Alma Mater Studiorum · Università di Bologna

Ingegneria Informatica
Corso di Laurea Magistrale in INGEGNERIA INFORMATICA

Calcolo di indicatori di performance aziendale in contesto Big Data

Relatore:
Chiar.mo Prof.
Claudio Sartori

Correlatore:
Francesca Marchi

Presentata da:
Fabio Ricca Rosellini

Sessione I
2017/2018

Indice

Capitolo 1

Introduzione

L'obiettivo di questa tesi è esplorare le caratteristiche e le potenzialità del mondo Big Data applicato al contesto aziendale, in particolare, creare un sistema automatico e configurabile per estrarre dai dati aziendali degli indicatori (KPI = Key Performance Indicator) utili agli utenti di business nel loro processo decisionale. Tale progetto permette la creazione di report e "dashboard" rappresentanti KPI basati sui contenuti televisivi messi in vendita dall'azienda. L'architettura che si è adottata per tale sistema è quella del Data Lake, per rendere possibile agli utenti più esperti la self service BI. Il Data Lake richiama la struttura di un corso idrico, da qui i nomi delle aree. Tale architettura prevede la creazione di 4 o più aree dove possono risiedere i dati. Queste aree servono per distinguere e organizzare i dati in base al loro livello di *purezza*. Nella *Swamp Area* sono contenuti i dati grezzi, in questa fase iniziale si tratta dei messaggi twitter. Essi vengono immagazzinati nella forma e struttura originaria della sorgente. Una peculiarità del Data Lake rispetto al Datawarehouse è la scelta d'immagazzinare tutti dati senza, per esempio, filtrarne le parti eventualmente ritenute inutili in quel momento. Tale approccio permette di non perdere mai alcuna informazione. Successivamente i dati vengono rielaborati e spostati nella *Working Area* grazie a processi di data integration, o ETL, atti a standardizzarli. Nella *Harbour Area* si immagazzinano i dati provenienti da altri sistemi, poiché essi presentano una struttura già standardizzata. Nella *Reservoir Area* si effettua l'ETL finale che lega i dati fra loro rendendoli pronti per la fase di Data Visualization. Questa architettura ha permesso di sperimentare diversi componenti open-source per Big Data, nello specifico Kafka, Kudu, HDFS e Impala. La necessità di creare un sistema pronto e adatto ai Big Data è nata non appena ci si è confrontati con la mole di dati in ingresso provenienti da Twitter. Twitter, e nello specifico i tweet, sono stati alla base della progettazione poiché data la velocità con i quali arrivano, e il numero degli stessi, richiedono una gestione particolare. Infatti l'uso di Kafka [9], una piattaforma distribuita per lo streaming, mettendo a disposizione delle code ove è stato possibile inserire i tweet tramite API Java, ha permesso di ordinarli e renderli fruibili alle successive fasi di data integration. Un altro aspetto interessante dell'uso di Kafka è la sicurezza

che esso aggiunge, infatti, su sistemi distribuiti, le code sono replicate, innalzando così il livello di fault tolerance del sistema. Per la memorizzazione e l'integrazione è stato utilizzato un altro componente opensource, Kudu [10]. Kudo è uno strato di storing che si posiziona sopra Hadoop, permettendo analisi efficienti su grosse mole di dati. Kudu é un sistema NoSQL, supera, ove necessario, le classiche logiche dei linguaggi SQL per permettere interrogazioni e analisi molto più efficienti. Per potersi interfacciare con tale *layer* si è dovuto impiegare Impala, che grazie al suo design architetturale moderno ha permesso analisi efficienti.

Impala [8] ha permesso analisi a bassa latenza e alta concorrenza sui dati. Capace d'interfacciarsi con Hadoop e Kudu, è stato lo strumento principale per DDL e DML. Un'altra caratterista molto importante di tale tecnologia è quella di *scalare* linearmente, ovvero è possibile attivare più istanze dello stesso esecutore parallelizzando il carico di lavoro. Inoltre utilizza la sintassi SQL, rendendo disponibili tutti costrutti dell'SQL classico. La peculiarità di tale sistema è quella della configurabilità. È infatti possibile inserire svariate sorgenti informative per arricchire l'analisi. Tale analisi ha come fulcro il titolo del contenuto. Si possono quindi estrapolare svariate informazioni, quali il ritorno economico del contenuto, per esempio, e la popolarità dello stesso, ricavata come *rumore* web e numero di telespettatori. Per poter visualizzare le analisi in modo chiaro ed efficiente, permettendo perfino la self service BI, si è scelto Tableau. Grazie a questo software è stato possibile creare svariate dashboard configurabili e interattive. L'importanza della Data Visualization non è seconda a quella dell'elaborazione effettiva dei dati, poiché l'utente di business si approccia al sistema attraverso i report e i grafici.

Capitolo 2

Background

2.1 Big Data

Si è passati in poco tempo da una realtà dove le aziende non avevano abbastanza dati per prendere decisioni corrette a uno scenario dove non si dispone di competenze e di strumenti adeguati per dare valore all'enorme vastità di dati che velocemente ci inondano. Tali dati sono definiti Big Data.

Per Big Data si intende l'insieme di tutte le tecnologie e metodologie di analisi di dati massivi. Si intende quindi la capacità di estrapolare, analizzare e mettere in relazione un'enorme mole di dati eterogenei, strutturati e non strutturati, per scoprire i legami tra fenomeni diversi e prevedere quelli futuri. Un sistema di Big Data è definibile tale poiché i dati da analizzare eccedono le capacità dei sistemi hardware e software usati comunemente per catturare, gestire ed elaborare i dati in un lasso di tempo ragionevole per un insieme ampio di utenti. I Big Data possono provenire da una moltitudine di sorgenti ed assumere una moltitudine di forme, fra le quali:

- pubblicità
- post e news feed
- raccomandazioni
- marketing
- pubblica sanità
- sicurezza
- ecc.

Tali dati possono essere sia storici che in real time, spaziando da un feed di tweet da Twitter, come nel nostro caso di studio, allo storico degli ordini dei clienti dell'azienda, ai dati dei sensori in ambito IoT o di log di un macchinario.
Come riporta Gartner [20] "i Big Data sono tutta quell'informazione ad alto volume, alta velocità e varietà che richiede nuovi sistemi di analisi e sistemi economici di storage che permettano analisi per il *decision making* delle aziende". Un'altra descrizione, proposta da Forrester [19] cita: "I Big Data sono quell'insieme di tecniche e tecnologie che permettono di operare su dati enormi in modo economico". L'aspetto economico è infatti cruciale; se non è *conveniente* immagazzinare e gestire tali dati, essi perdono completamente di valore.

Stiamo generando dati ad una velocità inimmaginabile in passato e il costo dello spazio di archiviazione è ai minimi storici. Negli anni 90 un GB costava 10k€, ora 0,07€. Un aeroplano ad ogni volo, per via di tutti i sensori su di esso installati, genera 5TB di dati. Facebook e Twitter dei petabyte ogni giorno. Riuscire ad estrapolare informazioni da questi dati è un vantaggio commerciale immenso.

Con l'arrivo dell'*internet of things (IoT)*[32], sempre più dati sono generati in ogni istante, senza contare tutti i social network. La sfida è quindi analizzare tutti questi dati, in tempi accettabili, efficientemente. Hadoop ha reso possibile tutto ciò impiegando *commodity hardware*. Prima che arrivasse il cloud non era possibile economicamente salvare tutti i dati in server dedicati. Ora con la flessibilità dei cluster di Hadoop tutti possono accedere al mondo dei Big Data; ora se si vogliono analizzare terabyte di data basta avviare un nodo Hadoop, fare le analisi e ridistribuire tali risorse per altri impieghi.

Tradizionalmente, i dati venivano salvati in una singola unità e tutte le richieste passavano attraverso quell'unico server. Se le prestazioni non fossero state adeguate o le risorse richieste fossero aumentate, si sarebbe aumentata la potenza computazionale aggiungendo memoria o aggiornando la CPU. Questo processo è chiamato *scale-up* o scaling verticale. Questo approccio si è usato per anni per colmare il gap fra performance richieste e risorse disponibili. Questo approccio evidentemente ha dei limiti. Il costo dell'hardware, via via che si sale come prestazioni, non è lineare, tutt'altro. Ci si troverà ad un punto dove aggiornare non sarà più economicamente fattibile.

La soluzione è quindi suddividere il carico in orizzontale, parallelizzando il più possibile. Si aggiunge del commodity hardware e le performance aumentano di conseguenza. Hadoop ha fornito il software in grado di parallelizzare il carico di lavoro su più macchine.

2.1.1 Big Data Characteristics

I Big Data possono presentarsi sia in uno stato strutturato, semi strutturato o totalmente destrutturato. I *data scientist* hanno individuato 6 caratteristiche, dette le *"6 V dei Big*

Data" per cercare di catturarne le peculiarità.

Volume

La loro caratteristica peculiare è proprio la quantità. Infatti prendendo solo Twitter e i dati prodotti da sensori di un macchinario, con una frequenza di rilevazione di 100ms, si può ben comprendere la magnitudine dei dati in questione. Inoltre il principio base dei Big Data è quello di salvare e storicizzare il dato indipendentemente dal suo immediato utilizzo e nella sua forma più grezza, dato che ogni operazione di pulizia, modifica o filtro potrebbero far perdere informazioni magari utili dopo molti anni. Inoltre ciò rende obsoleto l'uso dei classici RDBMS per memorizzare tali dati poiché ciò richiederebbe investire ingenti somme per lo storage e la capacità computazionale. Esistono MPP (Massive Parallel Processing) utilizzate nel data warehousing per velocizzare tali operazioni, ma falliscono nel fronteggiare un'altra peculiarità dei Big Data, l'eterogeneità. Per questo negli anni si è sviluppato fra le tecnologie Open Source lo standard de facto dei Big Data, ovvero Apache Hadoop. Questo dovuto alla sua capacità di memorizzare grandi moli di dati a costi contenuti usando hardware commodity.

Variety

Data l'incredibile varietà delle sorgenti, basti pensare ai post di Facebook e ai dati telemetrici di un motore, diventa impossibile il salvataggio di tali dati secondo un modello comune per tutti, come invece prevedono i database relazionali. Quindi la seconda caratteristica dei Big Dati, ovvero la Varietà, ha portato ad un cambio di paradigma, dato che appunto i dati possono essere destrutturati. Tale paradigma è chiamato schemaless database. Ad esempio molti sistemi NoSQL permette di analizzare e salvare i dati senza imporre una rigidità nella struttura.

Velocity

Un'altra caratteristica dei Big Data è la velocità con cui sono generati. Infatti oggigiorno ogni dispositivo elettronico è capace di generare dati con frequenza molto elevata, si pensi ai telefoni, alla nostra macchina, al nostro comportamento su internet ecc. La nuova sfida che si apre ora è non solo il salvataggio ma l'elaborazione, preferibilmente in real time. Questo perché per le aziende avere un vantaggio temporale nel trovare pattern nei dati, trend e in generali informazioni utili, è fondamentale nei mercati tanto vorticosi di oggi. Sono per questo nati altri strumenti open source quali Kafka e Kinesis, utili alla gestione di flussi streaming e capaci di interrogare i dati in tempi brevissimi.

Veracity

Un altra importante sfida alla quale i Big Data ci mettono di fronte è il controllo della correttezza e coerenza del dato. Questi problemi nascono dal fatto che principalmente si tende a centralizzare tali dati, creando potenzialmente delle incoerenze, data la diversità delle sorgenti. Bisogna quindi prestare attenzione ai processi automatizzati di inserimento dati che possono essere inaccurati.

Value

Un altro aspetto, forse il più importante, è il valore, o meglio trasformare i dati in informazione e quindi in valore. Infatti il data warehousing è lo strumento principe per gestire i big data che il business di una azienda utilizza per prendere decisioni per la propria azienda trasformando, di fatto, i dati in valore. Quindi è fondamentale riuscire a dare valore all'azienda che ha investito in una soluzione per big data.

Visualization

Ultimo ma non di minor importanza è la visualizzazione di tali dati. Infatti la soluzione di big data scelta deve poter comunicare l'informazione in modo chiaro all'utente di business. Se l'informazione non è comunicata in maniera chiara e incisiva, il dato perde valore. Per questo esistono diversi software di data visualization sia open source che a pagamento, qui di seguito introdurrò Tableau.

Per poter essere definiti big data tuttavia l'insieme di dati che si analizzano devono possedere un sottoinsieme di tali caratteristiche, ma non necessariamente tutte. Infatti se ci si trova davanti ad una grossa mole di dati strutturati i classici RDBMS, magari parallelizzati su più macchine, possono svolgere il compito egregiamente. La varietà dei dati può essere gestita usando codice ad hoc alla sorgente e alla destinazione. La velocità può essere gestita con tool specifici quali Microsoft SQL Server StreamInsight [29]. Non sempre i dati che si trattano sono big data, quindi non sono necessari tool specifici, sebbene ugualmente applicabili. Tuttavia questi tool, data la loro predisposizione a scalare e a fornire altre garanzie specifiche, non sempre sono i più leggeri o performanti dato l'intrinseco overhead.

2.2 Big data Management

Il mondo dei Big Data negli ultimi anni ha portato alla proliferazione di tecnologie. Hadoop in primis ha permesso, mantenendo i costi bassi, a molte aziende di entrare in tale

mondo.

Abbiamo avuto molta scelta per le tecnologie da usare dato il mercato opensource offre tale vastità di alternative. Le abbiamo analizzate, così da poter fare la scelta giusta per il nostro progetto.

2.2.1 Ingestion and Data-Processing of Real Time Data

La velocità con cui i big data possono arrivare e il crescente interesse nel volerli analizzare con bassissima latenza, ha portato alla nascita di servizi ad hoc capaci di processare tale mole di dati in tempi impensabili con le classiche tecnologie di Hadoop.

Ci sono quattro grandi nomi per quanto riguarda la ricezione e la gestione di streaming di dati in real time - Apache Kafka [9], Kinesis [1], Flume [4] e Storm [13]. Tutte e quattro sono capaci di gestire grossi flussi di data, siano log, dati da *IoT*, post da social media, ecc . Tuttavia ognuna ha i suoi punti di forza e debolezze.

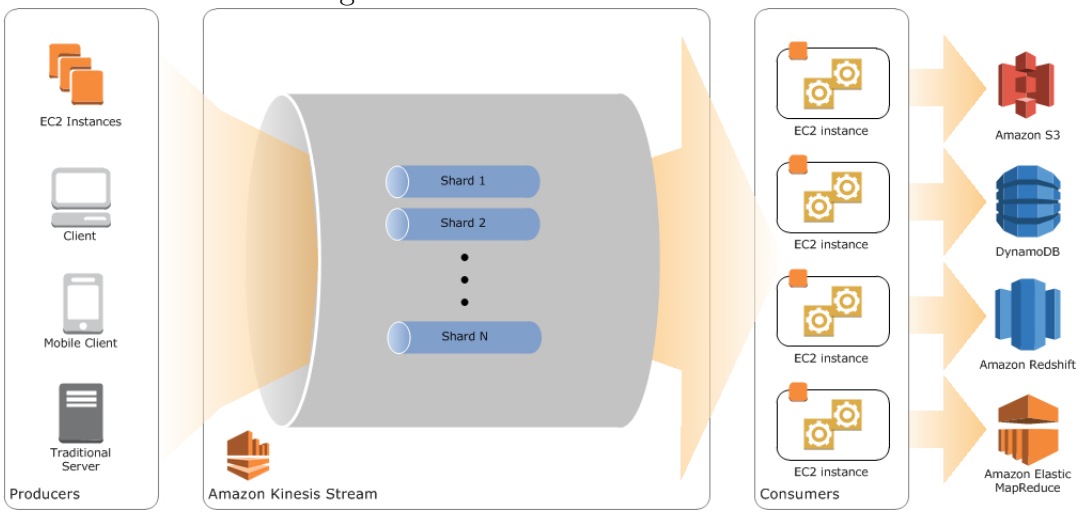
Kafka, la nostra scelta, è anche uno dei più famosi processatore di dati real time. Nata da Linkedin [23], è stata adottata da Netflix [24], PayPal [25], Spotify [28] e Uber [36]. Kafka, in breve, è un sistema distribuito di *messaggistica* che mantiene i messaggi in code chiamate topic. Caratteristica chiave di questa tecnologia è la possibilità di replicare su più nodi i dati, fornendo una sicurezza maggiore e un aumento del throughput. Vedremo nel capitolo 3.1 la sua struttura e il suo funzionamento.

Amazon Kinesis si comporta esattamente come Kafka. Tuttavia sebbene Kafka sia gratis, c'è molto lavoro di configurazione da fare. Kinesis invece è un pacchetto enterprise ben inserito nell'ecosistema Amazon. Kinesis permette la repliche dei dati in *shards*, quelle che Kafka chiama partizioni. Il pagamento è per shards-ora e per mole di dati.

Kinesis può essere usato per inserire dati da stream in modo continuo e consistente. Questi dati, come ricordato prima, possono essere log di sistema, post dai social network e flusso di click su un sito web.
Sebbene esso sia anche uno stream processor, ovvero capace di effettuare delle operazioni sui dati in real time, il suo utilizzo principale è quello di aggregare i dati prima di darli in pasto ad un data warehouse.

Come mostrato in Figuera 2.1, si può notare come i producer immettono i dati nello stream, diviso in shard, per poi essere consumati dai consumer.

Figura 2.1: Architettura di Kafka



I componenti interessati nel passaggio dei dati:

- **Kinesis Stream**:è una sequenza ordinata di record; ogni record è accompagnato da un numero univoco. Tale sequenza è divisa in Shrad.
- **Record**: ogni dato è un pacchetto detto record. Ogni record contine un ID, una chiave un *blob*. Il blob è una sequenza immutabile di byte di massimo 1 MB non ispezionata da Kinesis.
- **Shards**: è una sequenza univoca di dati. Ogni shrad ha una capacità massima predefinita di MB.
- **Partition key**: tramite questa è possibile suddividere i dati nelle varia shard.

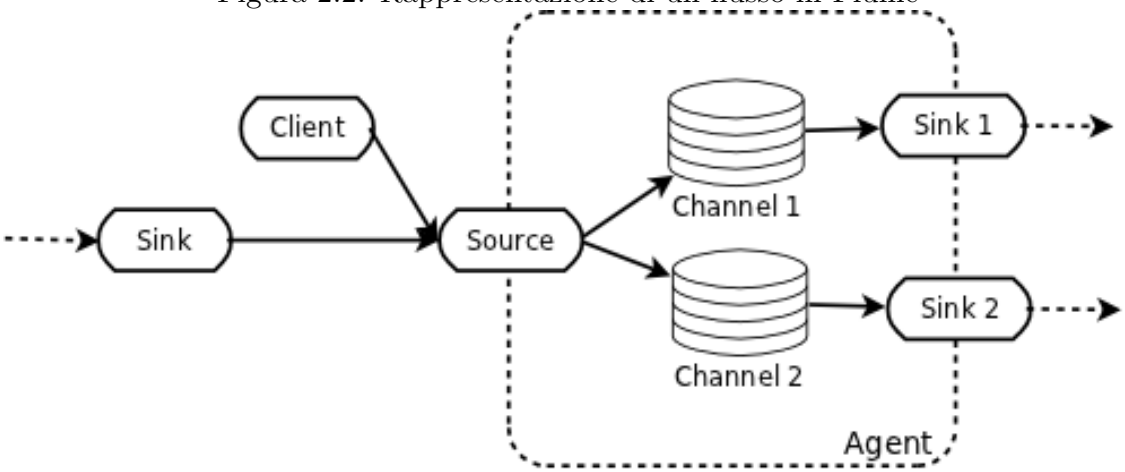
Apache Flume è un altro servizio di gestione di flussi di data, ma più incentrato sui log di sistema. Il suo obiettivo è quello di essere un servizio in gradi di trasportate i messaggi in modo affidabile e distribuito. A differenza di Kafka e Kinesis che delegano all'utente (*pull*) di prelevare i dati, Flume invia (*push*), tramite delle *sink*, i dati a tutti i client (che possono essere HDFS, HBase [6], Cassandra [2] e qualche db relazionale. Contrariamente su Kafka bisogna scrivere programmi ad hoc, utilizzando le API proprietarie, per creare il collegamento con le sorgenti e con i client. L'archietettura di Flume è composta da diversi componenti, che unitamente garantiscono a Flume di raggiungere tale obiettivo:

- **Event**: un evento è l'unità base del dato, esso dev'essere prelevato dalla sorgente ed inviato alla destinazione.
- **Flow**: il flusso è il movimento di tali dati, ad alto livello.

- **Client**: il client è quel componente che dalla sorgente fornisce i dati all'agente Flume.
- **Agent**: l'agente è il processo che coordina sorgenti, canali e destinazioni.
- **Channel**: il canale è un luogo temporaneo dove i dati risiedono prima che vengano consumati da una destinzione
- **Sink**: o destinazione, è l'interfaccia che preleva i dati e li rende disponibili al prossimo flusso o li scrive in una destinazione, tipo HDFS con *Flume HDFS sink*.

Il semplice funzionamento di Flume è rappresento in Figura **??**.

Figura 2.2: Rappresentazione di un flusso in Flume



Flume, a differenza di Kafka, non prevede repliche, quindi inutilizzabile in contesti dove la perdita di dati non è tollerabile.

Un'altra sfida aperta dai big data è stata la capacità di processare i dati. Qui infatti vedremo i *processing frameworks*. Il data processing può avvenire su dati in streaming in ingresso al sistema, o su dati già presenti.

Tali tecnologie si dividono in tre macro categorie in base al tipo di dato per cui sono disegnate. Infatti alcuni sistemi gestiscono dati in *batch*, gruppi, altri in streaming continui prima del salvataggio sul sistema e altri in entrambi in modi.

Batch Processing System

Il *batch processing* è il più maturo dei settori dato che si è sempre operato su grandi insiemi di dati statici, restituendo poi il risultato una volta che la computazione è ter-

minata.

Gli insiemi di dati su cui i sistema di batch processing operano sono:

- **Limitati**: i gruppi di dati sono finiti.
- **Persistenti**: i dati risiedono su supporti permanenti, non risiedono su memori volatili, quali la RAM.
- **Massivi**: spesso tali sistemi sono gli unici in grado di operare su enormi quantità di dati.

I batch processing sono adatti per le computazioni dove avere il set completo di dati è fondamentale. Si può pensare infatti al calcolo di somme e medie aritmetiche, dove è necessario avere una visione olistica dei dati e non una finestra limitata.

Tuttavia data la loro abilità di gestire grosse moli di dati li rende, per natura, non veloci. Non sono quindi impiegabili in sistemi dove la reattività nella visualizzazione di report e grafici è imperativa.

Fra tali tecnologie spicca **Apache Hadoop**, il primo *framework* di batch processing a guadagnare fama mondiale dopo le presentazioni e i paper di Google dove veniva descritto come loro lo utilizzassero per gestire i dati.
Le moderne versioni di Hadoop sono composte da numerosi componenti, che sinergicamente processano carichi di lavoro batch:

- **HDFS**: HDFS è il file system distribuito che coordina il salvataggio e le repliche dei dati attraverso tutti i nodi nel cluster. E' usato sia come sorgente dei dati, sia come appoggio intermedio che come destinazione per il risultato dell'elaborazione
- **YARN**: o (Yet Another Resource Manager, ovvero "ancora un altro gestore di risorse) è il componente coordinatore. E' responsabile di gestire le risorse per ogni *job*. Ha reso possibile l'esecuzione di carichi molto più diversi rispetto alle prime iterazioni di Hadoop agendo come interfaccia per le risorse.
- **MapReduce**: è il motore di calcolo nativo di Hadoop.

La sequenza di step della procedura di calcolo è la seguente:

1. Si leggono i dati da HDFS.
2. Si divide il *dataset* in blocchi e lo si divide su tutti i nodi.

3. Si eseguono i calcoli parziali su tutti i blocchi.
4. Si ricentralizzano i risultati raggruppati secondo delle chiavi
5. Si *Riduce* il valore per ogni chiave sommando e combinando i risultati dai vari nodi
6. Il risultato è poi riscritto su HDFS

Lo scrivere tutto su HDFS e non cercare di caricare tutto in RAM rende MapReduce in grado di operare solo su disco, permettendo la gestione di enormi quantità di dati su commodity hardware.

Stream Processing Systems

Lo *Stream Processing Systems* è quella tecnica di processing basata sull'analisi di dati in una finestra temporale prima che entrino nel sistema. Ciò richiede un modello totalmente diverso dal paradigma di batch. Qui le operazioni sono applicate ai dati non appena essi diventano disponibili, senza aspattare l'intero dataset.

I dataset su cui tali tecnologie operano sono:

- **Illimitati**: concettualmente non sappiamo quando lo streaming terminerà, se lo farà.
- **Sequenziale**: si considera un elemento alla volta.
- **Event-based**: il processamento non finisce finché non esplicitamente comunicato, quindi il risultato è immediatamente disponibile e rispecchia la realtà fino a quel momento; fino a che non nuovamente aggiornato.

La quantità di dati processabili è virtualmente infinita, ma vengono processati solo un elemento alla volta, veri stream processor, o pochi elementi alla volta, *micro-batch* processor.

Queste tecnologia trovano impiego in casi molto specifici, ovvero dove sono presenti requisiti *real time* o ad esempio in log di errore, dove la tempestività è cruciale.

Una delle principali tecnologie è il framework **Apache Storm** che vanta bassa latenza e quindi diventa la scelta migliore in ambiti real time.

Storm lavora orchestrando dei *DAG*, ovvero grafici aciclici orientati, in un framework chiamato *topologie*. Queste topologie rappresentano le varie trasformazioni che i dati possono subire via via che entrano nel sistema.

Le topologia sono composte da:

- **Streams**: il flusso di dati in arrivo.
- **Spouts**: sono l'ingresso dei dati, API, code ecc

- **Bolts**: sono gli step del processo che consuma lo stream; essì sono uno per ogni spouts e quanti ne servono all'interno della rete per le computazioni necessarie. Il bolt finale può essere l'input di un sistema connesso.

L'idea di base quindi è quella di creare operazioni discrete e connetterle organizzandole in topologie.

Hybrid Processing Systems: Batch and Stream Processors

Alcuni framework possono gestire entrambe le tipologia di lavoro. Ciò semplifica l'architettura perché rende disponibile entrambe le tipologie di API.

Tuttavia le modalità di raggiungimento di tale potenzialità da parte di Spark e Flink [**?**] variano significatamene. Ciò infatti dipende con che logiche le due modalità di analis, batch e non, sono unificate e come vengono pesate le relazioni fra dataset fissi e non.

Sebbene in taluni casi tecnologie ad hoc forniscano risultati migliori, l'approccio ibrido offre una soluzione generale. Inoltre tali tecnologia non portano solo la parte di processing, ma anche tante funzioni e librerie per la creazione di grafi, *machine learning* [**?**] e altro. **Apache Spark** è un **batch processor** con capacità di stream processor.

Creato secondo i principi di Hadoop, Spark porta ad un nuovo livello di performance per il batch processing offrendo computazione *in-memory*, ovvero caricata in RAM.

La sua elevata velocità deriva sia dall'ottimizzazione derivante dalla strategia in-memory, ma anche dalla visione olistica dei dati, creando per tempo, dei grafici aciclici di operazioni sui dati.

La parte di **stream processor** di Storm è presa in carico da **Spark Streaming**. Tale caratteristica è raggiunta grazie al concetto di *micro-batching*. Ciò si traduce nell'interpretare lo stream come piccoli gruppi di record, analizzandoli con le tecniche batch. Quindi Spark Streaming opera suddividendo il flusso in incrementi ogni pochi decimi di secondo, gestendo questi blocchetti come piccoli dataset. Le performance sono buone, ma non a livello di puri stream processor.

Un aspetto da non sottovalutare tuttavia è il suo funzionamento in-memory, che lo rende molto più costoso di un corrispettivo *disk-based*. Tuttavia in taluni contesti dove si pagano le risorse ad ore, l'estrema velocità può ripagare.

Apache Flink è uno stream processor con capacità batch. Al contrario di Storm, Flink considera i batch come stream finiti; ciò ha interessanti risvolti. Questo modello è chiamato *archiettura Kappa*, in contrasto con la più famosa *Lambda* dove il batch processing è la tecnologia principale, e la parte stream solo un modo di dare risultati temporanei.

Flink prevede quattro principali componenti:

- **Stream**: essi sono dataset immutabili, illimitati in ingresso al sistema
- **Operators**: sono gli operatori che possono agire sui dati.
- **Sources**: sono le sorgenti, i punti di ingresso per il sistema
- **Sinks**: sono le destinazioni, l'output. Possono rappresentare l'ingresso per altri sistemi in cascata.

L'aspetto batch è gestito da Flink estendendo il modello stream; dati sono letti da disco come fossero uno stream limitato. Gli algoritmi per i calcoli sono i medesimi.

Al momento Flink è l'unica tecnologia vialida di puro stream processing, rendendo Storm non ottimale data la latenza. Flink inoltre si ottimizza da solo, ai planner di query, ed è in gradi perfino di parallelizzare il carico di lavoro autonomamente. Ha una interfaccia web per il monitoring dei processi ed è ottimamente integrabile nell'ecosistema Hadoop.

2.2.2 SQL per i Big Data

Nel mondo di Hadoop e del NoSQL, sta crescendo l'interesse per i motori *SQL-on-Hadoop*. Ad oggi esistono diverse tecnologia in grado di fare questo. Con tali tecnologia è possibile collegare qualsivoglia tool di reportistica ad Hadoop. Prima era molto complicato poichè bisognava avere una profonda conoscenza di HDFS, MapReduce per poter interagire con i dati.

Discuteremo ora le più famose ed utilizzate.

Inzialmente sviluppato da Facebook [**?**], **Apache Hive** è una infrastruttura di data warehousing costruita su Hadoop per eseguire query, analisi, e processamenti dei dati per compiti intensivi.
E' molto versatile poichè è compatibile con grandi dataset salvati su HDFS e S3 di Amazon [**?**]. Inoltre espone un'interfacca in un linguaggio simile all'SQL -HiveQL - con caratteristiche di schema-on-read e la possibilità di trasformare in modo trasparente le query in job di Spark, MapReducde e Tez [**?**]. Hive ha le suguenti proprietà:

- Indicizzazione per accelerare le query.
- Supporto di svariati formati di file, fra i quali testo, HBase, e ORC.
- Salvataggio dei metadati in RDBMS, mantenendo bassi i tempi di accesso a tali dati durante i controlli per le query.
- Conversione autonoma di instruzioni in HiveQL in job MapReduce e altri.

- Offre molte funzioni per modifica di stringhe e date.

La sua automatizzazione per MapReduce lo rende una scelta obbligata se si ha familiarità con SQL e si vogliono creare job specifici. Sebbene abbia delle limitazioni, per taluni carichi di lavoro è molto adatto.

Apache Impala è un'ottima scelta per poter lanciare query su HDFS e, per esempio, Kudu ed HBase, dato che non richiede di muovere i dati prima dell'esecuzione. Può essere integrato facilmente nell'ecosistema di Hadoop dato che utilizza gli stessi metadati di Hive, Pig e MapReduce.
Impala utilizza una logica di storing colonnare, per ottimizzare la compressione dei dati e l'efficienza delle *scan*. La chiave delle sue alte performance è la modalità con cui esegue le query, replicandola sui nodi, senza dover spostare i dati.
Impala, non ricorrendo al MapReduce, è molto più veloce perché non si porta dietro tutto quell'overhead. Tuttavia non sempre è la scelta migliore. Infatti sebbene Impala sia molto più veloce, è molto *memory intensive* e non gira in modo efficiente se la mole di dati sono troppo grandi. Ad esempio fallisce in query contenenti molti join su tabelle grandi, dato che non è possibile mettere tutto in RAM.
Qui subentra Hive; se infatti c'è la necessità di fare operazioni *batch* su big data, si *deve* scegliere Hive. Se invece il processo real time è prioritario, meglio Impala.

Apache Spark SQL è pensato per i data scientist, piuttosto che all'ETL come Hive, o alla Business analytics di Impala. I suoi punti di forza sono la facile integrazione di query SQL in Java, Scala [**?**] o Python [**?**]. Fra le caratteristiche principali spiccano:

- Il supporto a numerosi formati, fra cui Parquet [**?**], Avro [**?**] e altri.
- Supporto completo ad HDFS e HBase.
- Supporta l'esecuzione concorrente delle query.

IL vantaggio di Spark SQL è che gli utenti Spark possono inserire query nei flussi Spark. Non è quindi concepito come layer *general purpose* di SQL per esplorare i dati. Tuttavia Spark SQL utilizzai i metadati di Hive, rendondolo pienamente compatibile con i dati di Hive. La caratteristica principale è la sua abilità a scalare fra migliaia di nodi in query anche molto complesse e lunghe, riuscendo a garantire una fault tollereance anche con query in esecuzione. Infine le performance rendono Spark SQL un'ottima alternativa.

2.2.3 Storage

Con la nascita dei big data si è dovuto pensare ad un nuovo paradigma per lo storing dei dati. Le vecchie tecnologie relazionali pongono molti limiti, in primis la difficoltà di scalare orizzontalmente. Nacquero così le tecnologie NoSQL.

Per NoSQL si intende *not only SQL* e si intendono tutti quei metodi di storing dei dati e le tecniche per accedervi con paradigmi diversi dalle classiche tabelle di database relazionale. Sebbene la denominazione NoSQL, è possibile interrogare tali sistemi con il linguaggio SQL, tipico dei relazionali, tramite layer e motori opportuni quali Impala, Hive, ecc.

Le motivazioni principali sono la possibilità di scalare orizzontalmente e la aumentata *avaiability*, ovvero la resilienza del sistema alle *failure*. Le varie tipologie di architetture NoSQL, tipo **key-value**, **colonnari**, a **grafo** e a **documento**, sono totalmente diversi dalle strutture dei database relazionali, per permettere un aumento in performance in certe operazioni.

Database a documenti

In questa architettura i dati sono organizzati in strutture simili ad oggetti, detti appunto *documenti*, ognuno dei quali con certe proprietà. La struttura dei documenti non deve essere fissa, ciò rende molto flessibile tale database.
Il documento qui, per creare una analogia con il mondo relazionale, corrisponde al record di una tabella, sebbene qui il numero di *campi* non sia prestabilito.

Database key-value

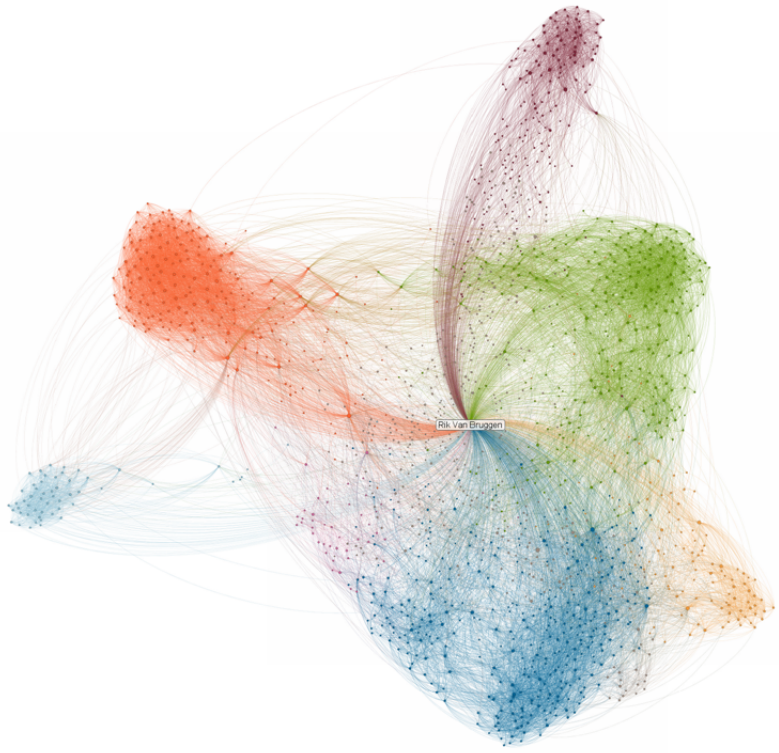
Hanno struttura e organizzazione interna molto simile a quella delle classiche`hashmap` del mondo informatico. I dati sono inseriti come tuple di due elementi, nello specifico coppie di chiave e valore.
La caratteristica che rende questi sistemi molto veloci nel recuperare i dati è la possibilità di poter recuperare il dato tramite la propria chiave. Un applicazione dove questa infrastruttura ha enormi benefici può essere quella dei sistemi di messaggistica, dove la chiave è l'account e il valore l'insieme di messaggi. Un'altra applicazione è quella dei log di sessione, dove l'ID della sessione la chiave, e come valore la lista dei dati.

Database a grafo

Proprio come nei grafi, qui i dati sono organizzati in nodi e connessioni fra esse. Il dato può risiedere sia sul nodo che sull'arco stesso. Il valore informativo aggiunto e la forza di tale database è la possibilità di individuare agilmente connessioni, anche complesse, fra i dati. La manutenzione e la gestione di una infrastruttura così organizzata non è banale, ma in alcuni contesti può essere la scelta migliore.
Si pensi ai Social Network, dove le relazioni fra gli utenti e i loro interessi, sono forse più importanti dell'utente stesso. Grazie a questa struttura quindi è agile suggerire a tali

utenti nuove amicizie, nuovi eventi o prodotti. Come si vede in Figura 2.3, la struttura può diventare molto complessa.

Figura 2.3: Database a grafo di Linkedin, visto da un utente



Database colonnari

Un database colonnare salva i dati in colonne fortemente tipate. Con una attenta organizzazione un database colonnare può raggiungere performance eccellenti per analisi e per i carichi di lavoro tipici dei data warehouse per diversi motivi.
Uno di questi è la *read efficency*, ovvero l'efficienza nella lettura, dato che in talune query è possibile leggere solo determinate colonne, senza per forza dover leggere tutta la tabella. Contrariamente, con un approccio basato sulle righe, per poter leggere una colonna, per esempio per aggregare determinati valori, si deve caricare tutta la riga.
La *data compression*, ovvero la compressione dei dati, è un altro aspetto molto importante dato che lo spazio è limitato. Questa peculiarità deriva dal fatto che essendo le colonne fortemente tipate, una compressione *pattern-based* risulterà molto più efficiente della stessa operata su una colonna ove risiedono diverse tipologie di dati. Combinata con la possibilità di leggere poche colonne, solo le necessarie, Kudu riesce a mantenere elevate e consistenti performance.

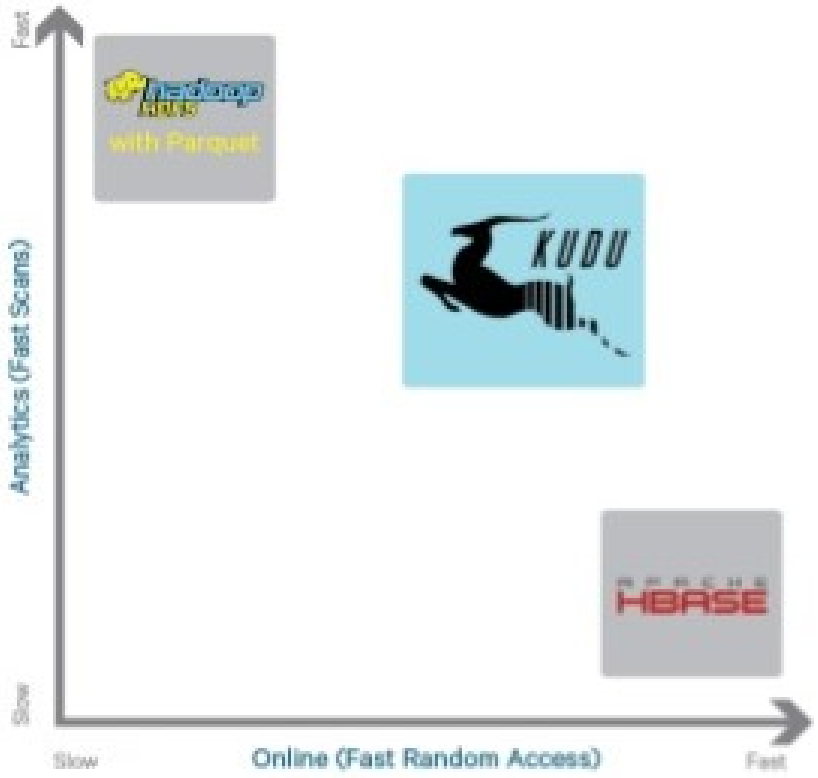
Un altro aspetto cruciale dei sistemi NoSQL è la garanzia che il sistema sia consistenze, disponibile e tollerante alle partizioni della rete (ovvero se alcuni o tutti messaggi vengono persi o rallentati per colpa della rete. La congettura di Brewer, detta *teorema CAP*, sostiene che solo due delle tre proprietà possono valere simultaneamente. E' evidente che un database relazionale è solamente CA, ovvero **c**onsistente e ***a**vailable*, ovvero disponibilità.

Quando si parla di storage in ambito big data saltano subito all'occhio quattro tecnologie; HDFS, cuore di Hadoop, HBase, Kudu e Cassandra.

HDFS non rientra in nessuna delle tipologie esposte precedentemente, per questo è molto veloce in scritture e letture, *scan*. HDFS è un file system distribuito disegnato per girare su commodity hardware. Una delle sue peculiarità è quella di essere altamente fault tollerant. Inoltre il throughput è molto elevato, rendendolo per molto tempo lo standard per queste applicazioni. E' disegnato per gestire dataset di dimensioni enormi (si parla di Gigabyte e Terabyte), infatti un singolo cluster, di centinaia di nodi, può gestire milioni di file. Tuttavia questa architettura pone diverse limitazioni, in primis l'impossibilità di comprimere i dati e di eseguire query SQL ottimizzate.

Cloudera [14] ha introdotto **Kudu** per porre rimedio a queste problematiche. Kudu è stato progettato per combinare il meglio dal mondo di HDFS e unirlo al mondo della business analytics rendendolo un *general purpose*.

Kudu si posiziona quindi *a metà* fra HBase e HDFS come mostrato in Figura :



Kudu è quindi molto utilizzato per via della sua grande flessibilità. Inoltre, data la sua natura colonnare fortemente tipata, è molto performante in quanto può comprimere molto i dati. Tale caratteristica rende anche molto veloci le analisi di raggruppamento, tipiche delle business analytics. Vedremo nel capitolo 3.2 la sua architettura e il suo funzionamento.

Un altra tecnologia è **Cassandra**. Essa segue la logica del chiave-valore ed è *eventually consistent*, a differenza di Kudu che utilizza il Raft consensus. Tuttavia, data la sua natura NoSQL non permette l'utilizzo dell'SQL (supporta però il CQL, linguaggio propietario). Cassandra è scelta ad oggi in molti campi dove sono richieste alte performance.

L'architettura di Cassandra contribuisce in maniera incisiva sulla sua abilità di scalare, performare e fornire una fault tollerance così elevata. La filosofia di Cassandra si basa sul concetto che problemi tecnici e disservizi della rete non solo possono accadere, ma *sicuramente* accadranno. Questo si traduce in un metodo nuovo di gestione dei dati e delle loro protezione rispetto ad un RDBMS.

Invece che usare il classico paradigma *master-slave*, Cassandra utilizza un approccio *peer-to-peer* distribuito molto più semplice da creare e mantenere. Fra i nodi non c'è il concetto di master, tutti sono paritari, e si scambiano informazioni secondo il protocollo

gossip.

Cassandra è sia capace di gestire petabyte di dati e migliaia di richieste da parte di utenti al secondo sia poche richieste su carichi molto leggeri, a differenza di una paradigma tipo quello di Hadoop MP, dove ogni richiesta ha un elevato overhead.

Una caratteristica peculiare di Cassandra, che la differenzia dagli altri sistemi NoSQL, è la *data consistency* configurabile per operazione. Ciò consente al programmatore di settare tale consistenza sia in maniera molto aggressiva per alcune operazioni, l'operazione verrà denominata completata solo dopo che tutti i nodi abbia risposto, sia molto lasca, ad esempio delle `INSERT` che saranno *eventually consistent*.

Un forte limite tuttavia è l'impossibilità di interrogare Cassansdra con SQL, tuttavia rende disponibile il CLQ, un linguaggio molto simile, ma meno potente. Il CQL, per questioni di efficienza, non supporta ne i JOIN ne le GROUP BY. Inoltre ogni comando di INSERT o UPDATE non legge il DB prima dell'esecuzione, creando risultati spesso inattesi. Molto limitativo per noi.

Tipologia di storing-system

Il panorama dei big data presenta due scelte su come organizzare i dati -data warehouse e data lake. L'approccio dei classici database relazionali poni molti limiti quando si comincia ad avere a che fare con grosse quantità di dati. Infatti un database totalmente normalizzato è molto inefficiente per le query, dato che per ogni minimo dettaglio si deve ricorrere a join. Inoltre ogni sorgente dati ha diverse interfacce e diverse modalità di interrogazione; potenzialmente anche diverse informazioni per lo stesso dato, rendendo il sistema inconsistente. Centralizzare tutta questa informazione è fondamentale per una vista chiara e coerente di tutti i dati.
L'scopo del Data Warehouse è proprio questo, ovvero centralizzare i dati, rendere efficienti le query e agevolare la gestione dei dati, facilitando, per esempio, l'individuazione di inconsistenze, prima che possano propagarsi e creare dei report errati.

2.3 Data Warehouse

Un data warehouse è un luogo centralizzato dove salvare tutte le informazioni, provenienti da molte e distinte sorgenti, in modo semplice e completo, rendendole disponibili agli utenti in una forma che possono capire e usare in un contesto business. Un data warehouse è orientato al soggetto, integrato, tempo variante e non volatile.

Orientato al soggetto

Il DW è disegnato per aiutare l'utente ad analizzare i dati, ad esempio può aiutare ad analizzare i dati di vendita. Creando un DW con obiettivo l'analisi delle vendite, è possibile eseguire query specifiche, come il trovare il cliente migliore ecc, in modo veloce ed efficiente. L'abilità di definire il DW in base all'applicazione, lo rende appunto orientato al soggetto.

Integrato

Una delle sfide principali del Dw è la sua abilità di integrare i dati da sorgenti diversi. La complessità di tale approccio prevedere di risolvere alcune sfide, tipo l'inconsistenza dei nomi e delle unità di misura.

Non volatile

Una volta che il dato è stato inserito non è più modificato. Inolte ne è creata una copia se il dato in questione è aggiornato o modificato nel tempo. Questa proprietà è logica, dato che lo scopo del DW è quello di creare report accurati, e la possibilità di vedere nel passato è imprescindibile.

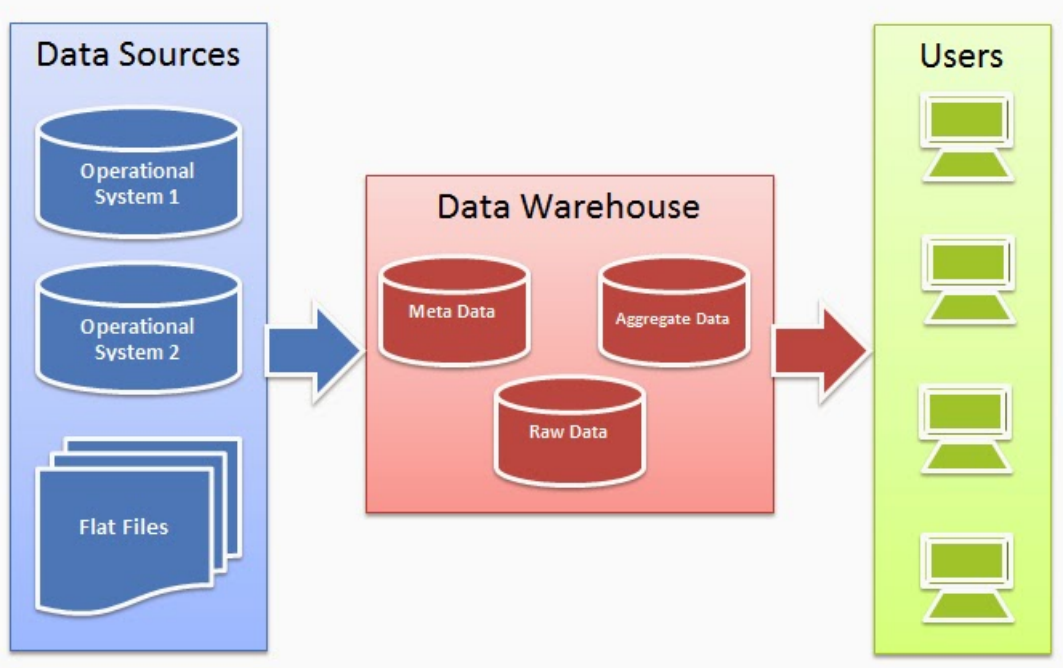
Tempo variante

Come anticipato prima, se un dato subisce una variazione esso non è sovrascritto, ma una sua copia, distinguibile dall'originale in termini di validità temporale, è creata.

Ciò che maggiormente lo differenzia da un DB relazionale è il numero di utenti che lo interrogano. Infatti un DW avrà poche centinaia di utenti, rispetto alle migliaia del classico DB. Questi pochi utenti inoltre eseguiranno per di più delle letture, e lanceranno query molto lunghe e complesse, non scritture frequenti di dati e query semplici. I dati poi che risiedono del DW sono tutta la storia del dato, in contrapposizione alla *istantanea* dei dati che mostra il DB.

2.3.1 Architettura Data Warehouse

Un DW, ad alto livello, presenta questa architettura.



Vantaggi

I vantaggi che il DW apporta all'azienda che lo utilizza non sono pochi. Infatti lo scopo del DW è di dare del valore aggiunto ai clienti dell'azienda, permettendo di accedere a maggiori e migliori informazioni. Permette quindi di riconciliare i dati da più fonti potendo così mostrare i trend dei dati su più dimensioni. Inoltre data l'alta ottimizzazione e personalizzazione dei dati inseriti, le performance delle query sono sempre molto elevate e ripetibili.

Svantaggi

Tuttavia i DW presentano anche degli aspetti negativi che ne possono limitare l'applicabilità in molti contesti. Infatti di DW non sono progettati per la gestione dei dati non strutturati. Soffrono inoltre di elevata latenza, poiché il dato, prima di essere disponibile all'analisi, deve passare da complesse fasi di estrazione e pulizia. La sua struttura di *schema-on-write*, ovvero i vincoli forti sulla struttura del dato quando esso è inserito, portano parecchie limitazioni. In primis non è possibile aggiungere dati liberamente, ma solo se appunto rispettano determinate regole. Ciò comporta anche che la manutenzione sia molto costosa, poiché ogni modifica comporta la riorganizzazione tutta l'implementazione.

L'architettura di storing dei dati, chiamata Data Lake.

2.4 Data Lake

Per gestire l'eterogeneità dei dati, per le ragioni introdotte prima, si è dovuto pensare a nuove tecnologie per lo storing dei dati. Dato che ora qualsiasi dispositivo produce dati è impossibile prevedere uno schema a priori, ma non introdurre tali dati farebbe perdere molta informazione. Nasce così il Data Lake [17], ovvero un posto dove è possibile salvare tutti i tipi di dati in tutte le forme possibili, dai perfettamente strutturati ai dati grezzi totalmente destrutturati. Il Data Lake seguendo la filosofia del bottom up, ovvero osservare il mondo e poi trovare teorie, sposa la tecnica del *schema-on-read*, quindi sono nel momento dell'analisi si crea un modello.

Una delle caratteristiche principali del Data Lake è la sua capacità di poter immagazzinare dati real time. Come anticipato prima, esso adotta l'approccio dello *store-all* che benissimo integra il concetto di streaming real time.

Un Data Lake inoltre è ottimo per fare della ricerca sui dati; proprio perché nulla è filtrato, nessuna informazione è persa. Questa caratteristica è fondamentale in un mercato così vario e dinamico, dove quindi la possibilità di avere più informazione, da più sorgenti, dei competitor è un vantaggio. Nel Data Lake infatti si effettua il ELT, ovvero, prima il dato è caricato, poi elaborato. Qui il data scientist può fare delle query ad hoc usando i dati grezzi, ovvero alla massima granularità possibile. Ciò è impossibile da fare nel Data Warehouse poiché il dato ha già subito delle modifiche, come filtri o raggruppamenti.

Un'altra peculiarità del Data Lake è che il dato non è classificato quando è inserito. Questo elimina la fase di *data preparation, cleansing* and *transormation*, che, come detto prima, priverebbe il dato di preziose informazioni. Salvare il dato nella sua forma più grezza permetterà di trovare risposte a domande che ancora non si conoscono o non si ritiene interessanti ora. In un Data Warehouse invece il dato è cucito e ritagliato per rispondere a domande precise.

Usando i sistemi tradizionali non è possibile trovare dei pattern nascosti nei dati che non si stanno collezionando. Al tempo dell'acquisizione non si conoscono ancora tutte le domande che si vorrebbero poter fare con i dati e neanche si conoscono tutti i dati di cui si possa avere bisogno. Il Data Lake cerca di porre soluzione a questo cruciale problema.

Le ragioni per cui sta prendendo piede questa tecnologia, in relazione ai limiti invece imposti dai comuni data warehouse, sono queste:

- I tradizionali data warehouse non sono progettati per integrare, scalare e gestire questa immensa crescita di dati non strutturati. Con l'arrivo quindi dei Big Data è nata la necessità e l'interesse di aggregare i dati per creare informazione. La

pletora di dati che abbiamo disposizione oggi sono testi, foto, immagini, dati dai sensori ecc ci permette di trovare un collegamento fra essi.

- L'impossibilità dei DW di integrare i dati ha lasciato proliferare i *silos*, rendendo frammentata e non omogenea la visione dei fatti da parte degli utenti.
- L'approccio *schema-on-write* dei classici DW impone la definizione del modello prima di caricare il primo bit. Questa metodologia è limitativa poichè è impossibile prevedere ad oggi quale opportunità prevederanno i Big Data. Inoltre gli *analytic framework* sono progettati per rispondere a domande specifiche identificate alla creazione del DW, magari anni addietro. Ciò non permette la *data discovery*.
- Gli approcci tradizionali sono fortemente ottimizzati per le analisi per cui sono progettati, ma totalmente inermi se i requisiti cambiano.

Il concetto di Data Lake è molto flessibile. Per catturarne l'essenza quindi lo si deve descrivere da molte angolazioni.

Il Data Lake è quindi un enorme repository dove è possibile immettere i dati nella forma più grezza possibile finché non torneranno utili. Si può erroneamente pensare che il DL sia un Database e possa esistere in autonomia. Ciò è sbagliato poiché il DL necessita di funzionalità complementari che solo un DW può fornire. Non è poi un database poiché in un database è possibile inserire solo dati strutturati, in un Data Lake è possibile inserire ogni tipo di dato.

La creazione di un Data Lake ci pone davanti a delle sfide. Infatti il Data Lake è una soluzione complessa dato che in esso coesistono diversi *layer*, ed ogni layer utilizza diversi strumenti di big data e tecnologie diverse per fornire quei servizi. Ciò richiede molti sforzi in termini di sviluppo, amministrazione e manutenzione.

Un altro aspetto importante è la *data governance*; dato che lo scopo del data lake è unificare i dati, esso dovrebbe essere costruito con cura, affinché non diventi un insieme di *silos*.

Uno dei motivi principali che ha permesso al DL di diffondersi è stato il crescente desiderio da parte degli utenti di fare analisi in autonomia, con approccio self-service [31].

Raggruppare i dati in un luogo solo ha molteplici vantaggi:

- Permette di centralizzare i dati, creando un unico punto di accesso più facilmente gestibile, senza dover credenziali ad hoc\permessi all'interno dell'azienda.
- Il Data Lake crea una zona isolata ove gli analisti possono *interagire* con i dati, fare le proprie query, senza avere conseguenze sul Data Warehouse di produzione.

Tuttavia una volta che il DL è stato creato e popolato, bisogna trovare una struttura tale da permettere agli analisti di trovare le informazioni di cui hanno bisogno. Per grosse aziende le sorgenti possono arrivare alle migliaia e il numero di tabelle segue di pari passo.

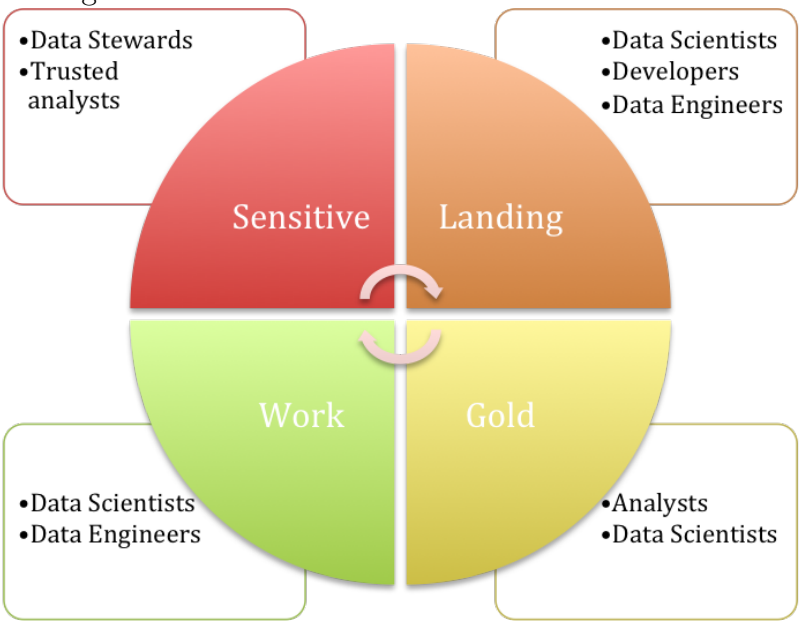
2.4.1 Architettura Data Lake

L'architettura del Data Lake non è unica e immutabile, ci sono però delle best practice concettuali che possiamo adottare.
Vediamo ora le macro categorie di *zone* o *aree* che si possono individuare in un DL (Figura 2.4) secondo la letteratura classica, soffermandoci successivamente su quelle che abbiamo individuato noi:

- **Raw o Landing zone**: in questa area sono messi i dati grezzi, così come arrivano dalle sorgenti. Qui finiscono sia i dati presi da flussi real time, sia quelli più *raffinati* presi da un DW.
- **Work Zone**: qui i dati subiscono una pulizia ed una elaborazione parziale. Inoltre è possibile dare spazio agli analisti permettendo loro di creara tabelle e appoggiarvi quelle temporanee
- **Gold Zone**: I dati qui sono nella loro forma più pura e standardizzata. Per molti analisti la Gold Zone è la sorgente principale dei dati. Qui, tramite tecnologie come Hive o Impala (o le altre dozzine che esistono), è possibile creare viste e supportare la creazione di report.
- **Encrypted Zone**: qui possono risiedere i dati sensibili, dove il controllo degli accessi è massimo.

Come si vede dalla Figura 2.4, nelle varie zone operano persone diverse con esigenze diverse.

Figura 2.4: Architettura standard di un Data Lake



Nel *nostro* Data Lake abbiamo invece individuato 5 zone, seguendo la filosofia delle 4 presentate precedentemente elencate.

Swamp Area

Qui troviamo il dato nella sua forma più grezza, ovvero la copia del dato tale e quale da come ci viene dalle sorgenti. Questa area è inoltre ottimizzata per la scrittura, dato che, essendo tutto storicizzato, non viene cancellato nulla.

Working Area

Questa area è ideata per contenere tabelle temporanee e tabelle di configurazione. Le tabella qui possono essere sia permanenti che temporanee, a differenza della Swamp. E' inoltre utilizzata come area di elaborazione per produrre dati per la Harbor e la Reservoir.

Harbor Area

I dati che sono già ripuliti o comunque hanno avuto alcune trasformazioni vengono messi in questa zona. Nel nostro caso di studio i dati provenienti dai relazionali sono stati posizionati qui. E' infatti il layer usato per mettere in comunicazione vari Data Warehouse.

Reservoir Area

Forse l'area più importante del Data Lake è la Reservoir. Qui infatti il dato è nella sua forma più raffinata. Qui inoltre si appoggiano tutti i tool di reportistica e analisi.

In questi due capitoli analizzeremo i vantaggi e gli svantaggi dell'adozione di un Data Lake rispetto all'utilizzo di un Data Warehouse. Tuttavia è bene ricordare che un DL non miri a sostituire un DW, ma ad affiancarlo ed integrarlo. Un Data Lake ha bisogno di un DW.

Vantaggi

Gli aspetti di principale utilità del Data Lake e i motivi per cui sta diventando così famoso e adottato dalle azienda possono essere racchiusi in 9 punti:

- **Scalabile**: Dato che esso si basa su tecnologie HDFS [5] o simili, aumentare lo spazio di archiviazione all'aumentare della mole di dati ha costo lineare. Inoltre un Data Lake può essere sviluppato su *commodity hardware*.
- **Sorgenti disparate**: A differenza di un Data Warehouse dove il dato dev'essere perfettamente strutturato e standardizzato, qui possiamo inserire dati da *ogni* fonte. Esse possono essere log, binari, XML, dati da sensori, post ecc. Questo approccio abbatte i silos e rende possibile integrare i dati velocemente.
- **Acquisizione di dati real time**: Per garantire prestazioni adeguate ai flussi di dati in streaming il Data Lake si appoggia a tecnologie quali Flume [4] e Kafka [9], come nel nostro caso di studio. Questo pone il DL su un altro livello rispetto al DW, ove ciò è inefficiente.
- **Salvataggio nella forma nativa**: Nel data warehouse i dati quando sono inseriti sono modellati per entrare nel cubo, rendendo molto efficienti le analisi per determinati scenari. Nel Data Lake ciò non è necessario e ciò offre una elevata flessibilità.
- **Schema**: Non essendoci uno schema predefinito il data lake si presta perfettamente alla *data exploration*.
- **SQL**: Una volta creato il DL è possibile interrogarlo con Impala, come si è fatto in questo progetto.

Svantaggi

Tuttavia il Data Lake nasconde per natura dei difetti, ai quali bisogna porre attenzione.

- **Confusione**: I dati sono inseriti senza identificatori unici e senza metadati, dando così all'estrattore il compito di partire dall'inizio ogni volta. E' quindi molto difficile cercare informazione in un mare di dati, dove quindi non esistono categorie o classi. Estrarre valore quindi non è banale.

- **Sicurezza**: Dato che in un DL tutti i dati vengono raccolti secondo il principio di ELT, non si può sapere se un dato è corrotto finché magari non è troppo tardi. Questo è avvenuto sia per la struttura del DL ma anche perché la sicurezza non era il requisito principale da soddisfare in fase di sviluppo della tecnologia.

- **Reportistica**: Gli strumenti di BI fanno fatica ad operare su dati che non sono ben organizzati, quindi almeno la strutturazione dei livelli superiori è fondamentale.

- **Integrazione**: Un'altra barriera è quella di una buona integrazione fra i dati, poiché le sorgenti sono eterogenee e il dato è spesso grezzo.

2.4.2 Similarità e Differenze

Ognuna di queste architettura presenta dei vantaggi e degli svantaggi. Per meglio capire quale tipo di architettura è la più adatta per il tipo di contesto in nostro interesse, è bene confrontare ogni peculiarità.
Come anticipato prima le differenze fra le due tipologie sono molte, rendendo tali sistemi adatti solo a talune applicazioni.

DATAWAREHOUSE	vs.	DATA LAKE
strutturato, processato	DATA	strutturato, destrutturato, grezzo
schema-on-write	PROCESSING	schema-on-read
costoso per grossi volumi	STORAGE	disegnato per abbattere i costi
meno agile, configurazione fissa	AGILITY	molto agile
maturo	SECURITY	sta maturando
professionisti di business	USERS	data scientists

Come è visibile nella tabella qui riportata le differenze sono particolarmente marcate. Se le performance di analisi e le sorgenti ingresso sono particolarmente statiche, il DW è la scelta migliore. Se invece si vuole creare un sistema dove poter organizzare ogni tipo di dato, persino in forma non strutturata, anche pensando al futuro, ovvero prendendo dati per i quali non si ha ancora un concetto chiaro, il DL è la scelta obbligata.

Tuttavia ci sono anche molte similarità:

- Entrambi sono delle repository per i dati.

- Hanno bisogno di uno scopo di business per esistere.

- Entrambi portano benefici

Scegliere quindi quale approccio sia meglio per la propria azienda è un ragionamento molto importante da fare. Molto spesso però la scelta migliore sta nell'adottare entrambe le soluzioni, per diverse necessità. Il DW è per query ad alte performance, molto ottimizzate, in modo constante e ripetibile. Il DL è per l'esplorazione, innovazione, flessibilità. Non c'è quindi motivo per cui una azienda non debba investire su entrambe le tecnologie.

Il data lake è stato scelto per questo progetto per la sua flessibilità, la sua capacità di adattarsi e di integrare diverse fonti.

Capitolo 3

Tecnologie Usate

3.1 Kafka

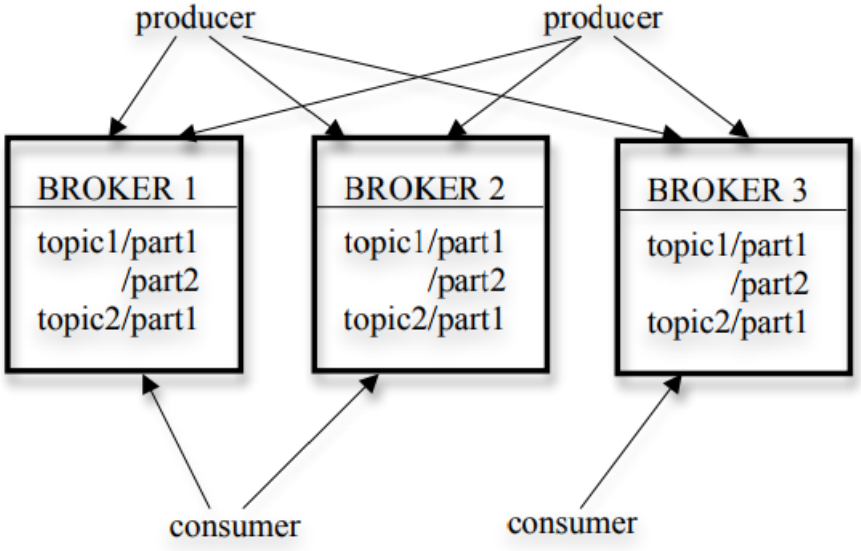
Come trattato prima non solo stanno aumentando i dati prodotti, e quindi collezionati dalle aziende ogni giorno, ma sta crescendo anche la volontà di analizzare questi flussi di dati in tempo reale. Si basti pensare alle interazioni sui social alle quale possiamo essere interessati, ai post stessi ma sopratutto ai comportamenti degli utenti, con *like*, condivisioni ecc. La produzione di dati con tale velocità ha richiesto la nascita di nuovi servizi e nuove tecnologie.

Questo ha reso inefficace il tradizionale metodo di salvataggio di dati; una tale mole di dati in così poco tempo può sovraccaricare il sistema, risultando in possibili latenze. Serviva quindi un sistema in grado di organizzare e ordinare i messaggi in ingresso. Per questo è nato Kafka, strumento utilizzato massivamente nel nostro progetto per via della sua versatilità, sicurezza e scalabilità.

Kafka è quindi un *aggregatore* di log basati sui messaggi. Ogni cosa qui è una sequenza di record ordinati. Ogni *sequenza* di messaggi di un dato tipo è chiamato *topic* dove un *producer* può pubblicare messaggi. I messaggi sono quindi salvati in un insieme di server chiamati *broker*. Un *consumer* può sottoscriversi a più topic e leggere i messaggi salvati sui broker.

3.1.1 Architettura Kafka

Figura 3.1: Architettura di Kafka



Come mostrato in Figura 3.1 ci sono 5 componenti. I *broker* ovvero i server dove risiedono i dati, i *topic*, le *partizioni*, i *producer* e i *consumer*.

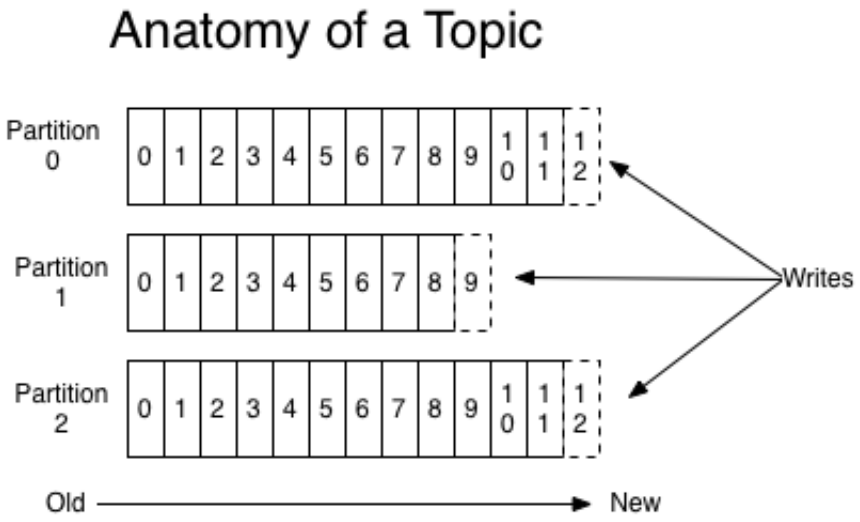
Essendo Kafka un *Messagging System* esso segue il modello del *publish-subscribe* supportando così il *multi consumer* e la parallelizzazione. Tuttavia non è l'unico modello di messaggistica esistente. Esistono tuttavia altri modelli di messaggistica, uno è Il modello di *queue*. Esso permette di suddividere li carico su più consumer, rendendo estremamente scalabile l'architettura in cui è inserito. Tuttavia un volta che il dato è stato letto, è perso.
Kafka, tramite il concetto di *consumer group*, permette sia di dividere il carico, sia di inviare il messaggio a più gruppi.

Topic

Concentriamoci sui topic, il cuore di Kafka. Un *topic* è una struttura nella quale i messaggi sono organizzati concettualmente e nella quale essi sono pubblicati. A un topic hanno accesso molteplici producer e consumer, molti possono scrivere e molti possono leggere.

Ogni topic è strutturato come in Figura 3.2.

Figura 3.2: Struttura di un topic



Ogni partizione è una sequenza di record *ordinata* alla quale i record sono appesi. Ogni record in ogni partizione ha un numero chiamato offset grazie al quale è possibile identificarlo univocamente all'interno della lista. Tale offset può essere utilizzato anche per prelevare messaggi inseriti precedentemente.

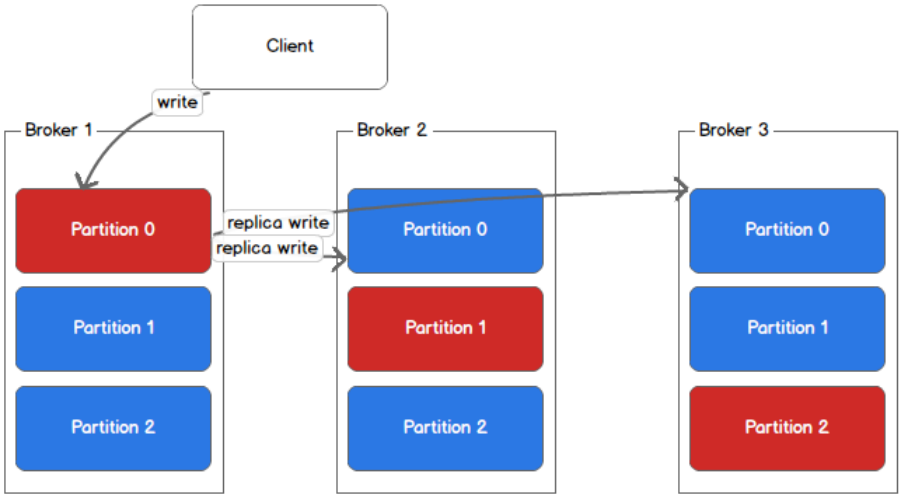
Kafka può anche essere utilizzato come storage system. Infatti ogniqualvolta un messaggio è inserito esso è mantenuto in memoria, per una durata configurabile, anche se non viene consumato.

Distribuzione

Le partizioni sono distribuite sui server. Ogni partizione è replicata su un numero configurabile di nodi per garantire la *fault tollerance*. In ogni server, o broker, vi sono più partizioni, alcune *leader* e zero o più che che *seguono* tale leader. Il leader si occupa della lettura e della scrittura, i follower si limitano a copiare tali modifiche passivamente. Se un leader si spegne, uno secondario prenderà il suo posto mantenendo la consistenza. In ogni server quindi risiedono sia leader che follower mantenendo così il carico bilanciato.

La logica che Kafka segue per garantire la replica dei dati organizza i broker in tal modo (Figura 3.3

Figura 3.3: Logica repliche su partizioni



Quando un dato è scritto su un topic esso è successivamente replicato su tante partizioni quante sono le repliche desiderate. Per garantire una fault tollerance più robusta chiaramente i broker dovrebbero risiedere su server spazialmente distanti.

Producer

Un *producer* può inviare messaggi a tutti i topic che vuole, potendo specificare inoltre su quale partizione. La logica di scelta della partizione può essere motivata da più ragioni. La prima si utilizza per gestire meglio la mole di dati, infatti se un topic è diviso in `n` partizioni equamente distribuite, `n` consumer possono *parallelamente* leggere i dati ed aumentare il *throughput*. Un'altra strategia è quella di organizzare i record in diverse partizioni secondo particolari funzioni di mapping `chiave - partizione`.

Consumer

Un *consumer* è organizzato in gruppi, e ogni messaggio pubblicato in un topic è inviato a un consumer per gruppo.
Se tutti i consumer sono sotto lo stesso gruppo, allora tutti i record saranno divisi equamente e il carico sarà bilanciato.
Se tutti i consumer appartengono a gruppi diversi, allora ogni messaggio sarà inviato a tutti i consumer.

Creando tuttavia più partizione si perde l'ordine dei record all'interno dei topic, dato che l'ordine è garantito solo all'interno della partizione.

Garanzie

Ad alto livello, Kafka da alcune garanzie:

- I messaggi appariranno nel topic secondo l'ordine in cui sono stati inviati. Se il producer P1 invia prima di P2, il messaggio di P1 apparirà per primo; allo stesso modo un messaggio m1 e m2 mandati in successione dallo stesso producer appariranno in tale ordine.
- Un consumer vede i record nell'ordine in cui sono salvati.
- Per un topic con replica `n`, Kafka tollera `n-1` server spenti prima di perdere il dato

3.1.2 Utilizzo Kafka

Per lanciare Kafka si deve avviare zookper [**?**]. Zookeper è un servizio centralizzato per la mantenere e gestire la configurazione e la naming in sistemi distribuiti. Esso gestisce anche la sincronizzazione fra tali sistemi. Nel nostro caso questo servizio è fornito da Cloudera Manager [14]. Cloudera Manager è il modo più veloce per rilasciare su un sistema Hadoop. Tramite tool automatizzati è possibile rilasciare tutti i componenti per creare cluster di qualsiasi dimensione, con le configurazione già settate al meglio per il sistema. Permette inoltre di avere un monitor sia sulle risorse sia sulle performance di ogni componente.

Avviare Kafka è semplice e basta un solo comando

```
1
2 cd /home/cloudera/Downloads/kafka_2.11-0.11.0.0 &&
3 sudo bin/kafka-server-start.sh config/server.properties
```

Nel file `server.properties` sono presenti i dati necessari per la corretta configurazione del broker, quale il numero di repliche, di partizioni e il nome del broker stesso.

Per leggere i dati possiamo sia usare le API, come vedremo dopo, sia un comando da linea di comando.

```
1
2 bin/kafka-console-consumer.sh --bootstrap-server localhost
    :9092 --topic 1 --from-beginning
```

Il producer usa delle API molto semplici:

```
1
2 ProducerRecord<String, String> record = new ProducerRecord<
    String, String>("topic", mesg.getJsonString());
3 producer.send(record);
```

Vediamo come dopo aver create l'oggetto `ProducerRecord` gli si passi il messaggio e il topic di interesse. Con la `send` il messaggio è inviato.

Il consumer quindi utilizza un *iteratore* che consuma messaggi finché ce ne sono, restando in attesa altrimenti.

```
1
2 while (consumerIte.hasNext()){
3         stringa = new String(consumerIte.next().message());
4         jsonString = new StringBuilder();
5         jsonString.append(stringa);
6         jsonObj = new JSONObject(jsonString.toString());
7         System.out.println(jsonObj.toString());
```

Come si nota è agile prendere il record è metterlo dentro la struttura dati più appropriata; come vedremo in sezione 4.4.2 preleveremo il dato e lo scriveremo su Kudu.

3.2 Kudu

Kudu [10] è un sistema di storing colonnare open source pensato per coesistere nell'ecosistema di Hadoop. Kudu salva i dati secondo una logica tabellare proprio come i database relazionali.
Ovviamente una tabella può essere sia una semplice coppia `key - value` sia una tabella di fino a 300 colonne (fortemente tipate). Ogni tabella *deve* avere una *primary key* composta da una o più colonne ordinate. Le righe di qualsiasi colonna possono essere efficientemente cancellate facendo riferimento alla loro chiave primaria.
Questo semplice modello dei dati permette di integrare applicazioni *legacy* poichè non ci si deve preoccupare di convertire in binario o in complessi JSON i valori presenti nelle celle prima di poterli utilizzare. Qui le tabelle si autodescrivono cosicchè da rendere possibile interrogazioni SQL o l'utilizzo di *engine* come Spark [12].
A differenza di altri sistemi di storing per l'analisi dei big data, Kudu non è un formato di file. E' uno storage system capace di restituire righe con una bassissima latenza, nell'ordine dei millisecondi.
Per accedere ai dati con filosofia NoSQL è possibile usare le API di Java, C++ e Python [**?**]. Ovviamente queste API possono essere usate in *batch* per integrare algoritmi di *machine learning* e *analytics*.

Le API di Kudu sono pensate per essere facili da utilizzare poiché, essendo le colonne fortemente tipate, si possono salvare i dati in formati primitivi, senza preoccuparsi delle serializzazione.

Sebbene non sia stato progettato per OLTP, se i dati possono essere spostati in memoria, ciò è possibile.

Kudu è stato disegnato per essere immerso nell'ecosistema Hadoop. Kudu prende posto egregiamente in tale ecosistema poichè può risiedere nello stesso disco dove è presente hdfs [5]. Inoltre occupa solo 1GB di RAM.

Un'altra peculiarità di Kudu è che è totalmente open source sotto la lincenza Apache 2.0 [3]

Come molti altri sistemi di storing analitico per big data, Kudu organizza internamente le tabelle secondo una logica colonnare. La memorizzazione colonnare permette una efficiente codifica e una alta compressione. Ad esempio una colonna con pochi elementi unici può essere compattata usando solo pochi bit per riga. Per via di queste numerose tecniche di compressione, Kudu è tanto veloce a leggere i dati quanto a scriverli. Un altro vantaggio di questa logica è quella ridurre drasticamente le operazioni di *IO* richieste per fornire risultati alle query. Grazie a tecniche quali la `lazy materialization` e la `predicate pushdown`, Kudu può eseguire drill-down e ricerche specifiche (ad *ago nel pagliaio*) fra miliardi di righe e terabyte di dati in secondi.

Un'altra peculiarità è la *fault tollerance*. Questa è conseguenza della sua logica interna. Per poter essere così prono allo scaling della dimensione dei dati Kudu salva le tabelle in piccole unità chiamate *tablet*. Questa suddivisione, come spiegato meglio successivamente, può essere fatta secondo diverse logiche. L'obiettivo è massimizzare la parallelizzazione.

Per mantere i dati al sicuro e consistenti Kudu utilizza l'algoritmo Raft [27] per replicare tutte le operazioni su altri tablet. Raft si occupa di garantire che una scrittura su un tablet sia almeno stata recepita da altri due tablet, assicurando che nessun dato venga perso in caso di rottura di un server in tale fase. Quando un server si blocca o si disconnette dalla rete, in pochi secondi le repliche dei tablet coinvolte si riconfigurano.
L'uso di tale algoritmo abbassa di molto le latenze perfino quando i server sono messi sotto stress da elevati carichi, sia per via di `job` di Spark sia per via di query Impala, per esempio.
A differenza dei sistemi *eventually consistent*, dove la consistenza è garantita solo dopo un certo delta di tempo, quindi una lettura immediatamente successiva alla modifica potrebbe essere errata, Raft riesce a mantenere consistenza fra le repliche grazie all'uso congiunti di *clock* logici e fisici.

3.2.1 Architettura

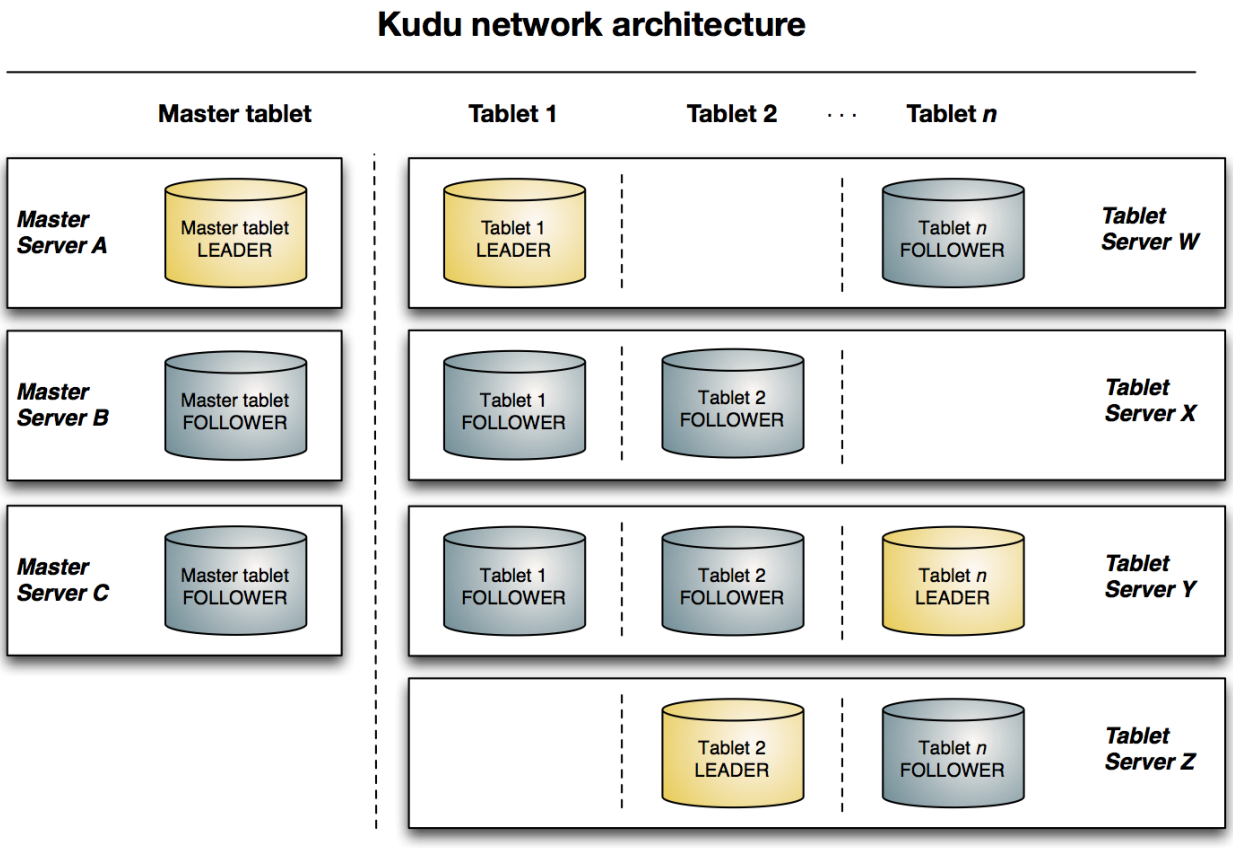
Kudu, agli occhi di un utente, non è altro che un sistema di storing basato su logica tabellare. Come in qualsiasi database tabellare è possibile definire per le varia tabelle i

tipi delle colonne come stringe, interi ecc e definire un sotto insieme ordinato e non nullo di colonne come chiave primaria.

Come anticipato prima una caratteristica che lo differenzia da altri sistemi NoSQL-like è l'uso forte della tipizzazione per le colonne. Da questo derivano diversi vantaggi:

- L'esplicitare il tipo permette al sistema di fare delle ottimizzazioni sulle colonne, come per esempio il `bit-packing` per gli interi
- E' possibile esporre dei metadati per la tabella in stile SQL cosicché l'integrazione con sistemi di BI e Data Exploration sia la più efficiente possibile.

Figura 3.4: Architettura Kudu



API

Kudu mette a disposizione API sia per la scrittura che per la lettura (*scan*) delle righe in una tabella. Vedremo il loro utilizzo successivamente.

Partizionamento

Come molti database distribuiti Kudu è organizzato orizzontalmente. Queste partizioni sono chiamate `tablets`. Virtualmente ogni riga può essere mappata su un diverso tablet, cosicché da rendere efficiente le `insert`. Per tabelle grosse si usano 10-100 tablet per macchina. A differenza di altri sistemi NoSQL, Kudu supporta la partizione con molteplici logiche. Quando si crea una tabella è possibile specificare una schema di partizionamento. La scelta della logica di partizionamento, e quindi del modello dello schema, è lasciato libero poichè ogni scenario è unico e non può esistere un modello unico che vada bene per tutto.
I principi per qui si tende a partizionare i dati sono molteplci:

- I dati dovrebbero essere partizionati in modo da rendere efficienti le letture e le scritture.
- Le tablet dovrebbero crescere in modo uniforme e i carichi di lavoro indirizzato di conseguenza.

Ci sono due metodi di partizionamento:

- Una partizione *hash* che consente di dividere le righe in sottoinsiemi secondo le chiavi primarie in `bucket` (secchi). La sintassi è `DISTRIBUTE BY HASH(id, ts) INTO 8 BUCKETS`.
- Una partizione a *range*, intervallo, che crea insiemi ordinati di chiavi.

Repliche

Per supportare l'*high availability*, ovvero l'alta disponibilità, Kudu può replicare i dati su più nodi. Quando si crea una tabelle è possibile specificare il numero di repliche, di solito 3 o 5.

Kudu Master

Per coordinare i servizi offerti da Kudu, serve un master. Esso ha diverse funzionalità:

- Agisce come un catalog manager, ovvero sa quali tablet esistono in ogni dato momento. Inoltre ogni qualvolta una tabella è creata o distrutta, tale informazione è distribuita su tutti i nodi.
- E' un cluster coordinator ovvero sa quali server sono attivi e agisce di conseguenza in caso di *failure*.
- E' un tablet director, ovvero tiene traccia delle repliche su ogni tablet.

3.2.2 Utilizzo Kudu

Per poter utilizzare Kudu, previa installazione come *stand-alone* o come *parcel* di Cloudera Manager, è necessario lanciare 3 servizi.

```
$ sudo service ntpd restart
$ sudo service kudu-master start
$ sudo service kudu-tserver start
```

Una volta fatto questo è possibile utilizzarlo in due modi. E' possibile interrogare il database sia con query scritte in SQL tramite Impala, sia usando le API NoSQL attraverso molteplici linguaggi di programmazione; qui le vedremo eseguite da un programma Java.

Vediamo ora la modalità tramite la API, dato che l'interazione con Impala verrà trattata nel Capitolo 3.3.3.

Per cominciare bisogna creare la connessione con il database.

```
cli = new KuduClient.KuduClientBuilder(KUDU_MASTER).build();
table = cli.openTable("table_name");
session = cli.newSession();
```

Come si vede si recupera l'indirizzo del master e si specifica con quale tabella si vuole interagire

Lettura

Per leggere dati si usano le *scan*.

```
KuduClient client = new KuduClient.KuduClientBuilder(
    KUDU_MASTER).build();

KuduTable table = client.openTable("table");

KuduSession session = client.newSession();
List<String> projectColumns = new ArrayList<>(2);
projectColumns.add("col_1");
projectColumns.add("col_2");

KuduScanner scanner = client.newScannerBuilder(table)
      .setProjectedColumnNames(projectColumns).build();
  while (scanner.hasMoreRows()) {
    results = scanner.nextRows();
```

```
15          while (results.hasNext()) {
16           RowResult result = results.next();
17           res = result.getString(0);
```

Dopo aver creato la connessione con il master, solitamente all'indirizzo `127.0.0.1:7051`, è possibile aprire la tabella desiderata. Si scelgono poi i nomi delle colonne che si vogliono leggere. Una volta scelte le colonne si crea lo scanner tramite il quale è possibile recuperare i dati contenuti nella tabella. Come si vede è necessario sapere esattamente il tipo del dato, questo è dettato dalla forte tipazione delle colonne di Kudu. Una volta che si è recuperato il valore della cella è possibile utilizzarlo nel programma.

Scrittura

Per poter scrivere i dati si utilizzano un altro set di API. Come prima è però necessario sempre instaurare una connessione con Kudu tramite l'indirizzo per master.

```
1
2 Insert insert = table.newInsert();
3          PartialRow row = insert.getRow();
4
5          row.addString("col_1",data_1);
6        ...
7
8 session.apply(insert);
```

E' possibile rieseguire queste scritture inserendo tali istruzioni all'interno di un ciclo.

Creazione tabelle

Tramite le API è possibile creare tabelle, specificando le colonne e il loro tipo. E' inoltre possibile

```
1
2 List<ColumnSchema> columns = new ArrayList(2);
3 columns.add(new ColumnSchema.ColumnSchemaBuilder("key", Type
      .INT32)
4     .key(true)
5     .build());
6 columns.add(new ColumnSchema.ColumnSchemaBuilder("value",
      Type.STRING)
7       .build());
8 List<String> rangeKeys = new ArrayList<>();
9 rangeKeys.add("key");
```

E' possibile inoltre specificare le proprietà, ovvero il numero di repliche e la tipologia di partizione.

```
addRangePartition(PartialRow lower, PartialRow upper);
...
setNumReplicas(int numReplicas);
...
addHashPartitions(List<String> columns, int buckets);
```

Come vedremo nella sezione successiva, con l'integrazione di Impala sarà molto più agile e intuitivo operare sulle tabelle.

3.3 Impala

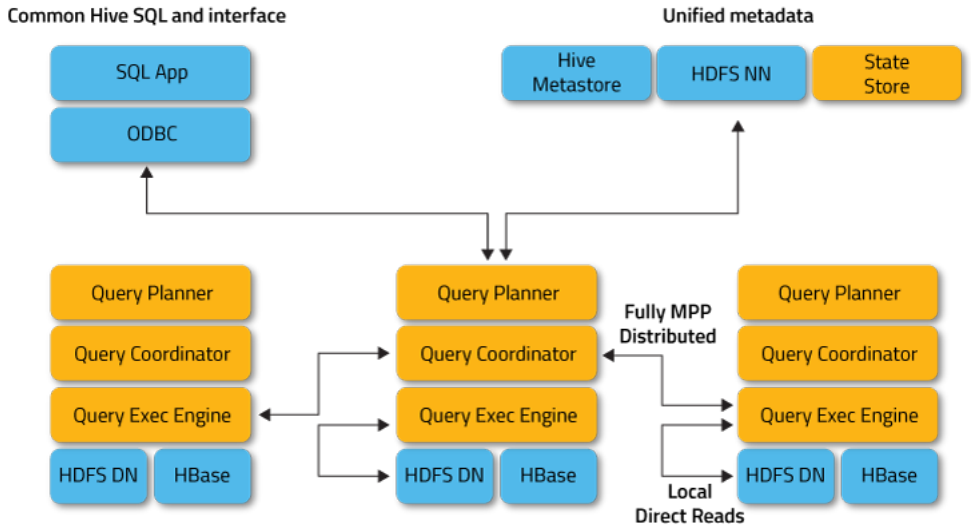
Apache Impala è un software open source scritto in C++ [15] sviluppato per analizzare grosse quantità di dati salvati su Hadoop Distribuited File System (HDFS). Impala garantisce elevate performance e bassa latenza rispetto agli altri motori SQL. In altre parole, Impala è il modo più efficiente per accedere ai dati su Hadoop, restituendo un'esperienza simile a quella che si avrebbe su un database relazionale.

3.3.1 Architettura

Per ridurre la latenza a livelli così bassi Impala utilizza una tecnologia basata su MPP (Massive Parallel Processing). Questo però eredita il difetto di assoluta non fault tollerance nel caso in cui un nodo cada. Se ciò accade l'intera query fallisce.
L'architettura di Impala, ad alto livello, si presenta così:
Per poter lanciare impala infatti bisogna lanciare 3 demoni:

```
$ sudo service impala-server start
$ sudo service impala-catalog start
$ sudo service impala-state-store start
```

Questi tre servizi sono `Impalad, Statestore, Catalogd`. Il demone impala (impalad) è responsabile di accettare le query dai client e condurre la loro esecuzione attraverso tutti i cluster. In ogni nodo ascolta anche le richieste dai demoni degli altri nodi.
Il demone responsabile per il salvataggio e la sincronizzazione dei metadati per tutti i nodi connessi al cluster è il servizio di catalogo. Qui è chiamato catalogd e ne basta uno per l'intero cluster.
Il servizio invece che si occupa di controllare la salute degli altri servizi è denominato state-store. Esso rende anche disponibile tale informazione a tutti i nodi. Come per il precedente, ne serve solo uno a cluster. Il suo compito è fondamentale dato che è lui che controlla se un servizio o nodo è andato offline, sia per una caduta fisica del server o per



un errore di rete, esso è in grado di informare tutti gli impalad che così non manderanno più nessuna query su tale nodo.

Vediamo ora quali sono gli step per l'esecuzione di una query:

1. L'utente lancia una query o dalla shell di impala (`impala-shell`) o da un programma tramite un connector ODBC.
2. Il servizio che ha ricevuto tale query dall'utente diventa il coordinatore e mette in atto i seguenti passaggi:
   (a) Leggendo i metadati controlla che la sintassi e la semantica della query siano corrette.
   (b) Prende i metadati dei blocchi di dati utili per la query dal namenode HDFS.
   (c) Distribuisce la query e le informazioni a tutti i demoni sui nodi paralleli.
3. Tutti i demoni che hanno ricevuto il compito dal primo demone leggono i loro dati locali e eseguono tale query.
4. Se tutti i demoni hanno completato l'esecuzione della query con i loro dati locali, il risultato è restituito al demone coordinatore e il risultato è fornito all'utente.

3.3.2 Utilizzo Impala

Ora mostreremo il funzionamento dell'`impala-shell`. Per lanciare la shell con il server locale possiamo usare il comando:

```
$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to quickstart.cloudera:21000
Server version: impalad version 2.9.0-cdh5.12.1 . . .
***********************************************************
Welcome to the Impala shell.
(Impala Shell v2.9.0-cdh5.12.1 (5131a03) . . .

. . .
***********************************************************
[quickstart.cloudera:21000] >
```

Ora, una volta connessi al server dove gira l'impalad, che sia in localhost o un server esterno, se esplicitato in fase di richiamo del comando, è possibile eseguire numerosi comandi. Per brevità ne mostreremo solo un sottoinsieme.

Possiamo vedere come sia immediato eseguire una query qualsiasi:

```
[quickstart.cloudera:21000] > use swa;
Query: use swa
[quickstart.cloudera:21000] > select count(*) from
    swa_tt_twi_pk;
Query: select count(*) from swa_tt_twi_pk
Query submitted at: 2017-11-29 16:04:03
(Coordinator: http://quickstart.cloudera:25000)

+----------+
| count(*) |
+----------+
| 2002612  |
+----------+

Fetched 1 row(s) in 0.14s
[quickstart.cloudera:21000] >
```

Dopo aver scelto il database sul quale fare la query è possibile interrogare le tabelle desiderate. E' inoltre possibile usare una differente notazione:

```
SELECT * FROM db_name.table_name;
```

Impala ha anche delle funzioni particolari per l'output e per fare delle interrogazioni in CLI senza aprire direttamente la shell. Come visto prima l'output è circondato da caratteri che ricordano una tabella, ciò può risultare scomodo. E' quindi possibile farsi restituire l'output come csv o come singola variabile, molto utile se si vuole intercettare

tale output. Nel nostro case study si è usato quest'ultimo metodo; grazie all'opzione (`-B`), per salvarsi il valore di un campo, è stato possibile automatizzare un algoritmo per la creazione di una tabella.

```
1
2 while [ -n "$X_tab" ];
3 do
4
5 X_tab=$(impala-shell -B -q "select tab_name
6         from wor.sources_config
7         where src_id="$S > 'out'; cat out)
```

3.3.3 Utilizzo Kudu con Impala

Presentiamo ora come è stato usato Impala per interagire con Kudu [21] nel caso di studio, mostrando quindi come usare concretamente le feature che esso offre.

Lanciando la shell di Impala con il comando `impala-shell` ci si connette al demone su `localhost` sulla porta `21000`.

Ora si può interrogare il database con le classiche istruzioni SQl tipo `SELECT`, `WHERE`, `JOIN`, `GROUP BY` in maniera piuttosto efficiente per le motivazioni esposte prima. Tuttavia Impala presenta qualche limitazione:

- Nessun supporto per le transazioni (ACID)
- Nessun supporto per cancellare una query una volta lanciata
- Nessun supporto per le *stored procedure*
- Nessun supporto per le chiavi autogenerate

Come anticipato precedentemente è possibile partizionare le tabelle per ottimizzare le performance. Idealmente ogni tabella dovrebbe essere divisa in tanti *tablet* quanti sono i server, per massimizzare il parallelismo.
Attualmente non è possibile creare partizioni *dopo* che la tabella è stata creata.

Con la keyword `PARTITION BY` è possibile scegliere fra due tipologie di partizione, per `RANGE` e per `HASH`

Impala rende disponibili molte altre funzionalità che esulano dal puro linguaggio SQL, sebbene la sintassi sia similare, ma che sono molto importanti nel *performance tuning*, ovvero, per esempio, nella gestione e calibrazione dei join, una delle operazioni più onerose in termini di risorse. E' possibile visualizzare `stats` riguardandi le performance delle tabelle, quali le partizioni, le dimenensioni, il livello di compressione ecc.

Capitolo 4

Case Study

Il caso di studio di questa tesi è la creazione di un sistema configurabile e modulare sul calcolo del ROI per un'emittente televisiva.
Basandosi sul rumore web di Twitter e sui dati già in loro possesso, come i dati dei diritti e palinsesto provenienti dal loro data warehouse. L'architettura che si è adottata per supportare l'enorme mole di dati provenienti dai social è stata quella del data lake con tecnologie all'avanguardia per i big data. Lo scopo principale di tale analisi e studio è stato quello di centralizzare i dati provenienti da sorgenti eterogenee. Questo ha permesso di fare analisi molto più accurate sull'andamento dei titoli. Più informazioni si hanno più l'analisi è veritiera. Rappresentando tali dati su tool di Data Visualization è stato possibile scoprire pattern e comportamenti degli utenti molto interessanti.

Per esempio è possibile scoprire che un titolo non ha più la stessa audience dopo che è stato ripassato molte volte, permettendo all'emittente di comprare meno passaggi dal fornitore. Inoltre può risultare utile per valutare se un titolo è ammortizzato correttamente, ovvero se i periodi scelti e le quote dell'ammortamento corrispondono all'effettivo valore utile.
Questo progetto è nato dall'idea di colmare quel gap fra la moltitudine di dati disponibili, quali costi, audience, rumore web ecc, e la possibilità di fare una analisi complessiva di tutto ciò, restituendo un report.
Come si vedrà nei capitoli seguenti ciò non è stato banale. Si è infatti dovuta creare una architettura in grado di gestire la mole di dati in ingresso garantendo sempre performance adeguate.

4.1 Contesto

Il contesto dal quale questa tesi nasce è quello della frammentarietà ed eterogeneità dei dati e dall'enorme potenziale in termini informativi nell'unificarli. Infatti poter estrarre informazione da molti dati permettere di fare scelte sempre più mirate e corrette. Infatti

la Base della *Business Intelligence* è quello di prendere decisioni basate sull'analisi dei dati. Più dati si hanno, più accurata sarà l'analisi.
Il motivo quindi che ci ha spinto a scegliere una architettura a DL è stata quella della flessibilità. Dato che il DL è schema-on-read è possibile inserire i dati senza avere ancora chiaro come saranno usati, non ponendosi limiti.

4.2 Architettura

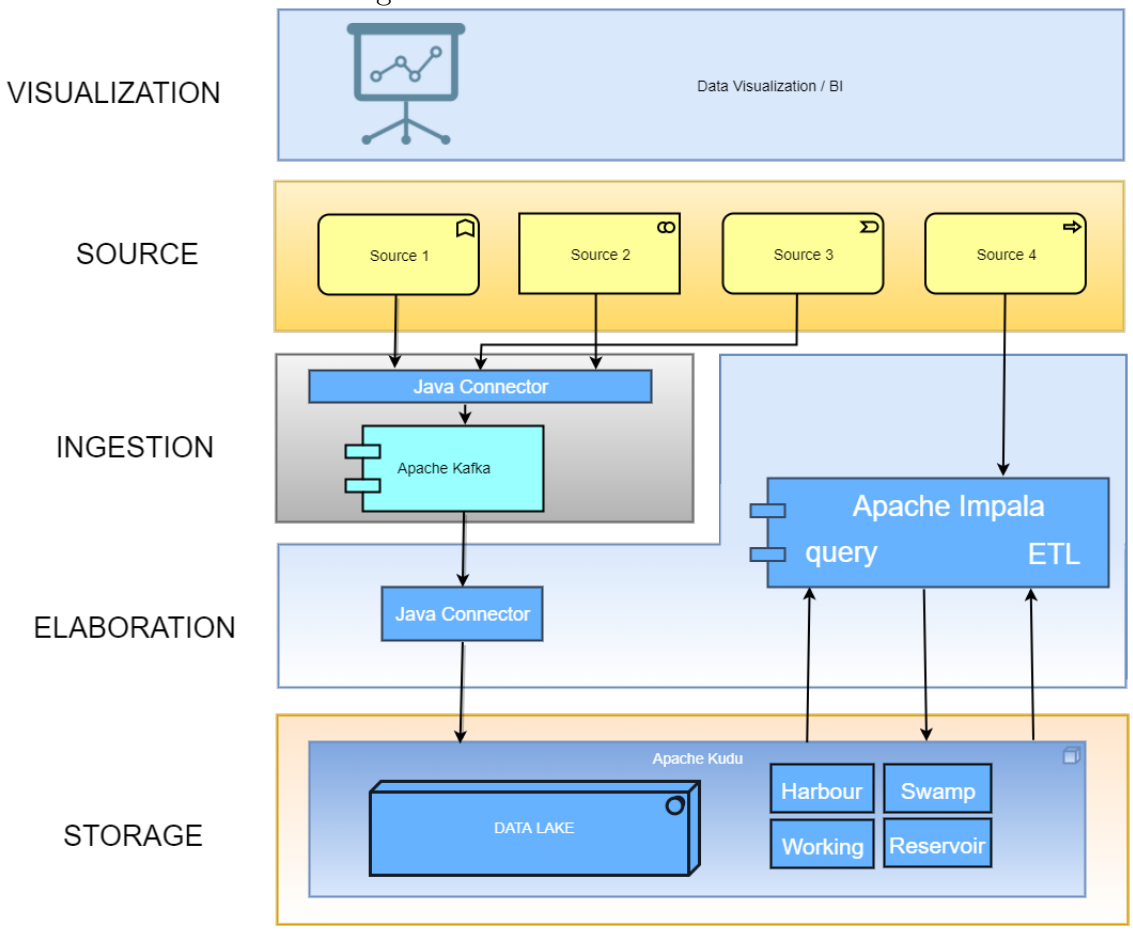
Come mostrato in Figura 4.1 l'architettura quindi dev'essere la più generale possibile.
Per la gestione dei flussi in real time si è scelto di utilizzare Apache Kafka. Lo storage è gestito da Kudu, cosicché, essendo il dato salvato in forma colonnare, le performance per le analisi saranno elevate. Con semplici flussi di Ingestion i dati possono provenire anche da Data Warehouse.

Per poter scrivere i dati da Kafka su Kudu si è creato un programma Java [22]. Esso ha il compito infatti, ogni volta che Kafka ha un messaggio pronto, di scriverlo su Kudu con le API opportune.
I flussi di ETL sono stati realizzati con Impala tramite degli script bash lanciati da crontab [16].
Kudu internamente è stato organizzato secondo l'architettura del data lake:

- Il data lake ha come landing area la Swamp. Qui infatti sono salvati tutti i dati della forma più grezza, senza essere filtrati o modificati.
- Dopo che i dati dalla swamp sono stati rielaborati vengono messi nella Harbor. Qui hanno già subito una pulizia iniziale, portandoli allo standard di qualità del DL. Vengono inseriti in questa zona anche i dati già puliti provenienti da altri data warehouse.
- Nella Working Table vengono messe le tabelle di configurazione, utili nella fase di integration. Risiedono inoltre le tabelle utili alla gestione del DL, come i log.
- Nella Reservoir invece sono messi i dati nella loro forma più pura, sia da tabelle *finali*, ovvero frutto dell'elaborazione tramite join ecc., sia come risultato di flussi ETL di pulizia. Inoltre qui è dove si appoggiano i tool di Data Visualization.

Per poter interrogare in modo efficiente Kudu si è optato per Impala. Impala infatti rende possibile lanciare query verso Kudu con la sintassi SQL. Impala è usato sia in tutte le fasi dell'ETL, sia da Tableau per poter comunicare con il database.
Tableau [30] è uno dei leader dei software di Data Visualization.

Figura 4.1: Grafico Architettura



4.3 Funzionamento Logico

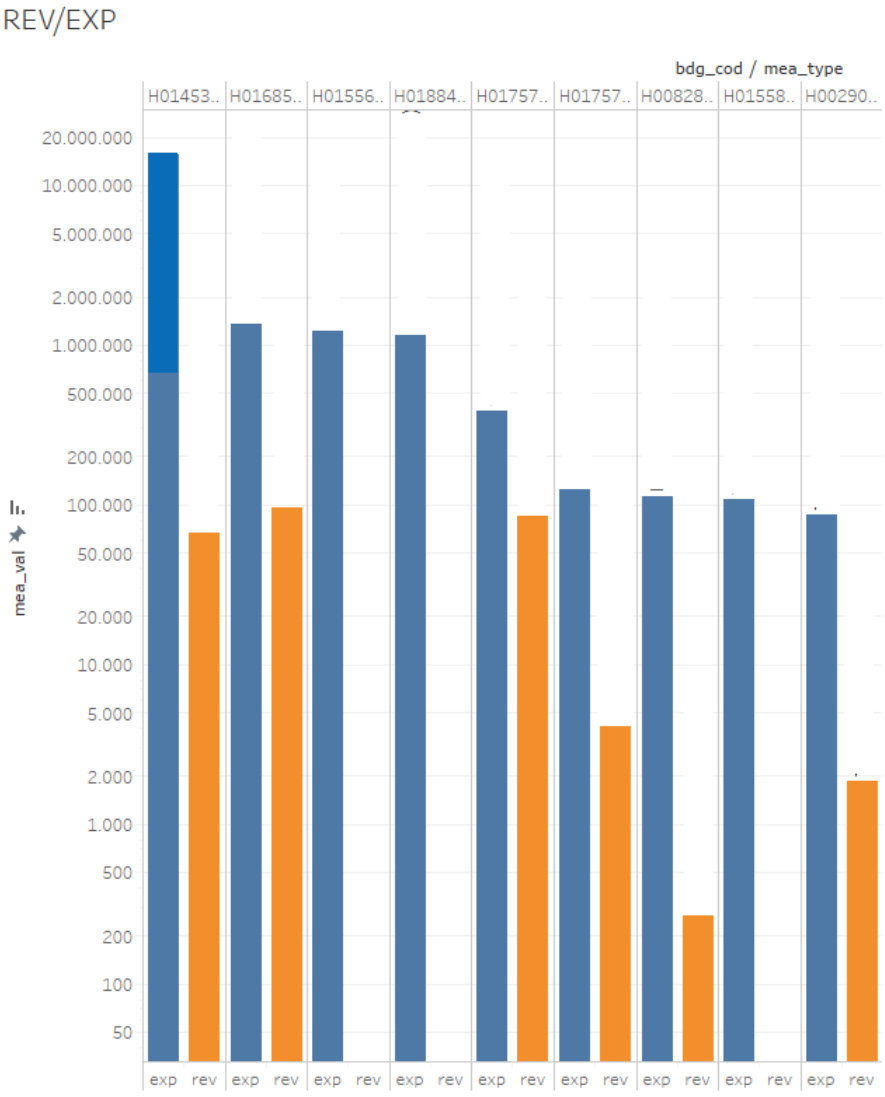
Il concetto chiave di questo progetto è quello di integrare i dati da molteplici fonti.
La caratteristica principale di questo lavoro è l'automazione del processo che crea un report unificato di tutti questi dati. Questo algoritmo parte da una tabella ove sono riportati il nome della tabella che ha la metrica che ci interessa, quale campo usare come metrica e a quale titolo si riferisce. Inoltre è riportato se tale numero è un numero puro, una somma in euro o altro. E' specificata inoltre la natura della misura, ovvero se è una spesa o è un ricavo.
Per esempio come fonti possiamo avere il numero di tweet fino a quel momento catturati e il costo del titolo. Nella tabella di configurazione quindi si inseriscono il nome di queste due tabelle e i rispettivi nomi dei campi. A fianco si specificano se sono spese o ricavi. Ora l'algoritmo inserirà nella tabella finale una riga per ogni titolo e per ogni data a cui

quel numero corrisponde.

H0xxxx	2017	12	1	0	rev	num-twi	234	number	1.0
H0xxxx	1	1	1	1	exp	cas-cos-eur	94529	eur	1.0

La creazione di questa tabella è molto utile poiché permette di automatizzare la comparazione di più misure per ogni titolo.

Figura 4.2: Grafico REV/EXP



Si può notare come per il titolo di codice `H0xxxx` il primo dicembre 2017 a mezzanotte ha totalizzato 234 tweet, che è un numero puro, di ricavi e ha peso 1. Il suo costo, si

veda la data fittizia atemporale, è di 94k euro e ha peso 1.

Grazie a questa automatizzazione si può vedere l'intero progetto come una scatola nella quale possiamo inserire le fonti più disparate, e come output avremo una vista d'insieme per ogni titolo.

4.4 Implementazione

Si veda ora come queste tecnologie sono state utilizzate nel nostro caso specifico.

4.4.1 Architettura

L'architettura come accennato prima si basa sull'utilizzo di quattro tecnologie. Nello specifico l'architettura risultante è quella esposta in Figura 4.3.

Per poter scaricare i tweet si è creato un programma in Java capace, tramite le API di Twitter [35], di collezionare tweet secondo vari criteri. Per la nostra applicazione si è scelto di scaricare solo quelli con *hashtag* interessanti per i nostri contenuti. Si è integrato sempre in Java il `connector` che scrive i tweet su Kafka. Sempre in Java è stato fatto il componente che legge da Kafka e scrive su Kudu. Kafka è stato scelto per poter gestire la mole di dati ad alta velocità che arriva da Twitter. Kafka infatti oltre che rendere fault tollerant l'invio dei record, rende anche ordinata la scrittura su Kudu.
Una volta che il dato è stato immagazzinato da Kafka dev'essere salvato. Per fare questo si è scelto Kudu. La scelta è stata dettata dalla sua posizione mediana fra le performance di lettura del dato, massima in HBase, ad esempio, e la velocità di analisi di HDFS. Inoltre essendo Kudu uno storage colonnare è molto efficiente per le interrogazioni analitiche (si pensi alle aggregazioni, come somme, medie ecc.). Per poterlo utilizzare in modo efficiente si è utilizzato Impala.

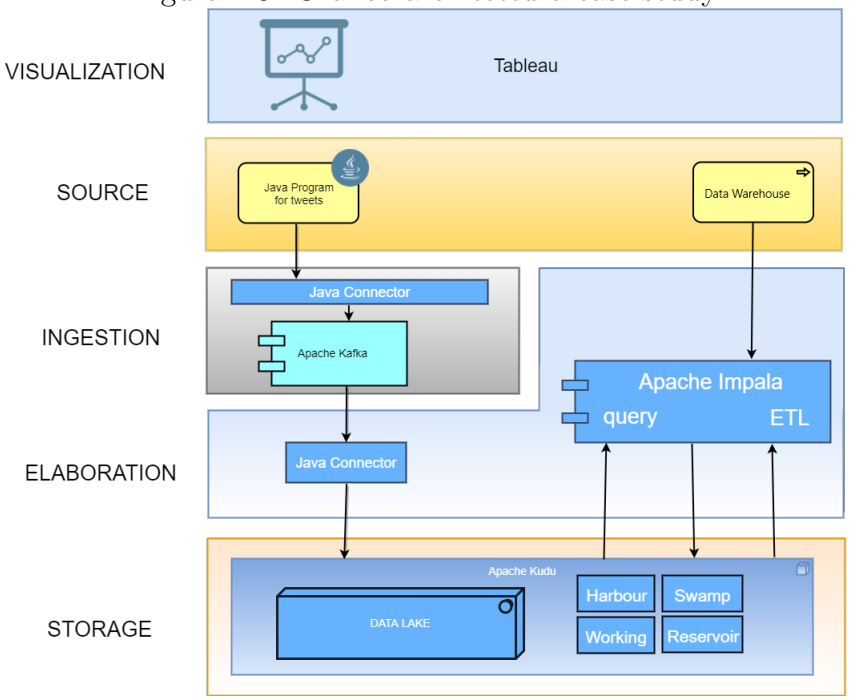
Per rendere efficiente e scalabile tale progetto si è utilizzata una struttura a Data Lake. La definizione del DL non è univoca ma è un insieme di best practice atte a rendere più chiaro che ruolo hanno i dati, a seconda di dove sono collocati. Per non rendere il DL una Swamp dove è impossibile trovare i dati, si è pensato di organizzarlo in questo modo:

- Nella Swamp Area si sono messi i dati nella loro forma grezza, ovvero i dati in arrivo da Twitter.
- Nella Working Area invece risiedono le tabelle di configurazione e quelle di log del'ETL.
- Nell'Harbor Area vengono messi i dati dopo una fase iniziale di pulizia. Qui inoltre vengono messi tutti i dati provenienti dal data warehouse dell'emittente televisiva, dato che sono già stati ripuliti. Tuttavia non sono ancora nella forma che serve a noi.
- Nella Reservoir Area invece vengono messe tutte le tabelle che hanno la struttura da noi desiderata. Qui infatti il dato è pronto. Inoltre in questa area il Data Scientist può fare self BI.

L'ETL invece è stato gestito tramite `crontab`, un servizio di Linux che permette di richiamare in modo controllato e temporizzato script bash. Nella directory `/home/cloudera/ETL` sono salvati tutti gli script. In base alla logica di quelle tabelle l'ETL può essere incrementale o full-refresh; questo si traduce in `INSERT INTO SELECT * FROM` preceduto o meno da una `DELETE` della tabella.

Dopo le fasi di Ingestion e Integration i dati convergono nella Reservoir Area. Questa area è l'unica dove gli utenti hanno accesso. In questa area è anche montato il tool di *data visualization*; per questo progetto si è scelto Tableau [**?**]. Esso ci permette di creare report ad hoc in modo semplice ed intuitivo. Inoltre permettere ad utenti esperti di interrogare il database come vogliono, lasciandoli la libertà di crearsi report personalizzati.

Figura 4.3: Grafico architettura case study



4.4.2 Sorgenti

Dati da Data Warehouse

Come sorgenti per questo progetto si sono presi i dati sia dai data warehouse dell'emittente televisiva stessa, in questa fase manualmente ma facilmente automatizzabile, sia da Twitter. Per importare le anagrafiche dei diritti, i passaggi del palinsesto, le tabelle dell'ammortamento e altro si sono presi i csv e, in due passaggi, si sono inserite su Kudu. Una volta avuto il csv si è creata una cartella di appoggio su HDFS

con il comando `hdfs dfs - mkdir /test`, successivamente è possibile inserire la tabella con `cat /home/cloudera/Downloads/test.csv | tr -d '\r' | Hadoop fs -put - /test/data.csv`. Successivamente all'import dei dati su HDFS si è dovuto, prima di creare la tabella su Kudu, creare un tabella di appoggio *ESTERNA* seguendo la sintassi di Impala.

```

CREATE EXTERNAL TABLE res_rights_raw
(
   bdg_cod          STRING,
   tit_bdg_des      STRING,
   sta_avail_dat    TIMESTAMP,
   end_avail_dat    TIMESTAMP,
   pur_cod          STRING,
   dea_cod          STRING,
   play_rig         STRING,
   cas_cos_eur      INT,
   ver_cod          STRING,
   sce_cod          INT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/test/'
TBLPROPERTIES ('skip.header.line.count' = '1');
```

Come si vede da questo esempio, qui si stanno caricando i dati presenti nella cartella test. Inoltre si può notare come sia possibile specificare il separatore dei campi per aumentarne la compatibilità. E' inoltre specificabile se la prima riga sia da saltare dato che di solito li ci sono le descrizioni dei campi. Impala prevede un ristretto numero di tipi di dato, fra i quali timestamp, interi e stringhe. In questa fase di caricamento in tabella non sono specificate chiavi, quindi tutte le righe sono caricate. Inoltre per essere sicuri si possono specificare tutti i campi a stringhe, risolvendo così tutti i problemi di compatibilità. Nel caso di questo progetto, dato che i dati arrivano da un data warehouse e quindi di qualità, si conoscono esattamente il tipo di dato di ogni colonna. Dopo aver caricato su hdfs, tramite Impala, i dati in forma grezza, è ora di creare, con sintassi simile, una tabella gestita da Kudu.

```


CREATE TABLE res.res_rights
(
   bdg_cod          STRING,
   tit_bdg_des      STRING,
   sta_avail_dat    TIMESTAMP,
```

```
8     end_avail_dat    TIMESTAMP,
9     pur_cod          STRING,
10    dea_cod          STRING,
11    play_rig         STRING,
12    cas_cos_eur      INT,
13    ver_cod          STRING,
14    sce_cod          INT,
15    primary        key(bdg_cod))
16 partition by hash(bdg_cod) partitions 2 STORED AS KUDU
17  tblproperties ('kudu.num_tablet_replicas' = '1',
18 'kudu.table_name'='res_rights',
19 'storage_handler'='com.cloudera.kudu.hive.KuduStorageHandler
       ',
20 'kudu.master_addresses'='localhost:7051');
```

Qui si noti come la struttura sia simile a quella proposta prima, ma con l'aggiunta di alcune proprietà. Infatti qui è specificato `STORE AS KUDU` che mostra il tipo di tabella che verrà creata. Le proprietà aggiunte servono proprio a definire i dettagli di Kudu, principalmente i percorsi del master, dello `storage_handler` e il nome della tabella. Una limitazione di Kudu è il non poter avere due tabelle con lo stesso nome sebbene in database diversi. Come naming convention si è adottata quella classica dell'azienda, ovvero 3 lettere per il database come prefisso del nome di ogni tabella, separato da underscore. Nella riga successiva alla definizione dei campi si trova la dichiarazione di chiave primaria e della modalità di partizionamento. In questo caso di studio tutto si svolgeva su un server in azienda singolo, quindi la replica è stata messa ad `1` e l'indirizzo del master Kudu è locale.

Twitter

Per poter invece collezionare i tweet si è dovuto creare un programma in Java utilizzando le API della Twitter Developer Platfrom [35]. Si è partiti creando la classe principale, responsabile del loop che immagazzina i tweet in una coda.

```
BlockingQueue<String> q = new LinkedBlockingQueue<String>(10000);
```

Per poter aver accesso ai tweet ci si è dovuti registrare su `Twitter Apps` [35], la quale associa al proprio profilo dei *token* segreti di autenticazione che saranno poi immessi come parametri delle API.

Successivamente, tramite le API che Kudu mette a disposizione per la lettura dei dati in tabella, si sono caricati nell'applicazione tutti gli hashtag che il motore Java userà per scaricare i tweet.

```
1
2  KuduClient client = new KuduClient
```

```
            .KuduClientBuilder(KUDU_MASTER).build();

    KuduTable table = client.openTable("swa_m_link");

    KuduSession session = client.newSession();
    List<String> projectColumns = new ArrayList<>(5);
    projectColumns.add("has");
    projectColumns.add("bdg_cod");
    projectColumns.add("vld");
    projectColumns.add("bgn");
    projectColumns.add("cha");
    KuduScanner scanner = client.newScannerBuilder(table)
        .setProjectedColumnNames(projectColumns).build();
    while (scanner.hasMoreRows()) {
      results = scanner.nextRows();
      while (results.hasNext()) {
        RowResult result = results.next();

        hash = result.getString(0);

        if (result.getString(2).equals("y")){
          hashtag.add(hash);
        }
        if (result.getString(2).equals("n")){hashtag.add("")
            ;
```

Tramite la `projectColumns.add()` si esplicitano le colonne che ci si aspettano dalla tabella. Fatto questo si recupera l'hashtag se è valido.

La parte ora più interessante è quella dello scrivere il tweet su Kafka. Si può notare che prima di essere processato da un *producer* Kafka il messaggio è convertito nel tipo dato *custom* `Tweet`. Il messaggio che arriva da Twitter è un Json [?], quindi facilmente manipolabile.

```

  while (true) {

      if (client.isDone()) {
        logger.error("Client connection closed unexpectedly:
            "
            + client.getExitEvent().getMessage());

        break;
      }

      String msg = queue.poll(5, TimeUnit.SECONDS);
      if (msg == null) {
```

```
12          logger.info("Did not receive a message in 5 seconds"
                );
13
14        } else {
15
16          KafkaWriter k = KafkaWriter.getInstance();
17          k.writeTweet(new Tweet(msg));
18
19        }
20      }
21
22      ;
23
24      client.stop();
```

Il tweet, di cui abbiamo scelto solo i campi che ci interessavano, quali nome dell'utente, testo, numero di follower e altro, ha quindi questa struttura:

```
1
2 public Tweet(String msg) {
3  try {
4   JSONObject t = new JSONObject(msg);
5   this.created_at = t.getString("created_at");
6   this.timestamp_ms = t.getString("timestamp_ms");
7   this.id_str = t.getString("id_str");
8   this.text = t.getString("text");
9
10  try {
11    JSONObject u = t.getJSONObject("user");
12    if (u != null) {
13      this.user__screen_name = u.getString("screen_name");
14      this.user__followers_count =
15                          u.getString("followers_count");
16      this.user__profile_image_url =
17                            u.getString("profile_image_url");
18    }
19  } catch (JSONException jex) {
20 }
```

Il producer Kafka è molto semplice dato che richiede solo poche proprietà, come il percorso di Zookeper [**?**]. Per poter scrivere il tweet sul corretto topic lo si indica come parametro della funzione `send` al momento dell'invocazione.

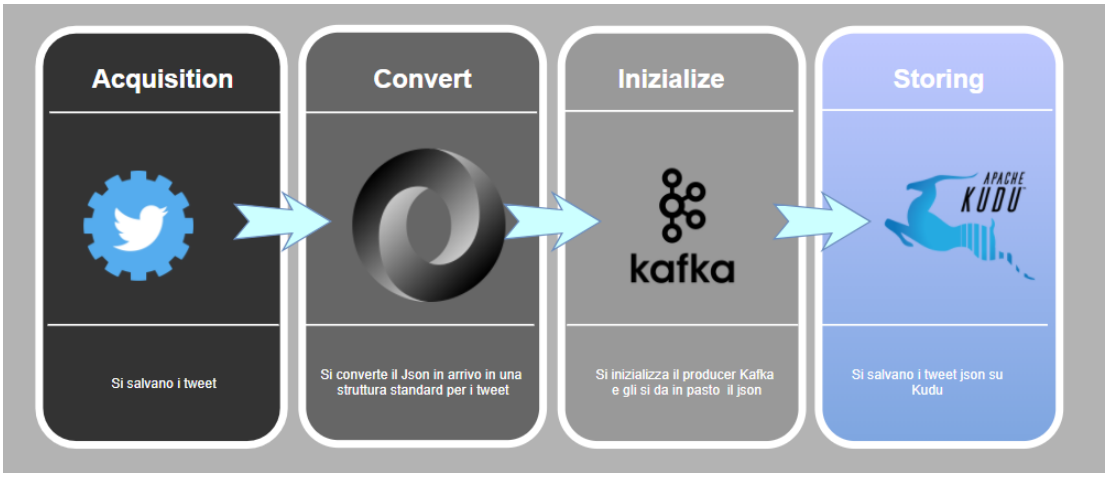
Una volta che il messaggio è su Kafka può essere replicato su più partizioni per fault tolerance. L'utilizzo di Kafka è stato scelto anche per dare ordini a tali messaggi senza sovraccaricare di richieste di scrittura Kudu.

Ora, virtualmente anche su un nodo distribuito, qui in locale per semplicità, si preleva il messaggio dal topic e lo si scrive su Kudu.

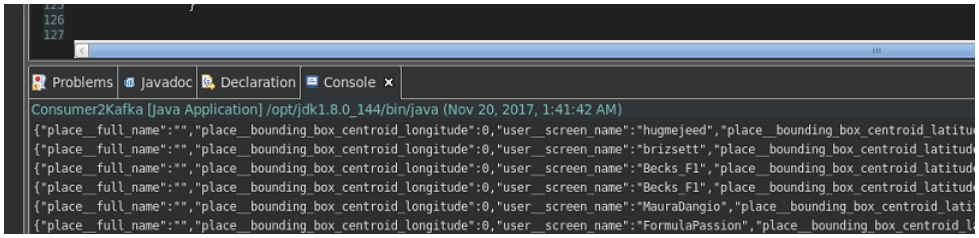
```
1
2  while (consumerIte.hasNext()){
3      stringa = new String(consumerIte.next().message());
4      jsonString = new StringBuilder();
5      jsonString.append(stringa);
6      jsonObj = new JSONObject(jsonString.toString());
7      System.out.println(jsonObj.toString())
8
9      Insert insert = table.newInsert();
10     PartialRow row = insert.getRow();
11
12     row.addString("created_at", jsonObj.get("created_at")
13                                        .toString());
14     . . .
```

Come si vede dal codice, dopo aver inizializzato il *consumer*, con un loop si iniziano a prelevare i messaggi presenti nella coda. Dopo che nel `Main` si è specificato in quale tabella di Kudu deve andare a scrivere, si comincia a creare ogni record prendendo i vari *campi* dal messaggio in arrivo. Dopo aver composto con la `addString()` la nuova row, la si può aggiungere alla tabella con il seguente messaggio:

```
1
2  session.apply(insert);
```



Come si può vedere per debugging si è tenuto visibile il feed dei tweet in arrivo. Qui si può notare come la visualizzazione rispecchi la struttura json; inoltre per il 95% dei casi, la localizzazione sia stata disabilitata dall'utente.



4.4.3 ETL

I processi fondamentali di qualunque Data Lake sono l'*ingestion* e l'*integration*. Per Ingestion si intendono tutti quei flussi ETL atti ad importare i dati dalle varie sorgenti al DW o al Data Lake. In questo caso i dati sono stati immessi manualmente tramite `.csv` in opportune strutture di Impala settando opportunamente i tipi di dato per le colonne. L'integration invece sono tutti quei flussi che modificano il dato e lo rendono sempre più pulito. Inoltre si occupano anche della creazione di informazione, ovvero creano tabelle che sono il risultato di join di più tabelle secondo varie logiche.
Si veda ora l'ETL nelle sua varie fasi, una ad una, che consentono quindi al dato grezzo di diventare dato strutturato.

Extraction

Il dato viene estratto decidendo se in logica incrementale o completa. Nel nostro caso i tweet vengono presi in modo incrementale tramite le API. I dati dai DW invece sono presi in full refresh, dato che è possibile che un dato vecchio muti, aggiornando, per esempio, il prezzo o un'altra dimensione.

Cleansing

E' la fase di trasformazione più basilare e automatizzabile, ovvero la rimozione di dati impossibili, tipici nelle date, e i dati duplicati, lanciando un'eccezione nel caso di violazione del vincolo di primary key. Essendo un DL nessuna colonna è ignorata e nessun valore è filtrato.

Transformation

Questa fase è invece specifica da progetto a progetto. Qui ad esempio possiamo denormalizzare le tabelle per migliorare le performance in fase di analisi. Possiamo inoltre aggregare su alcuni campi per trovare somme o conteggi.

Loading

Il dato è quindi caricato sui livelli più alti del DL o del DWH. Esistono due modalità per il salvataggio del dato, in modo incrementale o in modalità full-refresh, a seconda della logica adottata.

Le macro fasi sono le seguenti:

Ingestion

Ora mostreremo alcuni esempi di flussi ETL utilizzati.
L'orario da Twitter, ovvero un stringa, è stato convertito in un timestamp:

```
1
2  INSERT INTO swa.swa_tt_twi_pk
3     SELECT date_add(cast (
4       concat (
5          substr (created_at, 27, 4),
6           '-',
7         CASE
8             WHEN substr(UPPER(created_at),5,3) = UPPER('JAN')
9             THEN
10               '01'
11            WHEN substr(UPPER(created_at),5,3) = UPPER('FEB')
12            THEN
13               '02'
14            WHEN substr(UPPER(created_at),5,3) = UPPER('MAR')
15            THEN
16               '03'
17            WHEN substr(UPPER(created_at),5,3) = UPPER('APR')
18            THEN
19               '04'
20            WHEN substr(UPPER(created_at),5,3) = UPPER('MAY')
21            THEN
22               '05'
23            WHEN substr(UPPER(created_at),5,3) = UPPER('JUN')
24            THEN
25               '06'
26            WHEN substr(UPPER(created_at),5,3) = UPPER('JUL')
27            THEN
28               '07'
29            WHEN substr(UPPER(created_at),5,3) = UPPER('AGU')
30            THEN
31               '08'
32            WHEN substr(UPPER(created_at),5,3) = UPPER('SEP')
```

```
33              THEN
34                 '09'
35              WHEN substr(UPPER(created_at),5,3) = UPPER('OCT')
36              THEN
37                 '10'
38              WHEN substr(UPPER(created_at),5,3) = UPPER('NOV')
39              THEN
40                 '11'
41              WHEN substr(UPPER(created_at),5,3) = UPPER('DEC')
42              THEN
43                '12'
44              ELSE
45                 '99'
46              END,
47             '-',
48              substr (created_at, 9, 2),
49               ' ',
50        substr(created_at,12 8)) AS TIMESTAMP),
51                          INTERVAL 1 hours)
52         AS cre_at,
53       text,
54       user__screen_name,
55       user__followers_count,
56       place__full_name,
57       place__bounding_box_centroid_longitude,
58       place__bounding_box_centroid_latitude
59      FROM swa.swa_twitter_pk;
```

Integration

In questo caso il dato era per lo più pulito poichè arrivava da un DWH già ben strutturato. D'altro canto le logiche per la creazione di nuove tabelle non sono state banali. Questo perchè si sono dovute integrare più fonti d'informazione, mantenendo sempre lo stesso denominatore comune, ovvero il titolo.
Tutto questo in modo robusto, configurabile e automatizzato, per ridurre al minimo lo sforzo dell'utente nella fasi di data entry.

Ad esempio si è dovuto trovare un modo per agganciare ad ogni tweet il codice unico del titolo. Si è partiti con la tabella `swa_m_link` contenente gli hashtag relativi ai contenuti e, per ogni hashtag, il codice univoco del titolo. Questa tabella è quella usata anche dal programma Java per la raccolta dei tweet. Come si vede dal codice riportato qui sotto, per gli hashtag direttamente collegati ai titoli, il metodo per l'assegnazione

del codice `bdg_cod` è quello della semplice comparazione di stringhe; se la stringa con l'hashtag è compresa nel testo del tweet allora a quel titolo si può assegnare quel codice.

Nella prima parte della query invece è presente l'algoritmo che assegna al tweet un titolo nel caso in cui l'hashtag fosse1 generico. Infatti noi siamo interessati a raccogliere anche i tweet non diretti del tipo `#canaleCinema`. Si è scelto di confrontare l'orario del tweet con la programmazione del palinseseto.

```
1
2  INSERT INTO res.res_tweet
3  WITH twi_hash AS
4  (
5            SELECT     *
6            FROM       swa.swa_tt_twi_pk s
7            JOIN default.swa_m_link w
8            ON         Instr (Upper (text), Upper (has)) > 0
9            WHERE      bdg_cod = '***not_defined***' )
10 SELECT cre_at,
11        hp.pur_cod,
12        st.user__screen_name,
13        st.user__followers_count,
14        st.text,
15        st.has,
16        h1.bdg_cod AS 'bdg_cod or channel',
17        st.cha,
18        tit_ibms AS tit
19 FROM    har.har_plays hp
20 JOIN    twi_hash st
21 ON      (
22     date_add(Cast(Concat(Substr(pas_sch_dat,1,4),'-',Substr(
           pas_sch_dat,5,2),
23     '-',Substr(pas_sch_dat,7,2),' ',pas_sta_tim,':','00' )AS
            TIMESTAMP),
24     INTERVAL -15 minutes) < cre_at
25       AND    cre_at <date_add(cast(concat(substr(pas_sch_dat
             ,1,4),'-'
26         ,substr(pas_sch_dat,5,2),'-',
27         substr(pas_sch_dat,7,2),' ',pas_sta_tim,':','00' )AS
               TIMESTAMP),
28         INTERVAL 70 minutes))
29       JOIN res.res_rights h1 ON cast(h1.pur_cod AS STRING) =
              hp.pur_cod
30         WHERE  (
31               upper(hp.pas_cha_des) = upper(st.cha))
32 UNION ALL
33 SELECT    s.cre_at,
```

```
34          'no_pur' AS pur_cod,
35          s.user__screen_name,
36          s.user__followers_count,
37          s.text,
38          has,
39          w.bdg_cod AS 'bdg_cod or channel' ,
40          w.cha,
41          h.tit_bdg_des
42 FROM      swa.swa_tt_twi_pk s
43  JOIN DEFAULT.swa_m_link w
44 ON        instr (upper (text), upper (has)) > 0
45 LEFT JOIN res.res_rights h
46 ON        w.bdg_cod = h.bdg_cod;
```

Si vede dal codice come si sia scelta una finestra di 85 minuti che verosimilmente rispecchia la durata media di un contenuto. Se infatti il tweet con l'hashtag generico sopracitato cade in quella fascia temporale, allora a tale titolo è assegnato quel tweet. Ovviamente il rumore che si immette nelle analisi è notevole, ma si noterà che non è così rilevante dato che qualche tweet assegnato erroneamente, nel lungo tempo, non reca problemi dato che siamo interessati ai trend piuttosto che ai numeri.

Tuttavia i contenuti che creano più rumore web sono quelle delle serie televisive, che hanno un hashtag diretto, riducendo così l'incertezza intrinseca del metodo esposto precedentemente. Una logica simile è stata usata per associare gli ascolti al contenuto. Infatti gli ascolti arrivano per fascia oraria e canale, quindi non sono immediatamente collegabili ad un contenuto.

Inoltre con una tabella di configurazione si è reso automatico il congiungimento di metriche da diverse fonti mantenendo come fattore comune il `bdg_cod` (budget code). Data la mancanza di tool grafici open source atti ad automatizzare le fasi di ETL, si è dovuto creare uno script bash.

```
1
2 S=1
3 X_tab=a
4
5 /usr/bin/impala-shell -q "insert into wor.etl values (now(),
      'res_result.sh');"
6 /usr/bin/impala-shell -q "delete from res.res_result"
7
8 while [ -n "$X_tab"   ];
9 do
10
11 X_tab=$(impala-shell -B -q "select tab_name from wor.
     sources_config
```

```
12      where src_id="$S > 'out'; cat out)
13 X_name=$(impala-shell -B -q "select mea_name from wor.
       sources_config
14      where src_id="$S > 'out'; cat out)
15 Query="insert into res.res_result select bm.bdg_cod, bm.year
       ,bm.mon,bm.day,
16           bm.hour, sc.mea_typ,'$X_name' as name,
17           cast(sum(bm.$X_name) as int),sc.mea_unit, sc.
                mea_weight
18           from wor.sources_config sc join $X_tab bm
19           where tab_name ='$X_tab' group by bm.bdg_cod, bm.
                year, bm.mon, bm.day,
20           bm.hour, sc.mea_typ, name, sc.mea_unit, sc.
                mea_weight"
21
22
23
24 /usr/bin/impala-shell -q "$Query"
25
26
27  ((S++));
28  echo $X_name
29
30 done
31 echo "esco"
```

Questa fase di integration crea la tabella `res.res_result` che verrà poi utilizzata nel tool di reportistica Tableau per confrontare le spese e i ricavi. Si vede che con un ciclo si leggono tutte le righe della tabella contenente i *metadati* e con delle variabili di bash teniamo quei valori. Componiamo la string e la passiamo alla `impala-shell`.

4.5 Data Visualization

La Data Visualization non è la mera rappresentazione dei dati ma è il motivo per cui l'azienda investe nella *business analytics*. Non solo bisogna creare grafici e *dashboard* ma bisogna comunicare informazione.
Quando si creano le dashboard quindi bisogna tenere a mente alcuni principi:

- Si deve tenere presente a cui si sta mostrando tale report, ovvero chi è la persona più importante che lo vedrà. Non si può creare una dashboard *per tutti* senza perdere di specificità ed efficacia.
- Creare delle dashboard con delle informazioni veramente importanti e significative. Ad esempio misure come il market share che non variano molto spesso non

hanno molto impatto nella *decision making*, altri KPI invece possono risultare più allettanti per l'utente di business.

- L'obiettivo principale di tali analisi è prendere decisioni. Bisogna quindi domandarsi in fase di creazione del report quali scelte e quali decisioni tali dati possono supportare.

Esistono best practice per la creazione di un modello di visualizzazione dei dati. Si deve quindi scegliere il giusto tipo di grafico per il giusto tipo di dati:

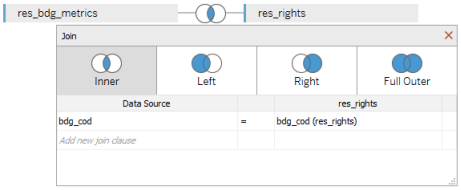
- **Grafici a linea**: per rappresentare il variare di un valore rispetto al tempo o ad una dimensione arbitraria. Sono particolarmente adatti per mostrare trend.
- **Grafici a barre**: ottimali per mostrare una misura in relazione e diverse categorie. E' importantissimo scegliere la giusta scala per gli assi. Per esempio i valori 3,04 e 3,70 possono sembrare equiparabili se si tratta di pesi, ma fondamentali se stiamo parlando di percentuali in borsa.
- **Grafici a dispersione**: o *scatter plot*, sono utili nella rappresentazione congiunta delle variazione di due gruppi di dati.
- **Dashboard interattive**: con i tool più evoluti, tipo Tableau usato nel nostro progetto, è possibile rendere tali grafici interattivi. Creando filtri e variabili e possibile con un semplice click modificare gli oggetti rappresentati, aggiungere dimensioni ecc, in modo tale da dare ancora più informazioni.

E' inoltre possibile superare la barriera delle 2 dimensioni, limite imposto dai monitor, con l'utilizzo di colori ed etichette.

4.5.1 Data Visualization with Tableau

Per questo progetto si é scelto di usare Tableau per molteplici motivi. Il primo è stato quello della sua elevata compatibilità con le più svariate sorgenti. Esso infatti permette di utilizzare come sorgenti sia file in formati Excel, json e csv sia i database relazionali, quali MySQL per esempio; supporta inoltre connector per il mondo hdfs. Questo ci ha dato la possibilità di connetterci con Impala, specificando l'IP e la porta del server.
La seconda motivazione è stata quella della sua semplicità ed immediatezza d'uso. Scrivendo semplicemente una query è possibile importare tali dati nel sistema.
Qui si aprono due possibilità: interrogare *live* il database, ovvero ogni query ricarica tutti i dati, o farlo in modalità *extract*, cioè caricando in memoria tale risultato, per permettere un aumento delle performance in fase di creazione delle dashboard.
Inoltre è possibile eseguire i join graficamente semplicemente trascinando le tabelle e scegliere la chiave di join e il tipo come mostrato in Figura 4.4.

Figura 4.4: Modalità di join grafico in Tableau



Una volta che la query è stata eseguita, per non occupare tutta la ram disponibile, Tableau mostra una piccola preview dei risultati. Spostandosi nella tab *worksheet* è possibile spostare i campi in righe e colonne creando grafici ad hoc.

Qui è selezionabile il tipo di rappresentazione che vogliamo come mostrato in Figura 4.5 a seconda di quante dimensioni abbiamo.

Figura 4.5: Modalità di join grafico in Tableau

4.5.2 DV nel Case Study

Abbiamo scelto di creare diverse *dashboard* per mostrare vari aspetti. Questo deriva dall'eterogeneità delle sorgenti che obbliga a creare report ad hoc per fotografare i vari scenari.

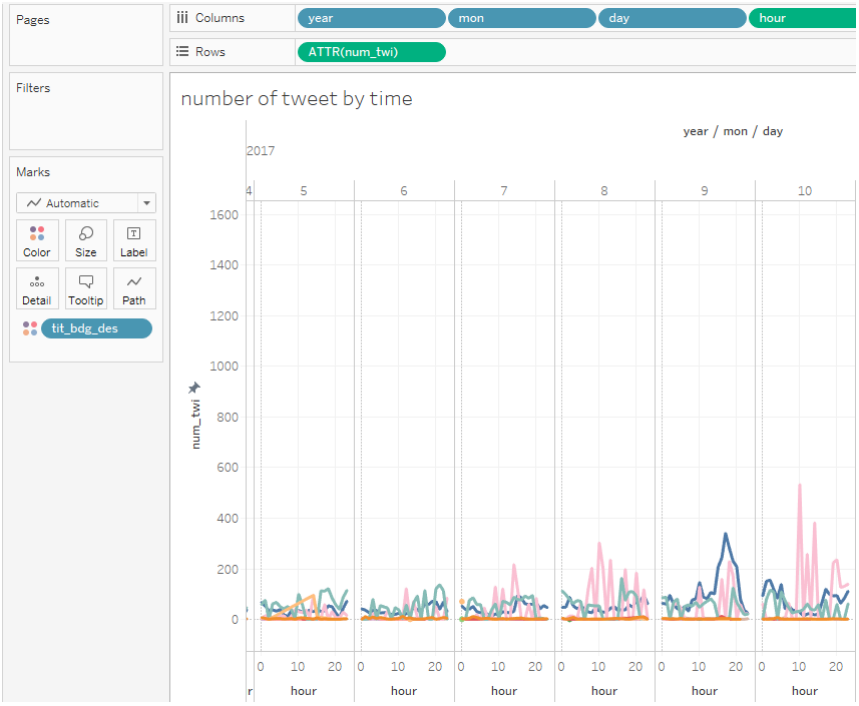
Si sono creati grafici dove il comune denominatore, il titolo, è messo, di volta in volta, in relazione rispetto a diverse dimensioni. Nei grafici è possibile sia vedere l'andamento del titolo secondo una determinata dimensione, sia l'evoluzione di tale titolo rispetto a

a diverse misure contemporaneamente.

Si mostrerà infine una calcolo del ROI e una rappresentazione dei costi e ricavi, provenienti dalle nostre sorgenti, creato in modo automatico.

Andamento dei tweet per titolo

Per mostrare l'andamento dei tweet rispetto al tempo, con granularità dell'ora, abbiamo scelto un grafico a linea.



Ogni colore è un titolo diverso e come filtro si può scegliere di visualizzare solo un titolo alla volta. Il grafico mostra il rumore web nei vari momenti permettendo di notare quando e se un contenuto ha creato interesse.

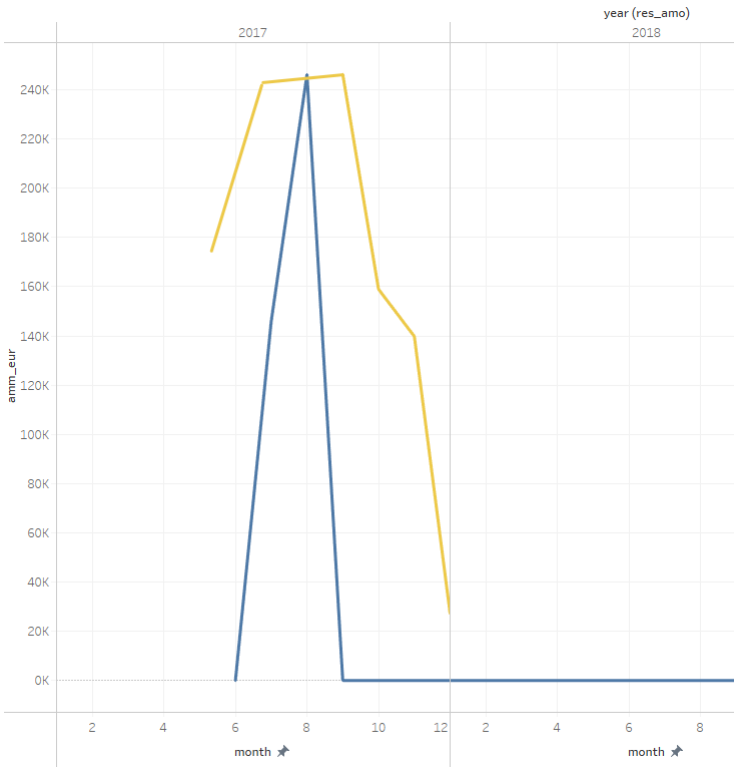
Numero di follower per titolo

Sempre in ambito Twitter abbiamo scelto come seconda metrica il numero di persone *coinvolte*. Ovvero si è deciso che 100 tweet da utenti con 1 follower contino come un tweet fatto da una persona con 100 follower perché tale tweet verrà verosimilmente letto da 100 persone. Inoltre questo metodo serve, in minima parte, per filtrare molti tweet spazzatura. Dato quindi che un tweet da un utente con un follower è probabilmente spam esso pesa 1, un tweet fatto da un utente *famoso* ha peso proporzionale alla sua fama.

Si è quindi creato un grafico con la stessa struttura di quello precedente.

Comparazione quote ammortamento - rumore web

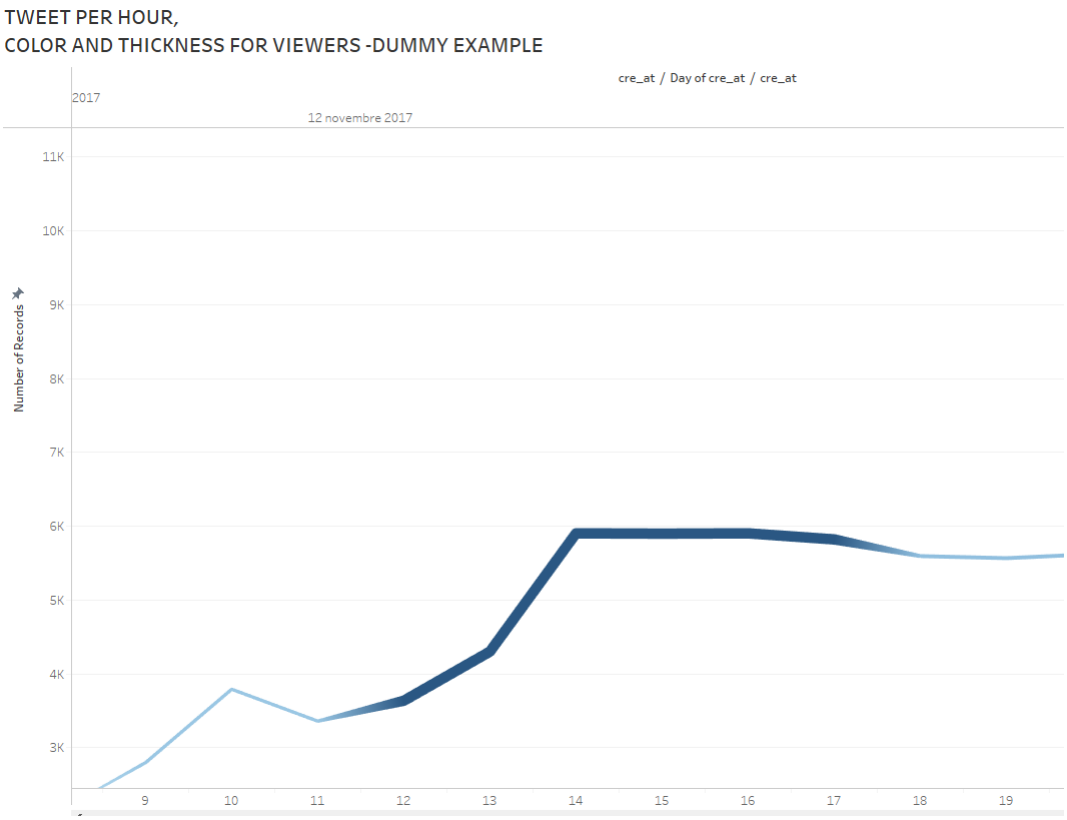
In questo grafico si è voluta mostrare la correlazione fra l'ammortamento, inteso come quota sul mese, e il numero di tweet ricevuti. Ovvero che il valore utile del titolo corrispondesse, per esempio, con l'*engagement* del pubblico dal social Twitter.



Si può efficacemente vedere che il titolo in questione è stato ammortizzato in modo corretto.

Grafico visualizzazioni - numero di tweet

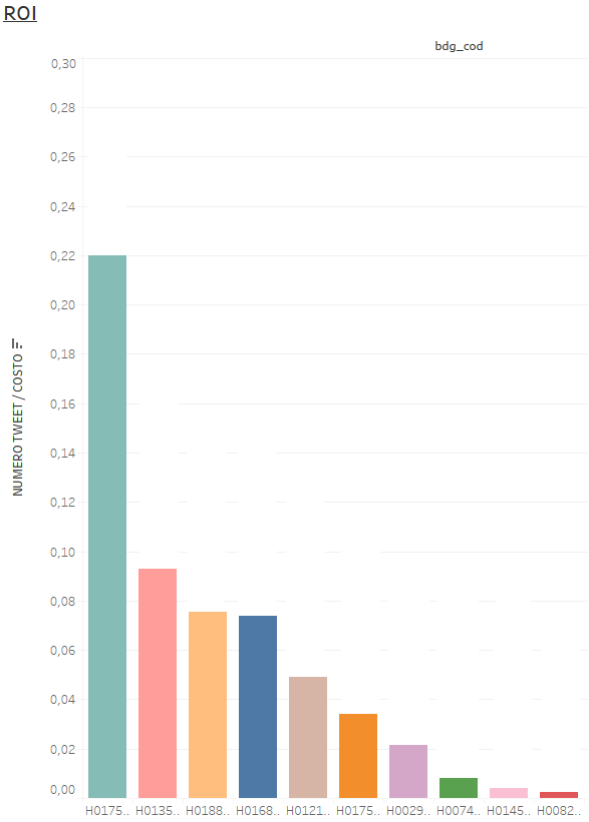
Questo grafico mostra la correlazione fra ascolti e interazioni nel mondo web. Si è scelto di rappresentare con l'altezza della linea il numero di tweet e con lo spessore (e il colore per chiarezza) il numero di ascoltatori collegati a quel canale in quel determinato momento.



Si noti come gli ascolti seguano di pari passo il numero di tweet. Inoltre che i tweet continuano anche dopo la fine della puntata o del film. Questo trend è ovviamente riscontrabile maggiormente in eventi live e serie televisive.

ROI

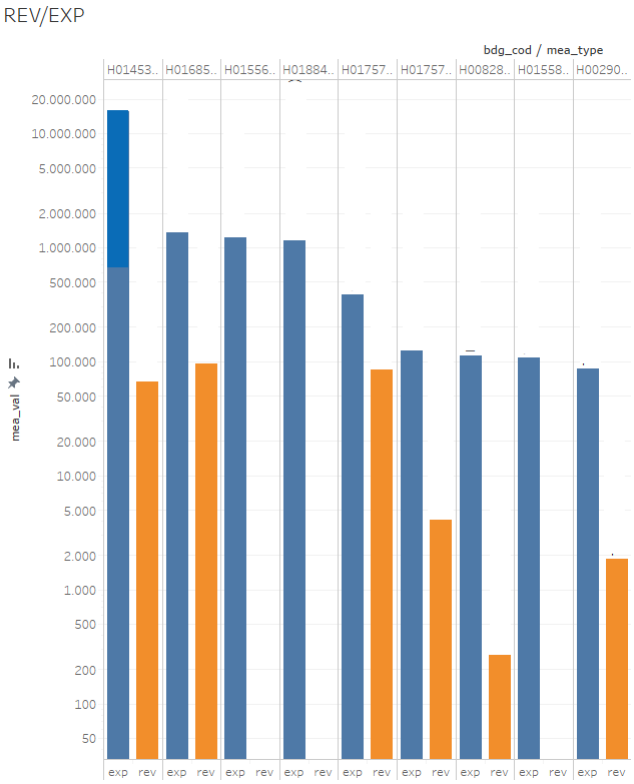
Per cercare di unire queste informazioni si è voluto creare un grafico che mostrasse, per ogni titolo, il valore di un certo KPI. I KPI, o indicatori chiave di prestazione in italiano, è un indice che monitora l'andamento di un processo aziendale o di qualità. Per questo progetto si è scelto il *ROI*.
La definizione canonica di ROI mal si sposa con gli oggetti che stiamo trattando,quindi per noi sarà ricavi su costi.



I titoli quindi possono essere messi a confronto in questo modo. Il titolo `H0175` ha totalizzato un ottimo ROI poichè o aveva un costo basso o ha ricevuto molti tweet. Diversamente il titolo `H0168` sebbene, per esempio, abbia ricevuto molti più tweet del precedente, il suo costo molto elevato ha abbassato drasticamente il suo ROI. Questa analisi ovviamente necessita di essere accompagnata da altre, per poter appunto decretare con precisione la natura di quel valore.

Grafico comparazione spese - ricavi

La peculiarità di questo sistema è la possibilità di creare un grafico automatizzato, per ogni titolo, di tutte le dimensioni che lo caratterizzano, suddividendole in costi e ricavi.



E' facile notare a colpo d'occhio se un titolo ha più spese o ricavi. Il fatto che sia tutto automatizzato permette all'utente di caricare varie sorgenti, secondo uno standard, senza preoccuparsi dell'implementazione.

Capitolo 5

Conclusioni

Con la presente tesi si è voluto provare a creare del valore aggiunto per l'azienda televisiva alla quale è indirizzato tale sistema. In particolare si è voluto fornire una visione sintetica e unitaria del gradimento degli utenti per i vari titoli cinematografici messi a disposizione.
Una delle problematiche con le quali ci si è dovuti confrontare è stata la parte di creazione e configurazione dell'architettura. Oltre alla complessa fase di progettazione dell'architettura a Data Lake, non è stato inoltre banale instanziare e configurare tutti i servizi necessari in modo che lavorassero in maniera sinergica. Per orchestrare il loro funzionamento si è usato Crontab. Esso, lanciando script di Impala in SQL, scandisce i tempi per la data integration. Ora gli script partono a distanze temporali preconfigurate, in futuro sarà possibile gestire la dipendenza dei flussi tramite segnali o ad eventi. Successivamente tale progetto verrà migrato su un vero cluster. Già in questo prototipo è stato comunque possibile mettere in evidenza gli interessanti risultati prodotti da questa analisi sinergica.

Si è potuta dare una visualizzazione d'insieme sui titoli mettendoli in relazione alla dimensione economica. Precedentemente infatti essi erano visualizzabili solo separatamente, rendendo complicate ed inefficienti le analisi. Grazie all'architettura Open source big data si è riuscito a creare un sistema adatto alla mole di dati in ingresso. Monitorando i tweet con hashtag relativi ai titoli si immagazzinavano circa 1 milione di tweet al mese, ovvero un tweet ogni 2 secondi. Questo numero è destinato a salire sia in momenti di picco, quali eventi live molto importanti, sia quando si monitoreranno molti più titoli. Tale flusso in ingresso inoltre aumenterà non appena si potrà avere accesso ai dati degli ascolti o quando si inizieranno a monitorare altri social network.

È infatti fondamentale avere accesso a molti dati semi-strutturati, la novità rispetto ad un sistema classico, per poter perfezionare le analisi. I dati semi-strutturati, sebbene contengano moltissima informazione, sono di difficile analisi. In questo progetto ci si è limitati a salvataggio e analisi sui volumi, senza scendere nello specifico del significato.

La naturale evoluzione di tale progetto sarà sicuramente la Sentiment Analysis. Tale scienza permettere di dedurre informazioni soggettive nei testi. Essa, basandosi principalmente su un dizionario per discernere la natura delle parole, permetterà di capire se il tweet analizzato era a favore o meno del contenuto analizzato e cercherà di capire lo stato d'animo dell'utente che l'ha creato. Infatti sarà possibile dare un peso ai tweet in ingresso in base al loro contenuto. È importante sapere se la persona che sta parlando del contenuto multimediale è soddisfatta o meno. Sebbene questa scienza non sia totalmente esatta per innumerevoli motivi legati anche solo alla natura ambigua della lingua, le analisi estrapolate saranno sicuramente d'interesse.

Un'altra possibile evoluzione di questo progetto può essere l'integrazione con Discovery Hub di Timextender [33]. Questo software, partendo da una varietà di data source, è in grado di creare un modello comune dei dati su quale appoggiare diversi tool di Data Visualization. Tale software permette a diversi utenti di utilizzare diversi tool, secondo le loro esigenze. Tale software permetterebbe agli utenti di business di utilizzare il software più adatto, ad esempio Tableau per gli utenti più esperti capaci di fare self service BI oppure Qlik Sense Cloud [26] per permettere agli utenti di interagire con dashboard e grafici ovunque loro siano, con un minimo input manuale.

Il futuro di questo progetto si basa sulla disponibilità di sempre più fonti e sempre più sorgenti. Infatti si prevede una maggior precisione nelle analisi quando si avrà accesso ai dati relativi agli ascolti. Per poi valutare con esattezza l'effettivo *ritorno* economico di un titolo si avrà la necessità di avere anche i ricavi dalle pubblicità passate durante il contenuto.

Per l'utente di business che lo andrà ad utilizzare ciò non comporterà un aumento della complessità dato che il sistema è totalmente automatizzato.

Bibliografia

[1] Amazon Kinesis *https://aws.amazon.com/it/kinesis/*

[2] Apache Cassandra *cassandra.apache.org/*

[3] Apache Foundation *https://httpd.apache.org/*

[4] Apache Flume *https://flume.apache.org/*

[5] Apache Hadoop *Hadoop.apache.org*

[6] Apache Hbase *https://hbase.apache.org/*

[7] Apache Hive *https://hive.apache.org/*

[8] Apache Impala *https://impala.apache.org/*

[9] Apache Kafka *https://kafka.apache.org/*

[10] Apache Kudu *https://kudu.apache.org/*

[11] Apache MapReduce *https://hadoop.apache.org/*

[12] Apache Spark *https://spark.apache.org/*

[13] Apache Storm *http://storm.apache.org/*

[14] Cloudera *https://www.cloudera.com/*

[15] C plus plus *http://www.cplusplus.com/*

[16] Crontab *http://www.adminschoice.com/crontab-quick-reference*

[17] Pasopuleti Pradeep, Purra Beulah Salome *Data Lake Development With Big Data*, Packt Pub Ltd (26 novembre 2015)

[18] Dremel *https://research.google.com/pubs/pub36632.html*

[19] Forrester Blog *https://go.forrester.com/blogs/*

[20] Gartner *https://www.gartner.com/it-glossary/big-data/*

[21] Kudu-Impala Integration *https://kudu.apache.org/docs/kudu-impala-integration.html/*

[22] Java *https://java.com/*

[23] Linkedin *https://it.linkedin.com/*

[24] Netflix *https://www.netflix.com/it/*

[25] Paypal *https://www.paypal.com/it/*

[26] Qlik Sense *https://www.qlik.com/it-it/products/qlik-sense*

[27] Raft *https://raft.github.io/*

[28] Spotify *https://www.spotify.com/it/*

[29] Stream *https://msdn.microsoft.com/en-us/library/ee391416(v=sql.111).aspx*

[30] Tableau *https://www.tableau.com/*

[31] Alex Gorelik, *The Enterprise Big Data Lake*, O'Reilly Media, Inc., (Marzo 2017)

[32] Andrew Minteer, *The Enterprise Big Data Lake*, Packt Publishing, (Luglio 2017)

[33] Timextender Discovery Hub *https://www.timextender.com/discovery-hub-4/*

[34] Twitter Inc. *https://twitter.com/*

[35] Twitter *https://dev.twitter.com/*

[36] Uber *https://www.uber.com/it/*