente fino alla root.

3.4.7 Scrittura intermedia Data Lake

Per risolvere il problema di scrittura in SQL da parte di worker multipli, si possono scrivere tutte le informazioni su un Data Lake e successivamente scriverle in bulk su SQL.
L'idea è quella di sfruttare il BCP, bulk copy program, per copiare tutte le righe nuove da datalake a SQL, e utilizzare MERGE per unirle alla vecchia tabella.

La MERGE può definire logiche di unione.
L'approccio evita la creazione di dirty pages.

4 Delta Lake

In un contesto big data esiste la sfida di rendere transazionali le operazioni. In particolare potrebbe essere necessario garantire consistenza e comunicazione tra i processi che stanno svolgendo particolari operazioni.

4.1 Transazionalità

Per garantire transazionalità è necessario un controllo sulla concorrenza.

4.1.1 Transazione

Sequenza di operazioni che soddifano le proprietà ACID.

4.1.2 Rollback

Riportare il sistema allo stato precendete all'operazione eseguita.

4.1.3 Proprietà ACID

Le proprietà che sono necessarie per garantire transazionalità sono:

- *Atomicity*: ogni operazione di una transazione è atomica
- *Consistentcy*: una transazione deve mantenere la consistenza del sistema
- *Isolation*: esecuzioni concorrenti portano allo stesso risultato di esecuzioni sequenziali
- *Durability*: una transazione non può essere ripristinata dopo un commit

4.2 Transaction log e shipping

Per garantire consistenza e replicazione si possono applicare *transaction log* e *transaction shipping*.

4.2.1 Transaction log

Storia delle operazioni eseguite da un database, possono essere utili per effettuare rollback in casi di errore di alcune transazioni che hanno lasciato il sistema in uno stato incosistente. Questi file possono diventare molto grandi.

4.2.2 Transaction shipping

I log possono essere mandati a un server per replicare lo stato del database e avere una copia esatta del sistema. È un'operazione molto pesante poichè bisogna scrivere tutte le operazioni che sono effettuate sulle tabelle.

4.3 Delta Lake

Delta lake ha l'obiettivo di garantire le ACID properties su HDFS, quindi su storage distribuiti.

4.3.1 Apache Parquet

Parquet è un formato con cui mantenere le tabelle. Usa memorizzazione per colonna piuttosto che per riga, per velocizzare le operazioni di select.
Dividendo per colonna poi si può ottimizzare lo spazio richiesto per la codifica dei valori nella tabella. Infatti, i valori di una colonna sono più omogenei di quelli presenti in un'intera riga, quindi richiedono meno spazio e si possono comprimere. Parquet non è abbastanza per garantire transazionalità.

4.3.2 Write-ahead log

Si mantengono quali oggetti fanno parte di una delta table, oltre ad aclune infomazioni statistiche quali massimo, minimo, count, etc ...
Si scrivono poi le operazioni da fare sulle tabelle nel log, prima di eseguirle, oltre a quelle da applicare in caso di rollback.

4.3.3 Funzionalità

Le funzionalità principali di un Delta Lake, che lo rendono propenso a un forte lock-in, sono:

- *Time travel*: esiste la possibilità di riportare lo stato dello storage a un determinato snapshot temporale.
- *UPSERT, DELETE e MERGE*: riscrivono in modo efficiente gli oggetti per applicare le operazioni richieste

- *Streaming efficiente*: le informazioni sono scritte in modo veloce poichè si suddividono in piccoli blocchi. I blocchi si uniscono dopo per velocizzare le letture
- *Caching*: i nodi del cluster possono cachare informazioni
- *Schema evolution*: si può cambiare lo schema delle tabelle continuando a leggere i vecchi file Parquet

5 Docker

Docker è un set di platform as a service che offre la possibilità di racchiudere software in containers.

5.1 Componenti

5.1.1 Container

I container sono isolati uno dall'altro in termini di librerie e configurazioni, ma possono parlare attraverso alcuni metodi. L'idea è quella di utilizzare *cgroups* del kernel Linux, che permette di limitare l'uso delle risorse da parte di un certo insieme di processi. In tal modo un container docker è meno impattante di una virtual machine.

5.1.2 Docker file

Contiene tutti i comandi necessari per assemblare l'immagine. Utilizzando *Docker build* si compila il docker file eseguendo i comandi.

5.1.3 Docker daemon

Si occupa di eseguire i comandi definiti nel docker file, ogni riga è eseguita e considerata come un singolo step. Modificando una certa riga, solo le righe successive ad essa verranno eseguite.

5.2 Sintassi

I comandi possibili in un docker file sono

- `FROM`: inizio di ogni docker file, specifica la parent image dalla quale si sta effettuando la build. Può essere preceduto da alcuni parametri.
- `RUN`: esegue qualsiasi comando bash-like come step.

- `COPY`: copia un file dalla stessa cartella del docker file in una qualsiasi locazione all'interno del container.
- `CMD`: esecuzione di un'operazione alla startup di un container.
- `ADD`: come `COPY`, ma supporta URL
- `EXPOSE`: usato per esporre un protocollo
- `ENV`: definizione di variabili locali
- `USER`: user che sta eseguendo un certo comando

5.3 Processo

Il processo di creazione di un container docker è la seguente:

1. Creazione di un docker file con i comandi necessari
2. Esecuzione di `docker build`
3. Esecuzione di `docker images` per vedere tutte le immagini disponibili
4. `docker run -image-` per eseguire il container
5. `docker ps` per vedere i container attivi, `-a` per vederli tutti
6. `docker start/restart/stop/kill`

È possibile orchestrare più containers con docker compose, tramite file di configurazione yml.

6 Service-oriented architecture

La base di SOA è quella di fare da ponte tra il mondo IT e quello business, pensando un'architettura come qualcosa basato su servizi. Essi collaborano e possono potenzialmente creare applicazioni molto complesse.

6.0.1 Strati SOA

Una SOA può essere vista a cinque strati:

1. *Sistemi operativi*: asset IT attuale
2. *Componenti dei servizi*: realizzano le funzionalità e garantisono la qualità dei servizi esposti, utilizzando le risorse del punto 1
3. *Servizi*: servizi deployati
4. *Processi di business*: layer che combina i servizi del livello sottostante per creare un sistema più complesso
5. *Clienti*: user che utilizza l'applicazione, oppure altra applicazione, etc.

6.1 Servizi

Un servizio è una risorsa identificata da un nome che esegue un compito ripetitivo. È descritta da una service specification, senza scendere nei dettagli, ma rimanendo ad alto livello. Si può vedere quindi come una black box.
Ogni servizio offre le proprie funzionalità sotto forma di contratto.

6.1.1 Elementi

Ogni servizio è costituito da:

- *Nome*
- *Versione*
- *Tipo di esecuzione*: asincrona, sincrona, ...
- *Metodo di logging*: before, after, ...
- *Message in*: configurazione dei messaggi in ingresso
- *Message out*: configurazione dei messaggi in uscita
- *Visibilità*
- Eventuale *binding* con server objects, definiti da un *application container* e un *integration implementation*

6.1.2 Server catalog

Catalogo di tutti i servizi disponibili, contiene una lista dei message in usati per configurare l'esecuzione dei servizi.

6.1.3 Service queue

Lista delle esecuzioni dei servizi. Monitorandola si può controllare lo stato dei servizi in real time.

6.1.4 Enterprise service bus

Sistema di comunicazione che permette ai servizi di comunicare con legacy systems. È utile per evitare la *system-to-system integration* ovvero integrazioni specifiche per ogni sistema. L'idea è quella di far comunicare ogni sistema con l'enterprise service bus, in modo da non occuparsene lato servizio, bensì lato legacy system.

6.1.5 Service broker

Implementa il broker pattern, ha come compito il mandare i servizi al corretto application server, controllando la service queue. Si occupa anche di controllare lo stato dei *message ins* e *message outs*. Fornisce inoltre la descrizione di un servizio quando richiesta da un client, e controlla che un determinato clien abbia i requisiti di accesso richiesti.

6.2 Modelli di comunicazione

6.2.1 Sincrono

Il client compila il message in, l'ESB guida la richiesta, il service broker si occupa di inviare al server corretto, il quale esegue e risponde sul message out. L'esecuzione è corredata da uno stato e un log. I log sono mantenuti all'interno della Service Queue Table.

6.2.2 Asincrono

Il client compila il message in, il service broker aggiunge il servizio da eseguire in coda. Quando il turno del servizio arriva, il broker troverà l'application server corretto. Esiste un ID, ritornato immediatamente, che il client può controllare per monitorare lo stato del servizio richiesto.

6.2.3 Streamed outbound file reference

Quando l'output è molto grande, è infattibile scrivere sul message out. Si può estendere la logica asincrona in modo da scrivere l'output su un certo file. È possibile iniziare a consumare l'output durante la scrittura.

6.2.4 Pub-sub service call-back

L'idea di un pub-sub è che un client possa iscriversi a un determinato topic e ricevere una notifica ad ogni evento. L'ESB può evitare al client la complessità del message bus, ad ogni nuovo evento un object id è passato al client.

6.2.5 Batch sync data process

Un job CDC permette di estrarre dati incrementali da un legacy system, salvandole su un data lake. Un servizio gateway invia queste richieste al target system, es SQL.

7 From big data to big money

Your solution is impressive... but, how much?

Nell'architettare una soluzine IT, bisogna tenere conto di due aspetti fondamentali, il beneficio e il costo.

7.1 Costo

Descrivere i costi di un sistema risulta talvolta complesso, bisogna tenere conto di molti aspetti a volte non quantificabili.

7.1.1 CAPEX

Capital expenditure, racchiude i fondi spesi per migliorare gli asset dell'azienda, ad esempio edifici, veicoli, macchinari o software.

7.1.2 OPEX

Operating expense, racchiude i fondi spesi per mantenere l'esecuzione di un prodotto, un esempio è una licenza software, o la manutenzione di risorse aziendali, tra le quali, il software.

7.1.3 Esempi

Alcuni esempi dei due tipi di costi appena menzionati.

EG	CAPEX	OPEX
Comprare software	X	
Comprare hardware	X	
Soluzione software ad-hoc	X	
Tassa mensile software		X
Servizio cloud		X
Colloquio consultant per start-up		X

7.1.4 One-shot

Costo di sviluppo iniziale, un costo one shot elevato puo limitare il running cost, che puo essere pericoloso. È un altro modo di vedere i costi, rispetto a CAPEX E OPEX.

7.1.5 Running cost

Costo di far girare una soluzione, può risultare pericoloso.

7.1.6 Costo variabile

Costo che cambia al variare dell'output prodotto, per esempio, il costo di aumentare il numero di utenti in una particolare applicazione.

7.1.7 Costo fissato

Costi che non dipendono da fattori variabili, per esempio, il salario dei dipendenti.

7.2 Tipi di progetto

Esistono sostanzialmente tre tipi di progetto, che variano a seconda di quanto siano chiare le varie parti dello stesso.

7.2.1 Turnkey project

Lo scopo del progetto è chiaro, il cliente pagherà quando il progetto è concluso. Il progetto è considerato un CAPEX

7.2.2 Time and Material

Lo scopo del progetto non è chiaro ma lo sono le attività da svolgere. L'idea è quella di assumere un lavoratore molto capace.

7.2.3 Body Rental

Il progetto e le attività non sono chiare, ma l'effort si. È necessario quindi un qualsiasi tipo di lavoratore, anche con poche skill.

7.3 As a service VS bare metal

7.3.1 X as a service

Si riferisce a qualcosa offerto come servizio, la cui complessità è nascosa al cliente finale. Alcuni esempi sono:

- *Infrastructure as a service*: hardware offerto da un certo provider

- *Software as a service*: possibilità di usare un software senza installarlo
- *Platform as a service*: permette di utilizzare piattaforme complesse senza preoccuparsi di configurazioni complesse

7.3.2 Bare metal

Il contrario del concetto as a service. Consiste nel creare i servizi richiesti, non appoggiandosi a nessun provider di servizi. Il costo CAPEX aumenta, ma probabilmente diminuisce l'OPEX. Bisogna però tenere conto del running cost della cosa sviluppata.

7.3.3 Fake as a service

A volte si pensa che trasferire tutti i legacy systems su cloud sia un esempio di IAAS, la verità è che porta a molti svantaggi.

7.3.4 Confronto

Cost	As a service	On premise	Fake as a service
CAPEX	Min	High	Min
OPEX	High	Low	Med
Costi nascosti	No	Manteinance	Manteinance
Scalabilità	By design	Need to plan	Need to plan
Costo 1 anno	Low	High	Low
Costo 3 anno	Med	Med	High
Costo 5 anno	Med	High	High

Se invece si facesse un buon piano, la scalabilità non sarebbe troppo un problema per la parte On premise, quindi, il costo finale sarebbe paragonabile a quello As a service.

7.4 Team development

7.4.1 Project management

Processo di portare il lavoro di un team verso il completamento di un certo goal. L'idea è quella di completare tutte le richieste in un determinato periodo di tempo.

7.4.2 GANTT chart

Grafico con task e settimane, con le relative associazioni.

7.4.3 Waterfall

Metodo di sviluppo a cascata, ovvero le fasi del progetto sono sequenziali. Non si possono cambiare i requisiti in corso, non è flessibile e potrebbe portare a problemi.

7.4.4 Deming cycle

L'idea è quella di procedere per cicli, iterando e migliorando la soluzione di iterazione in iterazione. Porta a risultati migliori del metodo a cascata.

7.4.5 Agile

Si suddivide il lavoro in sprints, ad ogni sprint gli sviluppatori consegnano una versione funzioante del sistema.
Prima di partire si raccolgono i requisiti ad alto livello, formalizzati in un *Product Backlog*.
Esistono vari ruoli:

- *Product owner*: si occupa di aggiungere le priorità del Product Backlog
- *Scrum master*: si occupa di controllare che non si rompa il modello Agile
- *Team*: gruppo di sviluppatori
- *Stakeholders*: valutano il prodotto

Esistono vari tipi di *meeting*, giornalieri, o meno recenti che controllano lo stato e la fine di uno sprint.
Ogni requisito nel Product Backlog diventa una User Story, ogni sprint si occupa di completarne un certo quantitativo.

8 Elasticsearch

Why do we need search?

In alcuni scenari è molto importante la ricerca, ad esempio in un Ecommerce oppure un search engine.
È impossibile creare una ricerca tramite un database classico, molto probabilmente il risultato che si vuole ottenere non corrisponde a una query singola, probabilmente si sta facendo una ricerca fuzzy.

8.1 Indicizzazione

Elasticsearch è un engine distribuito di ricerca e analisi per dati di qualsiasi tipo, strutturati o no.

È basato su Apache Lucene, che offre analisi e ricerche testuali, lo estende e lo semplifica. Offre una REST API e costituisce la base dell'EKL stack.

8.1.1 Processo

Per popolare Elasticsearch si seguono solitamente questi passi:

1. I dati si raccolgono da uan sorgente
2. Parsing e normalizzazione dei dati e indicizzazione Elastic
3. Query sull'indice, anche molto complesse
4. Data visualization con Kibana

8.1.2 Indice

Un indice Elasticsearch è una collezione di documenti che per qualche ragione sono raggruppati. Un indice ha varie caratteristiche:

- Ogni documento è un JSON
- Ogni indice ha un *inverted index* che mantiene le parole presenti in ogni documento, questo garantisce ricerche full-text molto efficienti
- Elastisearch promette di riuscire ad indicizzare ogni documento in meno di un secondo
- Un indice è distribuito in shards, garantendo ridondanza e scalabilità

8.2 Text preprocessing

8.2.1 Bag of Words

Dato un insieme di testi, si crea una lista di di parole univoche ed ordinate che compaiono in essi, questo sarà l'Absolute dictionary. Ogni documento ha associato un vettore, che contiene per ogni parola dell'absolute vector il numero di occorrenze nel testo. Si crea quindi una matrice di documenti sulle righe e parole sulle colonne.

8.2.2 Inverted index

L'approccio opposto al bag of word è quello di associare ad ogni parola, invece che ad ogni documento, i documenti in cui essa è contenuta. La matrice quindi è inversa, le parole sono sulle righe e i documenti sulle colonne.

8.2.3 Analyzer

Elasticsearch prima di indicizzare i testi si occupa di applicare alcune trasformazioni ai testi, per evitare alcuni problemi legati all'analisi testuale puntuale.

8.2.4 Lemming and stemming

Prima di procedere a un'analisi testuale è necessario un preprocessing. L'idea è quella di ridurre ogni parola alla sua forma base, in modo da unificare in un unico termine forme complesse dello stesso verbo, o singolari e plurali dei termini. Stemming è euristico, mentre lemming è deterministico.

8.2.5 Stop words removal

Un'altra cosa che bisogna fare prima di procedere ad un'analisi testuale è la rimozione delle così dette stop words, ovvero congiunzioni o termini specifici del linguaggio.

8.2.6 Mapping

Un indice elastic cerca di capire il tipo degli attributi degli oggetti contenuti nell'indice. Per ogni attributo si può poi definire come gestire l'indice, le opzioni possibili sono:

- *Analyzed*: si applica l'analyzer, ovvero stemming, lemming, tokenization etc.
- *Not analyzed*: indicizza il testo così com'è
- *No*: campo non indicizzato

Esiste poi la possibilità di avere tipi innestati, che sono appiattiti da Lucene.

8.3 Query

Ogni query assegna uno score ai documenti, in base a quanto matchano la query.

8.3.1 Query context e Filter context

Le query a un indice Elasticsearch sono in json, esistono vari tipi di query, che differiscono per complessità. Ad ogni documento viene assegnato uno score di matching.
Un esempio di query è quella *bool*, che permette di definire condizioni molto complesse, con clausole *must*, *must not*, *should*, *filter*.

Tutte le clausole a parte la prima fanno parte del *Query context*, la seconda invece del *Filter context*. È consigliato quindi utilizzare per query full text, condizioni che devono contribuire allo score nel query context, e tutto il resto del filter, il quale eliminerà molti documenti con altrimenti score nullo.

8.4 Shards e scalabilità

Elasticsearch è basato su uno storage object pattern, quindi la scalabilità orizzontale è garantita. L'architettura è abbastanza simile a quella di Hadoop.

8.4.1 Master node

Esiste un nodo master che si occupa di richieste generali come creare indici e gestire le risorse.

8.4.2 Name node, kinda

Non esistono veri e propri name nodes come in Hadoop, infatti, tutti i nodi contengono informazioni su dove sono i documenti, in modo da reindirizzare le richieste verso il nodo che contiene le informazioni richieste.

8.4.3 Shard

Ogni cosa è organizzata in shard, ognuno dei quali può essere primario o una replica di uno primario. Ogni nodo ha multiple shard.
Uno shard è un search engine funzionante, e può essere utilizzato a sè.

8.4.4 Scalabilità

Distribuendo gli shard tra i nodi si possono garantire elevate performace e più scalabilità.

9 NPL and semantic

Per effettuare analisi testuali si ricorre inizialmente a un modello Bag of Words, come discusso a sezione precedente. Bisogna poi però trovare un modo di confrontare due testi sotto forma di vettore. Si introducono quindi misure di distanza e di similarità.

9.1 Distanza e similarità

Computando la distanza tra due vettori, dove i vettori sono bag of words, si può calcolare la distanza di due testi.

9.1.1 Distanza Euclidea

La distanza tra due vettori è data da

$$dis(A, B) = \sqrt{\sum_{i=0}^{n}(A_i - B_i)^2}$$

È infattibile computare la distanza euclidea per vettori molto sparsi molto lunghi.

9.1.2 Cosine similarity

La similarità tra due vettori è data da

$$cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_{i=0}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=0}^{n} A_i} \cdot \sqrt{\sum_{i=0}^{n} B_i}}$$

La computazione della cosine similarity risulta meno intensa rispetto alla distanza euclidea. Infatti, considerando ad esempio una computazione fatta su Spark, utilizzando la distanza euclidea bisognerebbe mantenere tutti gli 0 nel vettore per ogni documento, infatti uno 0 può contribuire alla distanza nel caso in cui l'altro documento abbia valore 1.
Nel secondo caso uno 0 non contribuisce alla cosine similarity, quindi si possono mantenere, per ogni documento, solo i termini in esso contenuti.