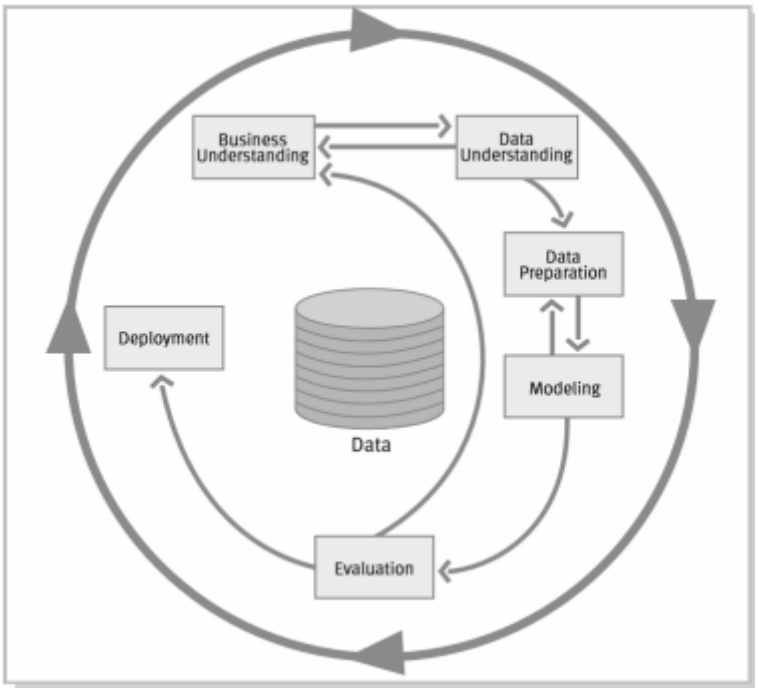
# Dispensa del corso di Data Management

[0]Pietro Carmine Valenti, CDLM Data Science, Università di Milano-Bicocca

# Indice

# Data Lifecycle

In un'epoca in cui i dati sono diventati un bene di grande valore, ve ne sono a disposizione un gran numero (**big data**) e vengono messi a punto metodi di analisi sempre più potenti (**machine learning**), il metodo scientifico di analisi dei dati (formulazione di ipotesi, test, modellazione) è stato messo molto in discussione; per tale motivo è fondamentale che gli esperti analisti svolgano tutte le principali fasi( **Crisp-DM reference model**):




- **Business understanding**: fase iniziale in cui si pone il focus sulla comprensione del progetto e dei suoi obiettivi; si estrapola conoscenza e la si utilizza per definire il problema. Dove trovo i dati? fonti interne o esterne? gratis/pubblici o a pagamento? Rispondendo a tali domande è infine possibile ottenere i dati necessari.
- **Data understanding**: fase in cui si prende confidenza coi dati, si identificano i problemi d qualità, si verificano le strutture preliminari e infine si formulano alcune ipotesi sulle informazioni nascoste.
- **Data preparation**: insieme delle tecniche volte a pulire i dati e a renderli utilizzabili per l'analisi.

# Data Variety

## 2.1 Data Modeling

Generalmente i dati vengono organizzati in modelli, ovvero schemi che associano a determinati oggetti le informazioni che li riguardano; ne esistono di diverso tipo, tuttavia i principali non relazionali vengono riassunti nel grafico sottostante, il quale mette in risalto le differenze sulla base della grandezza e della complessità:


Negli ultimi anni, a causa delle crescente quantità di dati e della loro natura multidimensionale e gerarchica, i modelli NoSQL hanno subito un notevole sviluppo, andando a sostituire in molti casi i classici modelli SQL (tabellari); alcuni esempi riguardano i *database* documentali e a grafo. Tale cambiamento è causato inoltre dalla rigidità dei modelli SQL, i quali hanno spesso bisogno di un gran numero di tabelle per essere organizzati e possiedono uno schema estremamente rigido, che poco si presta alla strutturazione di dati per fenomeni multi-relazionali.

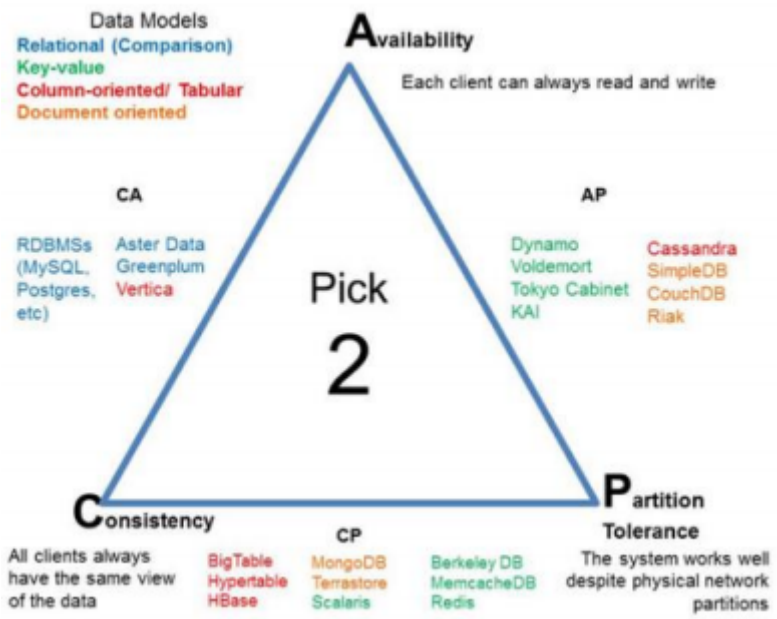
La caratteristica che invece rende i modelli NoSQL così adatti all'organizzazione di dati gerarchici risiede nella loro natura *schema-free*: tale proprietà consente di non definire a priori lo schema del modello, permettendo così di non modificarlo quando vi sia la necessità di aggiungere o aggiornare attributi (come invece avviene nei modelli relazionali, che richiedono di definire lo schema prima di procedere

all'inserimento dei dati). I modelli NoSQL di conseguenza possiedono uno schema che viene definito direttamente dall'inserimento dei dati.

**CAP Theorem** → definisce 3 proprietà che può possedere un sistema distribuito (il quale potrà possederne al massimo 2 simultaneamente):

- **Consistency**: tutti i nodi del sistema visualizzano i dati nello stesso istante
- **Availability**: ogni richiesta riceve una risposta, anche in caso fosse fallimentare.
- **Partition tolerance**: il sistema deve continuare ad operare, anche in caso una parte restituisca errori o perdite,

Solitamente i sistemi RDBMS sono di tipologia CA, mentre i modelli NoSQL possono essere CP (Dati coerenti ma il dbms non lavora 24/7) oppure AP (tuttavia i dati potrebbero a volte essere inconsistenti).




Un altra differenza tra i modelli relazionali e NoSQL risiede nelle differenti proprietà che devono essere verificate:

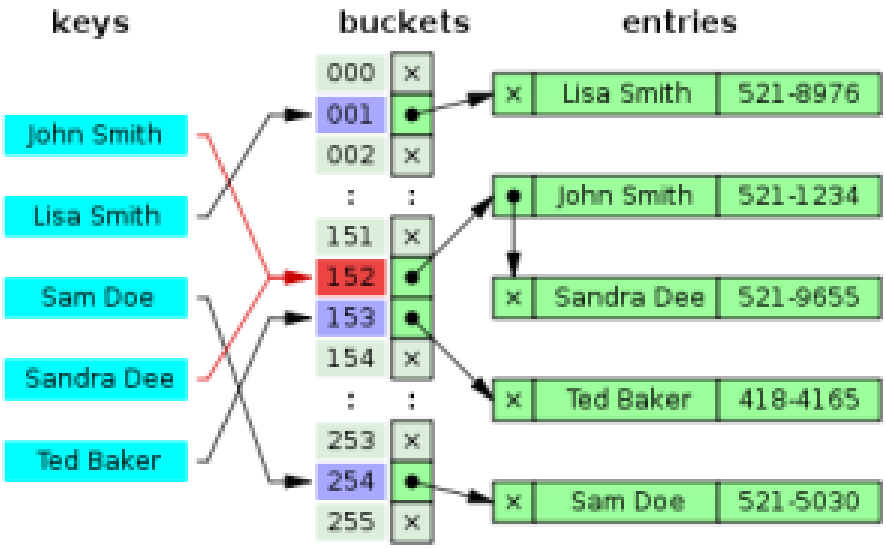
- Modelli relazionali → **ACID model**: devono sempre rispettare le proprietà di **Atomicity** (nelle transazioni vale il concetto *all or nothing*, e un fallimento in una transazione sui dati fa fallire l'intero processo), **Consistency** (se un elemento nella transazione rompe la consistenza, l'intera transazione fallisce), **Isolation** (ogni trasazione avviene prima o dopo le altre, nessuna di esse vede il prodotto intermedio delle altre) e **Durability** (le transazioni che hanno successo vengono memorizzate nel *transaction log*, e sono recuperabili in caso di fallimenti).

- Modelli NoSQL → **BASE model**: le proprietà ACID vengono rilassate in modo da adattarsi alla struttura dei modelli non relazionali. I modelli BASE posseggono le seguenti caratteristiche:
  - **Basic Availability**: grazie ad un approccio altamente distribuito (in cui non vi è un singolo *data store* ma diversi e con un alto tasso di replicabilità), è possibile gestire processi anche in presenza di numerosi fallimenti nelle transazioni (si supera il concetto *all or nothing*)
  - **Soft State**: rilassa il concetto di consistenza, dal momento che tale condizione è un problema dello sviluppatore, e non può essere gestita dal *database*
  - **Eventual Consistency**: riguardo alla consistenza si assume che, con il passare del tempo, i dati convergeranno alla consistenza.

## 2.2 Modelli non relazionali

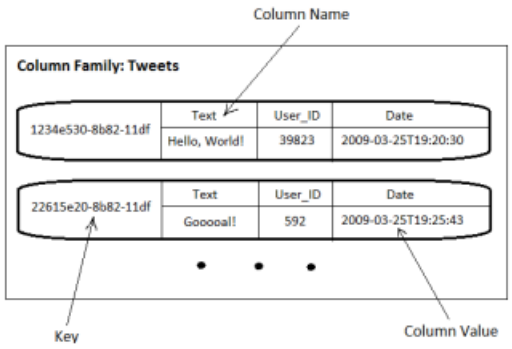
Tra i principali modelli non relazionali possiamo trovare:

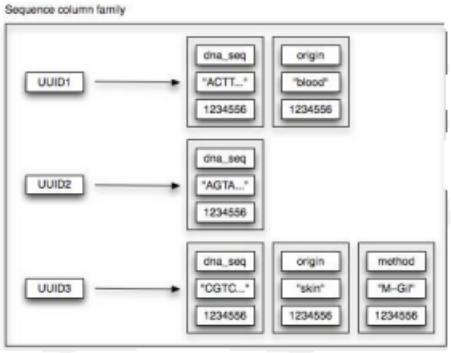
- **Key-Value stores** (Dynamo, Voldemort, Rhino DHT): sono database tabellari dove le chiavi si riferiscono a determinati valori, e la mappatura chiave-valore è basata sulle funzioni di crittografia di hash. A differenza delle tabelle relazionali le differenti righe sono indipendenti tra loro, e vi è la possibilità che abbiano attributi differenti (derivante dalla natura *schema-free*).



- **Column Family stores** (BigTable, Cassandra, HBase, Hadoop): le chiavi si riferiscono a determinati set di colonne, e sono molto utili per memorizzare *big data*.

| | | | | |
|---|---|---|---|---|
| **jim_87** | **Age** | **Name** | **Gender** | **Phone** |
| | 25 | Jim | M | 123456 |
| **jill_90** | **Age** | **Name** | **Gender** | **Phone** |
| | 22 | Jill | F | 654321 |




- **Document databases** (CouchDB, MongoDB, Lotus Notes, Redis): solitamente salvano in dati in formati JSON (*JavaScript Object Notation*), ovvero file basati su due strutture:una collezione di nomi (oggetti) e una lista ordinata di valori (*array*). I documenti vengono salvati univocamente attraverso una chiave (simile al concetto dei *column*), e sono rappresentabili da una struttura ad albero, con attributi nidificati (ovvero anche gli attributi possono essere a loro volta documenti). Sono dotati di schema flessibile, ovvero è possibile inserire nuovi campi (anche annidati) solo ad alcuni documenti (a differenza dello schema relazionale in cui bisogna aggiungere l'attributo ad ogni osservazione); se da un lato tale caratteristica risulta utile per rilassare la rigidità dei modelli relazionali, dall'altro lo schema variabile può risultare difficile da interpretare, in quanto non è sufficiente guardare alcuni documenti per conoscere lo schema degli altri (condizione che invece è garantita nei modelli relazionali).

- **Graph databases** (Neo4J, FlockDB, GraphBase, InfoGrip): sono *databases* organizzati in nodi e relazioni tra essi. I diversi nodi possono riferirsi a *entities*, le quali sono collegate con dati che si riferiscono a loro; non sono molto pratici quando si tratta di scalare *big data*.

- **RDF databases** e **Tuple stores**

### 2.2.1 MongoDB

E' un *document-based management system* che consente di memorizzare i dati in formato BSON (*binary* JSON, ovvero compilato), consentendo così di risparmiare notevolmente spazio, rendendolo più efficace. Volendo ridefinire gli elementi di uno

schema relazionale la tabella viene chiamata **collezione**, le righe **documenti** e le colonne **chiavi**. Possiede una sua sintassi, la quale differisce molto da quella SQL.

Viene spesso utilizzato come sistema di *storage* dei dati, come già detto in presenza di strutture gerarchiche, poiché consente di applicare procedure di cancellazione, eliminazione o aggiornamento degli attributi semplificandole molto. Un'altra importante funzionalità è il sistema di *query* interna a mongoDB, utilizzato per interrogare la base di dati oppure per applicare le procedure prima descritte; tale sistema viene inoltre reso maggiormente efficiente dalla possibilità di indicizzare le collezioni interne al database, riducendo l'ammontare di dati necessari al funzionamento delle *query* e snellendo considerevolmente le *query* stesse. L'insieme delle *query* applicabili al *database* viene denominato **CRUD operations** (*creation, read, update, remove*). Oltre alle operazione CRUD è possibile applicare tutte le operazione di *aggregation*, attraverso un *framework* relativamente complesso ma che risulta molto performante (operazioni matematiche sulle collezioni, raggruppamenti, etc).

### 2.2.2 Modelli a grafo

Con modello a grafo si intende uno schema di memorizzazione di dati caratterizzato da una struttura a rete, ovvero con nodi e archi; risulta concettualmente semplice da comprendere e con molte applicazioni reali, tra le quali spicca la modellazione di dati riferita ai *social network* (riescono a rappresentare le relazioni intrinseche del tema). Solitamente vi è la necessità di definire i *first class citizens* (ovvero le osservazioni di riferimento, nel caso dei *social* gli utenti), i quali vengono modellati.

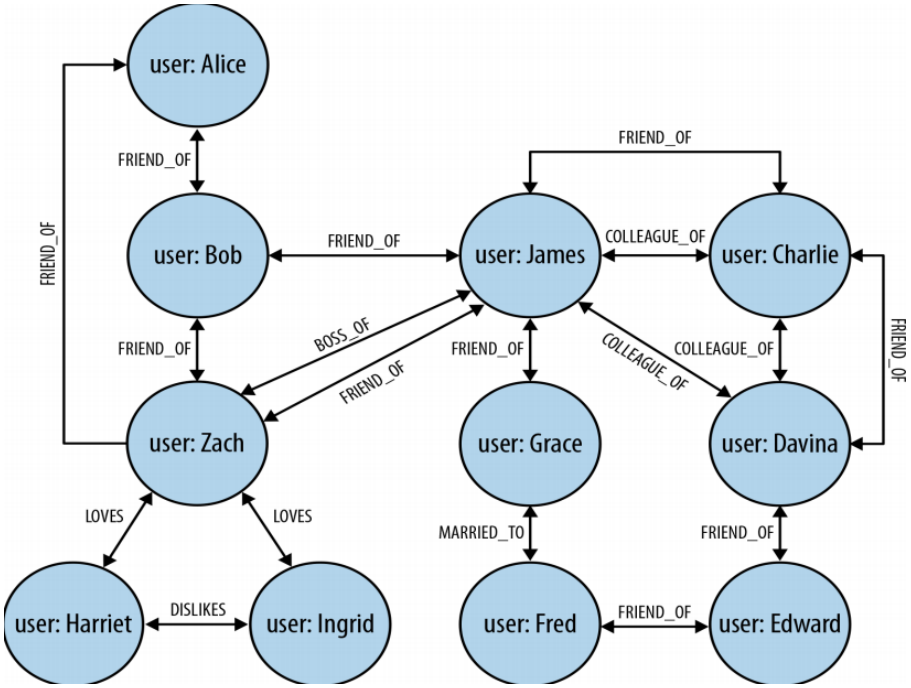
Nonostante le relazioni tra utenti all'interno delle piattaforme sociali siano tra i principali problemi trattati con strutture a grafo, vi sono molti altri campi di applicazione: lo *shortest path* dei sistemi di navigazione/geolocalizzazione e dei problemi logistici, individuazione di frodi nelle transazioni finanziarie, consigli di acquisto nelle piattaforme di *E-commerce* sulla base di altri utenti.

Il principale guadagno, in termini di onerosità computazionale, nel passaggio da un modello relazionale ad uno a grafo consiste nella riduzione del *join bombing*, ovvero il numero incredibile di operazioni di join da applicare quando vanno studiate relazioni di ordine elevato (gli amici degli amici...).

Le strutture a grafo a differenza di quelle documentali, rappresentabili con schemi ad albero, solitamente non risultano scalabili; tale criticità deriva dal fatto che una struttura ad albero consente di analizzare solo la parte d'interesse (due documenti possono essere considerati come componenti disgiunte), mentre in una struttura a grafo le relazioni tra nodi non consentono di applicare tali distribuzioni. Neo4j, uno dei principali dbms per la gestione di strutture a grafo, fornisce la possibilità di creare delle repliche del *database*, in maniera tale da svolgere interrogazioni in parallelo, ma non ha un sistema di scalabilità e quindi di distribuzione del *database* stesso.

Quando si modella una struttura a grafo vi è la possibilità di definire alcune proprietà riferite ai nodi, ottenendo in tal modo un **property graph**; il procedimento consiste nel caratterizzare nodi e archi tramite insiemi di coppie chiave-valore. Un esempio potrebbe essere le proprietà in riferimento ad alcuni studenti, i quali sono

nodi caratterizzati da alcune proprietà come matricola, anno di nascita, media dei voti, etc. Un esempio di *property graph* viene fornito nella seguente immagine.




La principale difficoltà aggiuntiva dei modelli a grafo, rispetto ai relazionali e ai documentali, consiste nella diversa metodologia di interrogazione della base di dati (derivante dal fatto che è una rappresentazione grafica). Vi sono alcuni linguaggi che risolvono tale problematica, per esempio Neo4j utilizza il linguaggio *cypher* (*graph-query language*), il quale è un linguaggio dichiarativo (simile a SQL per funzionamento), ovvero è un linguaggio *pattern-matching* (viene definito in maniera testuale il sotto-grafo che si vuole analizzare/visualizzare). Un altro linguaggio di interrogazione di strutture a grafo è Gremlin (*Graph-Trasfersal Language*); a differenza di cypher, tale linguaggio attraversa il grafo, in accordo con un determinato *pattern* richiesto, e ottiene come risultato un nodo; a differenza dei linguaggi dichiarativi è necessario specificare il percorso da compiere per arrivare al nodo desiderato (*vertex based*).

### 2.2.3 Modelli *key-value* e *wide-column*

- I modelli **Key-Value** sono tra le strutture NoSQL più semplici: si compongono di oggetti chiave-valore, i valori possono essere tipizzati (lista, stringa, *array*) ma non è possibile applicare interrogazioni. Nota la chiave è possibile applicare alcune operazioni come inserimento di una nuova coppia chiave valore, eliminazione, aggiornamento e identificazione. Sono invece ottimi dal punto di vista della scalabilità, poiché implementano una funzione matematica di Hashing (che consente di passare dallo spazio delle chiavi allo spazio dei valori di hashing, in maniera facilmente distribuibile). Alcuni prodotti di tipo *key-value* sono Redis, Memcached, Riak KV, etc.

- Alcuni esempi di prodotti che seguono lo schema **Wide-Column** sono *Big-Table* di Google e *Cassandra*. Tali modelli sono caratterizzati da righe che si strutturano come mappe multidimensionali ciascuna caratterizzata da una chiave, da una colonna e da un *timestamps*. Tra i principali sistemi di *storage* troviamo Hbase, il quale è ottimo dal punto di vista della scalabilità.

## 2.3 Integrazione e Arricchimento dei dati

Quando si tratta di unire due basi di dati provenienti da fonti diverse vi sono principalmente 2 possibilità:

- **Data Integration**: consiste nell'unione delle due basi di dati, seguendo il funzionamento di una *full-outer join* del SQL.
- **Data Enrichment**: consiste nell'arricchimento di una base de dati con alcune informazioni provenienti dall'altra; segue il funzionamento della *left-join* del SQL.

In pratica la differenza tra le due pratiche consiste nel fatto che nell'arricchimento dati viene definita una base di dati primaria, la quale andrà arricchita con l'inserimento di nuove informazioni dalla seconda base di dati, solamente per le osservazioni *matching*.

Molto spesso vi è la necessità di applicare integrazione/arricchimento tra due basi di dati che seguono uno schema differente (**eterogeneità nel modello** per esempio un SQL e un documentale); in tal caso è necessario decidere a priori quale sarà il formato della base di dati finali, al fine di definire come dovranno essere integrate/arricchite le basi di partenza. Vi possono essere numerosi conflitti legati all'eterogeneità delle basi di dati, riferiti principalmente a due situazioni: **eterogeneità nel nome e nel tipo**. Il primo è riferito alle situazioni in cui le entità/oggetti/attributi vengono etichettati con un nome differente (può essere di sinonimia $\rightarrow$ stesso concetto ma nome diverso, omonimia $\rightarrow$ concetti diversi ma stesso nome, iperonimie $\rightarrow$ presenza di concetti di livello più alto come persona-uomo), mentre il secondo riguarda la modellazione degli stessi. Nello specifico le eterogeneità di tipo conterranno i seguenti casi:

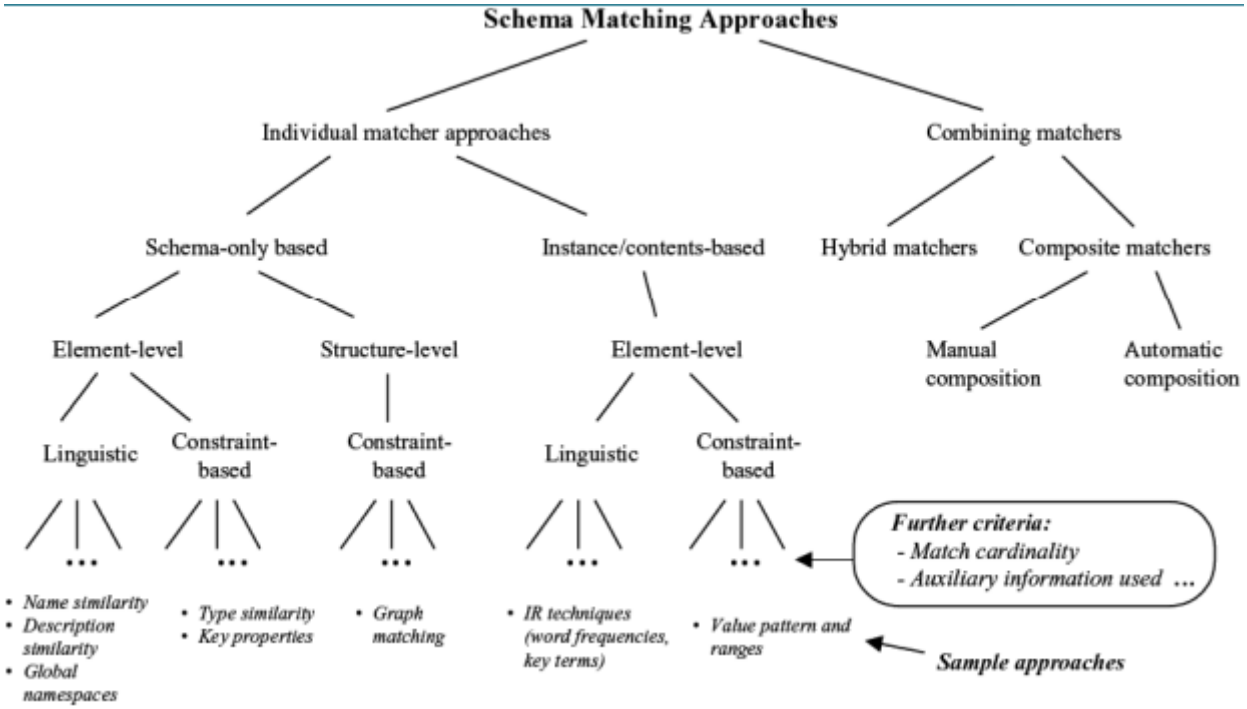
- Domini diversi per gli stessi attributi nei diversi schemi (attributo città, in uno schema si riferisce a città italiane nell'altro a quelle francesi).
- Un attributo in uno schema è derivato (dipendente), mentre nell'altro schema è indipendente.
- Lo stesso attributo viene definito come un'entità di secondo livello in uno schema (un arco, una *properties*, un grafo) mentre nell'altro schema come una di primo (un documento, un nodo, etc). Per esempio nel caso delle tasse universitarie i professori avranno uno schema semplice (pagate o no), mentre l'ufficio amminitrativo avrà uno schema più complesso (dichiarazione ISEE, nucleo familiare...).

- Uno stesso attributo potrebbe avere gerarchie di generalizzazione differenti a seconda dello schema considerato (in uno vi è persona, nell'altro uomo).
- Lo stesso attributo viene rappresentato come un nodo e come una relazione (arco) in due schemi (per esempio nelle pubblicazioni i coautori possono definirsi sia come nodi che come relazioni).
- Differenze di cardinalità o granularità
- Conflitti di chiave (per gli studenti in uno schema ho come chiave la matricola, nell'altro il codice fiscale).

Durante il **processo di data integration** andranno sviluppate diverse fasi:

1. Si scelgono le basi di dati iniziali, e si decide se applicare integrazione o arricchimento.
2. Si applica una trasformazione (*Schema transformation*) di schema per rendere i modelli omogenei tra di loro.
3. Si definiscono le regole di integrazione (*Schema matching*), finalizzate a definire la comunanza tra l'attributo $x$ dello schema A e l'attributo $y$ dello schema B.
4. Applicare l'integrazione (*Schemas integration*) vera e propria, ovvero la scrittura del codice che consente di applicare le regole precedentemente definite

Esistono una moltitudine di approcci per lo *schema matching*, i principali vengono riassunti nel seguente albero:



Fin'ora è stato considerato il caso in cui integrazione/arricchimento siano riferiti a solamente due schemi; in molti casi tuttavia tali procedure devono essere applicate ad **un numero maggiore di basi di dati**. Esistono diversi metodi per integrare un numero maggiore di 2 di schemi: il primo è sicuramente quello **a scala** (*composed*) il quale consiste nell'integrazione iterativa di coppie di schemi, dove ad ogni iterazione si integra lo schema precedentemente ottenuto con uno nuovo. Un altro metodo (che come quello a scala fa parte dei metodi binari) è quello **bilanciato** (*balanced*); tale metodo consiste nell'integrazione/arricchimento di più schemi contemporaneamente (poiché simili tra loro), con una conseguente progressiva integrazione degli schemi ottenuti. Vi sono poi i metodi n-ari, i quali possono essere **one-step** (integrazione in un colpo solo), o **iterativi** (funzionamento simili al bilanciato ma con più di 2 schemi integrati a ciascun livello).

Definito il metodo di *matching* e di integrazione è necessario risolvere gli eventuali conflitti, i quali possono essere classificativi, descrittivi, strutturali o di frammentazione. I conflitti più difficili da risolvere sono tuttavia gli *instance conflicts*, i quali possono assumere la forma di conflitti di chiave (osservazione identificata in maniera diversa, per esempio matricola-codice fiscale) o conflitti di valore (reddito di $x$ diverso tra le due basi di dati). Esistono diversi criteri per risolvere i conflitti di valore, i quali vanno selezionati in base al problema in analisi. Una possibilità è quella di assumere, sulla base di un certo criterio (dato più nuovo, banca dati più affidabile, valore medio, etc), un valore come corretto; chiaramente tale risoluzione può portare a distorsioni anche molto significative (sotto/sovrastima, etc).

Vi sono inoltre conflitti nei quali bisogna definire se due istanze, caratterizzate da una serie di attributi, facciano riferimento alla stessa entità (**record linkage**); una possibile soluzione consiste nel definire una **misura di distanza** (per esempio sintattica, Carlo e Crlo potrebbero riferirsi allo stesso nome) per determinare la probabilità che le due entità siano la stessa (minore sarà la distanza, maggiore sarà la probabilità). Solitamente vengono fissate due soglie di probabilità, una oltre il quale due istanze possono essere considerate uguali, l'altra invece sotto cui si esclude l'uguaglianza; vi è tuttavia un fascia centrale della distribuzione di probabilità nel quale non è possibile dire niente. La principale criticità delle misure di *matching* degli oggetti è la fortissima dipendenza dal dominio di riferimento (per esempio il cap in Italia avrà una forma differente da quello Giapponese, e il *matching* non funzionerà correttamente). Esistono forme di distanza più complesse, per esempio quella semantica, le quali richiedono tuttavia una conoscenza del dominio di riferimento molto più approfondita. Un'altra grande criticità del *record linkage* consiste nell'enorme numero di confronti da attuare quando si ha a che fare con basi di dati molto grandi; solitamente per ovviare a tale problematica si riduce lo spazio delle entità da analizzare (per esempio ordinandole e considerandone solo una parte). Possiamo riassumere i passaggi del *record linkage* con il seguente schema:

1. Normalizzazione dei formati (**preprocessing**); trasformare tutto in minuscolo, eliminare caratteri speciali e **blank**, etc.

2. Si riduce lo spazio di ricerca in blocchi più piccoli (**blocking**)

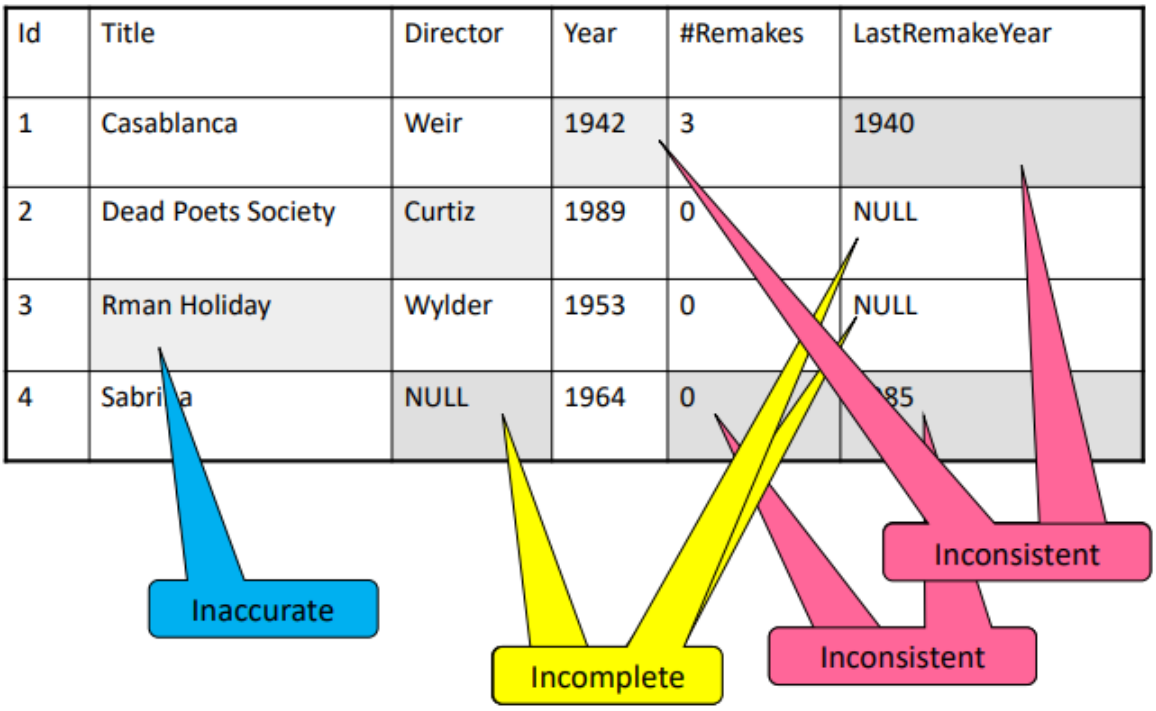
3. Viene definita la funzione di distanza (**compare**), la quale viene testata manualmente su un campione per verificarne l'efficacia.

4. Si decide se le entità corrispondono (**decide**).

## 2.4 Qualità dei Dati

Fornire una valutazione della qualità di un dato è un processo estremamente difficile, e lo è ancora di più se si considera il fatto che la stessa definizione di qualità, in riferimento ai dati, non è del tutto chiara. Una possibile definizione vede un "dato di qualità" come un dato utile al suo scopo: in quest'ottica non esiste una definizione universale di qualità, ma la si valuta sulla base del dominio e/o del suo utilizzo. Esistono inoltre diverse dimensioni attraverso il quale si può valutare la qualità, come per esempio l'accuratezza o la comprensibilità.

La valutazione della qualità di un set di dati, definita anche **assessment**, è una parte fondamentale del processo di analisi, e senza di essa i risultati ottenuti sono spesso errati. Spesso gli errori nei dati si presentano sotto forma di inconsistenze negli stessi (due dati in conflitto); la criticità principale è, in assenza di informazioni esterne aggiuntive, l'impossibilità di determinare quale dato sia veritiero. Tale procedura viene definita **improvement**, ovvero il miglioramento della qualità del dato attraverso una serie di correzioni di errori e inconsistenze, la quale tuttavia risulta spesso difficilmente applicabile, poiché potenzialmente richiede una moltitudine di informazioni aggiuntive.



Esistono un gran numero di **dimensioni** sotto cui valutare la qualità dei dati, le principali sono tuttavia **consistenza**, **completezza**, **accuratezza** e **dimensioni temporali** (Batini et al., 2009); un riassunto delle diverse dimensioni viene fornito nella seguente tabella:

| Cluster of Dimensions | Abstract Definition |
|---|---|
| **Accuracy, Correctness, Precision** | Proximity of data in representing a given reference |
| **Completeness, Pertinence** | Data represent all (and only) the aspects of the reality of interest. |
| **Minimality, Redundancy, Compactness** | Data represent all the aspects of the reality of interest only once and with the minimal use of resources. |
| **Consistency, Coherence** | Data comply to all the properties of their membership set (class, category,...) as well as to those of the sets of elements the reality of interest is in some relationship. |
| **Readability, Comprehensibility, Usability** | Data are easily perceivable and understood by users. |
| **Accessibility, Usability** | Data can be effectively "exploited" (consumed) by users. |
| **Currency, Volatility, Timeliness** | Data are up-to-date |

Definite le dimensioni sotto cui valutare la qualità dei dati è necessario valutare la metrica più adatta a quantificare i risultati; ne esistono di due tipi:

1. **Metriche Oggettive**: sono formali, precise e sono indipendenti dalla percezione/giudizio umani. Utilizzando funzioni matematiche, metodi statistici o procedure informatiche danno lo stesso risultato indipendentemente da chi le applica.
2. **Metriche Soggettive**: a differenze delle precedenti sono in gran parte basate su percezione ed esperienza umane. Restituiscono risultati in forma scala ordinale (scarsa, buona, ottima, etc ), differenti in base alla persona che le applica.

### 2.4.1 Dimensioni della qualità

Prendendo in analisi una delle dimensioni principalmente utilizzate, ovvero l'**accuratezza** (vicinanza di un valore $v$ ad uno $v'$, considerato rappresentazione corretta), possiamo distinguerla in due tipologie: accuratezza **sintattica** e **semantica**. Tipicamente la prima è più facile da valutare, impiegando alcuni set di dati contenenti i veri valori (elenco nomi italiani, elenco comuni, etc) e confrontandoli con il valore in analisi. Quando si parla di metriche per l'*assessment* di attributi stringa vengono utilizzate le **Edit distances**, ovvero date dalla somma di numero di inserimenti, di cancellazioni e di sostituzioni di simboli necessari a passare dal valore attuale $v$ al valore possibilmente corretto $v'$; è possibile fornirle anche in forma normalizzata (1-ED(v,v')/n, con $n$ = numero di caratteri nel dominio di $v$ e $v'$).

Tali metriche sono funzionali in presenza di stringhe piccole; in presenza di stringhe più ampie è consigliabile utilizzare porcedure di *assessment/improvement* basate su *token*.

Per quanto concerne la **completezza**, tale proprietà è stata principalmente pensata per il caso di modelli relazionali; è tuttavia applicabile anche al caso non relazionale. Una possibile definizione è la copertura con il quale il fenomeno osservato viene rappresentato nell'insieme dei dati; un esempio possibile é la quantità di impiegati rappresentati in una banca dati rispetto al totale degli impiegati dell'azienda. Ne esistono diversi tipi, come ad esempio la completezza di tupla (riga), di attributo (colonna) o di tabella (numero di valori non nulli su ciascuno di tali elementi). Solitamente una metrica di completezza utilizzata è il numero di valori nulli diviso la numerosità della riga, colonna o tabella.

La dimensione di qualità che si riferisce alle **proprietà temporali**, può essere valutata secondo 2 aspetti:

- Il **livello di aggiornamento** (*currency*): si riferisce al livello al quale i dati sono aggiornati rispetto al mondo reale. Una possibile misura è il ritardo temporale tra il tempo $t_1$ cui si riferisce il dato, e il tempo $t_2$ in cui si trova il fenomeno reale.
- La **tempestività**: si riferisce al livello al quali i dati vengono aggiornati, rispetto ad un determinato processo che li utilizza (orario delle lezioni dei corsi devono essere disponibili prima dell'inizio degli stessi). A differenza della *currency* è completamente dipendente dal processo cui si riferisce, e si valuta in relazione al momento temporale in cui dovrà essere disponibile per il processo. Esiste un possibile *trade-off* con la completezza e, a seconda dell'ambito di applicazione, bisogna valutare se sia meglio prediligere una proprietà rispetto ad un'altra.

Quando si ha a che fare con la dimensione temporale è importante definire uno storico delle osservazioni, ovvero applicare un *record linkage*, in maniera tale da riconoscere determinate osservazioni nei diversi periodi di cui si hanno i dati.

Una delle dimensioni della qualità più facile da verificare (ma non necessariamente da correggere) è la **consistenza**; ne esistono principalmente due tipologie:

- Consistenza dei dati rispetto a **vincoli di integrità** forniti in uno schema (il CAP deve essere consistente con la città di riferimento).
- Consistenza delle caratteristiche di uno stesso oggetto nelle diverse rappresentazioni della realtà racchiuse nella base di dati (indirizzo deve essere rappresentato nello stesso formato per tutte le basi di dati in cui viene definito).

Un caso tipico di inconsistenze nei dati, dovute anche alla dimensione temporale, riguarda il codice fiscale: tale codice è infatti legato al luogo di nascita, alla propria provenienza (il codice fiscale italiano è pensato per nomi italiani), e possono esservi inconsistenze in caso di nomi stranieri, comuni non più esistenti, etc. Tipicamente i **vincoli di consistenza** (*business rules*) possono includere:

- Un singolo attributo (i valori di $x$ devono rientrare in un dominio D).

- Più di un attributo (CAP e città non devono essere in conflitto).
- Attributi in più relazioni (come i vincoli di integrità referenziale).
- Espressioni in termini probabilistici (la probabilità che uno abbia 110 anni è molto bassa).

In fase di *assessment* è buona norma valutare se i dati rispettano tali proprietà di consistenza.

Esistono molte altre dimensioni secondarie come l'accessibilità (culturale, stato fisico e/o tecnologica), la quale misura la capacità degli utenti di accedere ad un dato.

### 2.4.2 *Quality assessment and improvement*

Poiché esistono diverse dimensioni della qualità del dato, più o meno utili a seconda del dominio di applicazione, esistono anche diverse metriche; tipicamente vi è la possibilità di fornire una misura aggregata (che tenga conto delle diverse dimensioni e metriche), per esempio utilizzando una media pesata. Se la parte che riguarda l'*assessment*, una volta definite dimensioni, metriche e e misure aggregate, può risultare relativamente semplice, la parte che riguarda l'**improvement** è leggermente più complicata; possiamo distinguere tra due vie principali:

- **Data-driven**: sono basate su confronti interni ai dati; tipicamente sono più semplici e hanno la possibilità di essere riproducibili in caso di correzioni ripetitive. Possono assumere la forma di *acquisition of new data* (inserire nuovi dati che permettano di sostituire quelli considerati non di qualità), *record linkage* (confrontare i set di dati "sporchi" con set considerati di qualità) e/o *source trustwothiness* (selezionare le fonti sulla base della qualità dei loro dati).

- **Process-driven**: si concentrano sul trovare la maniera ottimale di registrare e modificare i dati. Sono applicabili sotto forma di *process control* (si prova a controllare il processo nelle sue fasi di creazione e aggiornamento, analizzando standard di qualità) e/o *process redesign* (si cambia il modo in cui i dati vengono raccolti/costruiti/aggiornati).

Tipicamente i primi sono convenienti nel breve periodo, in quanto necessitano di un minor dispendio in termini di tempo e risorse, mentre i secondi risultano particolarmente efficienti per risolvere i problemi strutturalmente, e quindi nel medio/lungo periodo.

Le tecniche di *improvement* dipendono fortemente dalla dimensione della qualità che si considera:

- Per l'accuratezza ha senso utilizzare dei valori di riferimento per il dominio considerato.

- Per la completezza si utilizzano tecniche di *missing replacement* o *deletion*.

- Per la consistenza si utilizzano i vincoli di integrità e le dipendenze funzionali derivate dai dati.

# Volume

## 3.1 *Data Distribution*

Una componente essenziale di una struttura capace di gestire grandi (e crescenti) quantità di dati è la distribuzione dei *database*; prendendo come esempio un azienda che ha 3 uffici distanti (Londra, New York, Hong Kong) e che ha la necessità di registrare le informazioni afferenti ai suoi impiegati, tipicamente si avrà che ogni sede registra all'interno del proprio *database* i dati degli impiegati della sede stessa. La prima questione che si pone deriva dalla necessità di avere i dati della propria sede disponibili velocemente e al tempo stesso avere accesso ai dati delle altre sedi. La seconda questione invece riguarda invece gli eventuali guasti del sistema di uno degli uffici; in tal caso è buona pratica avere a disposizione delle repliche dei dati di un azienda all'interno del sistema delle altre, così da aumentare la disponibilità dei dati stessi. Quello che emerge da tale esempio sono i concetti cardine della *data distribution*: la **distribuzione e la replicazione dei dati**. L'idea che sta alla base di tali procedimenti è l'ottenimento di una struttura di gestione dei dati frammentata, più facile da gestire e meno penalizzata da eventuali guasti, piuttosto che un unica struttura centralizzata, molto più costosa sopratutto in caso di *crash* del sistema.

Quando si distribuisce un *database* bisogna compiere scelte circa l'omogeneità che si vuole ottenere; tipicamente le aziende cercano di ottenere degli *homogeneous distributed databases*, ovvero che utilizzano lo stesso *software* di gestione su ciascun nodo, vi è cooperazione nei processi di richiesta da parte degli utenti, i quali ne usufruiscono sotto forma di un unico sistema. Vi è ovviamente la possibilità di sviluppare dei sistemi eterogenei (opzione che si verifica spesso in caso di acquisizioni e fusioni). Le scelte che bisogna operare quando si decide di distribuire la propria base di dati riguardano quindi la **frammentazione**, la **replicazione**, l'**allocazione** e l'**omogeneità**.

Per quanto concerne la replicazione dei dati, esistono sia vantaggi che svantaggi:

- **Vantaggi**: i principali sono la maggiore disponibilità dei dati (in particolar modo in caso di fallimenti del sistema), la parallelizzazione delle interrogazioni (avendo due repliche si può interrogare metà da una e metà dall'altra) e la riduzione del trasferimento dei dati (le diverse sedi avranno a disposizione i dati delle altre localmente).

- **Svantaggi**: vi saranno costi più elevati in caso di aggiornamento (andranno aggiornate tutte le repliche), complessità maggiore per ciò che riguarda la gestione combinata (una scarsa coordinazione può condurre a inconsistenze in caso di aggiornamento delle repliche). Per ovviare a tali problematiche è possibile creare una *primary copy*, sulla quale applicare tutte le operazioni di aggiornamento (evitando in tal modo le inconsistenze).

Assieme alla replicazione della base di dati si affianca solitamente la frammentazione della stessa, la quale può essere orizzontale (suddivisione delle tuple in uno o più frammenti) o verticale (suddivisione delle colonne, ovvero del database in schemi più piccoli che condividono una superchiave primaria); tali frammentazioni sono ovviamente applicabili sia al caso di modelli relazionali, sia a quello dei modelli non relazionali.

Tipicamente un sistema, nonostante la sua natura distribuita, rispetta la proprietà di **Data transparency**: l'utilizzatore finale della struttura non si accorge della distribuzione dello stesso, né della sua replicazione, né della sua allocazione. Tale proprietà da un lato consente a chi opera le analisi di non preoccuparsi della struttura dietro il sistema, dall'altro può generare problemi causati da tale ignoranza (in particolare problemi di consistenza).

I sistemi distribuiti rispettano alcune proprietà:

- Ogni oggetto dev'essere assegnato ad un nodo e deve avere un nome unico.
- Dev'essere possibile risalire alla localizzazione di ciascun oggetto in maniera efficiente.
- Dev'essere possibile cambiare la localizzazione di ciascun oggetto.
- Ogni nodo deve poter creare e aggiungere oggetti autonomamente.

Esistono principalmente due strutture di sistemi distribuiti: la prima è il sistema centralizzato (**centralized scheme**, soddisfa i criteri 1-2-3 ma non il 4), ovvero in cui è presente un "capo" (*name server*) che conosce tutto del sistema distribuito (divisione in nodi, localizzazione, impiego,...), e una serie di componenti *software* più piccole le quali memorizzano porzioni di dati, eseguono semplici interrogazioni. La principale criticità di tale organizzazione è la natura di collo di bottiglia del *name server*, sia perché tutte le componenti devono fare le richieste ad esso, sia per i grossi danni in caso del suo fallimento. La seconda architettura, denominata **Distributed Query Processing** (architettura tra pari) si ottiene eliminando il *name server*; non esistendo un "capo" l'organizzazione del sistema può risultare molto complicata (dove si trovano i nodi e gli oggetti, quali sono i funzionamenti di ciascun nodo, ... non sono più conosciuti dalla componente centrale), tuttavia è molto meno soggetto ai problemi derivanti dalla natura di collo di bottiglia del *name server*. Tali sistemi garantiscono inoltre la tolleranza al partizionamento.

Qualunque schema si decida di utilizzare i **vantaggi di una struttura distribuita** (DDBMS) sono i seguenti:

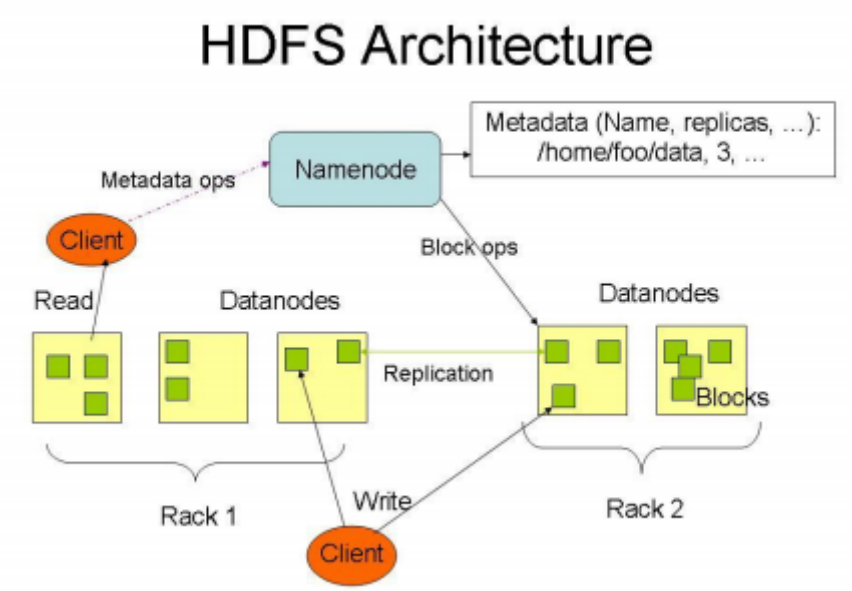
- Accessibilità a dati che sono fisicamente separati.

- Vengono migliorate la condivisione dei dati nel rispetto delle autonomie locali.
- Viene migliorata la disponibilità dei dati.
- Si migliorano l'affidabilità e la performance del sistema.
- Si ha un notevole risparmio rispetto ad un sistema non distribuito, poiché la scalabilità orizzontale prevede l'utilizzo di molte componenti *software* poco costose.
- Viene garantita la crescita modulare.

Vi sono tuttavia alcune **criticità in un sistema DDBMS**, come ad esempio la complessità, la sicurezza , eventuali costi di ottimizzazione possono essere molto elevati, difficoltà nel controllo dell'integrità del sistema, problemi di standard e difficoltà nella strutturazione del *database*.

Una delle principali problematiche di avere grandi quantità di dati organizzate in file è trovare una struttura di *file system* capace di memorizzarli e gestirli. In ottica di *data distribution* anche il sistema di gestione dei file viene distribuito, in maniera che ciascun *file system* debba gestire solo una porzione del file; tale logica sta dietro il sistema definito **Hadoop Distributed File System** (HDFS). HDFS serve di conseguenza a gestire in maniera distribuita un singolo file, spezzandolo in componenti più piccole. Esistono due componenti in un'architettura HDFS:

- **Name node**: responsabile di gestire la logica dell'organizzazione del file, conosce in quale nodo sono posizionati i file; è responsabile di fornire una risposta sulla localizzazione dei file al *client* quando viene fatta un'interrogazione. Tipicamente è presente anche un *name node* secondario, copia del primario, il cui scopo è sopperire ad eventuali fallimenti del principale.
- **Data nodes**: componente che contiene il dato, la porzione di file da gestire; sono i nodi a cui accedono i *client* dopo aver interrogato il *name node* e non conoscono la posizione degli altri file.



HDFS è un sistema che gestisce file con la logica *write one, read many*, ovvero che vengono inseriti un'unica volta e disponibili per l'analisi molte volte. Solitamente i blocchi di file hanno una dimensione di 128 mb, e ciascuno viene replicato su diversi *data nodes* (successivamente localizzabili dal *client* attraverso il *name nodes*).

Il motore di calcolo di HDFS viene definito **Map Reduce**, e il nome porta già in sé le due fasi:

- **Fase di Map**: si esegue lo stesso programma su un set molto ampio di dati, si estraggono i risultati interessanti i quali vengono ordinati e uniti.
- **Fase di Reduce**: aggregazione dei risultati intermedi e generazione dei risultati finali (i quali vengono immediatamente trasferiti su disco).

Il funzionamento è semplice: dato un programma, il quale ha lo scopo di essere applicato ai dati, esso viene copiato ed applicato su ogni partizione (fase di partizionamento) del file e infine si aggregano i risultati (fase di combinazione). Lo scopo di tale procedura è quello di "avvicinare" il programma ai file che deve gestire. **Hadoop** non è altro che un sistema HDFS il quale utilizza un sistema di calcolo *Map Reduce*.

## 3.2 *Data Warehouse*

La *data warehouse* è una tecnologia nata alla fine degli anni '80; la procedura per crearne uno passa per la raccolta dei dati, l'integrazione, la pulizia e infine la messa a disposizione della base di dati. Il *data warehouse* nasce per il mondo relazionale, e ancora oggi per la maggior parte viene applicato secondo tale logica. L'idea che sta alla base nasce dall'esigenza di avere a disposizione un accesso unificato che consenta di accedere ai diversi file; anche la *data integration* nasce per rispondere a tale esigenza, tuttavia lo fa in maniera "leggera" (*lazy*, parla più al *business* che alla struttura dati vera e propria). La principale criticità dell'approccio *lazy* è l'assenza di uno storico dei dati, e quindi l'impossibilità di fare interrogazioni di lungo periodo.

La *data warehouse* cerca di andare oltre la semplicità della *data integration*; l'idea di partenza è sempre l'itegrazione della base dei dati, ma applica 2 **principi fondamentali**:

- Salvare fisicamente i dati da un'altra parte (ovvero il *data warehouse* $\rightarrow$ magazzino dei dati).
- Riorganizzarli in maniera più precisa.

Le **fasi** di costruzione di un *data warehouse* sono particolarmente lunghe e complesse; le principali sono tre:

- **Costruzione**: può essere molto lunga poiché prevede la cooperazione di diverse aree operative. Tale complessità può portare a tempistiche molto estese, a volte di anni; è chiaro che un sistema progettato oggi e utilizzato tra un anno non sarà necessariamente in grado di rispondere alle esigenze che si avranno. Da questo punto di vista può risultare uno schema estremamente rigido.

- **Verifica della Qualità**: l'integrazione dei dati può portare a gravi problemi di qualità degli stessi.
- **Manutenzione**: il DW potrebbe richiedere delle parziali riprogettazioni in caso di aggiunte di nuove sorgenti dati o di nuove esigenze di interrogazione.

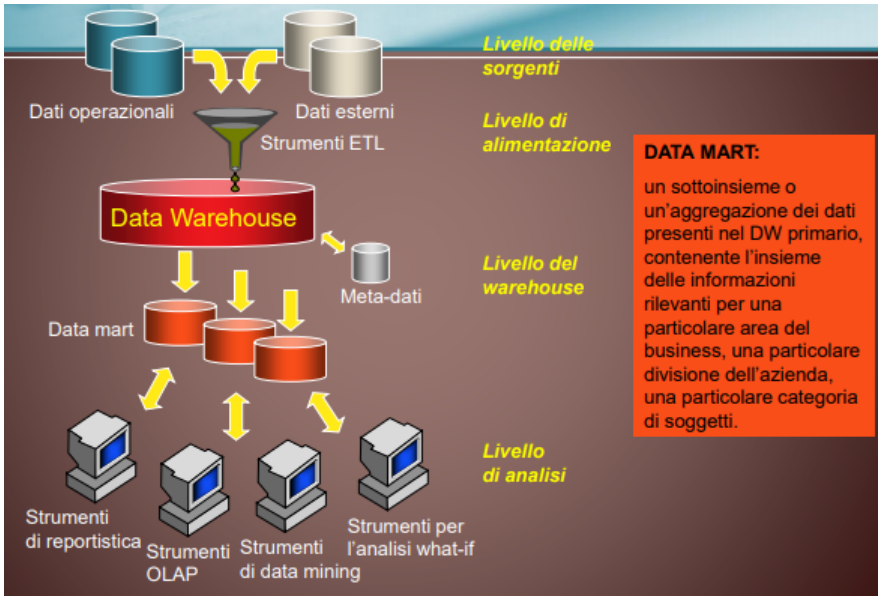
Se le fasi di progettazione e verifica del DW sono estremamente lunghe e costose, i **vantaggi** possono essere significativi (ecco perché ad oggi lo utilizzano molte aziende):

- **Query veloci**, tenendo tuttavia conto della possibilità che i dati non siano sempre aggiornati.
- **No interferenze con il carico dei dati**, il che consente di fare interrogazioni complesse senza sovraccaricare i sistemi operazionali.
- **Informazione memorizzata sul DW**, quindi possibilità di interrogare lo storico dei dati, ristrutturare/modificare/migliorare l'informazione e consente un maggiore controllo della sicurezza.

Un DW può essere quindi considerato un sistema di memorizzazione dei dati semplice, unico, completo e consistente ottenuto attraverso differenti risorse e reso disponibile all'utilizzatore finale in maniera da essere facilmente utilizzabile e comprensibile (B. Devlin, consulente IBM). Le principali proprietà sono quindi la sua orientazione ad un determinato argomento, integrazione, storicizzazione e persistenza del dato (W.H Inmon, 1992).

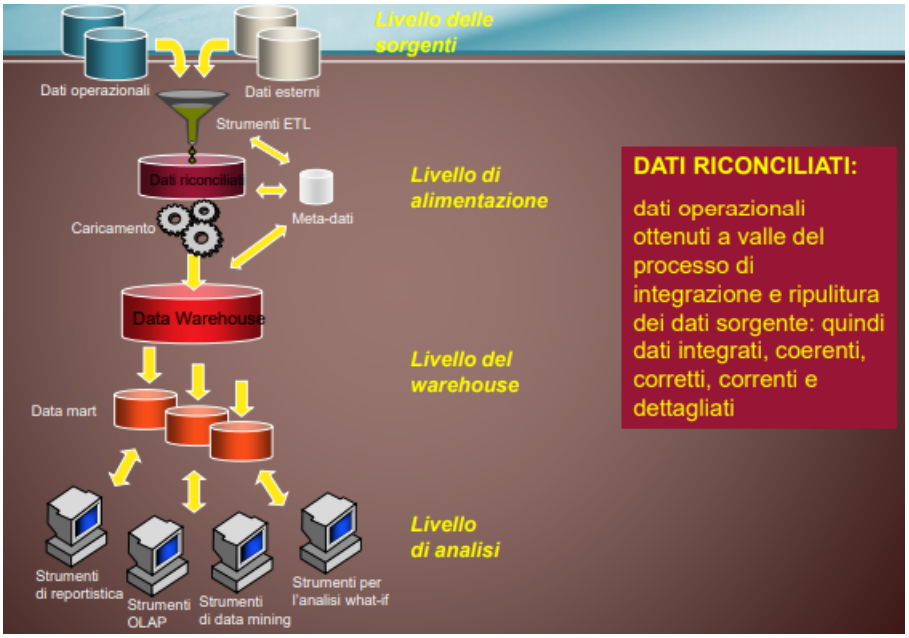
Negli anni si sono sviluppate alcune **possibili soluzioni** per la strutturazione di un DW, caratterizzate dai livelli della struttura del DW stesso:

- **Architettura a 2 Livelli**: si prendono i dati operazionali, si applicano delle trasformazioni con strumenti ETL, si costruisce un grande DW, vengono creati una serie di *data mart* (frammenti del DW specifiche su un dominio) i quali verranno interrogati dai diversi reparti (difficilmente applicabile per aziende grandi)



I principali **vantaggi** di una struttura a 2 livelli sono la continua disponibilità di informazioni di buona qualità (anche in caso di temporanea sospensione per motivi tecnici), l'interrogazione sul DW non interferisce con la gestione delle transazioni (essenziale per il funzionamento dell'azienda), l'organizzazione logica è basata sul modello multidimensionale (e non relazionale o semi-strutturato come nel caso delle sorgenti) e infine consente di impiegare specifiche tecniche per l'ottimizzazione delle prestazioni in termini di analisi e reportistica. La principale **criticità** è la quasi impossibilità della creazione di un unico grande sistema dove mettere tutti i dati dell'azienda (causa dei vincoli di costo e tempo).

- **Architettura 3 livelli**: è una variante che prevede un passaggio intermedio (*operational data store*) prima del DW, il cui impiego è quello di riorganizzare i dati operazionali in entrata e passarli gradualmente al DW. Rimangono tuttavia le difficoltà legata ai costi di tempo e denaro.




- **Architettura con data mart indipendenti**: viene utilizzata dalla maggior parte delle aziende; ogni organizzazione sviluppa il suo *data mart* in maniera indipendente. Consente di risparmiare tempo e di svolgere i progetti in tempistiche accettabili dal mercato (difficilmente rispettabili con gli oneri tempistici di un DW centrale). Si rinuncia ad avere una *single version of truth* (con rischio di incoerenze e/o inconsistenze), tuttavia risulta più veloce e maggiormente scalabile.

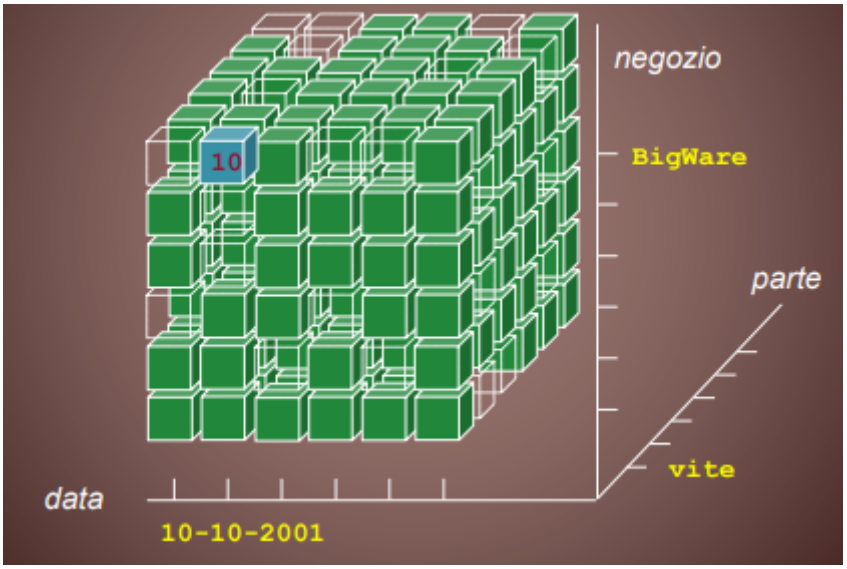
- **Architettura con data mart bus**: il funzionamento è simile al precedente, tuttavia i diversi *data mart* vengono integrati logicamente con dimensioni comuni (ciò riduce le inconsistenze). Spesso utilizzata quando vi è la fusione di due aziende poiché l'integrazione logica risulta meno onerosa di un'integrazione fisica.

Il DW quindi lo scopo di arricchire ed aggregare le informazioni contenute nei dati operazionali, e di conseguenza consente, seguendo una logica di interrogazione relazionale, di ottenere dei report generali, utili a rispondere alle tipiche domande di *business* (*business-oriented*). Esistono diversi metodi per analizzare i dati del DW, supportati da altrettanti strumenti: la **reportistica** (documenti diffusi periodicamente contenenti i dati aggregati), **OLAP** (*OnLine Analytical Processing* (richiedono all'utente di ragionare in modo multidimensionale, accedendo ad un'interfaccia grafica) e infine il **data mining** (richiede all'utente conoscenza dei principi alla base degli strumenti utilizzati).

La modalità OLAP è tra le principali tecniche impiegate per la fruizione dei contenuti del DW, poiché consente all'utente di analizzare i dati quando le necessità non siano facilmente identificabili a priori (e quindi di esplorarli). Il tutto si fonda sul **modello multidimensionale**, e l'utente deve conoscere i metodi di analisi costruendo una sessione di lavoro. Le **sessioni OLAP** consistono in un percorso di navigazione che riflette il procedimento di analisi; le interrogazioni della sessione sono di tipo multidimensionale, e si basano su 3 concetti (cubo multidimensionale):

- **La misura** (cella) secondo cui voglio misurare il fatto d'interesse.
- **La dimensione** (asse) dell'analisi.
- **La gerarchia di attributi** utilizzata per aggregare i dati memorizzati nei cubi base

Un esempio di rappresentazione multidimensionale potrebbe essere il seguente cubo delle vendite.



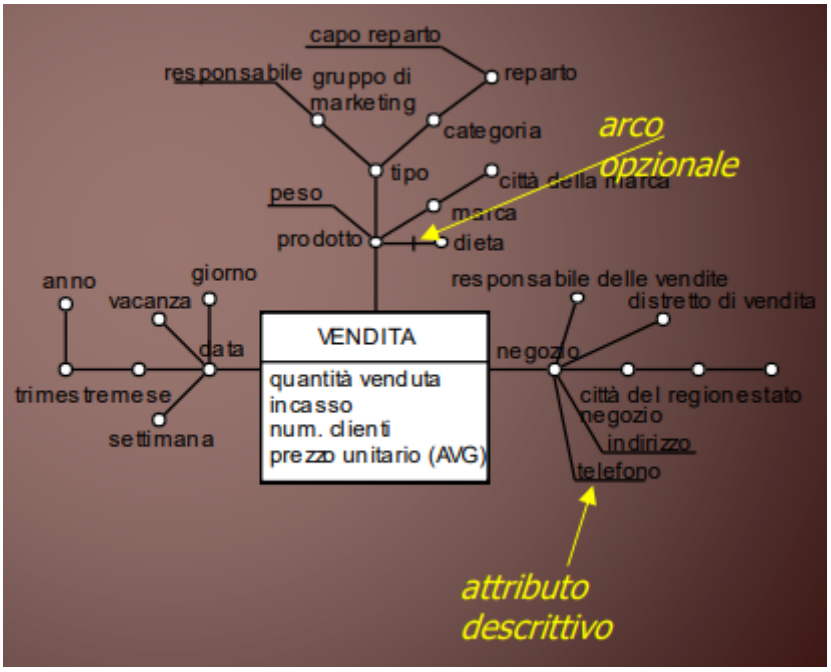

Una volta che si ha a disposizione il cubo è possibile applicare delle operazioni di **Slicing and Dicing**, ovvero prendere le porzioni di cubo che servono all'analisi. Le gerarchie sono una parte fondamentale dell'analisi OLAP, poiché consentono di aggregare gli attributi; per esempio data una serie di prodotti posso aggregarli in tipologie di prodotti (detersivi, latticini, carne, etc) e ancora in settori merceologici (alimentare, non alimentare), oppure per i negozi che possono essere aggregati per

comune, provincia o regione. Date le gerarchie le principali **operazioni OLAP** sono le aggregazioni (**roll-up**) le quali hanno lo scopo di aumentare il livello di aggregazione eliminando uno o più livelli di di dettaglio da una gerarchia (elimino i negozi e mi concentro sull'aggregato comuni). L'operazione opposta viene definita **drill-down**, e consiste nella riduzione dell'aggregazione introducendo uno o più livelli di dettaglio. Vi è inoltre un'ulteriore tipologia di operazione, definita **pivoting**, la quale comporta un cambiamento nella modalità di presentazione, con l'obbiettivo di analizzare le stesse informazioni sotto punti di vista diversi. Un'ultima operazione, utile quando vi è la necessità di unire due cubi, è il **drill-across**, che consiste nello stabilire un collegamento tra due o più cubi al fine di compararne i dati.

Il **Dimensional Fact Model** (DFM) è un modello concettuale grafico per *data mart*, il quale ha lo scopo di :

- Supportare in maniera efficace il progetto concettuale.
- Creare un ambiente su cui formulare le interrogazioni.
- Creare una piattaforma stabile da cui partire per le analisi.
- La rappresentazione generata consiste in un insieme di **schemi di fatto**, i quali modellano i fatti (concetto d'interesse per il processo decisionale, modella gli eventi, evolve nel tempo), le misure (proprietà numerica di un fatto che ne descrive un aspetto quantitativo), le dimensioni (proprietà con dominio finito di un fatto che ne descrive una coordinata di analisi) e le gerarchie (albero direzionato i cui nodi sono gli attributi dimensionali e i cui archi sono relazioni *maty-to-one*, subiscono variazioni con il tempo a causa delle aggiunte di attributi e oggetti).

La seguente immagine mostra come si potrebbe presentare un DFM in caso di un problema di vendita:

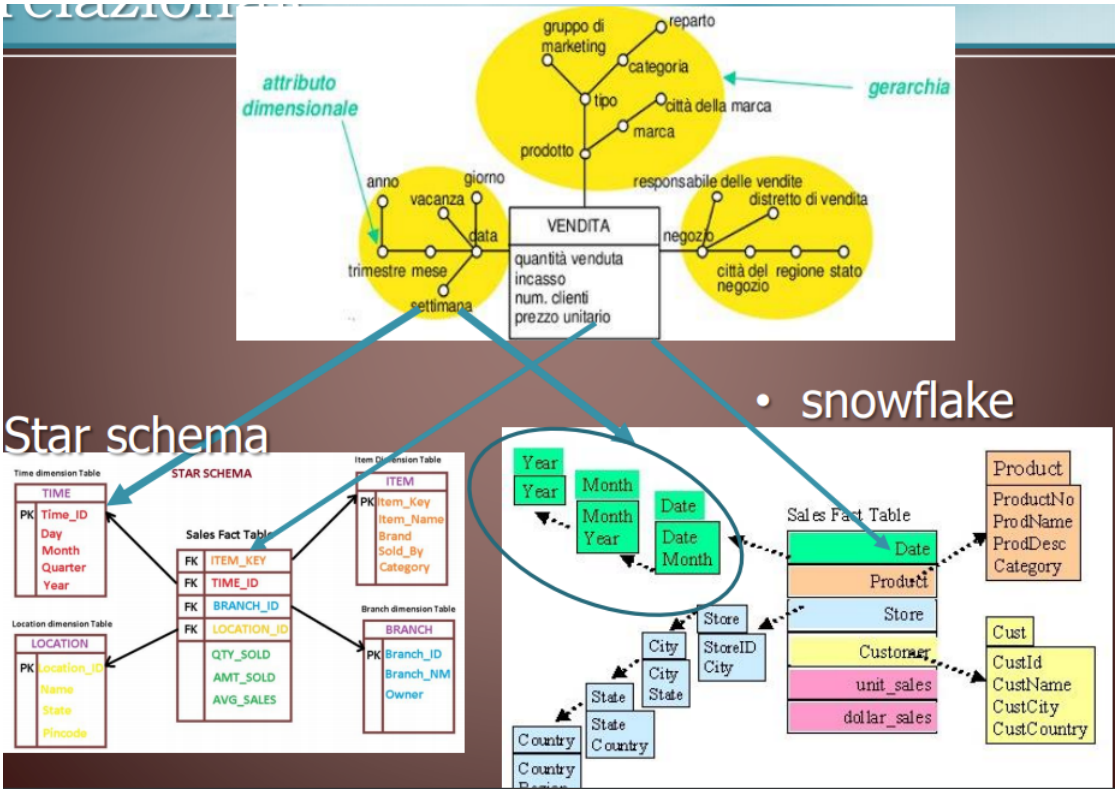


Quando si ha a che fare con problemi complessi vi è la necessità di utilizzare **costrutti** altrettanto **complessi**, come per esempio:

- **Attributi descrittivi**: contengono informazioni su un attributo dimensionale di una gerarchia; non si utilizzano per aggregazioni poiché contengono valori continui e derivano da associazioni *one-to-one*.
- **Archi opzionali**
- **Attributi Cross-Dimensionali**: attributo dimensionale o descrittivo il cui valore è determinato dalla combinazione di più attributi, anche appartenenti a gerarchie differenti.
- **Convergenze**: si ha quando due attributi sono connessi da più cammini direzionati e distinti; tali cammini rappresentano una dipendenza funzionale
- **Gerarchia condivisa**: una porzione di gerarchia viene ripetuta più volte nello schema.
- **Archi Multipli**: modellano associazioni molti a molti tra due attributi dimensionali.
- **Gerarchie incomplete**: per alcune istanze risultano assenti uno o più livelli (in quanto non noti né definiti).
- **Addittività**: si ha quando si legano due attributi attraverso archi addittivi; molto rischiosa in quanto può condurre a gravi errori.
- **Sovrapposizione di schemi**: si ha quando si applicano operazioni di *drill-across*. Le misure rappresentano l'unione delle misure presenti negli schemi originali, le gerarchie includono gli attrubuti presenti in entrambe le gerarchie e il dominio di ogni attributo sarà l'intersezione dei domini corrispondenti negli schemi originali.

Spesso si ha la necessità di accedere al DW (mediante operazioni OLAP) non attraverso il "cruscotto di analisi", ma attraverso lo schema logico, ovvero le tabelle relazionali; l'idea alla base è il passaggio da uno schema concettuale (di alto livello) ad uno logico. Il principale scoglio di tale procedura consiste nell'onerosità computazionale dovuta alla ridondanza dei dati (derivanti dalle operazioni SQL di *join*). Le principali operazioni da svolgere in fase di **progettazione logica** sono:

1. Scelta dello schema da utilizzare (*snowflake*, a stella, ...).
2. Traduzione degli schemi concettuali.
3. Scelta delle gerarchie dinamiche (per gestire i cambiamenti nel tempo).
4. Scelta delle viste da materializzare.
5. Applicazione di altre forme di ottimizzazione come frammentazione verticale o orizzontale.




Nel modello **star** i fatti diventano una tabella, le misure dei fatti diventano gli attributi dello schema logico e infine si aggiungeranno tante chiavi quante sono le dimensioni; è la maniera più semplice per operare il passaggio, nonostante il consistente numero di replicazioni. Il secondo modello, leggermente più complesso, è la rappresentazione a **snowflake**, il quale cerca di limitare la ridondanza degli attributi; spesso si hanno tabelle con un unico attributo identificati da una chiave che si trova nella gerarchia di livello più alto. Solitamente lo *snowflaking* viene considerato meno conveniente dello *star* per i costi maggiori in termini di tempo e complessità; risulta tuttavia utile quando vi sono gerarchie condivise.

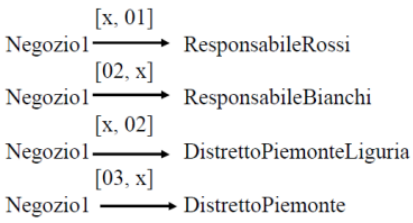
Nel passaggio a schema logico anche tutti i costrutti avanzati, precedentemente descritti, subiscono delle variazioni; tipicamente quando si trasforma un DFM in un sistema a stella bisogna valutare caso per caso i diversi costrutti, tuttavia una possibile traccia generale da seguire è la seguente:

- **Attributi Descrittivi**→ a seconda che si riferiscano a dimensioni o fatti verranno inseriti rispettivamente nella *dimension table* o nella *fact table*.
- **Attributi Cross-Dimensionali**→ poiché $b$ definisce una relazione *one-to-many* tra due o più attributi $a_1, ...a_n$, sarà necessario tradurre in una tabella che includa $b$ e abbia come chiave $a_1, ..., a_n$.
- **Gerarchie condivise**→ se le due gerarchie contengono gli stessi attributi basterà importare due volte la chiave nella medesima *dimensio table*; in caso condividano un'unica parte, bisognerà introdurre una terza tabella che metta insieme gli elementi.

- **Archi multipli** → serve introdurre una tabella ponte al fine di collegare i due attributi che condividono l'arco.

Le gerarchie consentono di associare alle dimensioni dei livelli di aggregazione successivi, attraverso connessioni di dipendenza funzionale. Poiché i DW cambiano nel tempo, anche le dipendenze funzionali associate alle gerarchie possono subire mutamenti. Lo studio delle trasformazioni delle gerarchi risulta fondamentale per la comprensione dei dati, e in particolare per le variazioni che subiscono. Vi sono alcune possibilità (ciascuna da valutare in base all'esigenza) per risolvere il problema delle **gerarchie dinamiche**:

- **Oggi o Ieri**: ciascun evento è riferito al valore delle gerarchie nell'istante di tempo in cui si è verificato. E' tra le tecniche che rappresentano più fedelmente la realtà. Vi sono alcune criticità in alcuni casi, come nell'esempio sotto presentato: dallo schema sembrerebbe che i negozi in Piemonte sono stati aperti solo nel 2003, quando in realtà era già presente il distretto Piemonte-Liguria. Lo schema sarebbe invece ottimale in caso si voglia studiare il premio alla produzione dato ai responsabili (Rossi x-2001, Bianchi 2002-x)




| Negozio1 | 2000 | 2001 | 2002 | 2003 | 2004 |
|---|---|---|---|---|---|
| **Responsabile** | Rossi | Rossi | - Bianchi- | Bianchi | Bianchi |
| **Distretto** | PiemonteLiguria | PiemonteLiguria | PiemonteLiguria | Piemonte | Piemonte |
| **Incassi** | 1000 | 1300 | 1200 | 1500 | 1400 |

- **Ieri per Oggi**: ciascun evento viene riferito al valore iniziale delle gerarchie. Tale schema non terrebbe conto della suddivisione del distretto o il cambio del responsabile: tipicamente è una soluzione da evitare

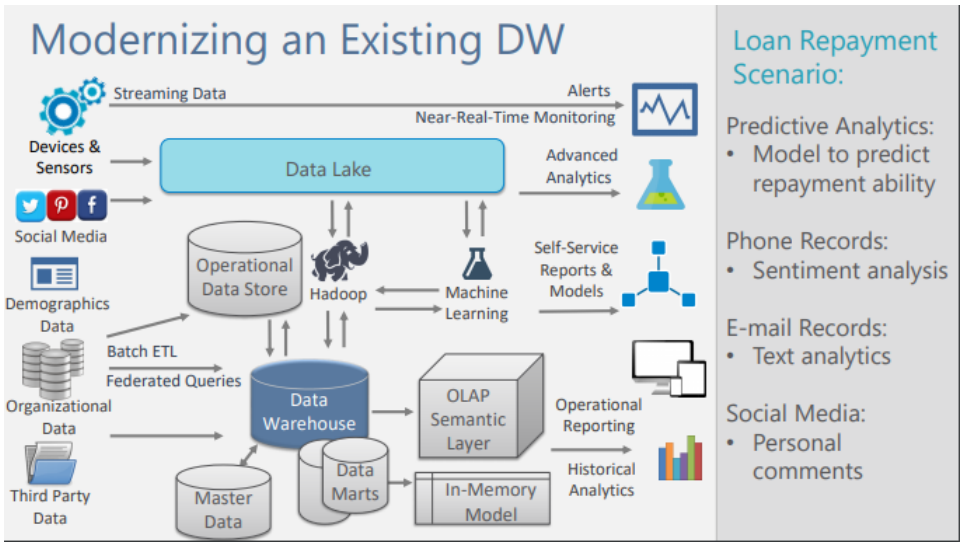
| Negozio1 | 2000 | 2001 | 2002 | 2003 | 2004 |
|---|---|---|---|---|---|
| **Responsabile** | Rossi | Rossi | Rossi | Rossi | Rossi |
| **Distretto** | PiemonteLiguria | PiemonteLiguria | PiemonteLiguria | PiemonteLiguria | PiemonteLiguria |
| **Incassi** | 1000 | 1300 | 1200 | 1500 | 1400 |

- **Oggi per Ieri**: gli eventi vengono descritti alla situazione delle gerarchie attuale (consente analisi storicizzate, inadatto per molti tipi di analisi)

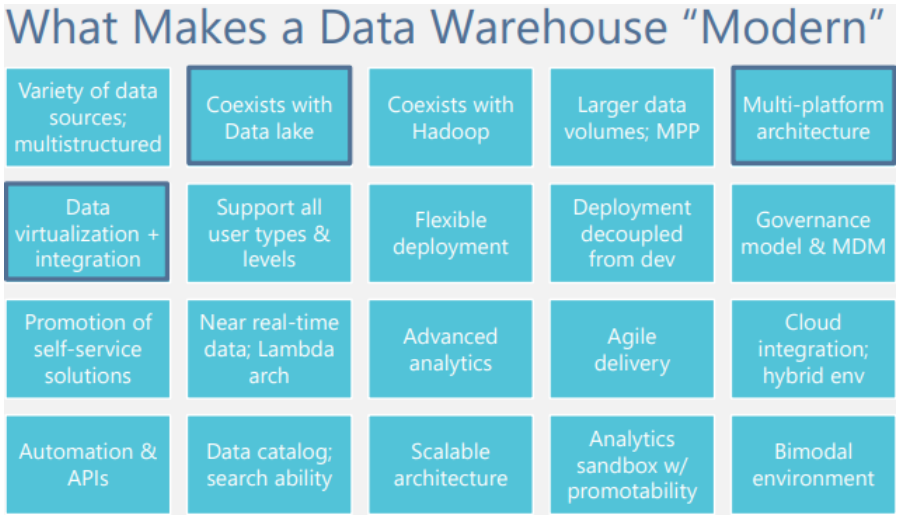
| Negozio1 | 2000 | 2001 | 2002 | 2003 | 2004 |
|---|---|---|---|---|---|
| **Responsabile** | Bianchi | Bianchi | Bianchi | Bianchi | Bianchi |
| **Distretto** | Piemonte | Piemonte | Piemonte | Piemonte | Piemonte |
| **Incassi** | 1000 | 1300 | 1200 | 1500 | 1400 |

## 3.3 *Data Lake*

Il *data lake* è la risposta alle esigenze di *storage* dei dati nell'era dei *big data*, esigenze difficilmente soddisfacibili attraverso l'impiego dei DW, i quali risultano spesso troppo rigidi. Negli ultimi dieci anni infatti si è aggiunta una serie di nuovi dati (per esempio di sensori, *social network*, *real time*, etc) e una serie di nuove tecnologie che hanno portato ad un significativo cambiamento nella gestione dei dati.

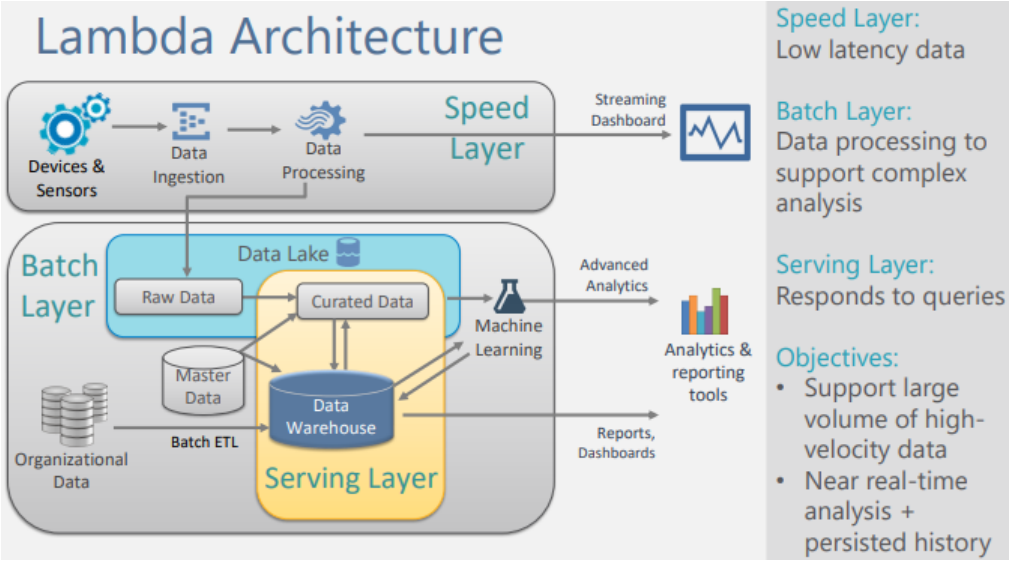



Nell'immagine viene messa in luce tale evoluzione, la quale ha portato alla nascita del *data lake*; la differenza fondamentale dal DW, ovvero un magazzino dove tutti i dati vengono organizzati con uno schema rigido, è la sua natura flessibile, all'interno del quale vi è un flusso di dati continuo. Tale condizione fa sì che all'interno del *data lake* arrivino dati nella forma in cui sono stati generati, senza particolari trasformazioni atte all'inserimento nello schema, e che solo in un secondo momento vengano rimaneggiati a seconda delle esigenze di analisi (sempre a differenza del DW nel quale i dati vanno inseriti seguendo un determinato schema). E' sempre possibile passare a posteriori i dati dal DL a DW, dopo aver applicato le analisi e trasformazioni necessarie

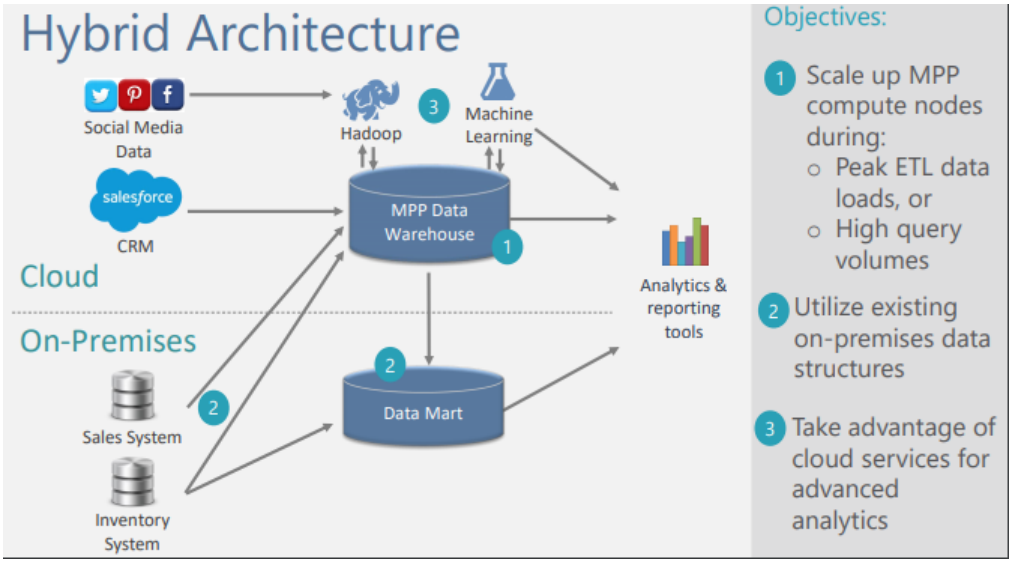


Esistono molte architetture per creare un sistema in cui comunichino DL e DW:

- **Lambda Architecture**: consentono sia una gestione dei dati in *real time* attraverso il *data lake* (*speed layer*, molto utile in caso di *marketing real time*), sia un'analisi più accurata (*serving layer*→ *curated data*).




- **Hybrid Architecture**:

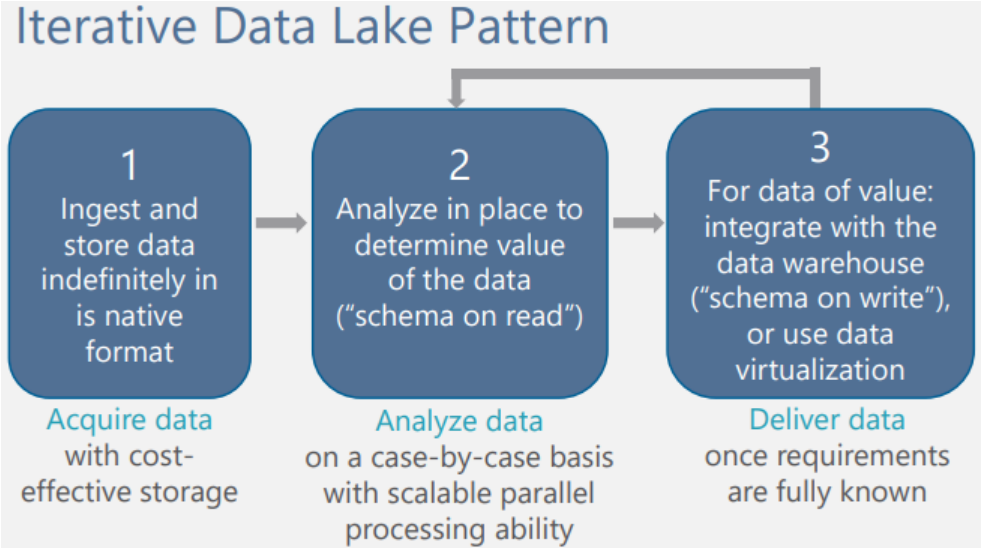



Per riassumere un *data lake* non è altro che un insieme informe di dati, i quali vengono salvati nel loro formato (relazionale, NoSQL, immagini, etc), i quali possono essere successivamente rimaneggiati. La **principale criticità** nella gestione di un *data lake* è che, per la sua natura di flusso non controllato, rende difficile identificare i file nel tempo (a differenza il DW etichetta ciascun file); per ovviare a tale problematica servirebbero strutture esterne che riescano a identificare o catalogare il contenuto (fatto, attributi, dimensioni, etc) dei *files* che vanno a far parte del DL.

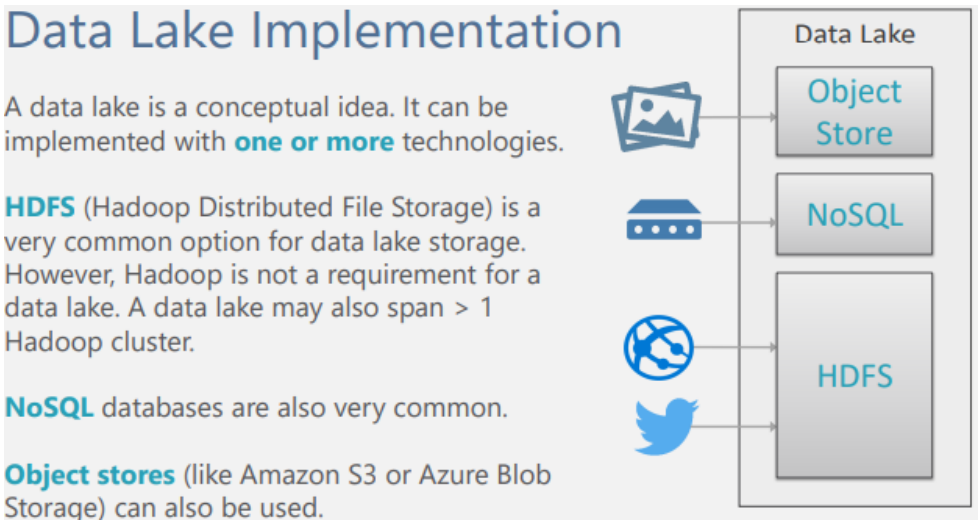
Gli **obiettivi** del Dl sono la riduzione del costo di acquisizione di nuove informazioni (derivante dal dover definire o aggiornare lo schema), rende più semplice l'acquisizione dei dati e consente di memorizzarne grandi quantità anche in caso di natura multi-strutturale.

L'approccio più utilizzato è il **case-on-case**:

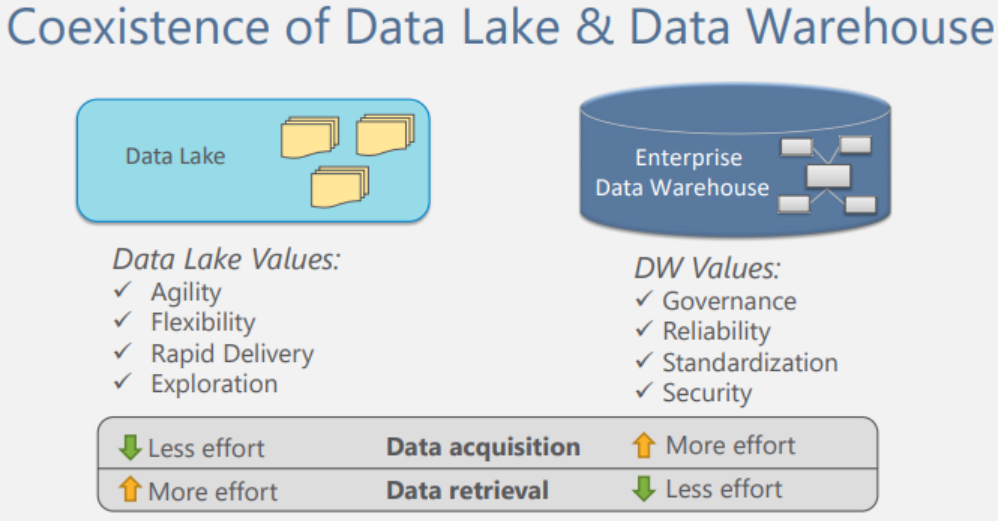
- I dati ottenuti vengono caricati nel DL nel loro formato nativo, senza particolari modellazioni.
- Successivamente vengono analizzati caso per caso utilizzando procedure parallelizzate.
- Infine vengono selezionati i dati ritenuti maggiormente interessanti o di valore, i quali verranno schematizzati e andranno ad arricchire il DW.




Esistono diversi **strumenti** per implementare un DL, i quali possono essere utilizzati in maniera combinata in strutture ibride:

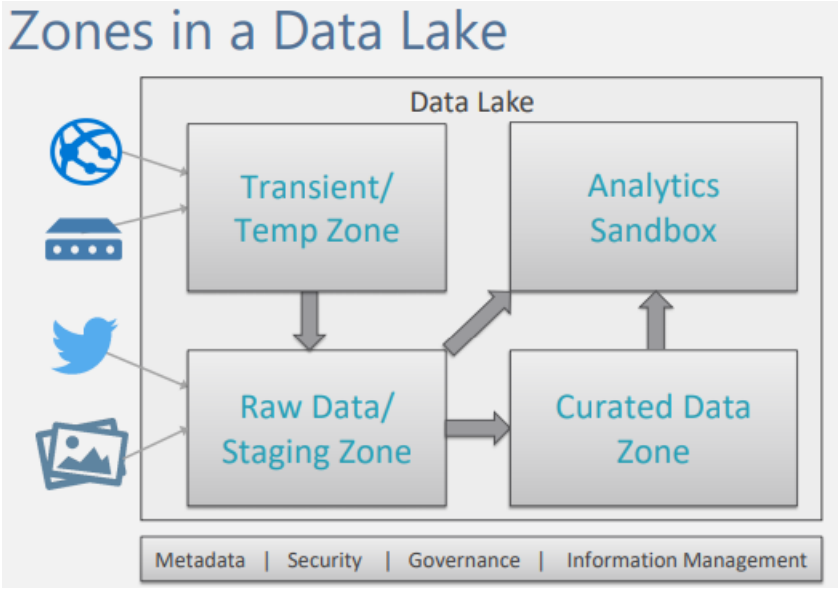


Le principali differenze tra un DL e un DW sono riassunte nella seguente immagine:




Un tema fondamentale quando si parla di DW, e ancor di più di DL, in ambiente di *business* è la **sicurezza delle informazioni**; essendo il dato una risorsa che l'azienda impiega per il miglioramento dei processi aziendali, o comunque per attività profittevoli, è importante che non venga visto da esterni. In questo senso la principale criticità di impiegare un DL, dove tutti i dati vengono memorizzati nello stesso "luogo", è l'enorme responsabilità data al responsabile (persona o reparto) che gestisce il DL stesso.

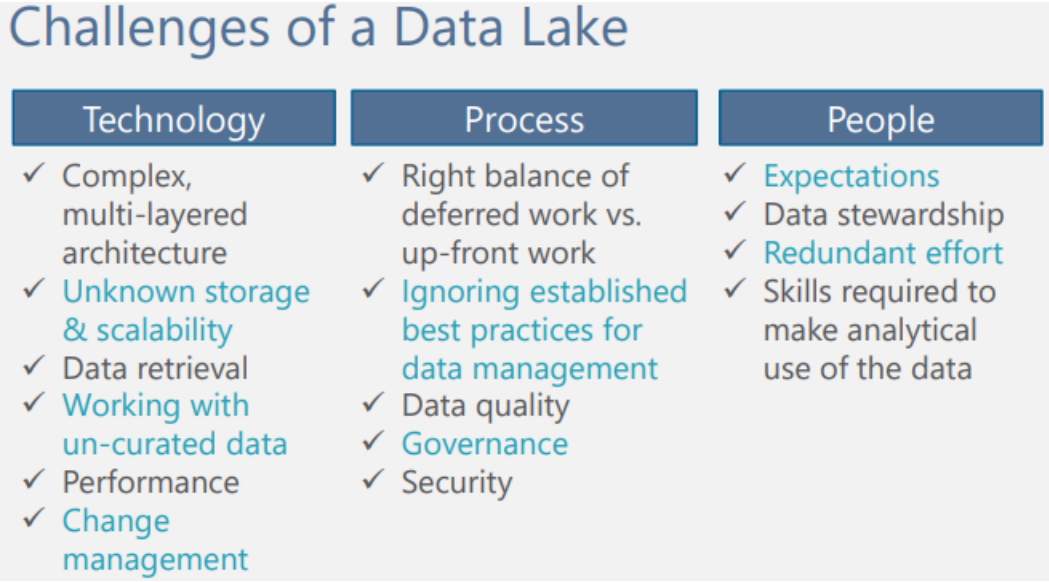
Lo schema interno di un DL prevede generalmente 4 aree:




dove la **transient zone** (acquisizione del dato) rappresenta il settore nel quale vengono applicate alcune pulizie preliminari del dato, prima del passaggio alla **raw zone** (*storage* del dato), mentre la **curated data zone** (*pre-processing/transformation* del dato) è il settore dove i dati vengono trasformati in maniera da essere utili alle

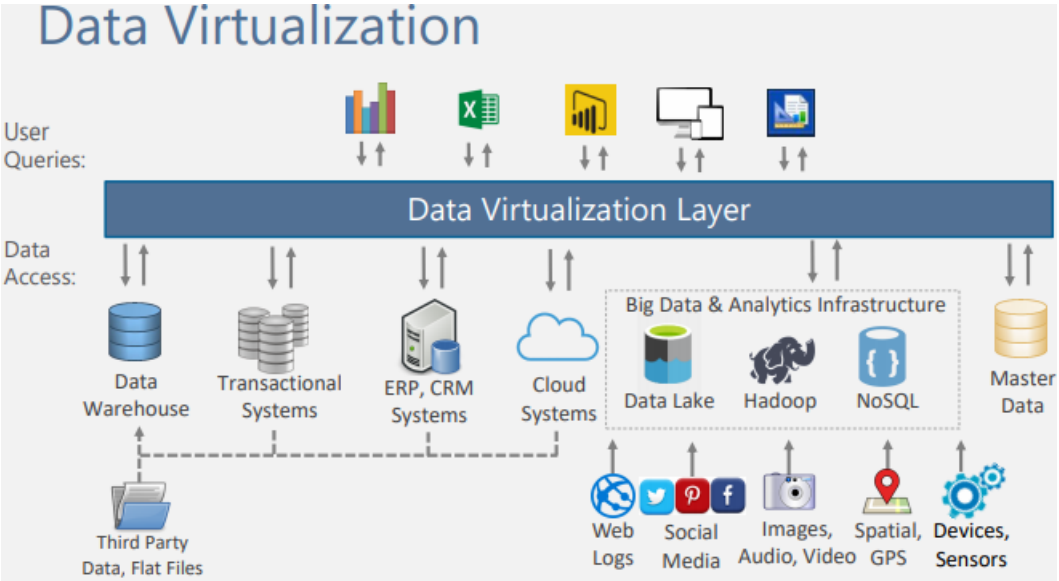
analisi (**analytics sandbox**). Al fine di strutturare un DL che sia condivisibile dai diversi analisti è necessario che tutti i passaggi e le trasformazioni del dato attraverso il DL stesso vengano adeguatamente documentate (*governance* del dato).

Per riassumere **le principali sfide** derivanti dall'introduzione di un approccio *data lake*:




**I principali vantaggi** sono invece la strutturazione di una "piattaforma di atterraggio" dove poter mettere i dati prima di (eventualmente) inserirli in uno schema *curated*, una zona in cui poter riportare dati strutturati per alleggerire il DW e infine garantiscono la possibilità di ottenere una vasta gamma di dati (forse il punto più cruciale nell'era dei *big data*).

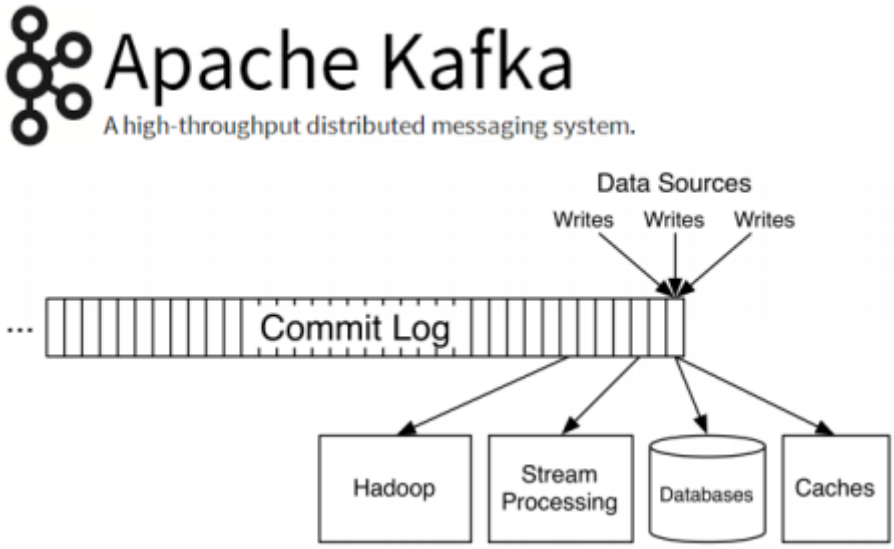
Lo **scopo finale** di un DW e di un DL è garantire dati in uscita che possano essere utilizzabili per le analisi, e per presentare i risultati sotto forma di dati strutturati, report e/o visualizzazioni; per tale motivo l'ultima fase viene denominata **data virtualization**:



# Velocity

Il concetto di *velocity* nasce nel momento in cui vi è la necessità di acquisire ed eventualmente processare dati velocemente o in tempo reale (*streaming data in real-time*). Alcuni esempi che coinvolgono architetture di questo genere sono l'analisi dei sensori, dei *social network*, del *real-time marketing*, informazioni sui battiti cardiaci (e *healthcare monitoring* in generale), analisi geo-referenziale (per esempio nei negozi, logistica, ...), etc. *Real-time* e velocità non sono lo stesso concetto: il primo si riferisce alla capacità di un sistema di rispettare dei vincoli temporali, impliciti o espliciti che siano.

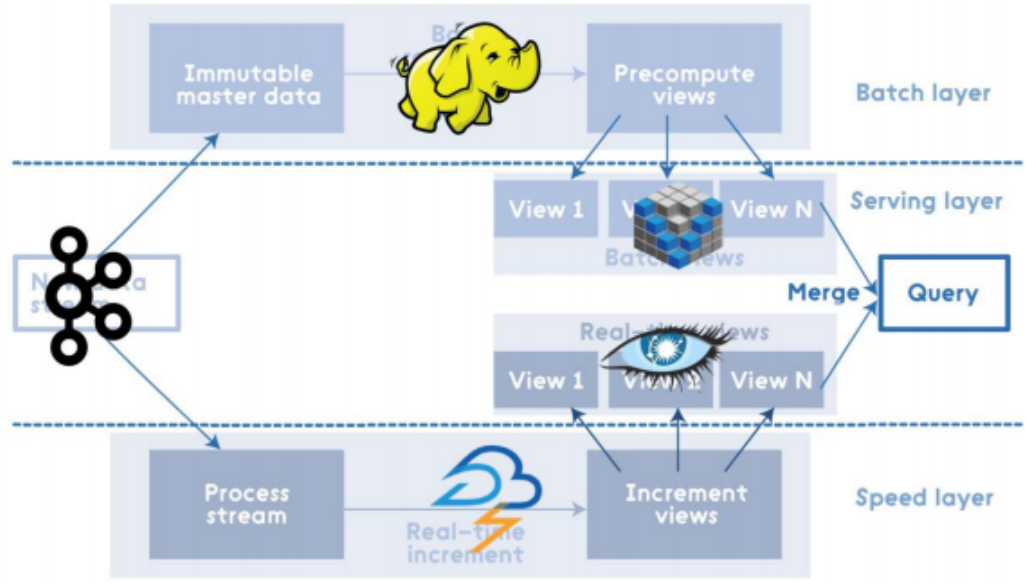
Al fine di costruire una struttura capace di gestire dati caratterizzati da velocità vi è la necessità di avere a disposizione uno strumento che intermedi tra il **produttore** di dati (ovvero la fonte, colui che scrive) e il **consumatore** di dati (ovvero chi legge e/o utilizza per le analisi); tale strumento viene definito **middleware** (come ad esempio Apache Kafka) e ha lo scopo di garantire una "coda" (*commit log* ,tutti i dati prodotti velocemente vengono inseriti) nel quale il produttore inserisce i dati.



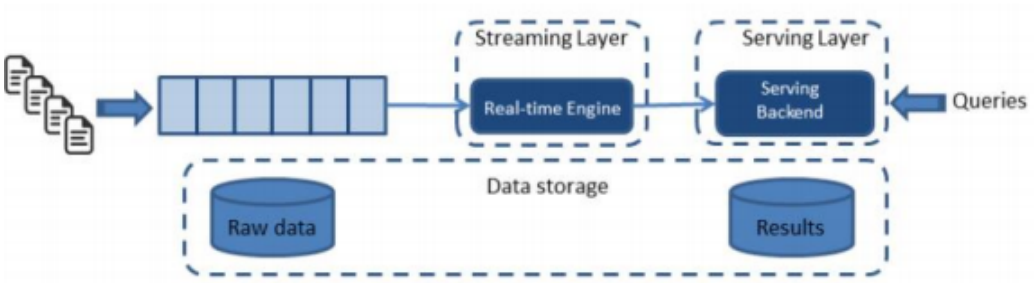

Un esempio è il caso dei post dei *social network*, nel quale si utilizza un *topic* (per esempio un *hashtag*): quando il consumatore si iscrive alla coda con un determinato *topic*, il *middleware* aggiungerà alla coda tutti i post prodotti inerenti, mentre il consumatore attenderà una risposta dall'intermediario per la lettura. **Lo scopo principale** è quindi il disaccoppiamento della procedura di ottenimento dei dati da

quella di lettura degli stessi; vi sono di conseguenza due flussi, uno di scrittura e uno di lettura (comunicazione asincrona).

A seconda della tipologia di problema si può avere la necessità di acquisire dati in tempo reale (ed elaborarli in tempi successivi) oppure di elaborare dati in *real-time*; per la seconda situazione esistono delle apposite strutture (come la **Lambda Architecture**), le quali hanno una componente definita *speed layer*, ovvero uno strato *software* che elabora i dati appena li riceve (per esempio Spark). La principale criticità di un'architettura Lambda consiste nella necessità di implementare 2 volte lo schema logico (a causa della sua natura a doppio ramo).




Un'altra architettura possibile è la **Kappa Architecture**; la differenza principale dalla precedente architettura consiste nella natura seriale del sistema, ovvero prima vengono svolte le operazioni dello strato veloce (*streaming layer*), e solo successivamente i dati vengono passati al *serving layer* (il precedente *batch layer* nell'architettura Lambda). I dati vengono di conseguenza raccolti/processati/elaborati velocemente, e successivamente (con il tempo necessario) i dati vengono memorizzati.




Esistono inoltri sistemi che si occupano unicamente di *streaming and monitoring*, ovvero architetture adatte a gestire dati veloci verso cui si ha come interesse principale il monitoraggio (come i battiti cardiaci), e solo come obiettivo secondario quello dell'elaborazione (modello predittivo per il rischio d'infarto). Un esempio di tecnologia di *real -time monitorig* è ELK (dall'acronimo dei 3 *software* che lo compongono: Elasticsearch, Kibana, Logstash).