

# Data Management

## Indice

<b>I</b>	<b>Modelli di dati</b>	<b>2</b>
<b>1</b>	<b>SQL.</b>	<b>2</b>
<b>2</b>	<b>NoSQL.</b>	<b>3</b>
2.1	Document Based: MongoDB[1, 2]. . . . .	4
2.2	GraphDB: Neo4J[5]. . . . .	6
2.3	Key-Value. . . . .	7
2.4	Wide Column: Cassandra[3] e BigTable. . . . .	8
<b>II</b>	<b>Distribuzione dei dati</b>	<b>11</b>
<b>3</b>	<b>Frammentazione.</b>	<b>11</b>
<b>4</b>	<b>Replicazione.</b>	<b>12</b>
<b>5</b>	<b>MongoDB: Shading[2].</b>	<b>13</b>

## Parte I

# Modelli di dati

Il *data modelling* è uno strumento per descrivere parte del mondo reale, permettendo di immagazzinare dati in un database ed effettuare interrogazioni sfruttando un linguaggio creato ad hoc. Il linguaggio di interrogazione deve essere comprensibile sia alla macchina che all'uomo, espressivo, semplice, flessibile e, se possibile, deve seguire degli standard. Il modello di dati più famoso è quello relazionale (rappresentato da SQL).

I modelli di dati seguono il Teorema di CAP, introdotto da Eric Brewer nel 2000:

- Consistency: tutti i nodi vedono gli stessi dati allo stesso istante;
- Availability: ogni nodo deve rispondere alle richieste rivoltegli;
- Partition Tolerance: il sistema funziona anche con dati frammentanti su una rete di calcolatori.

È impossibile per un modello soddisfare contemporaneamente tutte e tre le caratteristiche[4]. Il modello relazionale segue le leggi CA, mentre i modelli NoSQL sono CP o AP (dipende dal modello). Si tende a preferire la disponibilità alla coerenza per ottenere dati non aggiornati piuttosto che messaggi di errore.

## 1 SQL.

Nato negli anni '70, il modello relazionale è ben sviluppato e conosciuto e gode di un'ottima reputazione: è stato lo standard *de facto* per anni ed è ancora considerato il migliore per garantire l'integrità dei dati (permessa dal suo schema rigido). Il modello relazionale segue quattro proprietà rappresentate dalla sigla ACID:

- Atomicity: l'operazione è *atomica*, ovvero o avviene per intero (**COMMIT**) o restituisce un errore e il database ritorna alla forma iniziale (**ROLLBACK**);
- 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 tuttavia è fortemente influenzato dalla tecnologia degli anni in cui è nato: permette infatti di archiviare la maggior quantità di informazione occupando il minor spazio possibile su disco (allora gli hard disk erano costosi e ingombranti). Nel tempo dunque si sono notati difetti nel modello:

- un attributo può avere solo un tipo di valore;
- SQL non è compatibile con i moderni linguaggi di programmazione ad oggetti;
- la struttura del modello è molto rigida;
- non permette loop nei dati;
- la modifica di tabelle esistenti è difficile e dispendiosa.

Negli anni dunque si sono cercati modelli alternativi. Nei RDBMS la performance (cioè la velocità di esecuzione) dipende da vari fattori:

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

Inoltre il modello relazionale rende difficile scalare l'hardware: è stato pensato per poter girare su un'unica macchina fisica<sup>1</sup>.

## 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 NoSQL non hanno un modello prestabilito rendendo facile archiviare dati non strutturati e permettendo di aggiungere attributi senza modificare l'intero modello. Mentre il modello relazionale si basa sull'assunzione del *mondo chiuso*<sup>2</sup> (se il valore di verità non è noto, si considera la proposizione falsa), i modelli NoSQL adottano il modello del *mondo aperto*<sup>3</sup> secondo il quale non si può stabilire il valore di verità di cosa non è noto.

È possibile *scalare* (*scaling*) un DBMS potenziando l'hardware su cui è ospitato. Si distingue lo scalare verticalmente (*Scaling Up*), il potenziare la singola macchina, dallo scalare orizzontalmente (*Scaling out*), aggiungere macchine in una rete di calcolatori; quest ultimo caso 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.

Opponendosi alla rigidità del modello relazionale, le proprietà dei database NoSQL non potevano che essere riassunte dall'acronimo BASE:

- 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);

---

<sup>1</sup>Il modello *NewSQL* proposto nel 2011 tenta di risolvere a questo inconveniente

<sup>2</sup>[https://it.wikipedia.org/wiki/Ipotesi\\_del\\_mondo\\_chiuso](https://it.wikipedia.org/wiki/Ipotesi_del_mondo_chiuso)

<sup>3</sup>[https://it.wikipedia.org/wiki/Ipotesi\\_del\\_mondo\\_aperto](https://it.wikipedia.org/wiki/Ipotesi_del_mondo_aperto)

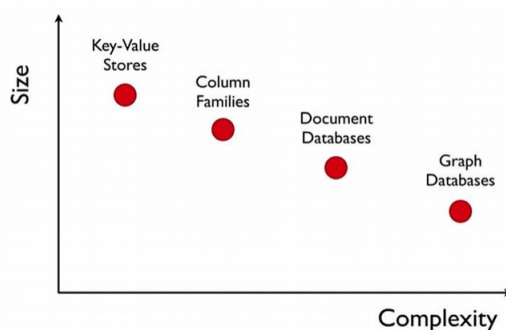
- Eventual Consistency: i sistemi NoSQL richiedono che, ad un certo punto, i dati convergano a uno stato consistente senza specificare quando; prima del raggiungimento della consistenza si possono avere valori non veritieri.

ACID	BASE
Forte Coerenza	Coerenza debole
Poca disponibilità	Forte disponibilità
Difficilmente scalabile	Facilmente scalabile
Rigido	Flessibile

I modelli NoSQL si dividono in quattro macro categorie[4, 5]:

1. 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;
2. 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.
3. Key Value (Dynamo, Voldemort, Rhino DHT): sono delle tabelle di coppie <chiave - valore> con chiavi che si riferiscono (o puntano) a un certo dato, è molto simile ai Document based.
4. Column family (Big Table, Cassandra): sono tabelle sparse e annidate contenenti coppie <chiave - valore>, sono in grado di salvare grandi quantità di dati.

Costando di meno lo spazio di archiviazione, questi modelli non tentano di minimizzare lo spazio occupato su disco ma puntano a massimizzare le prestazioni nelle operazioni di R/W. La ridondanza dei dati facilita questo compito pur aumentando notevolmente le dimensioni dei dati stessi: si sacrifica spazio di archiviazione a favore della velocità di lettura. La semplicità del modello permette di archiviare un maggior numero di dati:



## 2.1 Document Based: MongoDB[1, 2].

MongoDB è, come già affermato, un sistema di gestione basato sui documenti (*Document Based Management System*), in cui i dati sono archiviati in formato BSON (Binary JSON) e letti tramite indici. MongoDB non prevede operazioni di join. Cambiano i nomi delle entità rispetto al modello relazionale:

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

Lavorando su una rete di calcolatori, MongoDB può impiegare un tempo considerevole a importare documenti da un database preesistente; nel caso in cui si voglia velocizzare il processo, si potrebbe decidere di apportare alcune modifiche:

- 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 che ha funzione di backup.

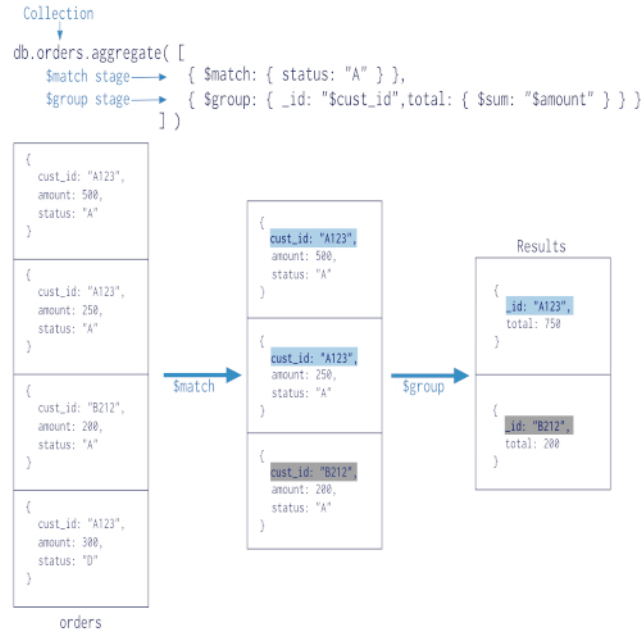
Bisogna stare molto attenti nel fare ciò, perché ogni perdita non sarà registrata e di conseguenza sarà definitiva.

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 per recuperare 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. Il comando è il seguente:

```
db.<collection>.ensureIndex({
  <field1> : <sorting>,
  <field2> : <sorting> ...
});
```

dove `<sorting>` assume il valore `+1` per ordinamenti in ordine crescente e `-1` per ordine decrescente. I campi `field` devono essere presenti in tutti i documenti della collection.

È possibile effettuare una sorta di `join` in MongoDB grazie all'algoritmo *pipeline*[4], che però impiega più tempo del corrispettivo SQL. Pipeline permette di eseguire operazioni di `$match` e `$group` su una collection così da eseguire una operazione su insiemi di risultati.



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.

## 2.2 GraphDB: Neo4J[5].

Un grafo è una collezione di nodi e archi, i quali rappresentano le relazioni tra i nodi stessi. È facile modellare numerose realtà, 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 valore 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 dei 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 adottato, soffrono perdite di prestazioni quando sono effettuate aggregazioni (soprattutto non indicizzati) dal momento che i collegamenti non sono nativi e manca il concetto di prossimità, presente invece nel grafo. 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 l’elaborazione a grafo.

Per eseguire una query nel modello a grafo, il tempo di risposta non dipende 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 le prestazioni calano 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 in cui i dati non abbiano la stessa struttura. Il processing engine usa “*index-free adjacency*” cioè 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 delle query che non sfruttano le proprietà del grafo viene peggiorata, ad esempio nel caso delle operazioni di R/W.

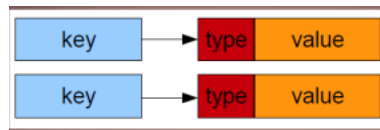
Neo4j implementa un linguaggio Cypher, di tipo dichiarativo, che permette query al grafo usando una sintassi simile a SQL o SPARQL, ma comunque ottimizzata per i grafi. Cypher è facile da leggere e capire ed espressivo, pur rimanendo compatto. Il linguaggio astrae il concetto di grafo permettendo all’autore di una query di indicare solamente cosa desidera tralasciando come.

Neo4J utilizza in parallelo il linguaggio Gremlin, parte del framework Apache TinkerPop, che invece permette di esplicitare le modalità con cui deve essere svolta una richiesta al database.

## 2.3 Key-Value.

Considerato il modello più semplice e flessibile, associa ad una stringa (la *chiave*) una sequenza di dati binari (BLOB, *Binary Large Object*) come possono essere immagini,

video o altri tipi di dato:



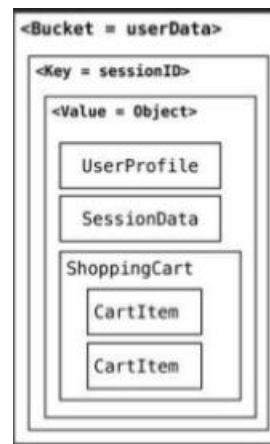
Le uniche operazioni lecite del modello sono;

- aggiunta di una nuova coppia <chiave - valore>;
- rimozione di una coppia <chiave - valore>;
- modifica di un valore (data la chiave);
- ricerca di un valore (data la chiave).

Data la natura *opaca* dei dati, non è possibile cercare la chiave dato il valore.

Esempi noti di Database key-value sono DynamoDB, sviluppato e usato da Amazon per registrare i sistemi di pagamento dei clienti, e Riak, che usa la seguente terminologia:

SQL	Riak
database	cluster
table	bucket
row	key-value
rowid	key



Grazie all'indicizzazione ad *hash*, questo modello riesce a scalare orizzontalmente in modo molto efficiente. L'hash è una funzione matematica che assegna un valore ad una chiave:  $h(x) = v$ ; nel caso specifico, generalmente il valore della funzione indica dove il dato si trova (non il dato stesso). I valori hash possono essere facilmente distribuiti su una rete di calcolatori di dimensione arbitraria: per esempio, considerando la funzione  $h(x) = x \% k$  (dove  $\%$  è la funzione resto della divisione intera e  $k$  il numero di nodi della rete), si può stabilire in quale nodo indirizzare la coppia in base al risultato. Per recuperare il dato, basta dunque interrogare il nodo in cui è presente la chiave e attendere che questo invii il valore.

## 2.4 Wide Column: Cassandra[3] e BigTable.

L'idea di archiviare i dati per colonne è nata negli anni '70, ma le prime implementazioni commerciali del modello risalgono solamente agli anni '90; di notevole importanza sono



BigTable, introdotto da Google e da cui nasce Apache HBase, e Cassandra, sviluppato da Facebook. Questo modello archivia i dati, raggruppati per colonne, su blocchi sequenziali dell'hard disk. Questo ottimizza le aggregazioni della colonna e le query eseguite su più colonne (il tempo dipende comunque da eventuali indicizzazioni e dal design del database). In un certo senso, questo modello è un'evoluzione del KeyValue: quando si tenta di apportare una struttura al valore indicizzato dalla chiave, si ottiene un risultato simile (a tal punto che Cassandra può essere considerato un KeyValue). Questo modello offre prestazioni elevate scalando orizzontalmente, quando sono richieste numerose operazioni di R/W al secondo su grosse quantità di dati e quando è necessario avere indicazioni temporali riguardanti ogni dato. Si può usare anche costruire un database ibrido tra questo modello e quello a grafo: nascono così progetti come JanusGraph. Il modello non permette join, ma si può ottenere un equivalente grazie alle funzioni `Scan()` e `Get()`.

### BigTable e HBase.

Sviluppato da Google, BigTable gestisce i dati costruendo delle mappe multidimensionali, ordinate, persistenti e sparse, accessibili grazie ad un indice di riga, ad un indice di colonna e a un *timestamp*. Ogni tabella è composta da *column-families*, le cui celle sono ordinate e sparse (ovvero contengono chiavi diverse). Si può rappresentare la struttura nel seguente modo:

row key	column family 1		column family 2		
	column 1	column 2	column 1	column 2	column 3
KEY	<i>cell</i>	<i>cell</i>	<i>cell</i>	<i>cell</i>	<i>cell</i>

Ogni riga può avere un timestamp diverso, quindi si possono avere più versioni della stessa tabella.

I dati all'interno della cella (e la chiave) non hanno tipo ma sono salvati come sequenza di byte, quindi per cercare un dato nota la chiave è necessario usare una funzione `.toBytes("Key")`. Le colonne sono dinamiche: è sempre possibile aggiungere una nuova colonna o modificare quelle esistenti (anche grazie ad API esterne). Ogni colonna può appartenere ad una sola column-family e appartiene alla riga con `familyNode:columnName` seguito dal valore:

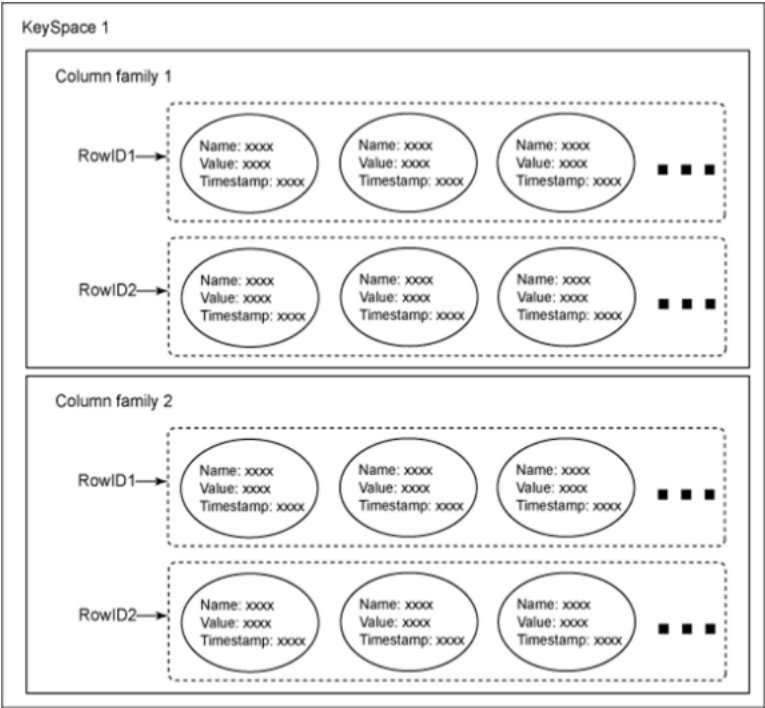
```
row-key : {
  info : {
    height : 170,
    state : "NY"
  },
  roles : {
    ASF : "Director",
    Hadoop : "Founder"
  }
}
```

Il numero di versione per ogni riga è univoco per ogni row-key: generalmente è usato un timestamp, ma può essere modificato dall'utente.

**Cassandra.**

Nato come supporto per la ricerca nell'inbox di Facebook, fu reso open source nel 2008 e in seguito è diventato un top-level project sotto Apache nel 2010. Il modello è identico a quello di BigTable e HBase con alcune differenze per gli algoritmi utilizzati e il modo in cui i dati sono distribuiti. Usa un linguaggio chiamato CQL (Cassandra Query Language) simile a SQL.

Il *Keyspace*, che sarebbe l'equivalente del database in un RDBMS, possiede nativamente un fattore di replicazione e un algoritmo di replicazione, ma rimane solamente un raggruppamento di column families.



Un esempio del Datamodel di Cassandra.

Non permettendo operazioni di join o chiavi esterne, il modello prevede l'uso di dati ridondanti e denormalizzati per gestire i collegamenti.

**Confronto tra i tre DBMS WideColumns.**

	Cassandra	Big Table	DynamoDB
<i>Tipologia:</i>	Column	Column	Key-Value
<i>Progettato per:</i>	Write often, read less	Large scalability	Large database
<i>Gestione parallela:</i>	MVCC	Locks	ACID
<i>Consistency:</i>		Sì	Sì
<i>Hight Availability:</i>	Sì	Sì	Sì
<i>Partition Tolerance:</i>	Sì	Sì	
<i>Persistenza:</i>	Sì	Sì	

## Parte II

# Distribuzione dei dati

Un database centralizzato è un sistema in cui il database è localizzato su una sola macchina: se questa non è funzionante (SPOF, *Single Point of Failure*), l'intero database non risulta accessibile. Per facilitare l'accesso ai dati in luoghi differenti, è possibile distribuirli o replicarli: nel primo caso si distribuiscono in luoghi diversi dati diversi, nel secondo i medesimi dati (per intero o solamente in parte). La distribuzione permette una maggiore disponibilità (*availability*) e la parallelizzazione delle operazioni. In fase di progettazione di un Database Distribuito, si costruisce la rete tra i nodi, si collocano questi geograficamente e si stabilisce il loro contenuto. Le macchine su cui gira un database distribuito non sono particolarmente potenti: si riduce notevolmente il costo dell'hardware ma la progettazione risulta più difficile. Bisogna tenere presente però che la prestazione totale di un database distribuito è, generalmente, pari alla prestazione del nodo più debole: è il fenomeno *bottleneck*. Le performance tuttavia migliorano linearmente rispetto al numero di nodi della rete, ma risultano influenzate dalla velocità di connessione.

In un Database Distribuito Omogeneo, tutti i *nodi* condividono le medesime impostazioni (non è possibile individuare una gerarchia): insieme collaborano a risolvere le richieste dell'utente, a cui appare come un singolo sistema. Invece in un Database Distribuito Eterogeneo i nodi hanno impostazioni diverse, rendendo difficili le comunicazioni per risolvere query e altre transazioni, che sono risolte limitatamente alla loro parte.

Un sistema distribuito tiene in memoria, nel complesso della rete, una copia di ogni dato su un numero di nodi stabilito dal *replication factor*. Ciò garantisce una maggiore tolleranza agli errori (è più difficile avere un SPOF): il dato è ottenibile da più fonti diverse, dunque se un nodo non è disponibile la rete gira la richiesta ad un altro. Inoltre la replicazione del database migliora con la sua frammentazione: i nodi replicano i dati spartendosi i frammenti (*churn*), riducendo le comunicazioni interne. In questo modo si possono distribuire le richieste ai singoli nodi, che elaborano in parallelo e inviano all'utente il risultato.

Tuttavia replicazione e frammentazione rischiano di compromettere la coerenza del dataset: sono da considerare il tempo di aggiornamento di ogni replica e la gestione di operazioni contraddittorie eseguite parallelamente (ad esempio, l'acquisto contemporaneo di due persone dell'ultimo biglietto per un volo aereo). Per ovviare a questo problema, una possibile soluzione è costruire una struttura gerarchica all'interno dei nodi: si individuano *master* che interagiscono con l'utente e *slaves* che hanno una mera funzione di replica (a meno che il rispettivo *master* non sia disponibile).

## 3 Frammentazione.

La frammentazione dei dati invece avviene dividendo i dati in  $k$  parti in modo tale però che sia possibile ricostruire la relazione originaria. La frammentazione può essere orizzontale (si dividono le osservazioni) o verticale (si dividono gli attributi) in base alla struttura del modello. In ogni caso dovrebbe garantire che i dati siano presenti nei nodi che li

processano più frequentemente; inoltre è permessa così la parallelizzazione dei processi (per osservazioni o per caratteristiche). Si possono anche avere frammentazioni ibride.

Si intende con *Data Transparency* la consapevolezza che ha l'utente finale della distribuzione dei dati: l'obiettivo è di far credere che si operi su un unico server.

Si individuano tre tipologie di architetture nei DDBMS (*Distributed DBMS*):

- *Share Everything*: è un sistema centralizzato con un'unica macchina e (virtualmente) un unico disco;
- *Share Disk*: i dati sono archiviati su diversi dischi connessi tra loro a cui accedono diverse macchine;
- *Share Nothing*: le macchine non condividono risorse ma operano all'interno di una rete (facile da scalare) sfruttata per unire i risultati.

Shared Disk	Shared Nothing
Migliore gestione del carico di lavoro	Opera su <i>commodity hardware</i>
High Availability	Facilmente scalabile
Performance migliori con letture frequenti	Performance migliori con grossi volumi (R/W)
I dati non sono partizionati	I dati sono partizionati in cluster

## 4 Replicazione.

Si individuano diverse strutture di replica:

- One2Many: una sorgente e più repliche;
- Many2One: più sorgenti e una sola replica (contenente tutti i dati);
- Peer2Peer: i dati sono replicati attraverso molteplici nodi che comunicano l'uno con l'altro;
- Bi-directional: un Peer2Peer a coppie;
- Multi-Tier Staging: la replicazione è divisa in più stages.

Per creare una replica bisogna avere accesso alla porzione di dati da replicare: è necessario prelevare i dati dal nodo o usare un backup come sorgente. Per sincronizzare le repliche durante l'operazione di trasferimento, si usano dei *log file*: sequenze di attività realizzate, in ordine cronologico, che possono tenere conto delle transazioni (*operations*) o di eventi di sistema (*checkpoint* e *dump*). Si definisce *checkpoint* un insieme di transazioni svolte in un determinato intervallo di tempo, mentre un *dump* è una copia dello stato del Database in un dato istante temporale.

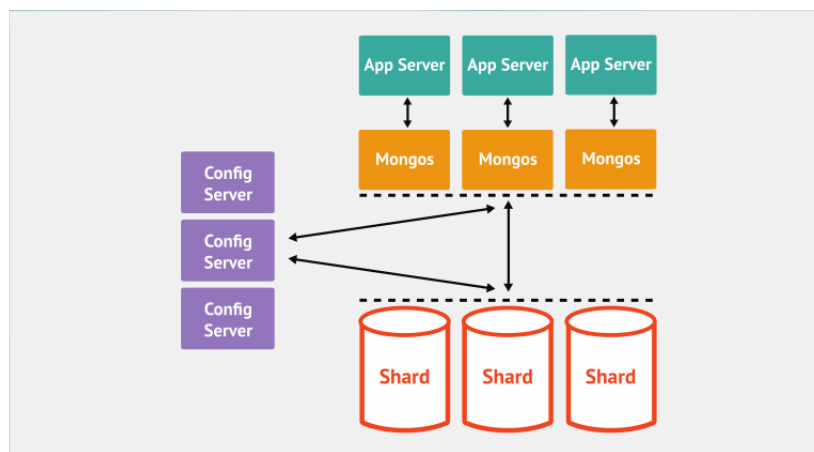
## 5 MongoDB: Sharding[2].

MongoDB divide una collezione in diversi server (chiamati *shards*, che nel complesso formano un *cluster*) secondo il valore assunto da una chiave presente in ogni documento, raggruppando così i dati in *chunks*. Inizialmente la collezione è composta da un solo *chunk* con valori limite  $(-\infty, +\infty)$ ; ma, impostando la chiave, è effettuata in automatico una divisione in modo tale che ogni *chunk* abbia un valore massimo di 200MB (sempre che sia possibile effettuare divisioni così precise), e che questi siano disgiunti ed esaustivi. I *chunks* sono così distribuiti tra i vari *shards* grazie ad un *balancer*, che però tenta di ridurne gli spostamenti: per evitare la saturazione della rete, un *chunk* è spostato da un nodo ad un altro solamente se c'è una differenza di almeno 9 *chunks* tra i due nodi. Così si tenta di equiparare il carico di lavoro tra i vari *shards* della rete, senza però sprecare risorse.

Il software incaricato di compiere queste operazioni è mongos, che mette in contatto i vari demoni mongod: è il punto di contatto tra il singolo nodo e il cluster, che permette la gestione dell'intera rete alla pari di una singola istanza. Altro compito di mongos è di instradare le query verso i nodi corretti, o di unire i risultati se la richiesta è fatta all'intera rete. Mongos non si occupa di archiviare la configurazione del cluster: a questo compito provvede il demone mongods (il *config server*), che conserva un log degli accessi alla rete.

Nel cluster sono quindi presenti tre tipi di processo:

- Mongod (i *shards*): i nodo del cluster, repliche o meno;
- Mongos: per incanalare le richieste allo *shard* corretto;
- Mongods: (config server): tiene traccia dello stato del cluster, se ne possono avere solamente 1 o 3.



Per l'aggiunta di nodi e repliche, la procedura è la seguente:

```
// inizializzazione di mongod su ogni server
localhost$ ssh server.ip
server% mongod
```

```

server% exit

// connessione al server remoto
localhost$ mongo server.ip/admin
MongoDB shell
connecting to: admin
> // inclusione nel cluster
> db.runCommand({
    "addShard" : "server_con_dati.ip:27018",
    "name" : "nome_identificativo"
});
{ "shardAdded" : "nome_identificativo", "ok" : 1 }
> // per aggiungere una replica
> db.runCommand({
    "addShard" : "server/server_da_replicare.ip:27018"
    "name" : "nome_identificativo"
});
{ "shardAdded" : "nome_identificativo", "ok" : 1 }
> // ora la rete è creata, si devono impostare i dati
> // selezione del database
> db.adminCommand({
    "enableSharding" : "database_condiviso"
});
{ "ok" : 1 }
> // selezione delle collection e delle chiavi
> db.adminCommand({
    "shardCollection" : "collection_condivisa",
    "key" : {
        "chiave_1" : 1,
        "chiave_2" : 1 //...
    }
});
{ "collectionSharded" : "collection_condivisa", "ok" : 1 }

```

Una *shard key* deve essere presente in tutti i documenti della collezione, non deve essere mutabile e viene automaticamente indicizzata da MongoDB. Inoltre deve avere valori diversi tra di loro (non si possono dividere valori tutti uguali) e deve facilitare le ricerche: è opportuno usare un attributo usato frequentemente nelle query). Possono nascere problemi con valori del campo *id* modificati in quanto possono essere unici solamente all'interno del proprio *shard* e dunque possono ripetersi in *shard* diversi (tenendo il valore di default è altamente improbabile che accada).

## Lecture di approfondimento

- [1] Kristina Chodorow. *MongoDB: The Definitive Guide*. Second Edition. United States of America: O'Reilly, 2013. ISBN: 978-1-449-34468-9.
- [2] Kristina Chodorow. *Scaling MongoDB*. United States of America: O'Reilly, 2011. ISBN: 978-1-449-30321-1.
- [3] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly, 2010. ISBN: 978-1-449-39041-9.
- [4] Marina Pernini. *NoSql DB. Analisi prestazionale e confronto col modello relazionale*. EAI, 2018. ISBN: 978-620-2-08288-4.
- [5] Ian Robinson. *Graph Databases*. Second Edition. O'Reilly, 2015. ISBN: 978-1-491-93200-1.