Data Management

1 Data Modelling.

Il data modelling è uno strumento per descrivere parte del mondo reale che ci permette di:

- immagazzinare dati (write data);
- interrogare i dati (read data).

Un data model è composto da un modello concettuale (teoria) e da un linguaggio (anche grafico a volte) per descrivere e interrogare i dati. Un data model è sempre disponibile in un database management system (DBMS) che supporta la semantica del modello. Alcune delle caratteristiche più importanti di un data model sono:

- Machine readability
- Expressive power
- Semplicità? Flessibilità? Standardizzazione?

Il data model più famoso e conosciuto è il modello relazionale.

1.1 Relational model.

Aspetti positivi:

1. Schema rigido.
2. Sfrutta le proprietà ACID :
    - Atomicity: l'operazione è *atomica*, ovvero o avviene per intero (`COMMIT`) o restituisce un errore e il database ritorna alla forma iniziare (`ROLLBACK`); ad esempio se si aggiornano i dati o sono aggiornati tutti o non è aggiornato nessuno.
    - Consistency: i nuovi dati inseriti rispecchiano lo schema prestabilito (o l'operazione di inserimento fallisce);
    - Isolation: le singole operazioni non influiscono sulle altre; per fare questo il database costruisce una coda di esecuzione dei processi, così lo stato del database non muta durante l'esecuzione di una singola richiesta;
    - Durability: la persistenza del file è garantita (virtualmente a tempo indefinito) anche in caso di crash del sistema; per ottenere questo risultato sono utilizzati backup e log files.

Il modello relazionale esiste da ormai 35 anni, è ben sviluppato e molto conosciuto: molti dati sono ancora salvati in questo formato ed è efficace ancora per molte operazioni. Tuttavia presenta numerosi svantaggi:

1. un attributo può avere solo un valore;
2. non è compatibile con i moderni linguaggi di programmazione ad oggetti;
3. il linguaggio è molto rigido;
4. non permette loop nei dati;
5. la modifica di tabelle esistenti è difficile e dispendiosa;
6. è poco prestante in alcuni contesti.

Nei RDBMS la performance (inteso come velocità) dipende da vari fattori:

- numero delle righe;
- tipo di operazione;
- algoritmo scelto;
- struttura dei dati.

Per *Scaling Up* si intende potenziare la singola macchina, mentre per *Scaling out* si intende aggiungere macchine in una rete di calcolatori; quest ultimo è molto difficile da effettuare su un database di tipo relazionale. Il costo è un altro dei problemi: l'acquisto di macchine di una certa potenza è molto alto e non garantisce un aumento lineare delle prestazioni, che anzi scendono asintoticamente.

1.2 NoSQL.

Per risolvere i problemi dei database relazionali, nasce un movimento informatico chiamato NoSQL (*Not Only SQL*) che non rifiuta il modello ma propone approcci alternativi. I database di tipo NoSQL, pur essendo molto diversi tra di loro, hanno caratteristiche comuni:

1. non hanno nessuno schema o modello prestabilito: la costruzione di un modello rigido per la registrazione dei dati è considerata una limitazione da superare, pertanto per aggiungere un nuovo attributo non è necessario modificare l'intero modello;
2. seguono l'assunzione del mondo aperto: ciò che non è vero è sconosciuto, ma non per forza falso, mentre SQL segue l'assunzione del mondo chiuso (è vero solo ciò che è noto lo sia).
3. seguono il teorema CAP: è impossibile per un sistema informatico distribuito (cioè un sistema interconnesso la cui comunicazione avviene solo attraverso messaggi) fornire simultaneamente tutte e tre le proprietà (si possono soddisfare solamente 2):
   (a) **(C)**onsistency (Coerenza): tutti i nodi vedono gli stessi dati allo stesso istante; se è assente allora una soluzione è mostrare il dato precedente alla modifica e non quello più recente.
   (b) **(A)**vailability (Disponibilità): il ricevere una risposta in un tempo determinato per ogni richiesta (fallita o meno); se assente non si riceve risposta in caso di fallimento della richiesta;

(c) **(P)**artition Tolerance (Tolleranza al partizionamento): il sistema funziona anche con dati frammentanti su una rete di calcolatori;

Il modello relazionale ha le proprietà CA, mentre i NoSQL generalmente sono CP o AP. Si può preferire la disponibilità alla coerenza quando è preferibile ottenere dati non aggiornati piuttosto che messaggi di errore.

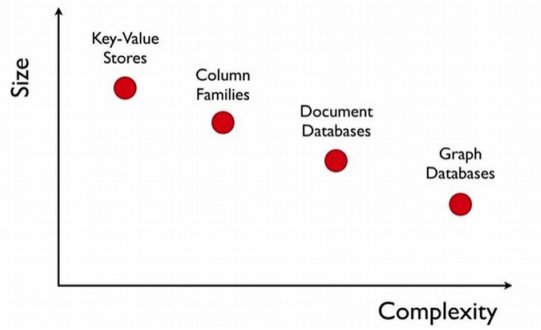
4. seguono il principio BASE (ovviamente, il contrario di ACID):
   - Basic Availability: la consistenza può anche non essere garantita interamente, mostrando solamente una parte dei dati realmente disponibili (è il tipico caso di crash di un nodo che quindi non può trasmettere i dati al resto della rete);
   - Soft State: i dati possono avere schemi diversi (a differenza del modello relazionale);
   - Eventual Consistency: i sistemi NoSQL richiedono che, ad un certo punto, i dati convergano a uno stato consistente senza specificare quando; prima del raggiungimento della consistenza si possono avere valori non veritieri.

ACID	BASE
• Forte Coerenza • Poca disponibilità • Pessimo "Multitasking" • Rigido	• Coerenza debole • Disponibilità come priorità e sacrifica per questo (CAP) • Veloce e Semplice

I modelli NoSQL si dividono in macrocategorie:

1. Key Value (Dynamo, Voldemort, Rhino DHT): sono delle tabelle con chiavi che si riferiscono (o puntano) a un certo dato, è molto simile ai Document based.
2. Column family (Big Table, Cassandra): in grado di salvare grandi quantità di dati, la chiave colonna si riferisce a un certo dato raggruppato in colonna.
3. Document based (CouchDB, MongoDB): di solito salvati con file JSON, contenente un insieme ordinato di coppie <chiave – valore>; è facile ricercare dati in questo formato;
4. Graph based (Neo4J, FlockDB): sfrutta il concetto matematico di grafo per archiviare i dati come nodi ed esaltare i legami tra di essi costruendo archi; è difficile da scalare per quanto riguarda l'immagazzinamento (è difficile tagliare un grafo) ma è molto rapido nelle query.

La semplicità del modello permette di archiviare un maggior numero di dati:



1.2.1 Document Based (MongoDB).

MongoDB è, come già affermato, un sistema di gestione basato sui documenti (document based management system), in cui i dati vengono archiviati in formato BSON (Binary JSON) e letti tramite indici. Volendo fare un confronto con i database di tipo SQL, Mongo DB non prevedere operazioni di join. Le principali differenze sono le seguenti:

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

Nel caso in cui si verifichi un massiccio caricamento di dati, si potrebbe decidere di apportare alcune modifiche al processo per fissarlo:

- Disabilitare il riconoscimento dei dati, che è un segnale trasmesso tra processi di comunicazione, computer o dispositivi, per indicare il riconoscimento o la ricezione del messaggio, come parte di un protocollo di comunicazione
- Disabilitare la scrittura su un file di tipo log

Bisogna stare molto attenti nel fare ciò, perché ogni ogni perdita non sarà registrata e di conseguenza sarà definitiva.

Per dataset di grandi dimensioni risulta molto utile l'utilizzo di indici: simili agli indici dei libri (o delle tabelle SQL), rappresentano un modo più veloce per recuperare informazioni. L'uso di indici rallenta l'inserimento di nuovi dati (perché appunto devono essere indicizzati) ma velocizza notevolmente le ricerche. Il campo "_id" è sempre indicizzato. Senza un indice, MongoDB analizza a tappeto tutti i documenti (esattamente come SQL). Metaforicamente dovrebbe leggere ogni volta tutto il libro perrecuperare l'informazione. L'indicizzazione evita proprio questo problema (che aumenta con le dimensioni della collection), organizzando il contenuto in una lista ordinata. Dato che l'indicizzazione rallenta le modifiche, è buona norma utilizzare solamente un paio di indici per ogni collection; il limite di MongoDB è di 64 indici.

L'aggregazione utilizza pipeline e opzioni. La pipeline di aggregazione avvia l'elaborazione dei documenti della raccolta (collection) e passa il risultato alla pipeline successiva per ottenere risultati, ad esempio: *matche* group. Possiamo utilizzare lo stesso operatore in diverse pipeline.

Per ragioni commerciali, MongoDB offre anche un'interfaccia SQL tramite il connettore BI (Business Intelligence): genera lo schema relazionale e lo usa per accedere ai dati.

1.2.2 GraphDB (Neo4J).

Un grafo è una collezione di nodi e archi, i quali rappresentano le relazioni tra gli stessi nodi. Ha un sacco di applicazioni, come: social media, raccomandazioni, luoghi geografici, reti logistiche, grafici per le transazioni finanziarie (per il rilevamento delle frodi), master data management, bioinformatica, sistemi di autorizzazione e controllo degli accessi.
Il modello a grafo con etichette (labeled-property graph model) ha le seguenti caratteristiche:

- contiene **nodi** e **relazioni**;
- i nodi contengono **proprietà** (coppie chiave-valore);
- i nodi possono essere etichettati con una o più **etichette**;
- le relazioni sono nominali e direzionali, con un nodo d'inizio e uno di fine;
- le relazioni, come i nodi, possono avere delle proprietà (e queste possono avere anche valori).

Le proprietà delle relazioni possono essere assegnate con un criterio **fine-grained** o **generic**. Considerando il caso della relazione `ADDRESS`, si può fare distinzione tra:

- fine-grained: `HOME_ADDRESS`, `WORK_ADDRESS` o `DELIVERY_ADDRESS` (sono tutte etichette diverse);
- generic: `ADDRESS:home`, `ADDRESS:work` o `ADDRESS:delivery` (l'etichetta è sempre `ADDRESS`, con un attributo che ne indica il tipo).

Generalmente è preferito il secondo metodo per la minore complessità nello scrivere query (come ad esempio elencare tutti gli indirizzi di una data persona).

Un database a grafo può usare un motore di archiviazione nativamente a grafo o usare altri sistemi di archiviazione. Il primo ottimizza la gestione de grafi, mentre il secondo archivia i dati in formato tabellare o tramite documenti per poi interrogare il database come se fosse un grafo. Il metodo tabellare per esempio archivia le relazioni su una tabella relazionale che può essere interrogata tramite *join bomb* (join con se stessa), tuttavia questo sistema degenera in fretta all'aumentare della distanza tra due nodi.

Il database a grafo risolve quindi il problema dei database relazionali (e di molti altri database NoSQL) a gestire le relazioni interne ai dati. Infatti anche altri modelli NoSQL, indipendentemente dal modello in uso, soffrono perdite di prestazioni quando sono effettuate aggregazioni di dati (soprattutto non indicizzati) dal momento che i collegamenti non sono nativi nel modello e manca il concetto di prossimità. Si può tentare di risolvere il problema con dati annidati tra di loro ma la struttura del database risulterebbe eccessivamente complessa e non permetterebbe altre query. Il DBA in base ai suoi bisogni (integrazione con altre applicazioni) può benissimo decidere di usare un database a grafo con una gestione dei dati non nativa senza che questo impatti sulla qualità del prodotto finale. In un archivio nativo a grafo gli attributi, i nodi e i nodi referenziati sono memorizzati insieme per ottimizzare l'engine di processamento a grafo. Per eseguire una query

nel modello a grafo, il tempo di risposta non tipende strettamente dal numero totale di nodi (che rimane più o meno costante) perché la query viene processata nella porzione locale del grafo connessa al nodo base, mentre nei modelli relazionali e altri modelli NoSQL peggiorano le prestazioni al aumentare dei dati (spesso in una maniera lineare). Inoltre è possibile aggiungere altri nodi e relazioni senza disturbare il modello già esistente anche nel caso i dati non abbiano la stessa struttura.
Il processing engine usa "index-free adjacency" cioé i nodi connessi sono collegati fisicamente tra di loro, ciò velocizza il loro recupero da una query, ma questa velocità ha un prezzo: l'efficacia dei query che non sfruttano le proprietà del grafo viene peggiorata, ad esempio nel caso delle operazioni di I/O.

Neo4j implements a declarative graph query language Cypher. Cypher allows graphs to be queried using simple syntax somewhat comparable to SQL or SPARQL, but particularly optimized for graph traversals.
The properties of the cypher language are:

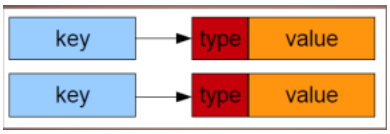
1. Pattern-matching query language
2. Humane language, easy to learn, read and understand
3. Expressive yet compact
4. Declarative : Say what you want, not how
5. Borrows from well known query languages especially SQL, yet clearly it's different enough to see that it deals with graphs and not SQL DBs.
6. Aggregation, Ordering, Limit
7. Update the Graph

A real example is the following :
We want to manage a server farm, we define a relational model for managing it. We know that a user access to application which runs on a VM and each application uses a DB and a secondary DB. Each of the VM is hosted on a server placed in a rack structure managed by a load balancer.
The initial stage of modeling is similar in any kind of DB, we seek to understand and agree on the entities in the domain and how to interrelate, usually done on whiteboard with a diagram which is a graph. Next stage we seek a E-R (Entity-Relationship) diagram, which is another graph. After having a suitable logical model we map it into tables and relations. We keep our data in a relational DB, with it's rigid schema. But keeping the data normalized slows down the query time so we need to denormalize it, because the user's data model must suit the database engine not the user. By denormalizing we involve duplicate data to gain query performance: denormalization is not a trivial task and we accept that there may be substantial data redundancy. The problems do not stop here, because once the DB is created, if we need to modify it -this is typically going to happen in order to match the changes in the production environment- we will need to do the whole work again!
In these cases, It is better to directly use the graph DBs: it will avoid the data redundancy and it can adapt really fast in case of a change in the DB itself.
Another graph traversal language is Gremlin, part of the Apache TinkerPop framework.

In this domain specific language (DSL) expressions specify a concatenation of traversal steps, so you basically explain to gremlin step by step what to do.

1.2.3 Key-Value Model

It's the most simple and flexible model of the NoSQL family, where every key is assigned to a value. It is possible to assign a type to a value. The values are not query-able, such as a BLOB, where a BLOB(Binary Large OBject) is a collection of binary data stored as a single entity in a DB, for example images, videos etc.:



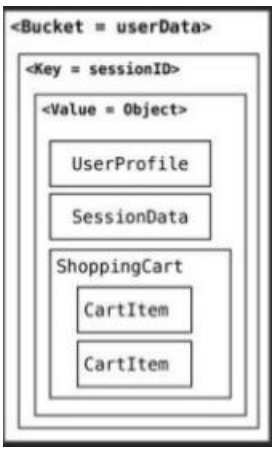
An example is the amazon's cart system which uses DynamoDB, a key-value system.

The basic operation in a key-valued models are:

- Insert a new key-value pair
- Delete a new key-value pair
- Update a new key-value pair
- Find a value given the key

There is no schema and the values of the data is opaque. The values can be accessed only through the key, and stored values can be anything : numbers, string, JSON, XML, images, binaries etc. An example of key-value model is Riak and has the following terminology:

Relational	**Riak**
database instance	Riak cluster
table	bucket
row	key-value
rowid	key



Some other examples are: Redis, Memcached, Riak, Hazelcast, Encache.
Thanks to the Hash based index, the key-value systems can scale out in a very efficient way. The Hash is a mathematical function that assign to a given key it's value, usually we have $h(x) = value$, usually it returns a pointer to where the data is stored not exactly the data itself, for example $h(x) = (x \ modulo \ y)$ where y is the max length of hash table. There might a problem with the conflicts but they can be managed. Hashing also enables

items in the DB to be retrieved quickly. The hash table can be easily distributed in a network, it is managed in pile so we can have a key(saved in the pile) x saved in a server and it's succ(x) stored in a different server, thus scaling out really fast. One reason to use hashing is so that we are able to evenly distribute the data across a certain number of slots. If we want to hash any number into 10 buckets, we can use modulo 10; then key 27 would map to bucket 7, key 32 would map to bucket 2, key 25 to bucket 5, and so on. In a DHT(Distributed Hash Table) it is pretty simple to insert a key-value `(k1,v1)`: basically take the key as input and route messages to the node holding that key and store the value there, and to retrieve the value of k1 the system simply finds the node with the key `k1` and return it's value `v1` from it.

1.2.4 Wide Column and BigTable

Intro to Columnar Storage: The essence of the columnar concept is that data for columns is grouped together on disk. In a columnar database, values for a specific column become co-located in the same disk blocks, while in the row-oriented model, all columns for each row are co-located. Indeed, one of the advantages to the columnar architecture is that queries seeking to aggregate the values of specific columns are optimized, because all of the values to be aggregated exist within the same disk blocks. The exact IO and CPU optimizations delivered by a column architecture vary depending on workload,indexing, and schema design. In general, queries that work across multiple rows are significantly accelerated in a columnar database.

These two models are an evolution, in a certain way, of the key-value models. When we start giving a structure to the value it becomes more complicated so we use wide column(Big Table): The first model was introduced by Google(HBase) and later on by Facebook (Cassandra), even though some consider Cassandra still as a key-value model and not a wide column.

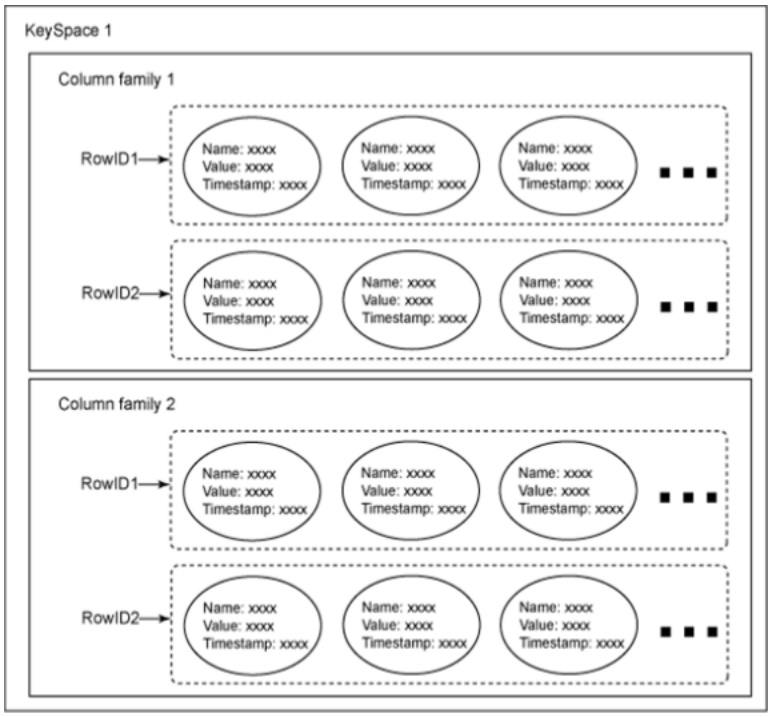
BigTable is a multidimensional map, which can be accessed by row key, column key and a timestamp. It is sorted, persistent and sparse. We will consider the **HBase BigTable** Model. The data is organized in tables, each table(the tables are multi-versioned) is composed by the column-families that include columns, the cells within a column family are sorted physically and are usually very sparse with most of the cell having `NULL` value so we can have different rows with different sets of columns. BigTable is characterized by the row key, column key, timestamp, the row has the keys, column contains the data and contents. The column is divided in families. The timestamp support the multi-version of modification to check how the data changed over time and still be able to access the latest one without any confusion. We can represent it as follows:

row key	**column family 1**		**column family 2**		
	column 1	**column 2**	**column 1**	**column 2**	**column 3**
	cell(data)	*cell(data)*	*cell(data)*	*cell(data)*	*cell(data)*
	cell(data)	*cell(data)*	*cell(data)*	*cell(data)*	*cell(data)*

Each row can have different timestamp, so we can have more versions of this table.

The data within the cells (also the key) has no type since it is saved in bytes. The columns are dynamic. So to get a data given a key we need to transform our data in Bytes with a comand .toBytes("Key"), we can also use python to modify the columns via APIs. This model can be useful for *-To-Many mappings. The row key is an array of bytes and serves as a primary key for the table. Each Column Family has a name and contains one or more related columns, the columns can belong to one column family only and is included inside the row with familyName:columnName followed the value, for example we have:
row key | info:{'height':'170cm', 'state':'NY'} roles:{'ASF':'Director', 'Hadoop':'Founder'}|
The version number of each row is unique within the row key, by default it uses the system's timestamp but they can be user supplied.
Best time to use HBase is when one need to scale out, need random write and/or read, when we need to do thousand of operations per second on multiple TB of Data, when we need to know all the modification done on the data, when we need a well known and simple access. One can combine a GraphDB with HBase, one example is JanusGraph. HBase does not support joins, but this can be done in application layer using Scan() and Get() operations of HBase.

Cassandra[Facebook]: It started as a support for the Facebook inbox search, it was open sourced in 2008 by Facebook and then it became a top-level project under Apache in 2010. The data model is the same as HBase, the only difference is the way it is stored (storage model) and the program algorithms. We can say it is a restricted BigTable, we can have only one column family. It uses a C* language very similar to SQL language.
Cassandra is a column oriented NoSQL system. The KeySpace is the outermost container, it's basic attributes are the Replication Factor (how many copies of the data we need), Replication Strategy and the Column Families. The Column Families, can be seen as a collection of rows or better a collection of key-value pairs as rows. A row is a collection of columns labeled with a name, the value of a row is itself a sequence of key-value pairs where the keys are the column's name and we need at least one column in a row. The columns have the key(name), the value and the timestamp. So a table is a list of "nested key-value pairs": (ROW x COLUMN key x COLUMN value) and this is inside a column family. The Key Space is only a logical grouping of columns families.



An example of Cassandra data model.

While with RDBMS, we can have that JOIN of normalized tables can return almost anything, with C* the data model is designed for specific queries and the schema is adjusted as new queries are introduced. In C* there are no JOINS, relationship or foreign keys and the data required by multiple tables are denormalized across those tables.
The Comparison between Apache Cassandra, Google Big Table and Amazon DynamoDB is:

	Apache Cassandra	**Google Big Table**	**Amazon DynamoDB**
Storage Type	Column	Column	Key-Value
Best Use	Write often read less	Designed for large scalability	Large database solution
Concurrency Control	MVCC	Locks	ACID
Characteristics	High Availability Partition Tolerance Persistence	Consistency High Availability Partition Tolerance Persistence	Consistency High Availability

2 Data Distribution

2.1 Fragmentation and Replication

Un sistema di DB centralizzato è un sistema in cui il database è localizzato, memorizzato e mantenuto in un'unica posizione. Se il processore fallisce, allora tutto il sistema fallisce. Se abbiamo bisogno di avere dati periodicamente in luoghi differenti allora possiamo distribuirli o replicarli. La distribuzione quindi consiste nel dividere i dati e memorizzarli in posti diversi, provvedendo così ad avere possibilità di eseguire operazioni in parallelo. D'altra parte possiamo anche replicarli (interamente o solo una parte) in diversi luoghi, migliorando cosi la loro disponibilità (availability).
In un Database Distribuito Omogeneo tutti i siti hanno identico software e concordano tutti a collaborare per l'elaborazione delle richieste dell'utente. Si arrendono alla loro autonomia per cambiare lo schema che appare all'utente come un singolo sistema.
In un Database Distribuito Eterogeneo diversi siti possono usare differenti schemi e software, creando maggior problemi per query e transazioni. I siti possono non essere a conoscenza l'uno dell'altro e quindi potrebbero fornire solo strutture limitate per la cooperazione nell'elaborazione delle transazioni.
Le sfide per la progettazione di un Database Distribuito sono decidere cosa va, che dipende dal pattern di accesso ai dati, e dal problema di dove allocare i frammenti ai nodi. Con la replicazione il sistema mantiene multiple copie dei dati, memorizzate in diversi siti, per un veloce recupero e tolleranza degli errori con la frammentazione dei dati divisi in frammenti archiviati in siti diversi. Possiamo combinare entrambi, in modo che i dati siano partizionati in diversi frammenti, e che il sistema mantenga diverse repliche degli stessi.
Dividendo/frammentando il nostro database in n database, sorge il fenomeno "bottleneck", dove la macchina più lenta detta la peggior performance per una query.
I vantaggi di una replicazione sono:

- Availability: l'errore di un sito non determina l'indisponibilità dei dati, in quanto la sua replica esiste altrove.
- Parallelism: le query possono essere processate da diversi nodi parallelamente, rendendo tutto più veloce.
- Reduced data transfer: poiché possono esistere repliche dei dati nel nodo stesso.

Mentre gli svantaggi sono:

- Aumentati i costi per gli update: in quanto ogni replica deve essere aggiornata.
- increased complexity of concurrency control e.g. due persone prendono lo stesso ticket nello stesso istante. Potremmo avere che repliche distinte possono portare a dati incoerenti a meno di qualche meccanismo di concurrency speciale venga implementato. Una soluzione potrebbe essere quella di scegliere una copia come copia primaria e applicare operazione di controllo della conucrrency solo sulla copia primaria.

Come già accennato, possiamo combinare la replica dei dati con la frammentazione dei dati. In particolare la frammentazione dei dati consiste in una divisone di relazioni r in

frammenti che contengono sufficienti informazioni per ricostruire la relazione r. Ora per semplicità, immaginiamo che sia un database relazionale. La frammentazione può essere orizzontale o verticale, l'importante che contenga informazioni sufficienti per ricostruire la "relazione" originale . In quella orizzontale ogni tupla deve essere assegnata a uno o più frammenti mentre nella frammentazione verticale, lo schema è splittato in due diversi schemi più piccoli, dove entrambi devono contenere una chiave comune per assicurare un join senza perdite, per esempio utilizzando una tupla-ID come attributo.
I vantaggi sono:

- Orizzontale:
  - Permette processi paralleli sui frammenti.
  - Permette lo split dei dati in modo che le tuple siano poi allocate nei posti in cui avviene l'accesso più frequentemente.
- Verticale
  - Permette alle tuple di essere splittate in modo che ogni parte delle tuple sia archiviata nel luogo in cui avviene più spesso l'accesso.
  - tuple-id permette un efficiente join di frammenti verticali.
  - Permette processi paralleli su una relazione.

Le due frammentazioni possono essere mixate insieme, e inoltre i frammenti possono essere successivamente frammentati fino ad una profondità arbitraria. Data Transparency è veramente importante perché indica il grado per cui un utente non è consapevole di come i dati sono stati archiviati in un sistema distribuito. Dobbiamo considerare i problemi sulla transparency in relazione a: frammentazione, replicazione e location dei nostri dati. Considerando la differenza tra un sistema centralizzato e uno distribuito, dal punto di vista dei costi, abbiamo che:

- Per un sistema centralizzato, il criterio principale per misurare il costo di una particolare strategia è il numero di accesso ai dischi.
- In un sistema distribuito dobbiamo invece considerare il costo per la trasmissione dei dai sulla rete e anche il potenziale guadagno in performance che si ottiene con l'ausilio dei diversi siti che entrano nel processo svolgendo query parallele.

Ricapitolando abbiamo che i vantaggi dei DDBMSs sono: riflette la struttura organizzativa migliorando determinate caratteristiche tra cui l'abilità di condivisione, autonomia locale, la disponibilità, l'affidabilità e la performance, riducendo il costo dell'hardware grazie anche ad una crescita modulare . Gli svantaggi sono: architettura e progettazione più complessa e costosa; la sicurezza e l'integrità sono più difficilmente controllabili; c'è una mancanza di standard ed esperienza nel suo utilizzo.
Ci sono 3 tipi di architetture nei DDBMS:

- Shared Everything: dominata dal mercato delle architetture fino al 2000(circa), abbiamo un grande e costoso database, che condivide ogni cosa, per cui molto lento; è il sistema centralizzato.
- Share Disk: abbiamo i dati salvati su diversi dischi connessi tra loro. I dischi sono tutti connessi insieme e comunicano con tutte le CPUs.

- Share Nothing: ogni cosa è separata, ogni disco ha la propria CPU e non comunica con le altre CPU o dischi; solo alla fine i processori vengono connessi insieme per combinare i risultati da ognuno di essi; Il modello è adottato dai modelli NoSQL; è molto facile lo scale out.

Shared Disk	**Shared Nothing**
Quick adaptability to changing workloads	Can exploit simpler, cheaper hardware
High availability	Almost unlimited scalability
Performs best in a heavy read environment	Works well in high-volume, read/write environment
Data need not to be partitioned	Data is partitioned across the cluster

Cosa scegliere tra Scalability o Availability? Dipende da quanto vogliamo spendere: più availability significa più denaro. Per esempio una disponibilità del 100% significa nessun down del sistema; diminuendo la disponibilità e quindi aumentando i down, potremmo avere disponibilità 99% con down compresi tra le 9 - 88 ore.
Replication:
Un log è un file sequenziale che è storato in una memoria stabile (che non dà mai errore; esso archivia tutte le attività realizzate dalle transazioni in ordine cronologico. Ci sono due tipi di record: transaction logs(operations) e system event (Checkpoint and Dump). Possiamo definire un checkpoint, quel momento in cui noi memorizziamo un set di transazioni in un dato tempo, mentre un dump è una copia piena dello stato di un DB in una memoria stabile; la sua esecuzione è offline, e genera un backup. Solo dopo che il backup è completo, che il dump recod viene scritto nel log.
Le strutture di replica sono:

- One2Many: una sorgente e alcuni target.
- Many2One: alcune sorgenti e un solo target.
- Peer2Peer: i dati sono replicati attraverso molteplici nodi che comunicano l'uno con l'altro.
- Bi-directional: concettualmente un Peer2Peer con solo due Peers.
- Multi-Tier Staging: la replicazione è divisa in più stages.

Per creare una replica possiamo:

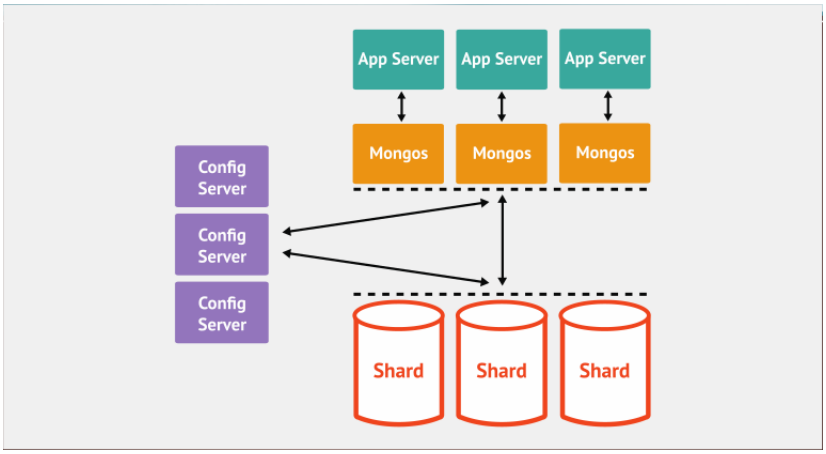
- Prendere i dati dal master, sia il backup o i dati stessi (se vogliamo stoppare l'attività del server) e muoverli (potrebbe essere utile se i dati non sono accessibili dall'esterno).
- use log file : per mantenere la replica uguale alla sorgente durante il trasferimento (quasi in tempo reale) si usa il Log per eseguire gli stessi comandi eseguiti sulla sorgente.

2.2 MongoDB's Approach: Sharding[4]

MongoDB usa lo Sharding per splittare una grande collezione in diversi server (chiamati cluster); ogni documento ha una chiave in mongodb, noi definiamo una chiave che definisce il range dei dati (chunk range). Gli spazi nella chiave sono come punti su una linea, mentre il range è il segmento di quella linea. Esempio: la chiave è il cognome, quindi decidiamo

che tutti i cognomi che iniziano tra A e K vengano salvati in una partizione mentre gli altri in una seconda. MongoDB non fa lo shard automaticamente una volta che abbiamo deciso di distribuire i dati. Di default la massima dimensione del chunk è di 64Mb; se tende ad aumentare la dimensione del chunk, MongoDB automaticamente lo splitterà in due più piccoli chunk, e se gli shard diventano sbilanciati, allora alcuni chunk migreranno ad altri shard cercando di correggere questo sbilanciamento. L'obiettivo del bilanciamento non è solo quello di mantenere ugualmente distribuiti i dati, ma è anche quello di minimizzare l'ammontare dei dati trasferiti. Il bilanciamento è piuttosto pigro, quindi non si attiva fino a quando i dati non sono molto squilibrati; lo fa per cercare di evitare lo spostamento di dati avanti e indietro, nel caso in cui dovesse bilanciare ogni minima differenza (si sprecherebbero troppe risorse)
Quando crei uno shard, MongoDB crea un singolo chunk con range $(-\infty, +\infty)$ dove con $-(+)\infty$ intendiamo il più piccolo/massimo valore che MongoDB può rappresentare. Da qui, se la dimensione del chunk è più grande di quella scelta, allora verrà splittata; ogni chunk range dovrà essere distinto e non sovrapposto, e inoltre la loro unione dovrà coprire il range iniziale.
Lo shard è in pratica un server, quindi bisogna usare mongos (s=server) per fare una query. Mongos è il punto di interazione tra l'utente e il cluster, che ti permette di trattare un cluster come un singolo server. Le query vengono instradate verso l'apposito shard grazie a mongos. Per le query senza target (es: with sorting) una richiesta è mandata a tutte le shard, e le query sono elaborate localmente; dopo che il risultato è ritornato, mongos mergia il risultato sorted e manda al client il risultato finale. Mongos attualmente non archivia alcun dato; la configurazione di un cluster è mantenuta in uno speciale mongods, chiamato config server, che tiene le informazioni circa gli accessi di chiunque a quel cluster. Il cluster consiste in tre tipi di processi:

- The shards: un nodo del cluster, può essere sia un singolo mongod che un set replica. MongoDB in automatico replica i dati in una shard dove avremo il chunk primario (con i dati storati che volevamo), e un chunk secondario dove vengono reolicati i dati come backup (è solo in lettura, non si possono fare query).
- Mongos processes: serve per guidare le richieste verso la posizione dei dati, e agisce come bilanciamento dei chunks. Non contiene dati locali. Possono esserci uno o più processi. Dopo che il bilanciamento è finito, il processo aggiorna anche il Config Servers con la nuova posizione e intanto il mongos che sta bilanciando i chunks, tira fuori un "balancer-lock", in modo che nessun altro mongos incomincerà il bilanciamento, che era già ultimato. Il lock è rilasciato solo dopo che il chunk è stato copiato, e il vecchio chunk cancellato dallo shard iniziale.
- Config servers: per mantenere traccia dello stato dei cluster. Archivia il chunk range e la posizione dei cluster. Si possono avere solo 1 o 3.



Let's see the steps for the configuration:

1. Start a config server, it start by default at port 27019:
   mongod –configsvr
2. Start the mongos Router, we need to initialize it like this even if the cluster is already running:
   For 1 configuration server: mongos –configdb <hostname>:27019
   For 3 configuration servers: mongos –configdb <host1>:<port1>,<host2>:<port2>,<host3>:<port3>
   We can have different mongos on same pc on different ports.
3. Start the shard database, it starts a mongod with the default shard port 27018
   mongod –shardsvr
   The shard is not yes connected to the rest of the cluster and it may have been already running in production.
4. Add the Shard:
   On mongos: sh.addShard('<host>:27018')
   To add a replica set: sh.addShard('<rsname>/<seedlist>')
5. Verify that the shard was added:
   db.runCommand({ listshards:1 })
   Obtaining as result:
   {
   "shards":
   [{ "_id":"shard0000", "host":"<hostname>:27018"}],
   "ok" : 1
   }

To enable sharding on a database simply use the command: sh.enableSharding("<dbname>")
To start a collection with the given key: sh.shardCollection("<dbname> .people",{"country" : 1})
Le proprietà di una Shard Key sono:

- Shard key è immutabile
- Shard key values sono immutabili

- Shard key deve essere indicizzato
- Shard key sono limitati a dimensioni massime di 512 bytes
- Shard key sono usate per guidare le query: scegliendo un campo comunemente usato in query
- Solo la shard key può essere unica tra le shards (the '_id' field is only unique within the individual shard)

2.3 HBase

We can consider HBase tables as potentially massive tabular datasets that are implemented on disk by a variable number of HDFS files called Hfiles. All rows in an HBase table are identified by a unique row key. A table of nontrivial size will be split into multiple horizontal partitions called **regions**. Each region consists of a contiguous, sorted range of key values (reminding of the MongoDB range-based sharding scheme).

Read or write access to a region is controlled by a **RegionServer**. There will usually be more than one region in each RegionServer. As regions grow, they split into multiple regions based on configurable policies (may also be split manually).

Each HBase installation will include a Hadoop **Zookeeper service** that is implemented across multiple nodes. Hbase may share this Zookeeper ensemble with the rest of the Hadoop cluster or use a dedicated service.

The HBase **master server** performs a variety of housekeeping tasks. In particular, it controls the balancing of regions among RegionServers. If a RegionServer is added or removed, the master will organize for its regions to be relocated to other RegionServers. An HBase client consults Zookeeper to determine the location of the HBase catalog tables, which can be then be interrogated to determine the location of the appropriate RegionServer. The client will then request to read or modify a key value from the appropriate RegionServer. The RegionServer reads or writes to the appropriate disk files, which are located on HDFS.

Summing up, the four major components (nodes) of HBase are:

1. The HMaster (only one): It is the implementation of Master server in HBase architecture. It monitors and coordinates all slaves in the cluster. It must assign regions, detect failures and has admin privileges.
2. The HRegionServer (many of them): It manages the data regions and serves client requests for reads and writes (using a log), this request is assigned to a specific region, where actual column family resides. However the client can contact directly the HRegionServer without having a permission of HMaster. HMaster is required only for operations related to metadata and schema changes. The Region servers run on Data Nodes present in the Hadoop cluster.
   The HRegions are the basic building elements of HBase cluster that consists of the distribution of tables, i.e., a subset of a table's rows, like horizontal range partitioning and it is done automatically.
3. The HBase client
4. Zookeeper: It is a centralized monitoring server which maintains configuration information and provides distributed synchronization between nodes. HBase lives and

depends on ZooKeeper, by default HBase manages the ZooKeeper instance (start and stop). HMaster and HRegionServers register themselves with ZooKeeper. If the client wants to communicate with regions, the servers client has to approach ZooKeeper first. During a failure of nodes ZooKeeper Quoram will trigger error messages, and it starts to repair the failed nodes.

HBase's architecture is similar to MongoDB's and they share the same flaw: If the master is broken the client cannot obtain any data and we have an error (Single point of failure).

2.4 Cassandra

In Cassandra there are no specialized master nodes. Every node is equal and every node is capable of performing any of the activities required for cluster operation. [1] It uses a peer-to-peer distributed system. The Data is partitioned among all nodes in the cluster and a custom data replication to ensure fault tolerance and since all the nodes are all considered same we have Read/Write-anywhere design. Cassandra is based on Google BigTable and Amazon Dynamo so it inherits their properties.
In Cassandra we can add or remove nodes with no downtime so we have a transparent elasticity and transparent scalability (performance grows linearly with the number of nodes) and due to it's P2P architecture we obtain also the High Availability. If a node were to fault we will have that we could obtain the data from the replicas in the other nodes. The Nodes are logically structured in Ring Topology, the hashed value of key associated with data partition is used to assign it to a node in the ring. The hashing rounds off after a certain value to support the ring structure and the lightly loaded nodes moves position to alleviate the highly loaded nodes. Cassandra uses ZooKeeper to find the replicas in other nodes, given a node we decide the N number of following nodes in which the replica will be saved.

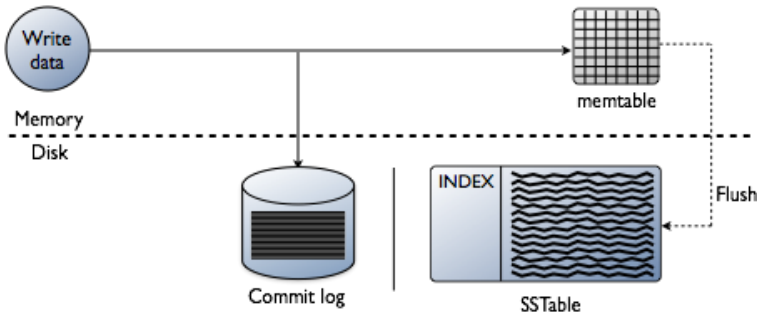


Figure 1: Data writing procedure in Cassandra.

As illustrated in Fig. 1, The data writing is separated in 3 stages :

[1] Nodes in Cassandra do, however, have short-term specialized responsibilities. For instance, when a client performs an operation, a node will be allocated as the coordinator for that operation. When a new member is added to the cluster, a node will be nominated as the seed node from which the new node will seek information. However, these short-term responsibilities can be performed by any node in the cluster.

1. The log file writing, which is essential: even if the data writing fails it doesn't matter because we have the log file.

2. Once written in the commit log, data is written to the mem-table. Data written in the mem-table on each write request also writes in commit log separately. Mem-table is a temporary storage of data in the memory while commit log logs the transaction records for backup.
   The first place where the read operations take place is mem-table and several may exist at once (1 current and other waiting to be flushed).

3. When mem-table is full, flush the data from the mem-table and store them disk in SSTables. The SSTables are immutable once written.

The consistency can be based on majority: if a certain number,which we decide(it can also be one), of the nodes agree that a certain data was written or read it is assumed to be true;
or it can be based on quorum which states the 50%(of the replication factor) + 1 nodes agree that something is written it's assumed true.
The commands for writing for write consistency one(quorum):
INSERT INTO table (column1, ...) VALUES (value1, ...) USING CONSISTENCY ONE (QUORUM)
If a node is offline, an online node makes a note to carry out the write once the node comes back online. For reading the data we have the following command: SELECT * FROM table USING CONSISTENCY ONE
To delete the data is it easier to consider the data not available and mark it for deletion, it is called a tombstone(like how recycle bin works in OS), and actually delete on a major compaction or configurable timer.
Compaction runs periodically to merge multiple SSTables reclaiming space, creating new index, merging the keys, combining columns and discarding the tombstones.

One of the advantages of a master node is that it can maintain a canonical version of cluster configuration and state. In the absence of a master node keeping a canonical version of cluster configuration, Cassandra requires that all members of the cluster be kept up to date with the current state of cluster configuration and status. This is achieved by use of the **gossip protocol**. Every second each member of the cluster will transmit information about its state and the state of any other nodes it is aware of to up to three other nodes in the cluster. One nodes gossip about other nodes as well as about their own state. This architecture eliminates any single point of failure within the cluster.[2]

One of the main reasons of gossip is related to node availability. Usually the only way to detect node failure is a system of heartbeats between nodes. However (in a distributed system) the heartbeats may be lost because of network issues rather than actual node failure. Cassandra failure detection is more probabilistic: nodes in the cluster become increasingly "worried" about other nodes, and if it seems likely that a node is down the operations will be redirected.

[2]Although distributed databases with master nodes have strategies to allow for rapid failover, the crash of a master node usually creates a temporary reduction in availability, such as momentarily falling back to read-only mode.

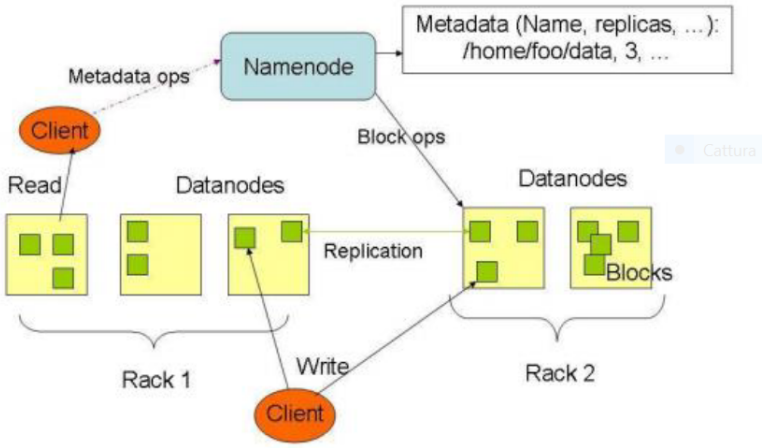
Cassandra distribute data throughout the cluster by using **consistent hashing**. The rowkey (analogous to a primary key in an RDBMS) is hashed. Each node is allocated a range of hash values, and the node that has the specific range for a hashed key value takes responsibility for the initial placement of that data. We usually visualize the cluster as a ring: the circumference of the ring represents all the possible hash values, and the location of the node on the ring represents its area of responsibility. When adding a new node, the system remaps all the hash ranges (exploiting a peculiar mechanism of Virtual node to optimize this operation)[3].

3 Big Data Architecture

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". Ad oggi sappiamo però che per definire i "Big Data" potremmo usare più di 3 V, aggiungendo anche variability e veridicity. 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 cui vantaggi sono già stati esposti nella sezione dedicata all'architettura distribuita. 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 (scale out), 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".

3.1 HDFS

L'architettura HDFS (Hadoop Distributed File System) è un file system distribuito progettato per girare su hardware base (commodity hardware).



HDFS Architecture

[3]More infos in the suggested course text-book

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, inoltre 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 e in un server master che gestisce il file system detto NameSpace e che regola gli accessi ai file da parte dei clients (secondo modello di accesso $Write-once-Read-many$). 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, e reindirizza le richieste di operazioni di lettura e scrittura da parte del file system dei clients ai DataNodes. 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.

3.2 Map Reduce

Map Reduce è un motore di computazione distribuito. Ogni programma è scritto in stile funzionale ed è eseguito in parallelo. "Mapred" risolve problemi legati alla gestione di processi di gestione/processing di BigData (ad esempio distributed pattern-based searching, distributed sorting) quali:

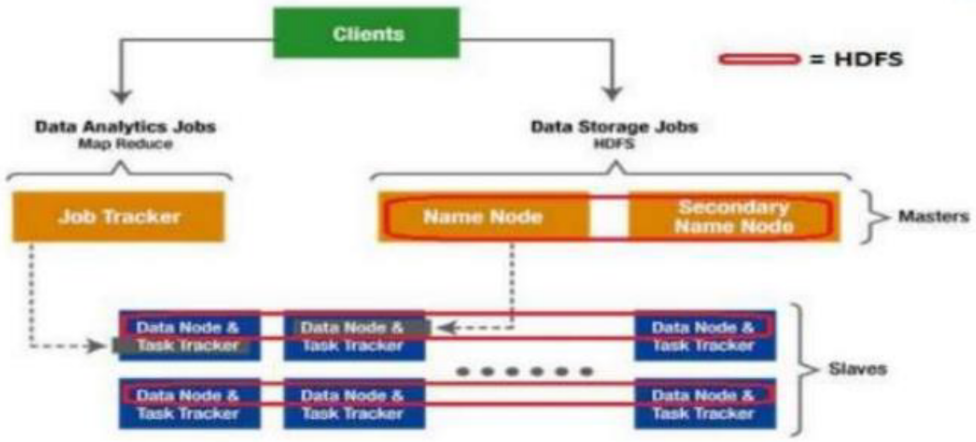
- Come assegnare i 'lavori' ai singoli 'worker'
- Cosa succede se ci sono più 'lavori' che 'worker'
- Cosa succede se i 'worker' devono condividere risultati parziali
- Come aggregare i risultati parziali
- Come scoprire se tutti i 'worker' hanno finito il proprio task
- Cosa succede se un 'worker' muore

Map Reduce

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 (`map (k, v) -> [(k', v')]`) e di reduce (`reduce (k', [v']) -> [(k', v')]`). Tutte le altre attività le svolge *MapRed*.

3.3 Hadoop

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



Schema ecosistema Hadoop

Main features[4]

[4]From "Next Generation Databases", Guy Harrison.

- It is an economical scalable storage model. As data volumes increase, so does the cost of storing that data online. Because Hadoop can run on commodity hardware that in turn utilizes commodity disks, the price point per terabyte is lower than that of almost any other technology.

- Massive scaleable IO capability. Because Hadoop uses a large number of commodity devices, the aggregate IO and network capacity is higher than that provided by dedicated storage arrays in which smaller numbers of larger disks are provided by even smaller numbers of processors.[5]

- Reliability: Data in Hadoop is stored redundantly in multiple servers and can be distributed across multiple computer racks. Failure of a server does not result in a loss of data; in fact, a Hadoop job will continue even if a server fails—the processing simply switches to another server.

- A scalable processing model: MapReduce.

- Schema on read: Data can be loaded into Hadoop without having to be converted to a highly structured normalized format. This makes it easy for Hadoop to quickly ingest data from various forms. The imposition of structure can be delayed until the data is accessed; this is sometimes referred to as schema on read, as opposed to the schema on write mode of relational data warehouses.

3.3.1 The Hadoop Ecosystem

L'ecosistema di Hadoop include una famiglia di utiliy e applications sempre in espansione, costruita sopra o progettata per lavorare con il core di Hadoop (Yarn, HDFS).

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.

[5]Furthermore, adding new servers to Hadoop adds storage, IO, CPU, and network capacity all at once, whereas adding disks to a storage array might simply exacerbate a network or CPU bottleneck within the array.

Vengono rilasciati inoltre altri applicativi quali Accumulo, Hbase e Spark [6], Kafka (distributed streaming messaging system), Oozie (workflow scheduler that allows complex workflows to be constructed from lower level jobs). In Cloudera sono presenti anche Impala, Mahout (machine-learning framework), Sqoop (utility per scambiare dati con relational databases, in import o export) e Flume (utility for loading file-based data (i.e. web server logs) into HDFS)

3.4 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.

Un data lake invece è una struttura capace di contenere un enorme quantità di dati salvati in ogni formato, in maniera non costosa. È una soluzione infrastrutturale in cui lo schema e i requisiti dei dati non sono definiti in partenza, ma vengono delineati al momento dell'interrogazione (*query time*): si tratta dello *schema on read*, caratteristico di un approccio "bottom up", caratterizzato dall'osservazione "sperimentale" come motore di un processo induttivo. Da questo "lago" ogni soggetto (autorizzato) può attingere, tipicamente passando per un processo di analisi e di campionamento accurato. Può avere lo scopo di catturare tutti i dati forniti in fase di "ingestion", e di passare solo quelli ritenuti rilevanti ad una Enterprise Data Warehouse (EDW): può essere quindi una fonte di dati per questa EDW. In questo modo si risparmia risorse relative all'EDW, e permette un'esplorazione iniziale dei dati disponibili, senza modellazione da parte dell'EDW team (*quick user access*). Una delle caratteristiche salienti del Data Lake è inoltre la facile scalabilità. As reported in Fig. 2, components of a typical data lake architecture are[7]:

- The data is first loaded into a **transient loading zone**, where basic data quality checks are performed using MapReduce or Spark by leveraging the Hadoop cluster.
- Once the quality checks have been performed, the data is loaded into Hadoop in the **raw data zone**.
- An organization can, if desired, perform standard data cleansing and data validation methods and place the data in the **refined zone**.
- From the trusted area, data moves into the **discovery sandbox**, for wrangling, discovery, and exploratory analysis by users and data scientists.
- Finally, the consumption **analytic zone** exists for business analysts, researchers, and data scientists to dip into the data lake to run reports, do "what if" analytics, and otherwise consume the data to come up with business insights for informed decision-making.

[6] If we think about Hadoop as a disk-oriented framework for running MapReduce-style programs, Spark represents a memory-oriented framework for running similar workloads

[7] See also `http://www.oreilly.com/data/free/files/architecting-data-lakes.pdf`. Note a slightly different zone denomination, compared to Maurino's Slides.

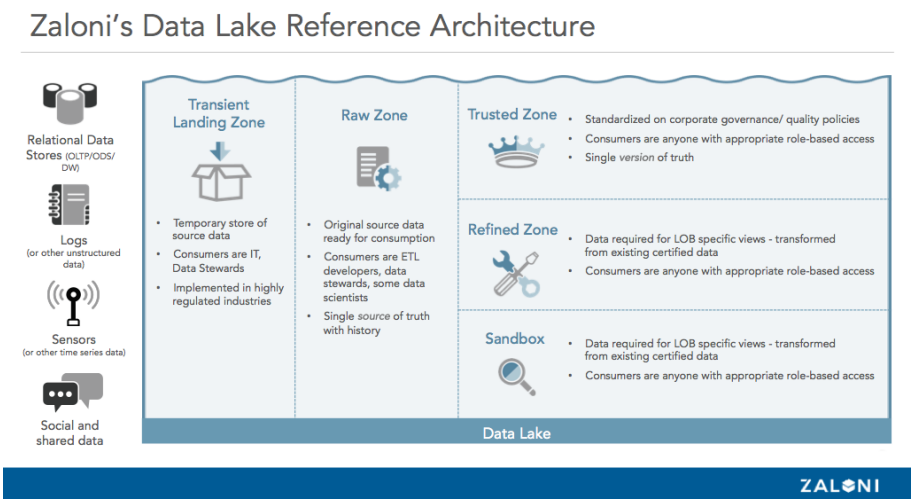
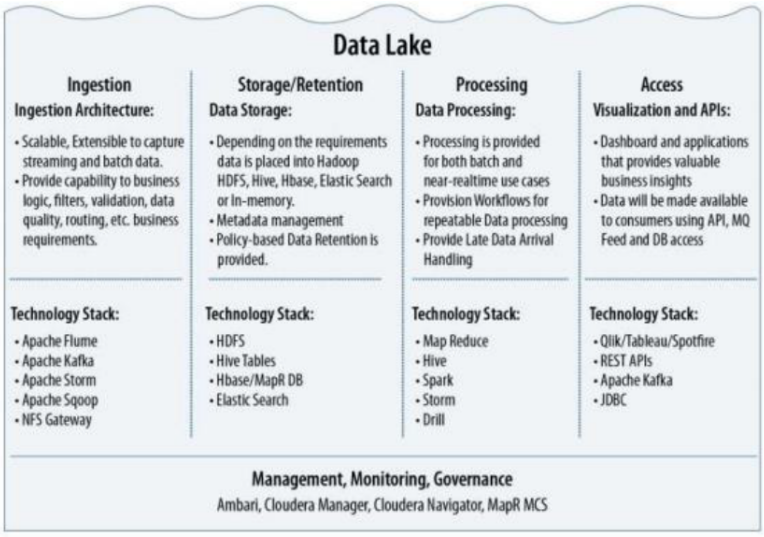


Figure 2: Typical data lake arch.



Schema riassuntivo Data Lake

Riassumento, keywords per i data lake sono:

- 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).
- Storage, basata su un'architettura HDFS.
- 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.

4 Data quality and integration

Fasi molto importanti nell'analisi dei dati sono: la "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).

The complete methodology of schema integration is composed of three possible steps:

- Schema transformation (or Pre-integration)
  Input: n source schemas
  Output: n source schemas homogeneized
  Methods used: Model transformation + Reverse engineering

- Correspondences investigation
  Input: n source schemas
  Output: n source schemas + correspondences
  Method used: techniques to discover correspondences (identifying semantic relativism)

- Schemas integration and mapping generation
  Input: n source schemas + correspondences
  Output: integrated schema + mapping rules btw the integrated schema and input source schemas

Method used: New classification of conflicts + Conflict resolution transformations: in particular, type of conflicts are

- $Classification\ conflicts$: corresponding elements describe different sets of real world objects
  Resolution: Introduction of a Generalization/Specialization hierarchy.
- $Descriptive\ conflicts$: corresponding types have different properties, or corresponding properties are described in different ways.
  Custom Resolutions depending on the specific type of conflict.
- $Structural\ Conflicts$: different schema element types, e.g.: class, attribute, relationship.
  Resolution: choose the less constraining structure.
- $Fragmentation\ Conflicts$: the same phenomenon of the real world is perceived as a single S1 object in one database and as several objects in the other
  Resolution: aggregation relationships.
- $InstanceLevel\ Conflicts$: same data in different sources have different values.
  Resolution: user defined $resolution\ function$ (taking two or more conflicting values of an attribute as input and outputs the result to the posed query). Common resolution functions are MIN and MAX. For non numerical attributes a typical solution is $CONCAT$.

4.1 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.

4.2 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 ($Pass\ It\ on$).
- $Conflict\ Avoiding$, applica una decisione unica per *tutti* i dati in conflitto, tipicamente scegliendo una delle fonti come "la più affidabile" e creando il record rappresentativo da quella fonte ($Trust\ Your\ Friends$ strategy).

- *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).

References

[1] Alex Ceccotti. "riassunto 2017/18". In: (2017).

[2] Jyoti Celestia. *TutorialsPoint*. URL: `http://www.tutorialspoint.com`.

[3] Kristina Chodorow. *MongoDB: The Definitive Guide*. Second Edition. United States of America: O'Reilly. ISBN: 978-1-449-34468-9.

[4] Kristina Chodorow. *Scaling MongoDB*. United States of America: O'Reilly, ISBN: 978-1-449-30321-1.

[5] guru99. *guru99*. URL: `http://www.guru99.com`.

[6] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly. ISBN: 978-1-449-39041-9.

[7] Ian Robinson. *Graph Databases*. Second Edition. O'Reilly, ISBN: 978-1-491-93200-1.