DATA MANAGEMENT

INTRODUZIONE

Catturare i dati: possono essere estratti da fonti interne (aziendali) o scaricati dalla rete (opendata, …). Dalla rete inoltre è possibile ottenere dati attraverso determinate applicazioni, per esempio, per scaricare dati da Twitter. Esiste infine una tecnica chiamata "scraping" che consente di acquisire dati altrimenti incaccesibili.

Memorizzare i dati: prima si cercava di organizzarli, ora si è introdotto il concetto di "data lake".

Controllare la qualità: è il lavoro più impegnativo che va effettuato.

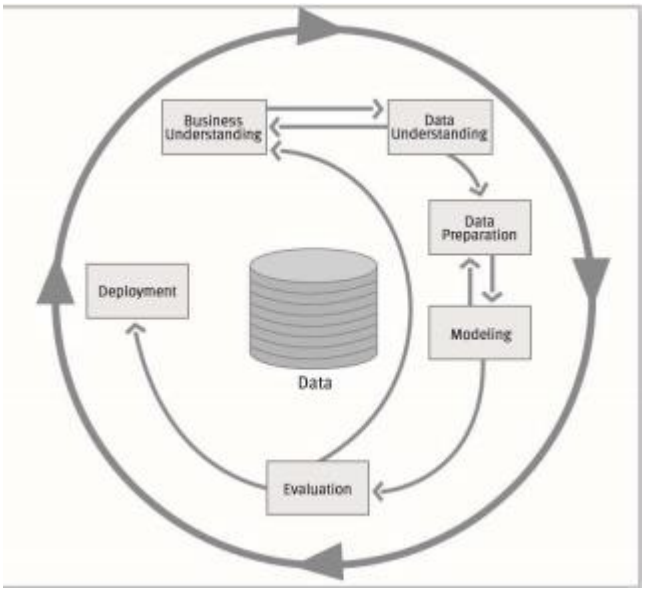
Integrazione delle informazioni: un esempio è il controllo incrociato che effettua l'agenzia delle entrate.

Analisi dei dati: consiste nell'utilizzo di tecniche statistiche e di machine learning.

Diffusione di metadati: inserimento dei dati in un contesto e quindi diffusione di informazioni quali trasformazioni effettuate e collocamento temporale (aggiornamento dei dati).

Una volta definito il ciclo di vita dei dati, bisogna porsi il problema della distribuzione di questi; in altre parole bisogna pensare di effettuare backup per salvaguardare i dati immagazzinati. Infine, verranno esposte alcune tecniche per immagazzinare ed interrogare i dati.

DATA LIFECYCLE



Avere più informazioni non sempre porta a prendere decisioni migliori. Il primo problema da porsi è la definizione di un obiettivo da raggiungere. Successivamente bisogna chiedersi di che genere di dati si ha bisogno e capire dove questi possano essere recuperati. Poi è necessario "preparare" i dati e quindi pulirli e trasformarli per renderli utili. La fase successiva consiste nell'esplorazione dei dati disponibili attraverso tecniche di visualizzazione. Una volta preparato il dataset si passa alla fase di modellazione. Infine è importante la fase di presentazione e di rilascio dei risultati ottenuti.

Poniamoci dunque la seguente domanda: dove posso recuperare i dati?

- All'interno della realtà aziendale, quindi sono il proprietario dei dati; in questa situazione è possibile avere problemi di collocazione dei dati (in che cartella sono?) o di accesso (ho i permessi?).
- Da fonti esterne, quindi non sono il proprietario dei dati; in questo caso posso avere dei problemi di costo (i dati sono gratis o a pagamento?). Tale costo è legato anche al formato e alla qualità dei dati.

Una volta identificata la fonte, esistono vari modi per ottenere i dati desiderati: download, SQL queries, API[1] (application program interface), scraper e app.

Raccolti i dati, bisogna valutarne la qualità, che dipende dallo scopo dell'analisi e si articola in accuratezza, completezza, consistenza e dilatazione nel tempo dei dati. Questa fase è necessarie perché generalmente i dati appena acquisiti sono "sporchi" a causa di:

[1] API rest: post (create), get (read), put (update) & delete (delete).

- Produzione errata dei dati (errore umano in fase di input, errori sistematici durante l'acquisizione, fonti differenti con rappresentazioni diverse della realtà, …)
- Storage (formati differenti, formati insufficienti, …)
- Uso (capacità di analisi e di processing insufficienti, cambiamento dei requisiti di IQ, problemi di sicurezza e di accesso, …)

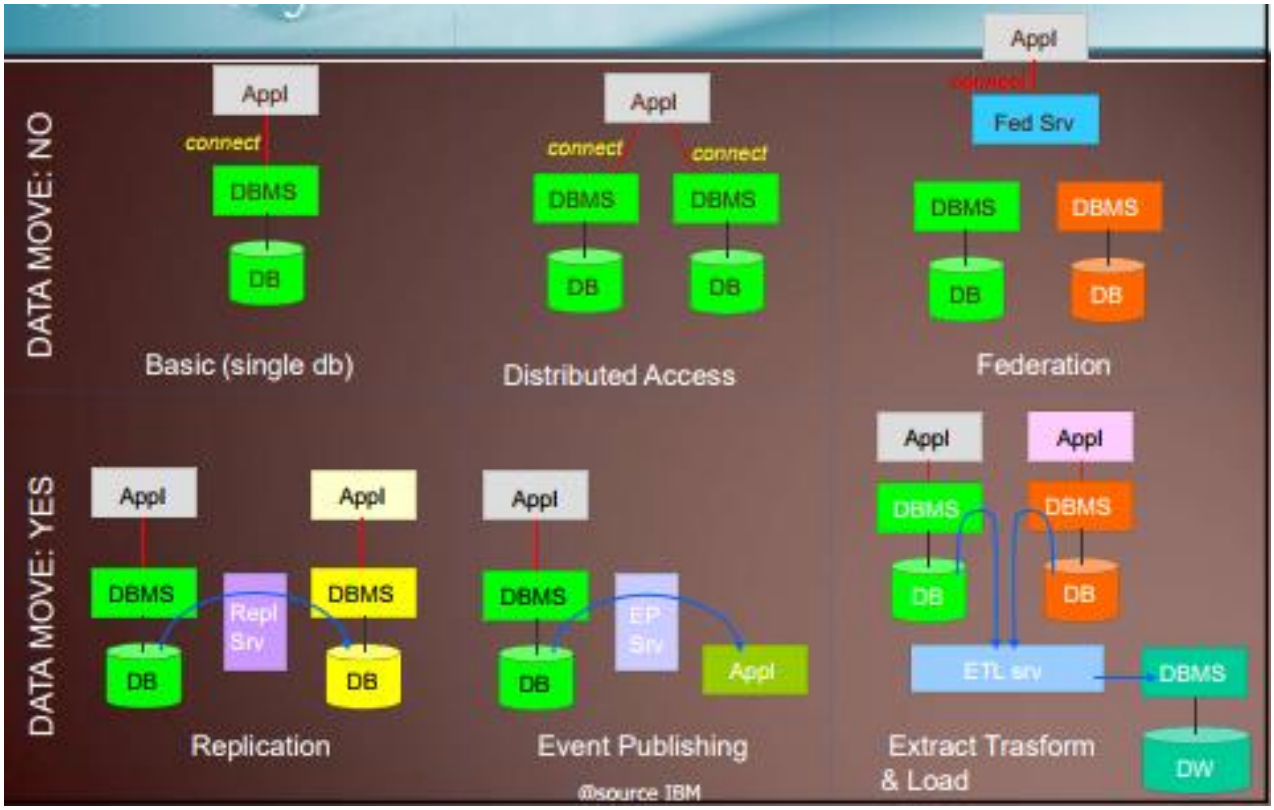
Una volta superato il problema della qualità dei dati, si passa alla "data integration". Generalmente, nel corso di un progetto, si utilizza più di un dataset e l'integrazione dei dati presenta problemi di semantica. Esistono perciò tecniche appropriate che vedremo in seguito. Infine bisogna affrontare il problema dello storage, che nel caso dei big data necessita di essere distribuito.

In sostanza, il data management consiste nell'orchestrazione di tutti i processi riguardanti il ciclo di vita dei dati, che nell'ottica moderna sono il "new oil" (we need to find it, extract it, refine it, distribute it and use it to drive economic prosperity).

Citiamo inoltre il problema del trasporto dei dati (wallet gardner – giardini protetti); facendo un esempio, è possibile mandare un messaggio da un telefono con operatore Tim ad uno con operatore Vodafone, ma non è possibile mandare un messaggio da Telegram a Messenger.

ARCHITETTURE DISTRIBUITE

Perché distribuire i dati? (1) Dimensioni troppo elevate (2) Necessità geografiche (3) Sicurezza – backup (4) Efficienza e problemi di accessi multipli. Lettura 1 (file nella cartella drive)



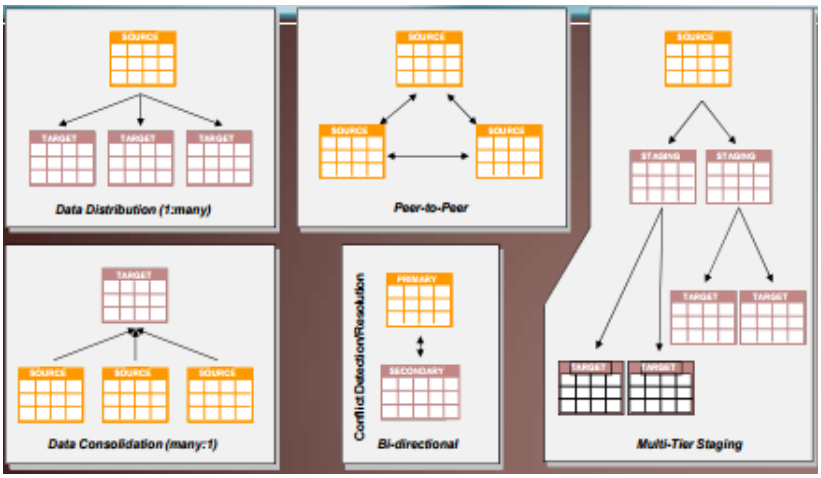
Tipi di architettura:

- Shared everything, ovvero un'architettura avete un DB centrale a cui si collegano i vari terminali.
- Shared disk, ovvero un'architettura distribuita in cui tutti i dischi sono accessibili da ogni cluster di nodi. Le sue caratteristiche sono: rapida adattabilità ai cambiamenti di carichi di lavoro, alta disponibilità, performa meglio in ambienti di "lettura pesante" e non c'è necessità di partizionare i dati.

- Shared nothing, ovvero un'architettura distribuita in cui ogni nodo è indipendente e autosufficiente. I suoi punti di forza sono: capacità di sfruttare hardware semplici e poco costosi, una scalabilità quasi illimitata, lavora bene in ambienti in cui si trattano grandi volumi di dati. Tuttavia, i dati non sono condivisi, ma partizionati tra i cluster.

Una domanda sorge spontanea: cosa si intende per "alta disponibilità"? È un misto tra architetture tecnologiche, persone e processi, mentre non è un concetto vicino a scalabilità e maneggevolezza. La disponibilità di un'architettura può essere interrotta per aggiornamenti, blackout, etc.

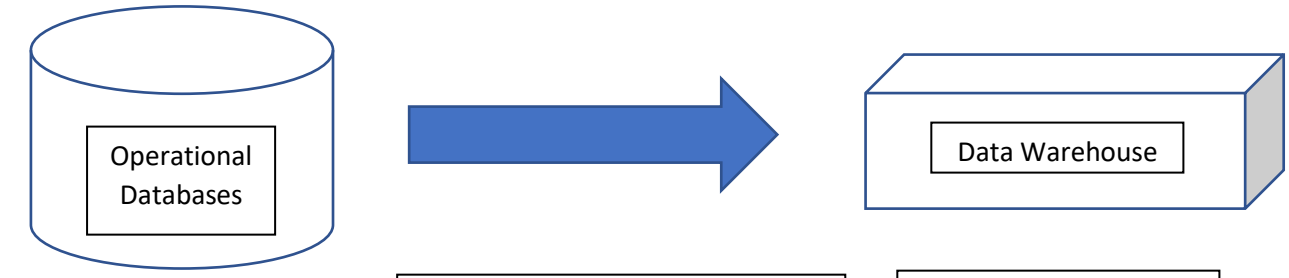
Un file di log è un file sequenziale memorizzato in una memoria stabile ("never fail"). Il log contiene tutte le informazioni riguardanti le operazioni svolte, ad esempio su un DB. Vengono memorizzati due tipi di informazioni: transazioni e eventi di sistema (checkpoints & dump). Il file di log è dunque un file sequenziale che permette di recuperare alcune informazioni. Per replicare i dati è sufficiente avviare tutto il file di log su un'altra macchina.



Come creare una replica:

- Staccare il disco, copiarlo e riattaccare tutti i dischi sulle macchine interessate.
- Fare il backup, copiarlo e reimportare le copie su altre macchine.
- Fare un full backup attraverso il file di log all'inizio per aggiornarlo poi tramite altri file di log.
- Publisher, distributor, subscribers

Il datawarehouse è un prodotto software che diventa utile solo attraverso un processo di data warehousing. Possono esserci problemi di accesso (nonostante la grande mole di dati), di incongruenza tra fonti di uguale valore e di scarsa qualità.



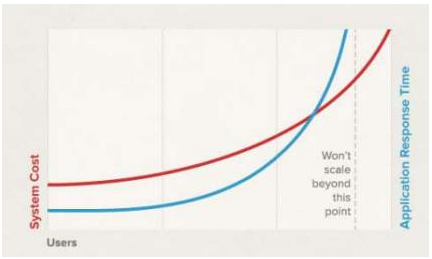
Reservation, Analysis & Management of patients

Processo di estrazione, pulizia, validazione, filtraggio, trasformazione e caricamento

Business oriented & Focus on application and procedure

NoSQL (Not Only SQL)

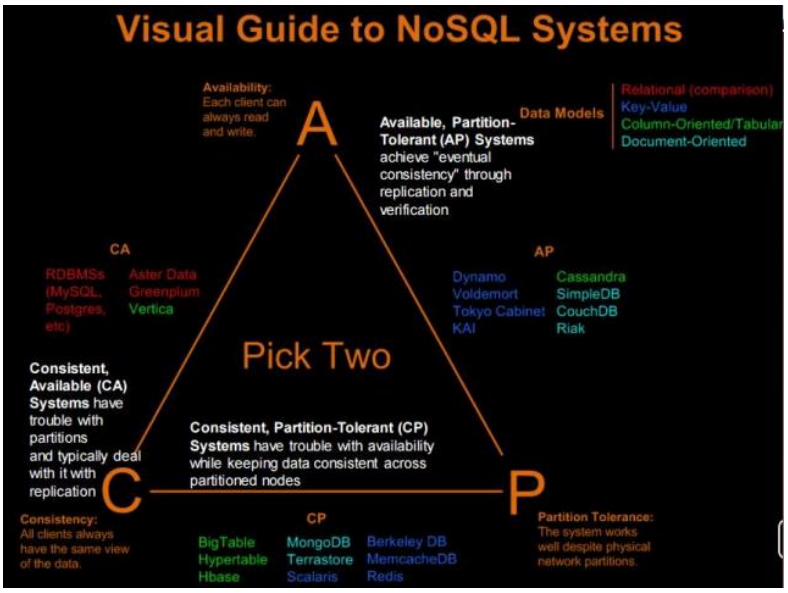
Il modello SQL gode delle cosiddette proprietà "ACID": atomicità, consistenza, isolazione e durabilità. Gli RDBMS sono scalabili aggiungendo server all'architettura, tuttavia non è possibile aggiungere server all'infinito.



NoSQL è concepito su un mdello “schema free” o “schemaless” ed è basato sul “CAP Theorem”. Inoltre, gode delle cosiddette proprietà “BASE”, ovvero Basic Available, Soft state & Eventually consistency.

CAP Theorem: è impossibile per un sistema distribuito computerizzato soddisfare simultaneamente tutte le seguenti tre grazie:

- Consistency – consistenza (tutti i nodi vedono gli stessi dati nello stesso momento)
- Availability – disponibilità (una garanzia che ogni richiesta riceva una risposta di successo o insuccesso)
- Partition tolerance – corretto funzionamento delle partizioni (il sistema continua ad operare anche se una parte del sistema fallisce)



Un sistema distribuito può soddisfare al massimo due di queste garanzie contemporaneamente. I sistemi RDBMS sono generalmente CA (a volte è possibile renderli anche CAP). I sistemi NoSQL sono principalmente CP (non funzionano 24/7) o AP (a volte sono inconsistenti).

I modelli NoSQL sono:

- Key-Value Stores → tabelle in cui una determinata chiave fa riferimento ad un preciso valore. La mappatura delle chiavi-valori è basata sull’algoritmo di hash ed è molto efficiente per la distribuzione dei dati. [Dynamo, Voldemort, Rhino DHT, …]
- Column Family Stores → tabelle multicolonnari con chiavi che fanno riferimento ad un set di colonne. Permette di gestire grandi moli di dati. [BigTable, Cassandra, HBase, Hadoop, …]
- Graph DB → è organizzato in nodi e archi, ovvero relazione tra nodi. Sia nodi che archi possono avere delle proprietà. Non è facilmente scalabile. [Neo4j, FlockDB, GraphBase, InfoGrip, …]
- Document Based → basato su documenti (tipicamente JSON). Tali documenti sono individuati nel DB attraverso la loro chiave univoca. È uno schema estremamente flessibile basato sulla corretta indentazione (embedding), necessita di pochi indici e per questi motivi è molto performante. [CouchDB, MongoDB, Lotus Notes, Redis, …]
- RDF Databases → verrà trattato nel corso di datasemantics.

MongoDB

È un “document based management system”. I dati sono immagazzinati in file JSON binari (BSON). L’accesso ai dati avviene tramite un sistema di indicizzazione. Non sono permesse operazioni di join.

Uno dei punti di forza di MongoDB è la sua scalabilità orizzontale (data dalla possibilità di aumentare il numero di nodi). Questa scalabilità è consentita dallo “sharding”, ovvero una

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

tecnica per la creazione di un database distribuito in cui ogni nodo contiene una porzione del database. Tale tecnica è basata sulla definizione di una shard key, ovvero un campo indicizzato che deve essere presente in tutti i documenti di una collezione e che stabilisce come essa debba essere suddivisa in parti. La shard key può essere "range-based" (valori di chiave simili stanno nello stesso cluster), "hash based" (la partizione viene divisa in base all'hash del valore di uno shard key) o "tag aware". Poi parla di "massive data upload", ma dalle slide non si capisce bene (neanche la parte prima non si capiva tanto).

GraphDB & Neo4j

Un grafo è una struttura matematica costituita da nodi e archi, oppure un insieme di nodi con un certo tipo di relazioni tra loro. In un grafo, il concetto più importante è quello di nodo (come l'entità nel modello ER) e il modo in cui questi sono semanticamente connessi è modellato tramite gli archi (relazioni), i quali possono avere o meno un senso di percorrenza. I grafi sono molto utilizzati in ambito sociale (social networks, …), di raccomandation systems, geografico, logistico, di transazioni finanziarie, bioinformatico, etc. Esistono due approcci per gestire un grafo: Graph Database (un DBMS capace di gestire in modo nativo i grafi - OLTP) e Graph Compute Engines (un motore di analisi per grafi - OLAP). Le operazioni consentite su un Graph Database sono: create, read, update e delete. I Graph Databases sono utilizzati in contesti transazionali per dati complessi e le interrogazioni sono risolte tramite l'attraversamento di un grafo. Possono avere uno storage nativo (ottimizzati e progettati per la gestione dei grafi) o non nativo (trasformano i dati dal formato grafo ad altri formati). Inoltre, anche il loro "processing engine" può essere nativo o meno. I grafi sono molto performanti in lettura, ma perdono tantissimo in scrittura.

Il modello a documenti è un caso specifico del modello a grafo (quando non ci sono cicli → albero).

La modellazione di un grafo avviene "come su una lavagna". Risulta quindi essere una modellazione più simile a quella umana. Per interrogare i grafi ci sono molti linguaggi; il più usato è Cypher, basato su una logica pattern-matching e facile da interpretare in quanto molto espressivo e simile al linguaggio umano. È un linguaggio dichiarativo (come SQL) e permette operazioni di aggregazione e ordinamento oltre a consentire di aggiornare un grafo. Neo4j supporta questo linguaggio per le query. Neo4j può analizzare grafici singoli contenenti decine di miliardi di nodi, relazioni e proprietà. Inoltre, a differenza di un RDBMS, non deve effettuare molteplici operazioni di join, ma parte da un nodo identificato attraverso un'opportuna indicizzazione e agisce tramite una combinazione di puntatori e sistemi di corrispondenza per trovare i dati richiesti. Il tempo di esecuzione non dipende dalla grandezza complessiva del grafo, ma solo dalla query. Lui parla anche di "non functional characteristics" dei Graph Databases, ma non sono spiegate.

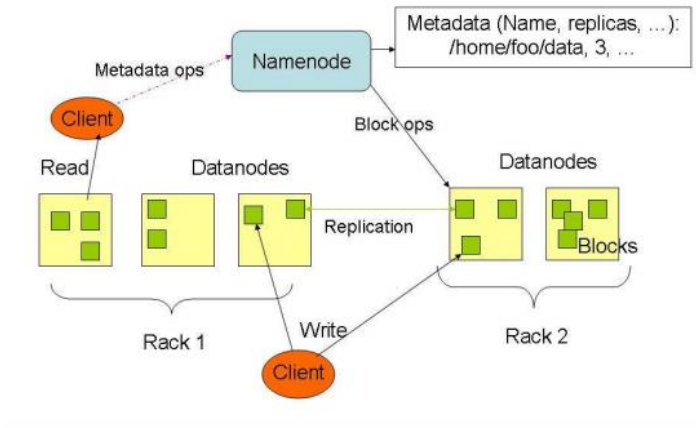
BIG DATA

La prima definizione di big data viene data da Gartner nel 2012: "Big data is high volume, high velocity and/or high variety information assets". Analizzare big data su un singolo server "big" è un processo lento, costoso e difficile da realizzare. La soluzione è effettuare un'analisi distribuita su hardware poco costosi tramite un sistema di calcolo parallelo. I problemi principali della distribuzione dei dati sono la sincronizzazione, i cosiddetti "punti morti" (deadlock), la larghezza della banda, la coordinazione tra i nodi e i casi di fallimento (failure) del sistema. Questo genere di architettura ha comunque molti vantaggi, tra cui: scala linearmente, l'attività di calcolo è rivolta ai dati e non viceversa (cambio di paradigma), gestisce i casi di fallimento (failure) e lavora con hardware con potenza di calcolo "normale".

HDFS

L'architettura HDFS (Hadoop Distributed File System) è un file system distribuito progettato per girare su hardware base (commodity hardware). Sebbene ci siano molte similitudini con altri sistemi di file system distribuiti, le differenze sono comunque significative. In particolare, HDFS è fortemente tollerante agli errori

ed è progettato per girare su macchine poco costose. HDFS fornisce un accesso ad alta velocità ai dati delle applicazioni ed è ideale per applicazioni con data set di grandi dimensioni.



HDFS ha un'architettura master/slave. Un cluster HDFS consiste in un singolo NameNode, un server master che gestisce il file system NameSpace e regola gli accessi ai file da parte dei clients. Inoltre, sono presenti un certo numero di DataNodes, generalmente uno per ogni nodo nel cluster, che gestiscono lo storage collegato ai nodi su cui vengono eseguiti. Internamente, un file è splittato in uno o più blocchi (tipicamente di 128 MB ciascuno) e questi sono memorizzati in un set di DataNodes. Il NameNode esegue le operazioni del file system NameSpace, ovvero apertura, chiusura e "rename" dei file e delle directories. Inoltre, determina la mappatura dei blocchi nei DataNodes. I DataNodes sono responsabili invece delle richieste di operazioni di lettura e scrittura da parte del file system dei clients. I DataNodes permettono anche la creazione, l'eliminzazione e la replica dei blocchi sotto istruzioni del NameNode. I Meta-Data (lista dei file, lista dei blocchi, lista dei DataNodes, attributi del file, …) sono tutti memorizzati nella memoria principale. Esiste inoltre un Transaction Log che registra la creazione, l'eliminazione e qualunque altra operazione avvenga su un file. La strategia di piazzamento dei blocchi del file tra i DataNodes è la seguente:

- Una replica sul nodo locale
- Una seconda replica su un rack (collezione di nodi) remoto
- Una terza replica sullo stesso rack remoto
- Delle repliche piazzate in modo randomico

I client leggeranno il file dalla replica più vicina a loro (rack awareness). Esiste poi un sistema per verificare la correttezza dei dati. Può succedere infatti che un blocco inviato dal DataNode arrivi corrotto. Il client software HDFS implementa un controllo checksum dei file. Quando un utente crea un file, genera anche un checksum per ogni blocco del file e memorizza questi checksum in un file nascosto separato nello stesso NameSpace HDFS. Quando un client richiede l'accesso al file, verifica che i dati ricevuti da ogni DataNode combacino con il checksum associato. Se così non è, il client deciderà di reperire quel determinano blocco da un altro DataNode che possiede una replica di tale blocco di dati. Il NameNode verifica inoltre che i DataNodes, i quali inviano continuamente "segnali di vita", siano tutti funzionanti e gestisce gli eventuali malfunzionamenti. Tuttavia, il NameNode rappresenta un "single point of failure". Per questo motivo i Transaction Log sono memorizzati in più directrories: nel file system locale e in uno remoto.

Formati di file supportati: Text, CSV, JSON, SequenceFile, binary key/value pair format, Avro*, Parquet*, ORC, optimized row columnar format.

Map Reduce

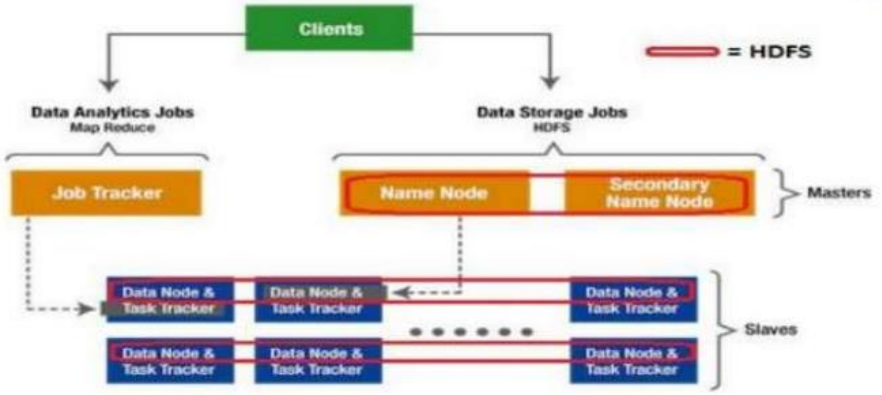
Map Reduce è un motore di computazione distribuito. Ogni programma è scritto in uno stile funzionale ed è eseguito in parallelo. "Mapred" risolve problemi legati a Map Reduce quali:

- Come assegnare i “lavori” ai singoli “worker”
- Cosa succede se ci sono più “lavori” che “worker”
- Cosa succede se gli “worker” devono condividere risultati parziali
- Come aggregare i risultati parziali
- Come scoprire se tutti gli “worker” hanno finito il proprio task
- Cosa succede se uno “worker” “muore”

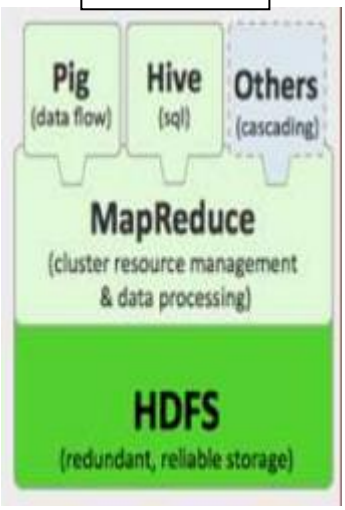
La fase di Map esegue lo stesso codice su un grande ammontare di record, estrae le informazioni rilevanti, le ordina e le unisce. La fase di Reduce aggrega i risultati intermedi e genera l’output. Il programmatore dovrà esclusivamente definire le due funzioni di map e di reduce. Tutte le altre attività le svolge il mapred.

Hadoop

Hadoop è un framework che supporta applicazioni distribuite con elevato accesso ai dati. Hadoop unisce il sistema MapReduce (parallel and distributed computation) e l’HDFS (hadoop storage and file system).

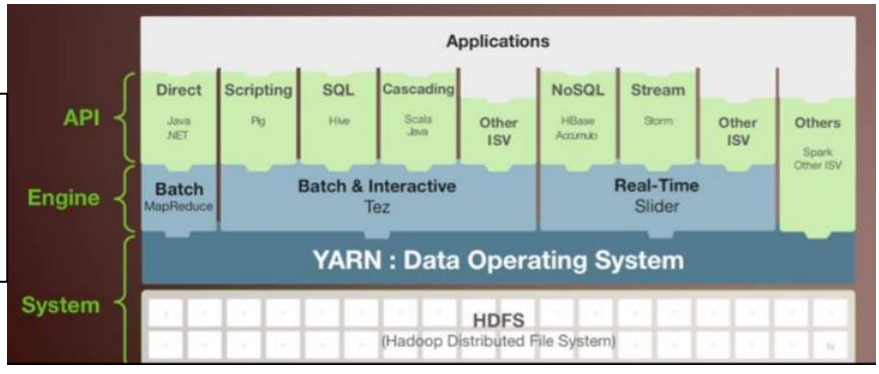


Hadoop 1.0



Pig è uno scripting language che esegue l’analisi dei dati. Gli script in “pig latin” sono tradotti in lavori di MapRed. Hive è un’interfaccia simile a SQL per dati memorizzati in HDFS. I piani di esecuzione sono generati automaticamente da Hive. Hive rappresenta quindi un data warehousing basato su Hadoop. Questo applicativo è utilizzato per report giornalieri, misure di attività degli utenti, data e text mining, machine learning e per attività di business intelligence come pubblicità e individuazione degli spam. Tuttavia, il sistema di MapRed ha dei limiti: è difficile da comprendere, i task non sono riutilizzabili, è incline agli errori e per analisi complesse richiede molti lavori di MapReduce. In Hadoop 1.0 dunque, ogni “jobtracker” deve gestire molti compiti: gestire le risorse computazionali, scandire i task dello stesso lavoro, monitorare la fase di esecuzione, gestire i possibili “failure” e molto altro ancora. La soluzione a questo problema è stata splittare la fase di gestione in gestione dei cluster e gestione dei singoli lavori. Questo sistema è stato messo in pratica da YARN (Yet Another Resource Negotiator) in cui è presente un ResourceManage globale e un ApplicationMaster per ogni applicazione.

Hadoop 2.0

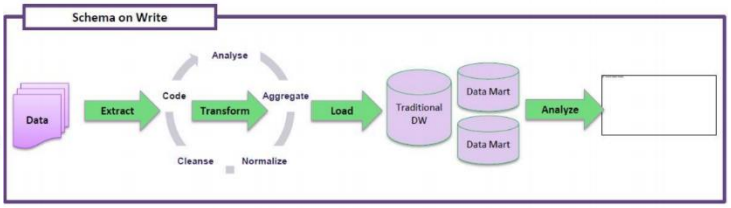


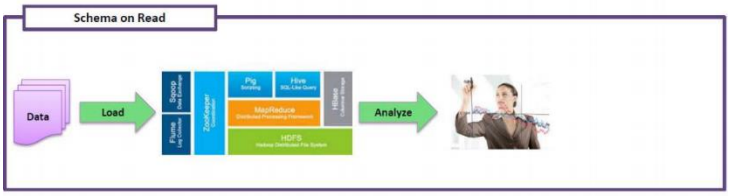
Vengono rilasciati inoltre altri applicativi quali Accumulo, Hbase e Spark. In Cloudera sono presenti anche Impala, Mahout, Sqoop e Flume. Inoltre, anche Hortonworks è basato sulla medesima architettura.

DATA LAKE

Il termine “data lake” è stato coniato nel 2010 da James Dixon, il quale ha distinto due approcci di gestione dei dati: Hadoop e i data warehouses. Quest’ultimo (anche detto data mart) consiste nel memorizzare i dati in modo pulito e strutturato per una facile “consumazione” futura. I data lake invece sono una grossa mole di dati non strutturate e non puliti da cui ogni soggetto (autorizzato) può attingere, ma passando per un processo di analisi e di campionamento accurato. Per realizzare un approccio “bottom up” non c’è necessità di definire uno schema dei dati prima di caricarli, ma i dati devono essere modellati a seconda dei task. Nei sistemi EDW (Enterprise Data Warehouse) i task sono fortemente collegati con il sistema di modellazione dei dati.

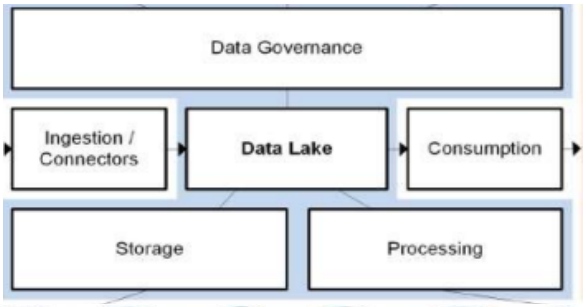
Attribute	EDW	Data lake
Schema	Schema-on-write	Schema-on-read
Scale	Scales to large volumes at moderate cost	Scales to huge volumes at low cost
Access methods	Accessed through standardized SQL and BI tools	Accessed through SQL-like systems, programs created by developers, and other methods
Workload	Supports batch processing, as well as thousands of concurrent users performing interactive analytics	Supports batch processing, plus an improved capability over EDWs to support interactive queries from users
Data	Cleansed	Raw
Complexity	Complex integrations	Complex processing
Cost/ efficiency	Efficiently uses CPU/IO	Efficiently uses storage and processing capabilities at very low cost
Benefits	• Transform once, use many • Clean, safe, secure data • Provides a single enterprise-wide view of data from multiple sources • Easy to consume data • High concurrency • Consistent performance • Fast response times	• Transforms the economics of storing large amounts of data • Supports Pig and HiveQL and other high-level programming frameworks • Scales to execute on tens of thousands of servers • Allows use of any tool • Enables analysis to begin as soon as the data arrives • Allows usage of structured and unstructured content from a single store • Supports agile modeling by allowing users to change models, applications, and queries

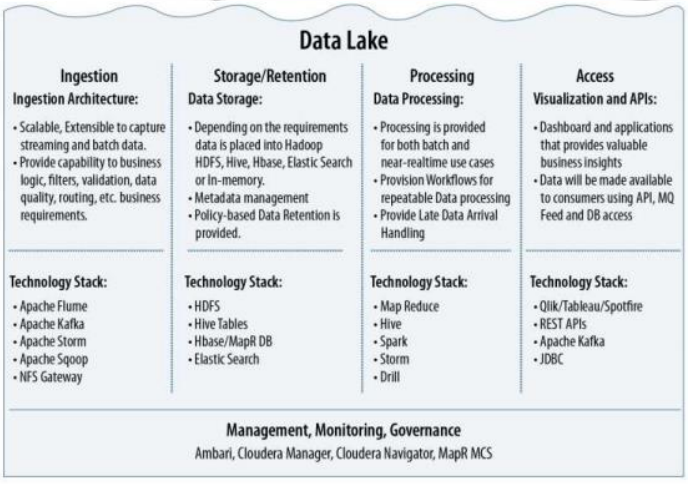




Le componenti di un’architettura data lake sono le seguenti:

- Storage, basata su un’architettura HDFS (vedi lezione precedente).
- Data Ingestion, ovvero l’acquisizione di dati da fonti esterne che avviene tramite sistemi come Sqoop (RDBMS), Flume (Web Servers), Kafka o Storm (Streaming Data).
- Data Processing, che si suddivide in tre fasi: data preparation, data analystics e result provisioning for consumption. Per effettuare tutte queste operazioni ci si serve di software quali MapReduce, Spark, Flink o Storm. Per eseguire molte attività di manipolazione viene spesso utilizzato NiFi, ovvero un tool workflow based che integra vari applicativi.
- Data Governance, che comprende Lineage, Integration, Authentication and Authorization, Search, Quality, Audit Logging, Metadata Management, Lifecycle Management e Security.





Per acquisire dati in real-time è necessario catturare ogni aspetto di tali dati e con il minimo ritardo possibile. Inoltre, talvolta è necessario sia immagazzinare questi dati in streaming che analizzarli all'istante. La principale limitazione dell'architettura Lambda è che per fare ciò, la logica architetturale va implementata due volte, spesso con tool diversi (Storm, Flink, Spark, …). L'architettura Kappa risolve questo problema. Per maggiori dettagli su tutta questa lezione → Lettura 2 (file nella cartella drive)

DATA QUALITY & DATA INTEGRATION (RECORD LINKAGE & DATA FUSION)

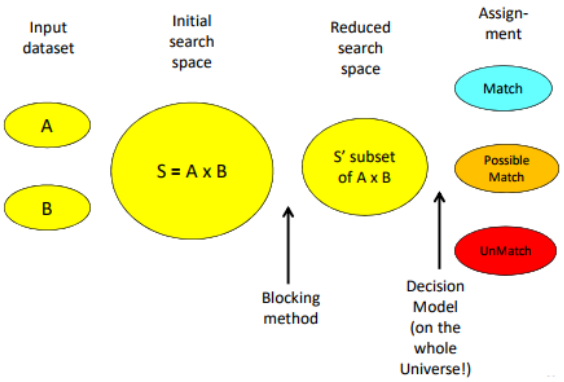
In questa lezione ci focalizzeremo principalmente sulle fasi di "verifica della qualità dei dati" (Data Understanding), di "pulizia" e di "integrazione" dei dati (Data Preparation). Come primo aspetto, bisogna concentrarsi sui casi di "deduplication"; in altre parole bisogna chiedersi: ci sono record nella stessa tabella che si riferiscono allo stesso oggetto/persona della realtà? Si possono stabilire molteplici regole per decidere se due record si riferiscono allo stesso oggetto, sia sintattiche che semantiche. Una volta individuati due record "analoghi", bisogna unirli (data fusion) in un solo record contenente le informazioni "corrette". Una volta analizzato questo aspetto di "qualità" dei dati, immaginiamoci di dover integrare una tabella con un'altra e quindi di dover cercare i record delle due tabelle diverse che sono collegati tra loro. Questo processo si chiama "record linkage" e può essere svolto, ad esempio, individuando le chiavi delle due tabelle (una primaria e una esterna, ad esempio), e se queste corrispondono si procede con il "merge" dei due record. Tuttavia, se la data quality non è delle migliori (a causa di input sbagliati o di standard non condivisi), di conseguenza anche la data integration sarà scarsa.

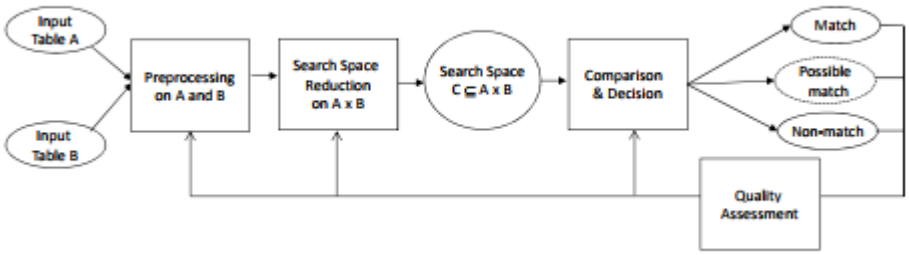
La data integration si articola in due fasi: record linkage (identificazione dei set di record che identificano lo stesso "oggetto reale") e data fusion (scelta di un record unico rappresentativo del set precedente).

Record Linkage

Esistono vari sinonimi per questa fase: Object Identification, Deduplication (su un dataset), Object Matching e molti altri. L'output di un algoritmo di record linkage può essere: "matching tuples", "not matching" o "don't know" (o "possible matching). Esistono più tecniche di record linkage:

- Empirica, ovvero due tuple vengono unite se la loro distanza (in termini sintattici o anche altre forme di distanza) è piccola
- Probabilistica, ovvero una generalizzazione sulla popolazione di regole estratte da un campione
- Knowledge Based, ovvero decisioni prese seguendo regole prestabilite
- Mixed, ovvero un misto tra il secondo e il terzo approccio





Il record linkage può essere effettuato anche tra set di dati in formati diversi.

Fasi del record linkage con approccio probabilistico:

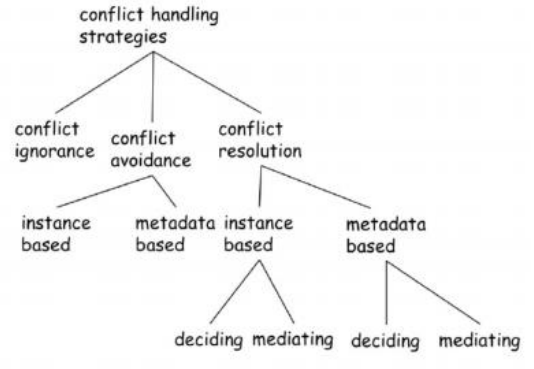
1) Preprocessing – Normalizzazione dei formati

2) Blocking – Blocco con riduzione dello spazio di ricerca (ad esempio con la tecnica “Sorted Neighbour”, che consiste nello scegliere una chiave, ordinare i dataset per quella chiave univoca e generare delle finestre mobili che si muovono attraverso i file ordinati)
3) Compare – Scelta di una funzione di distanza
4) Compare – Scelta di un campione di coppie di tuple di cui si sa a priori l’accoppiamento (o meno) e valutazione per ogni valore della funzione di distanza il matching (o meno)
5) Decide – Valutazione della distanza tra coppie della popolazione; inoltre, basandosi sullo step 4, scelta della soglia minima (dmin) e massima (dmax)
6) Decide – Per tutte le coppie di tuple nella popolazione, se d < dmin “matching”, se d > dmax “not matching” e se dmim < d < dmax “possible matching”

Una possibile funzione di distanza è la “edit distance”. La edit distance non normalizzata UED[v1,v2] è pari al numero di inserimenti, cancellazioni e sostituzioni di simboli alfanumerici necessari per trasformare v1 in v2. La edit distance normalizzata ED[v1,v2], dove v1 e v2 sono valori nel dominio D in cui il massimo numero di simboli è n, è pari a 1 - UED[v1,v2]/n ed assume valori in [0,n]. Ad ogni modo, esistono diversi tipi di normalizzazione, di blocking, di funzioni di distanza e di campionamenti. Per valutare la qualità delle metriche scelte si utilizzano misure come recall e precision. Talvolta, nell’algoritmo si inserisce la presenza di un errore deterministico a cui è associato un peso minore nella funzione di distanza (ad esempio per le lettere “r” e “t” nella tastiera QWERTY).

L’approccio basato sulla conoscenza inserisce dei vincoli/regole di integrità alla funzione di distanza.

Data Fusion



Durante questa fase si possono incontrare possibili conflitti. Esistono perciò delle strategie per gestire tali conflitti:

- Conflict Ignoring, che ignora il problema lasciando la risoluzione all’utente.
- Conflict Avoiding, che gestisce il problema generalmente scegliendo una delle fonti come “la più affidabile” e creando il record rappresentativo da quella fonte.
- Conflict Resolution, che fa attenzione a dati e metadati prima di decidere sulla risoluzione del conflitto. Questa strategia può essere divisa in “deciding” (quando viene scelto un valore tra quelli esistenti) e “mediating” (quando viene scelto un valore che non appartiene per forza a quelli esistenti, per esempio la media).

Strategy	Classification	Short Description
Pass it on	ignoring	escalates conflicts to user or application
Consider all possibilities	ignoring	creates all possible value combinations
Take the information	avoiding, instance based	prefers values over null values
No Gossiping	avoiding, instance based	returns only consistent tuples
Trust your friends	avoiding, metadata based	takes the value of a preferred source
Cry with the wolves	resolution, instance based, deciding	takes the most often occurring value
Roll the dice	resolution, instance based, deciding	takes a random value
Meet in the middle	resolution, instance based, mediating	takes an average value
Keep up to date	resolution, metadata based, deciding	takes the most recent value