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 prima di tutto da un modello concettuale (teoria) e poi da un linguaggio (anche grafico a volte) per descrivere e interrogare i dati. Un data model è sempre disponibile in un databasse management system 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. Molto rigido come regole.
2. Il RDBMS sfrutta le proprietà ACID :
    - A : Atomicity : serve affinché l'operazione o avviene su tutti i dati o non avviene, ad esempio se vi è un aggiornamento nei dati questo aiuta affinché i dati non vengano aggiornati solo parzialmente.
    - C : Consistency : I dati o il nuovo dato che viene aggiunto rispetta lo schema prestabilito. E se si viola la consistenza con un'operazione tutta l'operazione fallisce.
    - I : Isolation : gestisce come l'integrità delle transazioni sono viste dall'utente. Inoltre garantisce che durante un'operazione (query) non vengano svolte altre operazioni, e che lo stato del database venga modificato solo prima della fine della query.
    - D : Durability : garantisce che una transazione effettuata sopravvivrà per sempre, anche se il sistema crasha. Dovuto grazie ai server di backup e log files.

Il RDBMS esiste già da 35 anni quindi è ben sviluppato e molto conosciuto, molti dati sono ancora salvati in questo formato ed è efficace ancora per molte operazioni. Gli aspetti limitanti sono:

1. un attributo può avere solo un valore.
2. non è compatibile con molti linguaggi moderni.
3. molto rigido come linguaggio
4. non accetta i loop.
5. Difficile modificare le tabelle.
6. La performance.

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

- Numero delle righe
- Tipo di operazione
- Algoritmo scelto
- La struttura dati scelta

Per Scaling Up intendiamo potenziare le macchine, mentre per Scaling out intendiamo aggiungere macchine, per i RDBMS è più facile Scale up che Scale out. Il costo è un altro dei problemi, installare il software richiede un costo molto alto e hardware molto complesso. Inoltre se continuiamo a aggiungere server (scaling out) il prezzo dell'Hardware aumenta esponenzialmente mentre il guadagno complessivo sul tempo di risposta scende asintoticamente.

1.2 NoSQL

Di fronte a questi svantaggi sorge una nuova "tecnologia": i NoSQL(Not Only SQL). Le proprietà dei NoSQL sono:

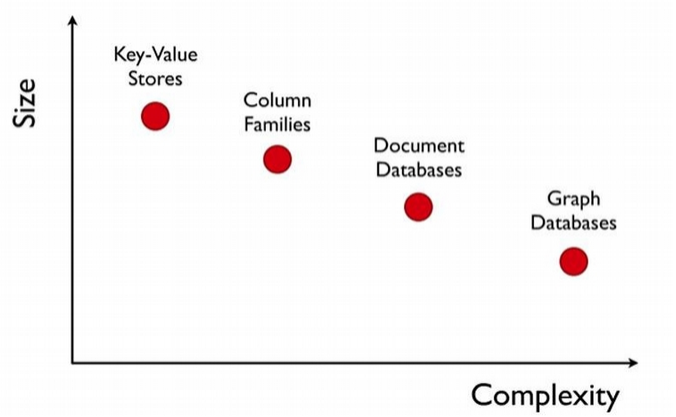
1. Non hanno nessuno schema o modello prefissato
   - A differenza dei modelli SQL nei quali bisogna prima definire il modello, nel caso NoSQL non esiste un modello rigido.
   - Per aggiungere un nuovo attributo non vi è bisogno di cambiare il modello a differenza dei SQL.
   - I modelli NoSQL seguono 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 (solo ciò che è noto come vero è vero)
2. Segue il teorema CAP:
   - E' impossibile per un sistema informatico distribuito(vuol dire sistemi interconnessi tra loro e la comunicazione avviene solo attraverso messaggi) fornire simultaneamente le tre garanzie (infatti si possono soddisfare solo 2 di esse):
     (a) **(C)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 cioè non quello più recente.
     (b) **(A)**vailability(Disponibilità) : La garanzia che ogni richiesta ottenga una risposta su ciò che è fallito e ciò che ha avuto successo. Se è assente aspetterò per lunghi tempi senza ricevere una risposta.
     (c) **(P)**artition Tolerance(Tolleranza sulle partizioni) : Il sistema funziona anche dopo aver perso un numero arbitrario di pezzi del sistema.
   - I sistemi RDBMS sono dei software CA, ed è possibile creare dei modelli RDBMS basati sul CAP
   - I sistemi NoSQL sono dei sistemi solitamente CP o AP.
   - Uno preferisce la disponibilità sulla coerenza, perché è meglio vedere un vecchio dato non coerente che vedere un errore di fallimento di caricamento dei dati.
3. Segue Principio BASE:
   - Basic Availability : completare una richiesta anche se parzialmente consistente, ad esempio nel caso di fallimento. E' possibile fare ciò grazie al fatto che usa server sparsi ovunque con un grado di replicazione del database e in caso di malfunzionamento del database richiesto non tutto il sistema cede la disponibilità.
   - Soft State : Abbandonano la richiesta di consistenza dei ACID praticamente completamente, cioè i dati possono avere schemi diversi.
   - Eventual consistency : I sistemi NoSQL richiedono che a un certo punto i dati convergeranno a uno stato consistente (non si fa garanzie sul quando), e quindi prima di allora ho una consistenza ritardata cioè prima del momento dello stato consistente posso ricevere come risposta qualunque valore come risposta di una query.

ACID	BASE
• Forte Coerenza • Poca disponibilità • Pessimo "Multitasking" • Complesso	• Coerenza debole • Disponibilità è la cosa principale e sacrifica per questo (CAP) • Veloce e Semplice

Esempi e tipologie di modelli NoSQL :

1. Key Value (Dynamo, Voldemort, Rhino DHT) : Sono delle tabelle con chiavi che si riferiscono/puntano a un certo dato, è molto simile a 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 ragruppato in colonna.
3. Document based (CouchDB, MongoDB) : Di solito salvati con file JSON, salvati come una <chiave – valore>. E' facile ricercare dati in questo formato. JSON è basato su due strutture :1) chiave – valore per gli oggetti e 2) lista ordinata di elementi.
4. Graph based (Neo4J, FlockDB) : Uso i vertici/nodi e archi per rappresentare i dati e legami tra di loro. E' difficile fare scaling con i grafi per quanto riguarda l'immagazzinamento ma è molto rapido nelle query.

Dobbiamo sacrificare o la dimensione del database o la sua non complessità:



1.2.1 Document Based[MongoDB]

MongoDB is -as already mentioned- a document based management system, with the data stored in Bson (Binary Json), and the access to data is possible thanks to indexes. Compared to the SQL DBs, Mongo doesn't have the join feature. The name changes are:

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

In case of a massive data upload one might decide to do a few changes in the process to fasten it up:

- Disable the acknowledgment(ack) of the data, which is a signal passed between communicating processes, computers or devices, to signify acknowledgment or receipt of message, as part of a communications protocol.
- Disable the writing on a log file

One must be attentive when doing so because any loss will not be registered and lost forever.
For large data-sets it is useful to use some additional structures called indexes: those are similar to the book indexes and act as a faster way to retrieve information. They might require more time during the insertion but makes the queries faster later. One primary index (basic one) is always the defined for the id.

How does it exactly works? Basically without an index Mongo will perform a simple table scan (like SQL) in which it has to look through all the "book" to find a query result. The analogy with the book is if we don't have the index for our books we will have to read it whole until finding the point where we wanted to be. Indexing avoids this problem (more problematic if the database is large), providing an ordered list that points it's content. Since indexing slows down the modifications one must choose just a couple of indexes for any given collection, the tricky part is to decide which one. MongoDB gives, by default, a limit of 64 indexes!

The aggregation uses pipeline and options. The aggregation pipeline starts processing the documents of the collection and passes the result to the next pipeline in order to get result for example: $match and $group. We can use the same operator in different pipelines.

For many (mainly commercial) reasons, Mongo offers also a SQL interface; we need the connector BI (Business Intelligence): it generates the relational schema and use such schema to access the data.

1.2.2 GraphDB[Neo4J]

A graph is a collection of nodes and edges which represents their relations. It has a lot of applications such as: social media, recommendations, geo, logistics network, financial transaction graphs (for fraud detection), master data management, bioinformatics, authorization and access control.
The labeled property graph model has the following characteristics:

- It contains **nodes** and **relationships**;
- Nodes contains **properties** (key : value pairs);
- Nodes can be labeled with one or many **labels**;
- Relationships are named and directed, with a start and end node;
- Like nodes, also Relationships can contain properties.
- The relationship can be **fine-grained** or **generic**:
  let's consider the *Address* relationship case: we can distinct between:
  - fine-grained relationship like `HOME_ADDRESS` or `WORK_ADDRESS` or `DELIVERY_ADDRESS`
  - or we could choose only `ADDRESS` and specify which kind of address it is `ADDRESS{type:'home'}`. This method is called generic relationship.

  Usually the generic one is preferred, especially in cases where I need to find all the address of a client: all I need to do is find the `ADDRESS` relationship, whereas in the fine-grained case I would need to find one by one all kind of addresses (just imagine if we had 100 types of addresses!). On the other hand to find the `DELIVERY_ADDRESS` all I need to do is to identify the `ADDRESS{type:'delivery'}`.

A Graph Database (GD) can use either the native or non-native storage and processing engine.
Native Graph Storage is optimized for the native graph management, whereas the non native graph storage treats the data in non graph based model while still supporting a graph query language. Examples: Relational, Object oriented DB, Wide Column. In a relational for example in a join bomb we can use a graph to connect two tables. The problem with relational DBs and most of NoSQL DBs is the lack of relationships. Moreover in SQL joining tables adds more complexity, and especially in case of sparse table with null-able column the operation requires special checking in the code. In the case of a market, just to see what a customer bought,

we need to do a lot of expensive joins on the customers that buy a specific product with other products for the recommendation systems. For the NoSQL DBs, whether key-value, document or column-oriented, we might use the aggregation technique to see the relationship, but the relationship between aggregates aren't citizen in the data model. Indeed the aggregation is a costly operation since it doesn't exploit index free adjacency (the data is physically near) and since they stay inside of aggregates with structure in form of nested maps and even after the aggregation there is no back link to point backward and run other interesting queries.

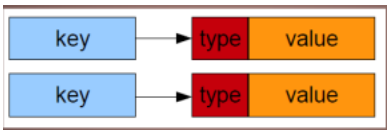
In a native graph storage the attributes and nodes and the referenced nodes are stored together, to optimize the graph processing engine. Performing a query the graph-model, the response time does not strictly depend on the total number of nodes (it remains nearly constant) because the query is processed locally to the portion of the graph which is connected to the base node, while the other SQL and NoSQL models will suffer in performance speed with the increase of data. Moreover we can add more data/nodes and relationship without disturbing the already existing model.
The processing engine uses index-free adjacency, meaning that connected nodes physically points to each other in the DB, this makes sure that the retrieval is faster but it comes at a cost: the efficiency of queries that do not use graph traversal, for example the writing time etc.
It's important to note that it's neither good nor bad to use native or non-native engine, simply one needs to make a choice based on his/her needs.
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.:



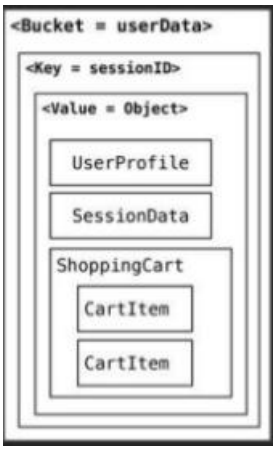
An example is the amazon's cart system which uses DynamoDB, a key-value system.

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.
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.

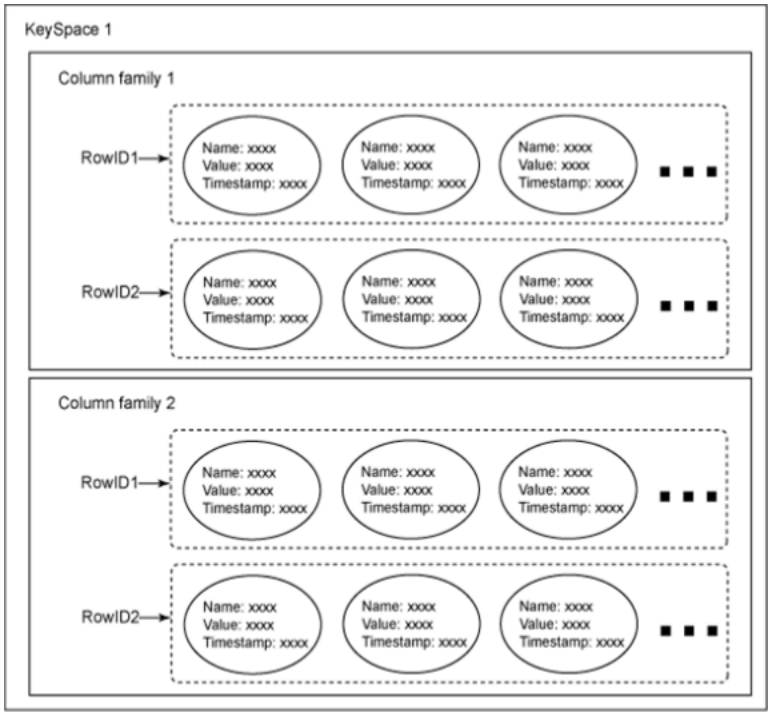
BigTable is a multidimensional map, which can be accessed by row key, column key and a timestamp. It is sorted, persistent and sparse. We will consider the **HBase BigTable** Model. The data is organized in tables, each table(the tables are multi-versioned) is composed by the column-families that include 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

A centralized DB system is a system where the database is located, stored and maintained in a single location. If the processor fail all the system fails. If we need data periodically in different location we can distribute it or replicate it. The distribution is to divide the data and have it stored in different places, it provides opportunities for parallel execution. On the other hand we can also replicate it or replicate part of data in different location improving also the availability.
In a Homogeneous Distributed Database all sites have identical software and aware of each other and agree to cooperate in processing user requests. They surrender their autonomy to change the schema appearing to user as a single system.
In a Heterogeneous Distributed Database different sites may use different schema and software which creates a major problem for query and transaction processing. The sites may not be aware of each other and may

provide only limited facilities for cooperation in transaction processing.
The challenges of a distributed database design is deciding what goes, which depends on the data access pattern, and the problem of where to allocate fragments to nodes. With replication we have that system maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance and with the fragmentation the data is divided into several fragments stored in different sites. We can combine both of them so the data is partitioned into several fragments and the system maintains several identical replicas of such fragment.
Dividing/fragmenting our database into n databases, the bottle-neck phenomenon arises, in which the slowest machine dictates the worst performance for a query.
The advantages of replication are:

- Availability: Since the failure of a site does not result in unavailability of it's data since it's replicas exist elsewhere.
- Parallelism: Queries may be processed by several nodes in parallel thus faster.
- Reduced data transfer: Since replicas of the data may exist in the node itself.

While the disadvantages are:

- increased cost of updates: Since each replica must be updated.
- increased complexity of concurrency control e.g. two people book a ticket at the same time solution. We may have that distinct replicas may lead to inconsistent data unless some special concurrency mechanisms are implemented. A solution could be to choose one copy as the primary copy and apply concurrency control operation only on the primary copy.

As already mentioned we can combine data replication with data fragmentation. In particular data fragmentation consists in a division of relation r into fragments which contain sufficient information to reconstruct relation r. Now for simplicity imagine the database to be relational. The fragmentation can be horizontal or vertical, but it must contain sufficient information to reconstruct the original "relation". In the horizontal each tuple of the data is assigned to one or more fragments while in the vertical the schema (in the case of relational) is split into several smaller schemas, and all of them must contain a common candidate key to ensure lossless join for example using a tuple-id attribute.
The advantages are:

- Horizontal:
  - Allows parallel processing on fragments.
  - Allows the data to be split so the tuples are located where they are more frequently accessed.
- Vertical
  - Allows tuples to be split so that each part of tuples are stored where they are more accessed.
  - tuple-id allows efficient joining of vertical fragments.
  - Allows parallel processing on a relation.

The two fragmentation can be mixed together and the fragments may be successively fragmented to an arbitrary depth. Data Transparency is very important because it indicates the degree to which the users are not aware on the way the data is stored in a distributed system. We have to consider transparency issues in relation to: fragmentation, replication and location of our data.
Considering the difference between the centralized and the distributed system from the point of view of costs we have that:

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, instead, other issues must be taken into account: in first place we have to consider the cost of a data transmission over the network and then also the potential gain in performance from having several sites process parts of the query in parallel.

Summarizing we have that the advantages of DDBMSs are: It reflects the organizational structure, improves share-ability and local autonomy, improves availability, reliability and performance, reduces the hardware cost and have a modular growth. The disadvantages are: Architecture and database design is more complex, cost, Security, Integrity control is more difficult, there is a lack of standards and experience.
There are 3 types of architecture in DDBMS:

- Shared Everything: Dominated the architecture market until 2000(approximately), we have one big complex costly data base, it shares everything and can be very slow. It is the centralized system.
- Share Disk: We have data saved on different disks connected between them, no one uses it. The disks are all connected together and communicates to all the CPUs.
- Share Nothing: Every thing is separated every disk has it's own CPU and the disks does not communicate with other CPUs or disks, it is very easy to scale it out. Model adopted by NoSQL models. Only at the end the processors are connected together to combine the results from each of them.

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

What to choose between Scalability or Availability? It depends on how much one wants to spend: more availability means more money. For example a 100% availability mean no downtime, decreasing the availability increases the downtime we can have, a 99% availability means a downtime of 9 - 88 hours.
Replication:
A log is a sequential file that is stored in a stable(never failing) memory, it stores all activities realized by all transaction in a chronological order. The two types of records are the transaction logs (operations) and the system events (Checkpoint & Dump). About this we can define checkpoint as the moment when we store the set of running transactions in a given time point while a dump is full copy of the entire state of a DB in a stable memory, his execution is offline and, summarizing, it generates a backup. Only after the backup is completed, the dump record of log is written.
The replicas structures are:

- One2Many: One source and many targets.
- Many2One: Many sources and only one target.
- Peer2Peer: Data is replicated across multiple nodes that communicates with each other.
- Bi-directional: Conceptually a Peer2Peer with only two Peers.
- Multi-Tier Staging: The replication is divided in more stages.

To create a replica we can:

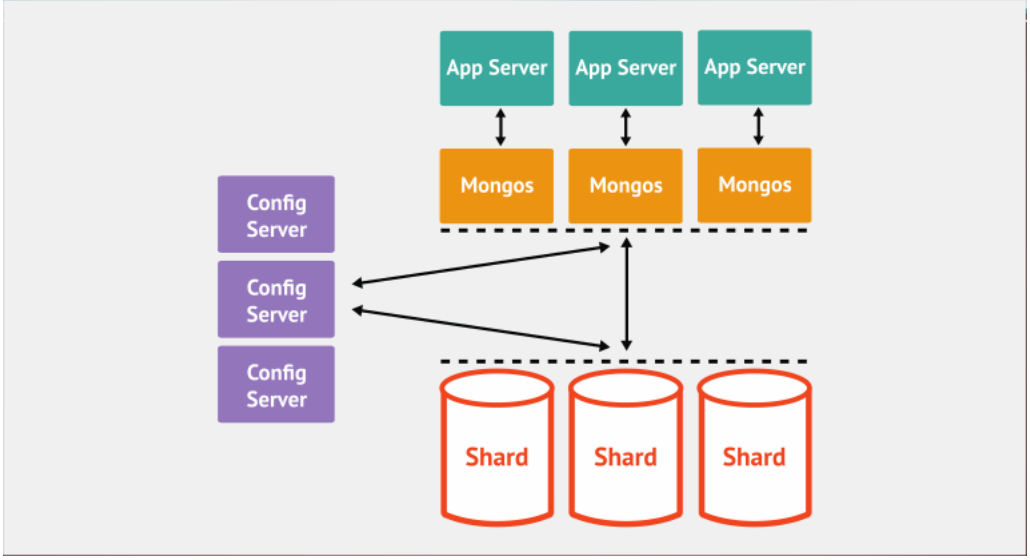
- take the data form master either the backup copy or the data itself (if we can stop the activity of the server) and move it. (It could be useful if the data is not accessible from outside)
- use log file : to maintain the replica the same as the source even during the transfer (near real time) using the Log to execute the same commands executed on the source.

2.2 MongoDB's Approach: Sharding[ScalingMongoDB]

MongoDB uses Sharding to split a large collection across several servers(called a cluster): Every document has a key in mongodb so we define a shard key which defines the range of data(chunk range). The key space is like points on a line and the range is a segment of that line. Example: Key is surname so we can decide all the surnames staring from A to K save in first partition and the others in the second. MongoDB does the sharding automatically once we tell it to distribute the data. The Default maximum chunk size is 64Mb, as

a chunks gets bigger, MongoDB will automatically split it into two smaller chunks and if the shards become unbalanced, some chunks will be migrated to other shards with less chunks correcting the imbalance. The goal of balancer is not only to keep the data evenly distributed but also to minimize the amount of data transferred. The balancer is lazy so it won't activate itself until the data is very imbalanced, it does so to avoid moving data back and forth, if it balanced every tiny difference it would just waste resources.
When you first create a shard, MongoDB creates a single chunk with range $(-\infty, +\infty)$ where with $-(+)\infty$ we mean the smallest(largest) value MongoDB can represent. From here if the chunk size is bigger that the chosen size it will split and every chunk range must be distinct and not overlapping and their union should cover all the initial range.
The shard is basically a server so we need to use mongos (s=server) to make a query on a shard. Mongos is the point interaction between users and the cluster, mongos lets you treat a cluster as a single server. The queries are routed to the appropriate shard thanks to mongos. For the not targeted queries (with sorting) a request is sent to all shards and the queries are performed locally and then after the result has returned mongos merges the (sorted) result and returns results to client. Mongos doesn't actually store any data, the configuration of a cluster is held on a special mongods called config servers, they hold the definitive information about the clusters for everyone's access.
The cluster consists basically of three types of processes:

- The shards: a node of the cluster, it can be either a single mongod or a replica set. MongoDB also automatically replicates the data and in one shard I will have a primary chunk where I store the data I wanted and secondary chunks where MongoDB replicated other data as backup just in case, the secondary is a read only data so I cannot do update queries on it.
- Mongos processes: for routing requesting to the correct data and acting as a balancer. It contains no local data and we can have 1 or many of it. After it finishes balancing it must also update the Config Servers with the new positions and while one mongos is balancing the chunks it takes out a "balancer lock" so no other mongos will start the balancing. The lock is released only after the chunk has been copied and the old chunk is deleted from the initial shard.
- Config servers: for keeping tracks of the cluster's state. It stores the cluster chunk ranges and locations. We can have only 1 or 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})
The properties of the Shard key are:

- Shard key is immutable
- Shard key values are immutable
- Shard key must be indexed
- Shard key are limited to 512 bytes in size
- Shard key are used to route queries: so choose a field commonly used in queries for the shard key
- Only the shard key can be unique across 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.

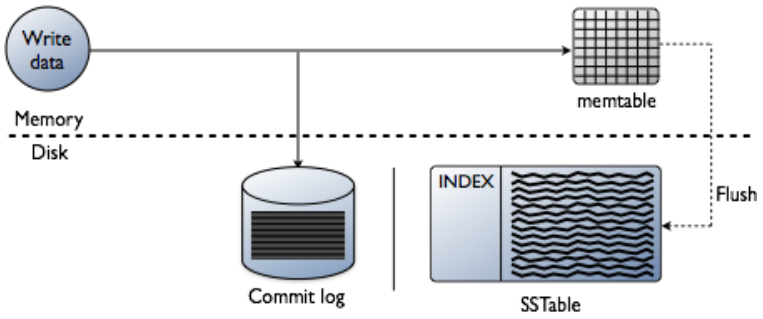


Figure 1: Data writing procedure in Cassandra.

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.

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

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

[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.

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.

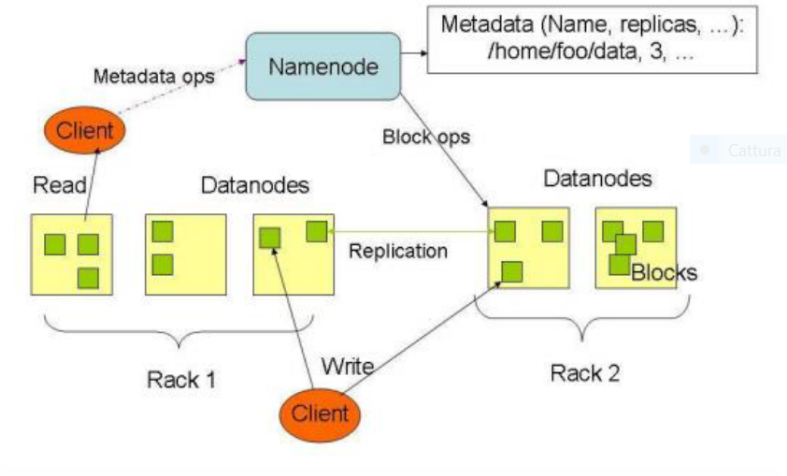
3 Big Data Architecture

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".

[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.

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

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:

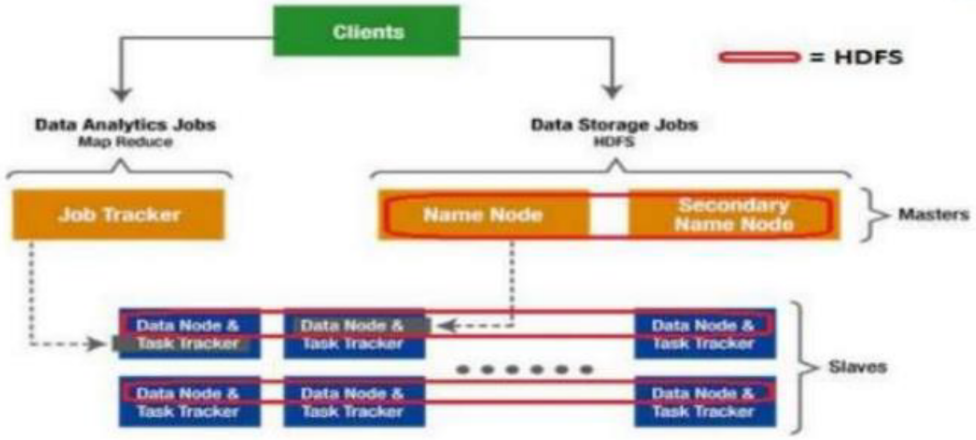
- Come assegnare i 'lavori' ai singoli 'worker'
- Cosa succede se ci sono più 'lavori' che 'worker'
- Cosa succede se i 'worker' devono condividere risultati parziali
- Come aggregare i risultati parziali
- Come scoprire se tutti i 'worker' hanno finito il proprio task
- Cosa succede se un 'worker' muore

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[3]

- 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.

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

- 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.[4]

- 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.

Vengono rilasciati inoltre altri applicativi quali Accumulo, Hbase e Spark [5], 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

[4]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.

[5]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

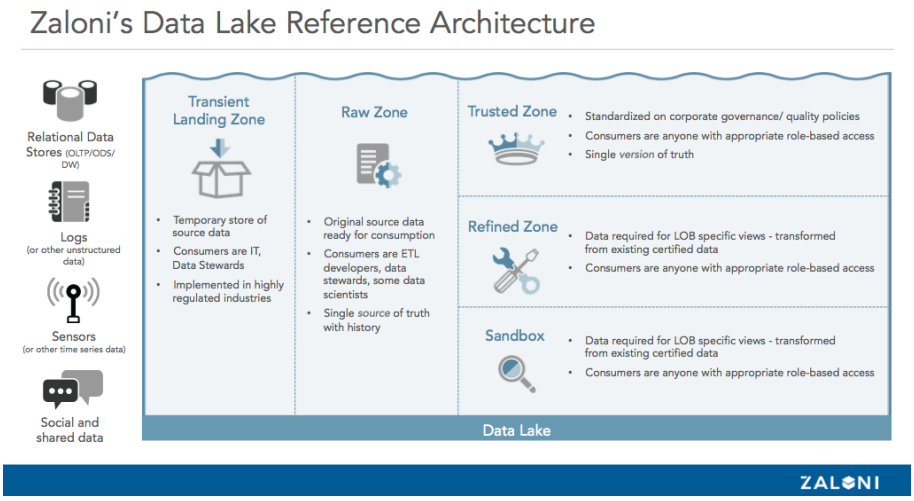


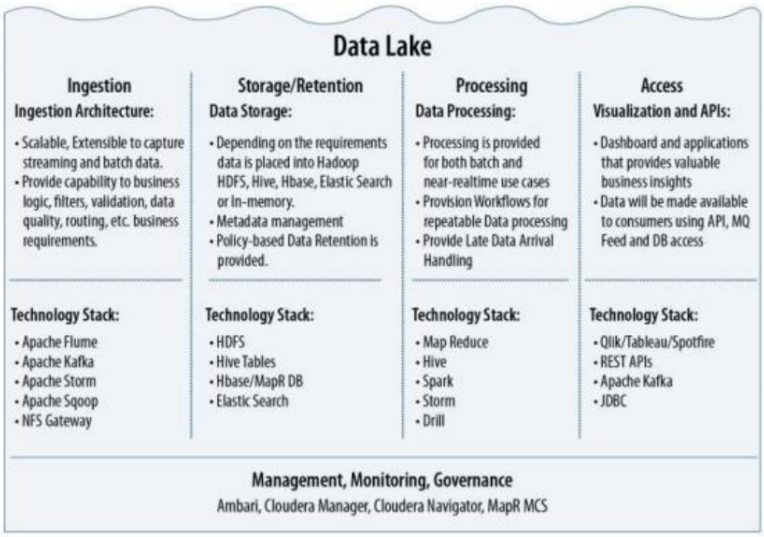
Figure 2: Typical data lake arch.

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[6]:

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



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).