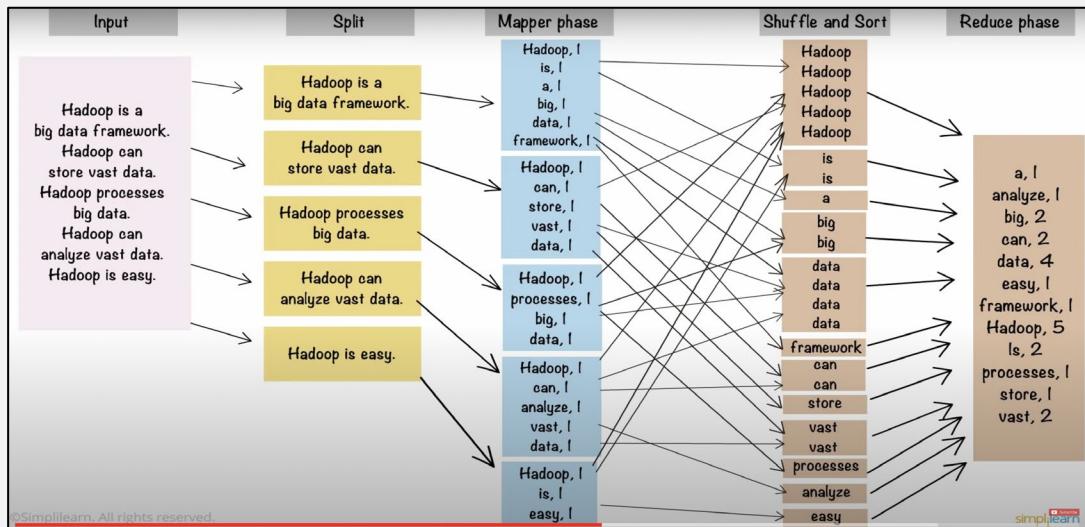


# Big Data Management - Simone Maiorani

## Capitolo 2 – Map-Reduce Data Processing Model

Modello di elaborazione dati per grandi datasets.

**Obiettivo:** scrivere programmi in Java che utilizzano algoritmi di tipo MapReduce.



**Struttura base: coppie chiave-valore.**

L'input viene spartito in una serie di coppie chiave-valore.

Sta a noi capire come rappresentare il nostro dataset secondo tali coppie.

**Ruolo del Programmatore:** i programmi scritti con questo modello possono essere automaticamente parallelizzati ed eseguiti su un cluster.

L'implementazione di MapReduce è trasparente per il programmatore.

L'implementazione di MapReduce fornisce lettori di input che trasformano i datasets in input (documenti, database, file XML, ecc.) in un set di coppie chiave-valore.

L'utente può definire lettori di input personalizzati.

Definire una **funzione Map** che viene chiamata per ciascuna coppia chiave-valore.

Tale funzione può creare altre coppie oppure no.

La funzione Map è di tipo stateless.

Tutti i valori associati a una data chiave sono raggruppati insieme e ordinati per chiave.

Questa fase (**Shuffle Phase**) può essere parzialmente personalizzata.

Genera in output una lista di valori intermedi che saranno l'input per la funzione Reduce.

Definire una **funzione Reduce** che viene chiamata solo al termine dell'esecuzione della funzione Map su tutte le coppie.

Tale funzione viene eseguita su ogni chiave univoca e la sua lista di valori, producendo in output una nuova lista di coppie chiave-valore, questo è l'output del programma.

Genera l'output finale di coppie chiave-valore.

L'output della funzione Map dev'essere dello stesso tipo dell'input della funzione Reduce.

L'output della funzione Reduce può invece essere diverso.

**Multiple Rounds:** alcuni algoritmi MapReduce fanno uso di Round multipli.

I round sono coordinati da un driver esterno, può essere ad esempio un semplice algoritmo centralizzato.

Se il numero di round non è predeterminato, il driver può decidere quando interrompere le iterazioni in base ad alcune condizioni di arresto.

**Memoria:** è un aspetto importante in quanto può avere un impatto anche sulla rete in caso di trasferimenti. La **Memoria Locale (ML)** di un algoritmo è la quantità massima di memoria principale richiesta, in ogni round, da una singola invocazione della funzione map o reduce, per memorizzare l'input e l'eventuale struttura dati necessaria per l'invocazione. La Memoria Locale limita la quantità di memoria principale richiesta per ciascun lavoratore. **È fondamentalmente la memoria RAM.**

È un limite superiore per un processo di calcolo.

La **Memoria Aggregata (MA)** di un algoritmo è la quantità massima di memoria secondaria (spazio su disco) che viene occupata dai dati archiviati all'inizio/fine delle funzioni Map o Reduce di qualsiasi round.

La Memoria Aggregata delimita la quantità complessiva di spazio (disco) che la piattaforma in esecuzione deve fornire.

È lo spazio da avere disponibile per eseguire e salvare tutto.

**Complessità:** si utilizzano per l'analisi asintotica principalmente tre indicatori di performance ovvero numero di Round R, Memoria Locale (ML), Memoria Aggregata (MA). Gli algoritmi di solito mostrano dei compromessi tra gli indicatori di performance.

Ad esempio: una ML piccola consente un parallelismo elevato ma può comportare un R grande. Occorre avere un buon Trade-OFF.

### Esempi di applicazioni Map Reduce

#### - Calcolo del numero di co-occorrenze (2 parole insieme sulla stessa riga).

A livello di Pseudocodice utilizziamo 2 cicli for annidati:

(blocco una parola e a giro scorro tutte le altre).

Può interessarci o meno l'ordinamento, ad esempio (<Pippo, Pluto>, <Pluto, Pippo>) possono essere due chiavi diverse oppure no.

Indichiamo con  $k$  il numero di righe e con  $N_{MAX}$  il numero di parole per riga.

Questo metodo (metodo "Pairs") può generare un numero alto di coppie <chiave, valore>.

**Soluzione:** invece di farne tante utilizzo dei dizionari (metodo "Stripes").

Nel **Reduce** dovrò prendere la stessa <key> per ogni dizionario e sommarne i valori.

In generale, **meglio poche chiavi ma grandi che tante chiavi ma piccole**.

"Stripes" meglio di "Pairs".

a	b	c	d	f	h	
1	3	5	7	1	1	
b	c	f	h	l	m	
1	2	3	1	1	2	
a	b	c	d	f	h	l
1	4	5	9	4	2	1
						2

Metodo Stripes

Stesso ML e MA di "Pairs" però con molto meno ordinamento e mescolamento delle coppie chiave-valore.  
È però più difficile da implementare.

Il dizionario non può essere troppo grande per evitare il paging della memoria.

In generale noi vogliamo che **all'aumentare dei cluster aumenti lo speed-up (velocità)**.

Ma non sempre questa cosa accade. Inoltre, avere troppe macchine potrebbe essere addirittura controproducente perché vanno ad aumentare le comunicazioni tra i nodi (che potrebbero congestionare la rete).

**Possibile soluzione:** aumentando le risorse dei nodi (senza aumentare il loro numero) dovrei avere tempi di esecuzione minori.

### - Modellare query SQL.

Con operazioni di tipo SELECT/WHERE mi basta solo il metodo Map.

Non serve il metodo Reduce perché faccio solo operazioni di filtraggio e non merge nulla.

Con operazioni di tipo SELECT/WHERE WITH JOIN serve anche il metodo Reduce.

Il join ci indica la chiave che ci serve! Scegliere bene la chiave è fondamentale!

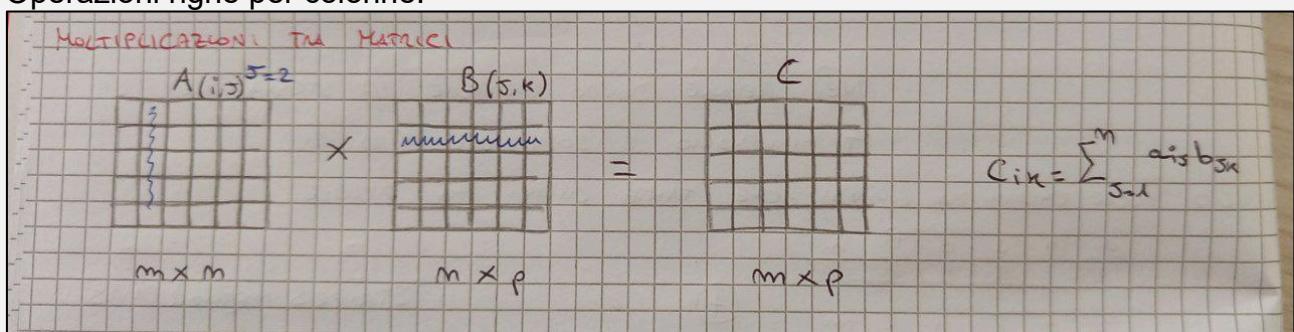
Nel Map verifichiamo le condizioni sulle tuple.

Nel Reduce verifichiamo le condizioni degli aggregati (esempio: GROUP BY).

In generale noi vogliamo un **algoritmo output-sensitive**, ovvero un algoritmo il cui tempo di esecuzione dipende dalla dimensione dell'output, anziché (o in aggiunta) alla dimensione dell'input.

### - Moltiplicazioni tra matrici.

Operazioni righe per colonne.



L'indice j è quello in comune, i & k quelli che variano.

Trovo tutti i prodotti per quel j & poi li sommo.

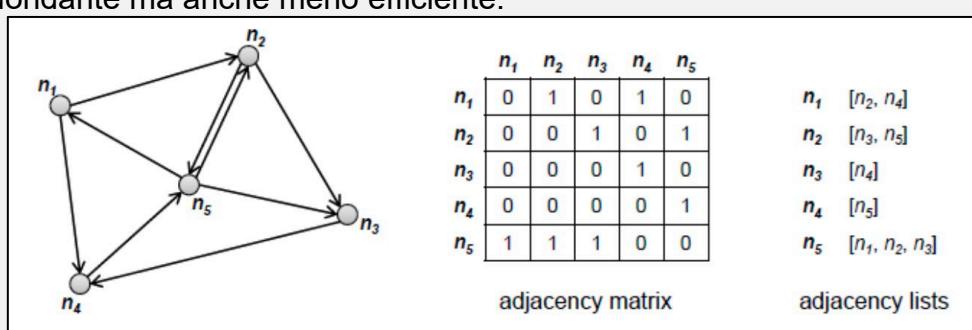
### - Algoritmi sui grafi.

**Un grafo può essere rappresentato attraverso una matrice di adiacenza.**

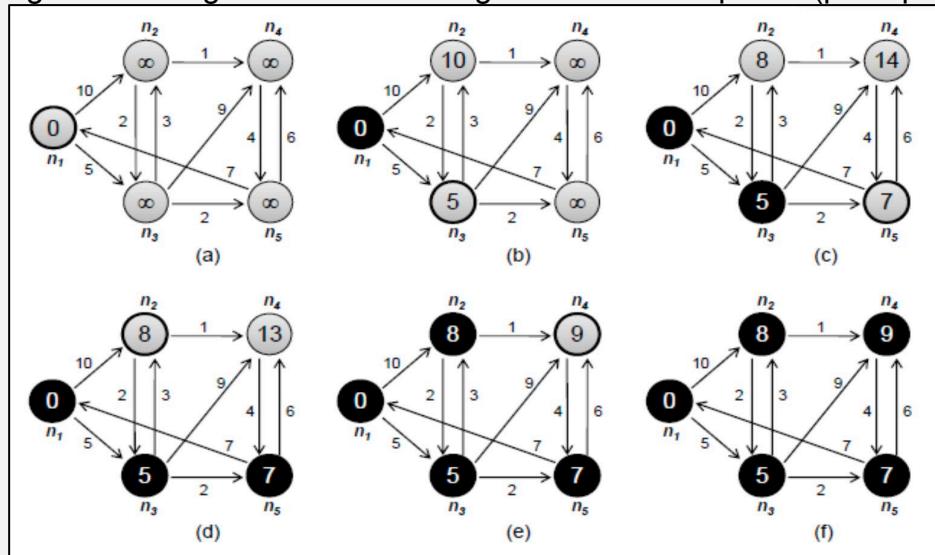
**Problema:** molti grafi sono sparsi e quindi molte celle sono a zero. Questo implica sprecare spazio per poca informazione.

**Un grafo può essere rappresentato attraverso una lista di adiacenza.**

È meno ridondante ma anche meno efficiente.



L'**algoritmo di Dijkstra** è un algoritmo utilizzato per cercare i cammini minimi in un grafo (con o senza ordinamento) con pesi non negativi sugli archi. Ovvero i percorsi più brevi da un vertice sorgente a tutti gli altri vertici in un grafico con archi pesati (peso positivo).



Questa è una versione centralizzata, **come lo implemento in modo distribuito?**

Per implementare l'algoritmo Dijkstra all'interno del paradigma MapReduce, dobbiamo applicare una visita BFS distribuita del grafo.

- Non possiamo utilizzare una struttura dati globale come una coda di priorità.
- L'algoritmo avrà bisogno di più round MapReduce.
- La struttura del grafico viene passata da un'iterazione all'altra.

Ogni vertice informa i suoi adiacenti riguardo la sua distanza dalla sorgente e riguardo il peso dell'arco per arrivare a loro.

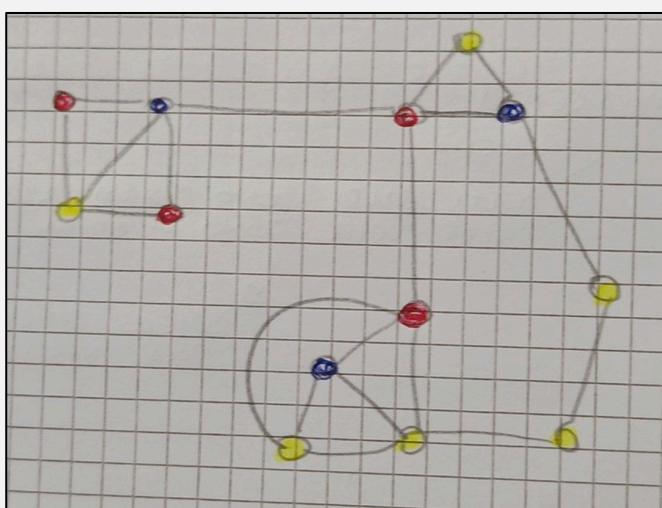
L'informazione viaggia nella rete attraverso dei messaggi.

L'algoritmo converge (si ferma) quando nessun vertice aggiorna più la sua distanza.

Il Driver guarda l'unica struttura centralizzata, ovvero un BOOLEAN dove ci sarà un 1 se qualche nodo ha aggiornato la propria distanza o uno 0 se tutti i nodi hanno aggiornato la posizione.

*Il numero massimo di iterazioni è pari al percorso più lungo tra sorgente e un nodo.*

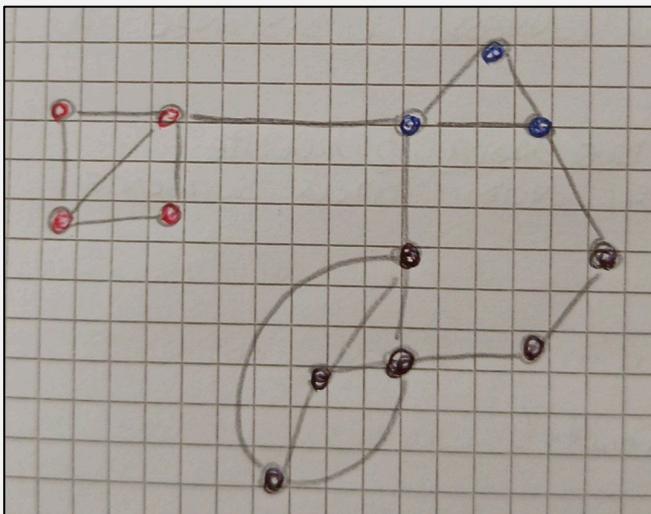
Caso peggiore: iterazioni = grandezza input.



Per comunicare in questo modo le coppie **<key, value>** tra i nodi gialli occorrerà per forza usare la rete.

Stessa cosa per i nodi rossi o per i blu.

[**Soluzione SUB-OTTIMA**].



Ogni cluster ha tutti i nodi connessi quindi comunica in locale senza usare la rete.

[Soluzione OTTIMA]

Per ottenerla occorrono appositi algoritmi di Path-Partitioning.

E principi di:

- Località: tutti i nodi di un cluster vicini.
- Bilanciamento: ogni cluster ha più o meno gli stessi nodi.

#### Limitazioni degli algoritmi dei grafi in MapReduce.

- Nessuno stato globale (struttura dati) può essere mantenuto e comunicato durante il calcolo.
- Le informazioni devono essere propagate attraverso coppie chiave-valore.
- In caso contrario, è possibile utilizzare strumenti esterni (come database distribuiti) per ottenere uno stato globale ma è pericoloso a causa di possibili colli di bottiglia.
- Il calcolo può richiedere diverse iterazioni coordinate da un programma esterno.  
Se l'output di ogni iterazione fosse archiviato in una memoria secondaria, ciò potrebbe rappresentare un collo di bottiglia significativo per il calcolo.

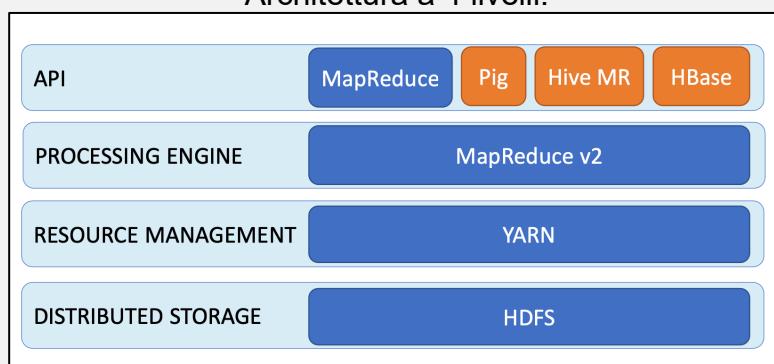
### Capitolo 3 – Apache Hadoop

Framework open source basato su Java.

Il suo **ecosistema** è composto da 4 moduli principali:

- **Hadoop Common**: classi e interfacce per files systems distribuiti e I/O.
- Hadoop Distributed File System (**HDFS**): file system distribuito che fornisce un accesso ad alta velocità effettiva ai dati dell'applicazione.
- **Hadoop YARN**: un framework per la pianificazione del lavoro e la gestione delle risorse del cluster.
- **Hadoop MapReduce**: motore di esecuzione per dati distribuiti, basato su YARN.

Architettura a 4 livelli:



Tutti i moduli sono progettati partendo dal presupposto fondamentale che avere problemi hardware o software è normale vista l'elevata quantità di macchine.  
L'importante è gestirli automaticamente all'interno del framework.

## File System Distribuito – HDFS

Implementazione open source basata su Google File System (GFS).

Partizioniamo il nostro dataset su più macchine.

Il rilevamento dei guasti e il ripristino rapido e automatico degli stessi è un obiettivo architetturale fondamentale di HDFS.

**Bisogna ottimizzare il throughput** (dati processati al secondo).

In modo secondario ci interessa avere anche una bassa latenza (poco tempo per aprire un file piccolo). Viene utilizzato per gestire grandi dataset.

**Una volta caricato il dataset esso non cambia più perché lo scopo è analizzarlo.**

Non è possibile modificare un file in un punto arbitrario ma posso inserire un file in APPEND.

L'idea chiave è **spostare la computazione verso i dati e non viceversa** (anche se a volte occorre farlo, ad esempio nella fase di shuffle).

HDFS è stato progettato per avere un'elevata portabilità.

Riassumendo, gli svantaggi invece sono:

- Accesso a bassa latenza.
- Gestione di file di piccole dimensioni.
- Non supporta scritture multiple o modifiche arbitrarie.

A livello architetturale **i file sono divisi in blocchi** e sono sparpagliati su diverse macchine (sharding).

Per evitare di avere file corrotti duplico i blocchi su più nodi. Questo porta ad un aumento delle dimensioni (1 GB replicato 3 volte → 3 GB) ma è necessario.

A livello architetturale abbiamo un'**architettura di tipo master-slave**.

Master: gestisce il namespace del FS.

Slave: memorizzano i blocchi di dati.

Un **blocco** è l'unità minima di informazione che può essere letta o scritta (default 128 MB).

Bisogna **sempre ottimizzare il throughput** in funzione della latenza minima.

- Esempio: se il tempo di ricerca è 10 ms e la velocità di trasferimento è 100 MB/s, un blocco di 100 MB assicura che il tempo di ricerca richieda solo l'1% del tempo di trasferimento (1 s).

**Più un blocco è grande più tempo ci vuole per processarlo ma meno ci vuole per individuarlo** (compenso).

Un blocco se importante posso replicarlo su tutti i nodi così lo trovo ovunque, in questo caso abbiamo una località + disponibilità massima per quel nodo.

Un **namenode** è il **nodo master (daemon)** che gestisce l'intero filesystem (namespace).

Mantiene la struttura ad albero per tutti i file e directory.

Contiene al suo interno un log edit (registro delle modifiche) con il quale tiene traccia di tutte le modifiche avvenute su file. Grazie ad esso il namenode mantiene in salute il filesystem poiché se dall'ultimo snapshot ho perso qualche "pezzo" posso ripartire dal vecchio e riprocessare.

Coordina le operazioni su file ma non gestisce i dati trasferiti.

Se perdo il master il lavoro cessa.

Può esserci un namenode secondario, sincronizzato con il primo, il quale entra in scena in caso di fallimento del nodo master. Chiaramente dev'essere posizionato su switch/rack diversi.

Il namenode secondario viene aggiornato periodicamente così se serve la sua presenza è il più aggiornato possibile.

Un'altra possibile soluzione può essere quella di utilizzare più nodi master (multi-master).

Per cluster molto grandi con molti file (miliardi), la memoria del namenode rappresenta un limite per la scalabilità (altro motivo per cui i blocchi dovrebbero essere grandi).

- La federazione HDFS consente la scalabilità orizzontale di un cluster aggiungendo più namenode, ognuno dei quali gestisce una parte del namespace del filesystem.

Per una questione di robustezza vorremmo repliche di blocchi il più distanti possibile, ma più sono lontani più ci vuole tempo per tornare indietro e più aumenta quindi la latenza.

Occorre un buon tradeoff tra robustezza e larghezza di banda. Politica: uso 3 repliche.

La banda dipende dalla distanza client – nodo.

Da processi sullo stesso nodo.

Da nodi diversi sullo stesso rack.

Da nodi su rack diversi nello stesso cluster (data center).

I **datanodes (daemons)** sono gli slave nodes ed hanno il compito di memorizzare i dati.

Il namenode conosce a quali datanodes sono associati i blocchi di dati.

I datanodes inviano degli "heartbeat messages" al namenode (default ogni 3s).

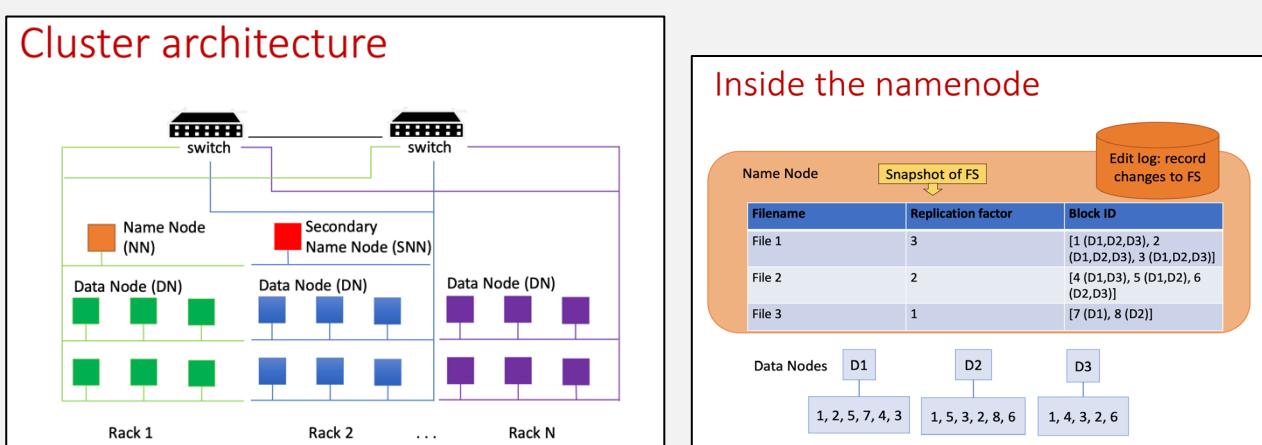
Se il namenode non riceve questo messaggi entro l'intervallo prefissato allora procede con la replica dei blocchi del nodo perso su un nodo sano.

I datanode sono anche responsabili dell'integrità dei dati.

Il **client** permette di visualizzare il FS come se fosse normale, mostra l'interfaccia del FS.

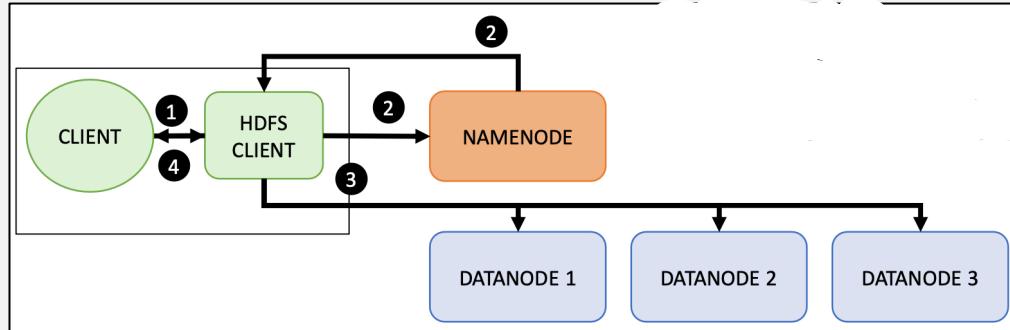
L'utente finale non vede la struttura che c'è dietro.

Il client comunica con tutte le altre entità per operare al meglio.



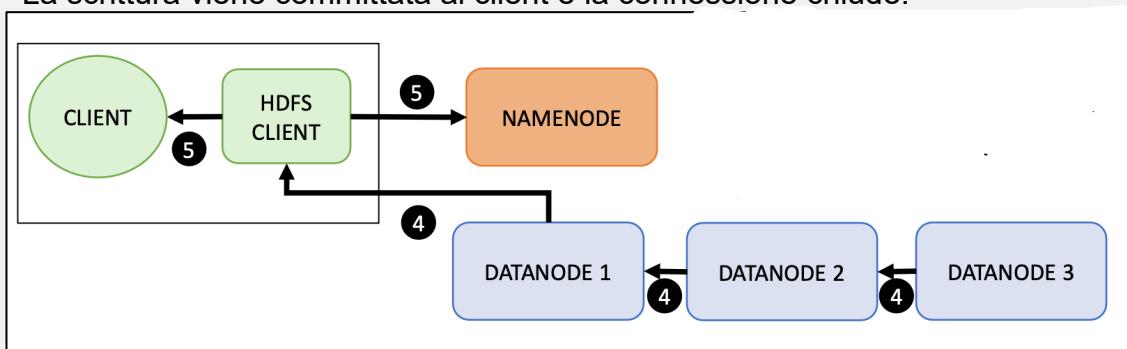
### Operazioni di lettura (read):

1. Il client fa una richiesta "read file" al client HDFS.
2. Il client HDFS fa una richiesta al namenode per ottenere l'elenco dei blocchi e le relative posizioni, che contengono le informazioni che il client vuole, da esso.
3. Il client HDFS legge i blocchi dai rispettivi datanodes.
4. Comunica le informazioni al Client e la comunicazione viene chiusa.



### Operazioni di scrittura (write):

1. Il client fa una richiesta "write file" al client HDFS.
2. Il client HDFS fa una richiesta al namenode per ottenere l'elenco dei blocchi e le relative posizioni, che contengono le informazioni che il client vuole, da esso.
3. Il client HDFS scrive i file sui rispettivi datanodes.
4. Il client HDFS riceve un ACK dai datanodes di avvenuto successo.
5. La scrittura viene committata al client e la connessione chiude.



**Integrità dei dati:** HDFS esegue il checksum, di tutti i dati scritti su di esso, il quale deve solo dire se c'è un errore (overhead < 1%). Non deve correggerlo sennò ci vuole più tempo. Se c'è un errore (blocco compromesso) lo butto via e uso una replica.

La replica è costosa: lo schema di replica 3x predefinito in HDFS ha un sovraccarico del 200% nello spazio di archiviazione e in altre risorse (esempio: larghezza di banda di rete). Tuttavia, per i datasets caldi e freddi con attività di I/O relativamente basse, si accede raramente a repliche di blocchi aggiuntive durante le normali operazioni, ma consumano comunque la stessa quantità di risorse della prima replica.

HDFS può utilizzare Erasure Coding (EC) al posto della replica.

Esso fornisce lo stesso livello di tolleranza agli errori con molto meno spazio di archiviazione.

Dataset **freddo** → Erasure Coding → disponibilità ma non località.

Dataset **caldo** → Repliche → località.

Datasets caldi si accede con minore frequenza e vengono archiviati in uno spazio di archiviazione leggermente più lento, mentre i datasets freddi sono raramente accessibili e archiviati in uno spazio di archiviazione ancora più lento.

Bisogna garantire la portabilità delle applicazioni facendo riferimento all'interfaccia e non all'implementazione.

## Yet Another Resource Negotiator (YARN)

Modulo di Hadoop responsabile della gestione delle risorse nonché dello scheduling e del monitoraggio applicazioni.

Abbiamo i **container** definiti come unità singole di lavoro (consistono di fatto in un insieme di risorse: core della CPU e MB di RAM).

Una singola unità di lavoro gira su un container.

Un'applicazione di tipo Map-Reduce può girare attraverso più container, in realtà *cerca* proprio *di distribuirsi* su più container possibili.

Un nodo può allocare più container ma un container può appartenere a un solo nodo.

Dentro un nodo posso mettere più container, favorendo la parallelizzazione.

I **Node Manager** sono i responsabili dell'allocazione di un container in un nodo.

Gestisce e controlla il ciclo di vita dei container.

Un demone Node Manager viene eseguito su un singolo nodo.

Su ogni nodo gira il proprio demone Node Manager.

Richiamo random su cosa significa demone: un demone, in informatica e più in generale nei sistemi operativi multitasking, è un programma eseguito in background, cioè senza che sia sotto il controllo diretto dell'utente, tipicamente fornendo un servizio all'utente.

Principalmente viene utilizzato sui server ma anche su normali PC.

I **Resource Manager** sono le entità che gestiscono le risorse tra le varie applicazioni, per garantire un utilizzo ottimale del cluster. Dispone di un algoritmo di scheduling per l'allocazione delle risorse. Si occupa anche di monitoraggio e salute globale grazie ai report che riceve dai node manager.

Più o meno ha lo stesso ruolo del Namenode in HDFS.

**L'Application Master** fa le veci dell'applicazione stessa.

È un'istanza di una specifica libreria di uno specifico framework.

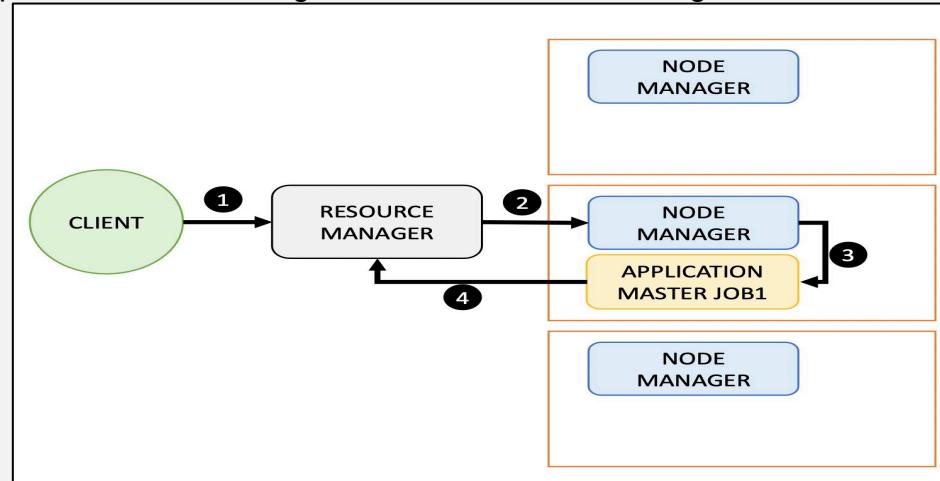
Conosce le richieste specifiche delle applicazioni.

Negozi le risorse con il Resource Manager e collabora con il Node Manager per l'assegnazione delle risorse.

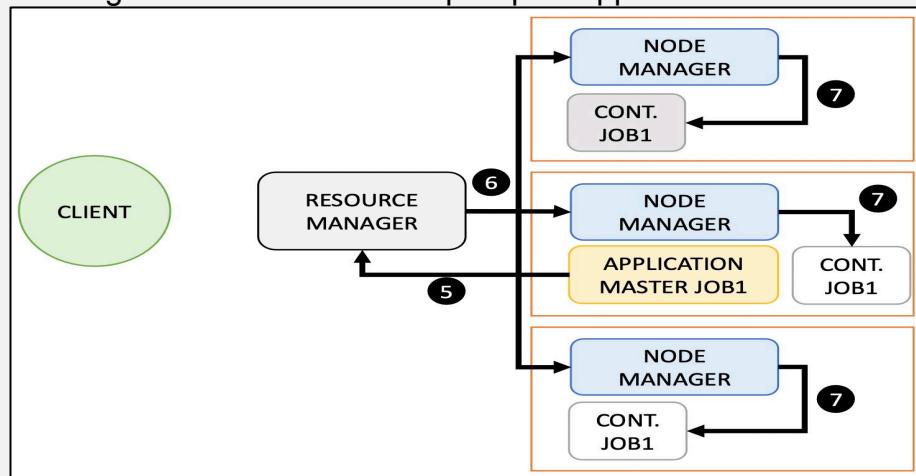
Se un task fallisce, l'Application Master negozierebbe le risorse dal Resource Manager e tenterà di rieseguire l'attività.

Possono coesistere framework diversi (ad esempio Giraph e Spark) poiché il Resource Manager può parlare con più Application Master.

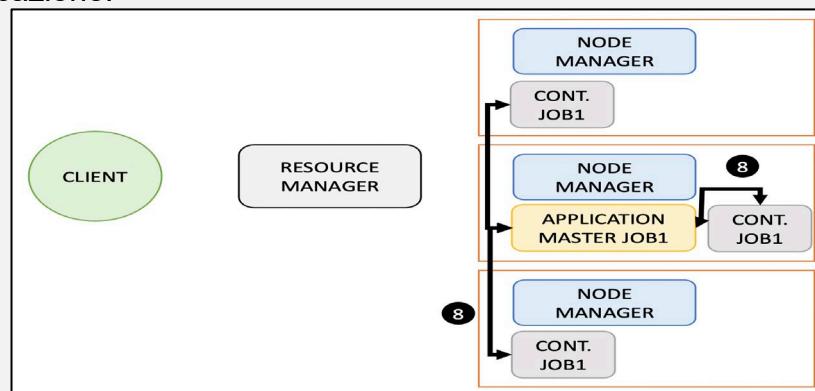
1. Un programma client invia la domanda (specificando il tipo di domanda).
2. Il Resource Manager negozia le risorse con un Node Manager per acquisire un container su quel nodo in cui avviare un'istanza dell'Application Master.
3. Il Name Manager alloca un container per l'Application Master (Application Master JOB1, sarebbe dentro al main in java).
4. L'Application Master si registra con il Resource Manager.



5. L'Application Master richiede i container al Resource Manager per l'esecuzione delle sue attività.
6. Il Resource Manager negozia le risorse con i Node Manager per acquisire i contenitori richiesti.
7. I Node Manager allocano i container per quell'Application Master.



8. Il codice dell'applicazione va in esecuzione su ogni container associato e segnalano l'avanzamento del processo all'Application Master tramite un protocollo specifico dell'applicazione.



## Job scheduling

I lavori dell'utente sono pianificati in base a uno scheduler.

Lo scheduler predefinito adotta una politica FIFO che supporta l'uso delle priorità.

Al termine di un job, viene avviato il lavoro successivo con la priorità più alta.

Non supporta la prelazione, ovvero le attività di lavori a bassa priorità non possono essere eliminate per liberare risorse per lavori ad alta priorità.

- Il **Fair Scheduler** cerca di gestire le risorse nel modo più equo possibile.
  - Se è in esecuzione un singolo job, esso ottiene tutto il cluster.
  - Man mano che vengono inviati più lavori, ad essi vengono forniti contenitori liberi in modo tale da offrire a ciascun utente una giusta quota del cluster.
  - I lavori vengono inseriti in **pool** e ogni utente ottiene il proprio pool.
  - Gli utenti ottengono la stessa quantità di risorse (in media) indipendentemente dal numero di job che hanno inviato.
  - Supporta la prelazione, può uccidere i container se necessario e allocarli a chi magari e ha bisogno ed è under capacity.
- Il **Capacity Scheduler** utilizza una gerarchia di code, ciascuna con una propria capacità allocata.
  - Funziona in modo simile a Fair Scheduler, tranne per il fatto che all'interno di ciascuna coda i lavori vengono schedulati utilizzando la pianificazione FIFO (con priorità).
  - Ciò consente agli utenti o alle organizzazioni di simulare un cluster separato con pianificazione FIFO per ciascun utente o organizzazione.

In generale, divide le risorse per team.  
Ogni team usa poi la politica FIFO nel suo spazio risorse.  
In questo modo un team che ha pochi job non viene sottomesso da un team con più job di lui.  
In queste applicazioni i singoli task richiedono poche risorse.  
Però i task sono tanti quindi conviene utilizzare più container.

## Framework MapReduce

API e classi java.

Le due classi principali sono Mapper e Reducer.

Entrambi metodi astratti che corrispondono alle rispettive funzioni, scriveremo extends Mapper e extends Reducer sulle nostre classi.

La classe main è quasi un copia e incolla in cui verrà specificato dov'è l'input, la funzione da lanciare e dove salvare il risultato. Può essere associato a un file di configurazione.

## Jobs e Tasks

Un MapReduce **job** è una singola unità di lavoro che il client richiede, esso contiene i dati in input, il programma MapReduce e le informazioni di configurazione.

Successivamente il framework MapReduce esegue il job dividendolo in **tasks** che possono essere di tipo Map o Reduce.

Hadoop divide l'input in parti di dimensione fisse chiamati **splits**, in generale, pari alla dimensione dei blocchi HDFS (di default 64 MB). 1 blocco = 1 split.

Ogni split contiene un set di records (esempio righe di un file di testo).

Il framework MapReduce crea un **map task** per ogni split, che esegue il metodo map su ogni record dello split.

- Se possibile, il map task viene eseguito sul nodo (in realtà uno dei  $k$  nodi, dove  $k$  è il *fattore di replica*) in cui si trova lo split, per ottimizzare la località dei dati e ridurre al minimo i trasferimenti di rete.

L'esecuzione del map task genera coppie chiave-valore intermedie.

- Queste coppie chiave-valore sono partizionate e ordinate per chiave all'interno di ciascuna partizione sul nodo che ha eseguito l'attività di mappatura.

Il numero totale di partizioni è pari al numero di reduce tasks per job.

I **reduce task** possono essere zero, uno o più. In base all'esigenza del programma e alle impostazioni di configurazione.

- Ogni map task memorizza il proprio output intermedio sul proprio disco locale e i reduce task recuperano tali output intermedi attraverso la rete.

Una volta recuperate tutte le coppie chiave-valore intermedie, i reduce tasks le raggruppano e infine l'output viene salvato in HDFS.

Con più reducer si ha un ordinamento a blocchi.

Con un solo reducer si ha un ordinamento totale.

Per partizionare su più macchine:

$$k \bmod p = x \in [0, p - 1] \quad p: \text{reducer}$$

*Vantaggi:* algoritmo chiaro per tutti, garantisce velocità.

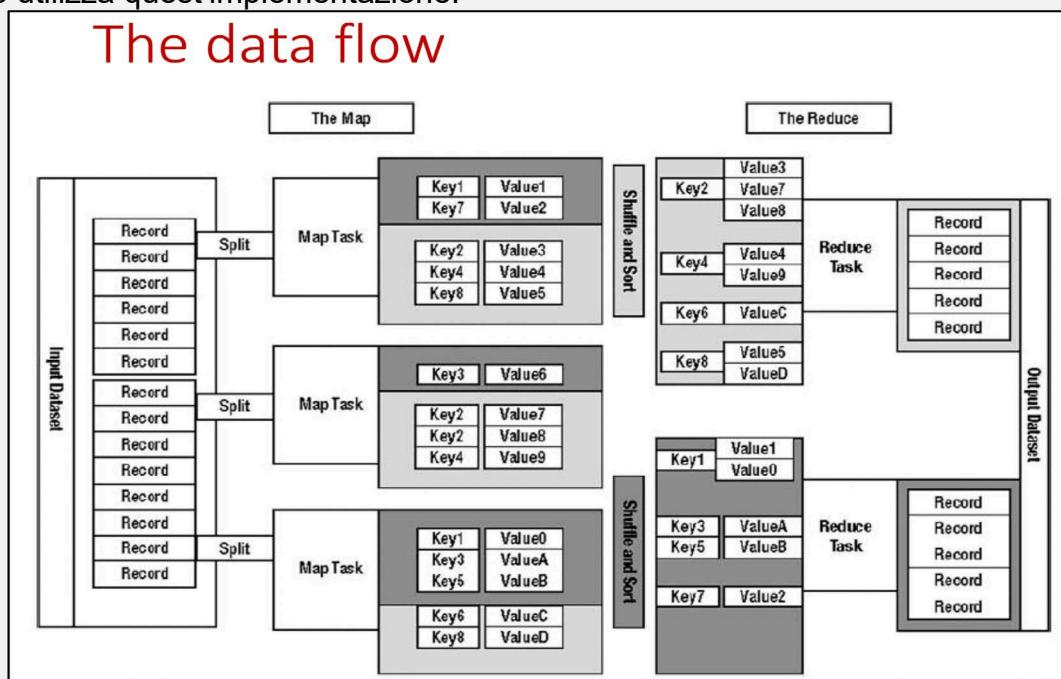
*Svantaggi:* molto sensibile alle chiavi. Non garantisce un bilanciamento ottimo.

Un possibile miglioramento è:

$$h(k) \bmod p = x \in [0, p - 1]$$

Ovvero utilizzare una funzione di hash che garantisce una distribuzione uniforme.

Hadoop utilizza quest'implementazione.



### Funzioni Combiner

Una funzione di combinazione è un metodo che viene eseguito sull'output di ogni map task e il suo output costituisce l'input per le reduce task.

È un'implementazione della classe Reducer, poiché esegue l'aggregazione locale degli output intermedi, che aiuta a ridurre la quantità di dati trasferiti tra le map tasks e le reduce tasks.

Non vi è alcuna garanzia su quante volte verrà eseguita su ciascun record intermedio. Quindi eseguirla zero, una o più volte non dovrebbe modificare l'output del Reducer. Devo essere sicuro che il risultato non cambia.

In base all'applicazione la fase di Combiner può esserci o meno: se ho un'app che fa la somma posso anticipare alcuni reducer se sono già pronti rispetto ad altri, se ho un'app che fa la media non posso perché devo aspettare tutti.

### Quanti map tasks per nodo?

Il numero di map tasks è generalmente determinato dalla dimensione totale dell'input, ovvero il numero totale di blocchi dei file di input.

Sembra che un intervallo che va da 10 a 100 sia l'ideale.

Un task per essere ragionevole deve durare almeno un minuto.

### Quanti reduce tasks?

Il numero corretto di reduce tasks sembra essere 0.95 o 1.75 moltiplicato per il numero di nodi e il numero di container massimi per nodo.

- Con 0.95, tutte le reduce tasks possono essere avviate immediatamente e iniziare a trasferire gli output delle map appena terminano.
- Con 1.75, i nodi più veloci termineranno il loro primo round di reduce tasks e avvieranno una seconda ondata di attività svolgendo un lavoro di bilanciamento del carico molto migliore. Quando ho task pesanti meglio usare il valore 1.75.
- L'aumento del numero di reduce task aumenta il sovraccarico del framework, ma aumenta il bilanciamento del carico e riduce il costo degli errori.

In memoria mi sta bene qualsiasi tipo di formato.

Quando invece devo trasferire preferisco un formato più portatile possibile.

La **serializzazione** è il processo di trasformazione di oggetti strutturati in un flusso di byte per la loro trasmissione su una rete o per la loro scrittura su una memoria persistente.

La **deserializzazione** è il processo inverso, trasforma un flusso di byte in una serie di oggetti strutturati.

La serializzazione compare in due aree distinte dell'elaborazione dei dati distribuiti, ovvero: durante la comunicazione tra processi, durante l'archiviazione persistente.

Hadoop utilizza un proprio formato per la serializzazione, chiamato **Writables**: compatto, veloce, non facile da estendere e basato su java.

L'interfaccia Writable definisce i metodi utilizzati per la serializzazione e la deserializzazione.

Viene implementata da vari wrapper per tutti i tipi primitivi Java.

- |   |
|---|
| <ul style="list-style-type: none"><li>• byte → ByteWritable</li><li>• short → ShortWritable</li><li>• int → IntWritable</li><li>• float → FloatWritable</li><li>• long → LongWritable</li><li>• double → DoubleWritable</li></ul> |
|---|

Tutti questi wrapper hanno un metodo get e un metodo set per recuperare e memorizzare il valore wrapped.

## Come trasformiamo il nostro input in coppie <chiave, valore>?

InputFormat è l'interfaccia per i formati di input di MapReduce.

Fornite da API di Hadoop (ma anche da esterni).

Sono organizzati gerarchicamente:

- FileInputFormat: utilizzati per leggere da file.
- TextInputFormat: utilizzati per leggere file formattati (default).
- NLineInputFormat: garantisce che negli splits ci siano lo stesso numero di righe.
- KeyValueTextInputFormat: legge la chiave direttamente da file di testo (delle linee), funzionale e veloce.

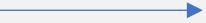
OutputFormat è l'interfaccia per i formati di output di MapReduce.

I file di output sono salvati nel File System.

## Configurazione

Vengono riportati nei vari nodi.

- 1) Possono essere tanti file modificabili:

- Locali (per singole applicazioni). 
- Globali (sempre attivi).  File XML.

- 2) Possono essere script su linea di comando. In questo caso occorre un Parse.

Hadoop utilizza l'interfaccia Tool:

- o Metodo Run: lo scrivo io.
- o ToolRunner: fa tutto lui.

## Capitolo 4 – The Think Like A Vertex Graph Processing Model

### Graphs and graph algorithms

I grafi sono onnipresenti: un modello di base di entità con connessioni tra di loro.

- Il Web, i social network online, le relazioni tra gli utenti che valutano i film in un database di film e le reti biologiche sono solo alcuni esempi di grafi.
- Che si tratti di un contesto aziendale o scientifico, vedere i dati come connessi aggiunge valore ad essi: l'analisi di queste connessioni può aiutarti a estrarre informazioni in modi altrimenti non possibili.

In questo contesto ci si basa molto sulle iterazioni (moltissime), eseguite tante volte su ciascuna entità grafica, sull'accesso random dei dati e su esplorazioni.

Per questi e altri motivi, **il paradigma MapReduce non è adatto a questi contesti.**

Si avrebbe un eccessivo overhead a causa delle molte iterazioni.

Un algoritmo grafico richiederebbe più lavori MapReduce (all'incirca uno per ogni iterazione dell'algoritmo).

- Tra lavori consecutivi, i dati vengono archiviati e ricaricati, sprecando tempo di I/O e larghezza di banda della rete.
- Potrebbero essere necessarie ulteriori iterazioni di MapReduce per verificare la terminazione dell'algoritmo.

## The BSP model (bulk synchronous parallel)

È un modello di calcolo. Considerato di basso livello.

Può essere considerato un'astrazione di hardware e software paralleli e supporta un approccio al calcolo parallelo indipendente dall'architettura e scalabile.

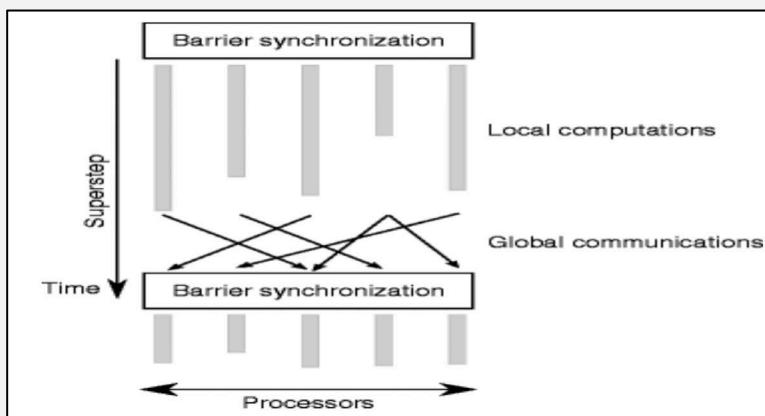
Ogni algoritmo MapReduce può essere simulato all'interno del modello BSP.

In input non deve ricevere per forza un grafo.

Possiamo vedere questo modello come una rete di  $p$  processori: ogni processore ha una memoria locale personale. Ogni iterazione dell'algoritmo prende il nome di *superstep*.

Ogni processore riceve un proprio input e lavorano in modo *asincrono*, al termine comunicano tutti il proprio output. Le comunicazioni possono essere locali o globali.

Al termine di ogni superstep viene attivata quella che prende il nome di *barriera di sincronizzazione*: essenzialmente, tutti i processori aspettano che tutti abbiano finito il loro lavoro (quindi aspettano il processore con superstep più grande) in modo da essere sincronizzati al termine di ogni iterazione e pronti per la prossima.



## Quanto costa l'algoritmo?

Abbiamo:

- $w_i$  = numero massimo di operazioni aritmetiche  $i$  eseguite da ciascuno dei  $p$  processori in un superstep  $i$  (supponendo che qualsiasi operazione aritmetica possa essere eseguita in 1 unità di tempo).
- $h_i$  = massima quantità di dati in input/output inviati/ricevuti da ciascuno dei  $p$  processori nel superstep  $i$ .
- $g$  = tempo impiegato per inviare un messaggio di dimensione 1.
- $l$  = tempo di una barriera di sincronizzazione.

Quindi il **costo totale su  $S$  supersteps** è pari a:

$$\sum_{i=1}^S w_i + g * \sum_{i=1}^S h_i + S * l$$

Poiché  $g$  e  $l$  non dipendono dall'algoritmo, il costo di quest'ultimo può essere definito in base ai seguenti tre contributi:

Tempo di calcolo:  $W = \sum_{i=1}^S w_i$

Costo della comunicazione:  $H = \sum_{i=1}^S h_i$

Numero di supersteps:  $S$ .

Il costo totale può quindi essere riscritto in modo compatto come:  $W + gH + Sl$ .

## The TLAV model

È un modello di calcolo. Considerato ad alto livello. Appositamente per grafi.

Introdotto da Google, si basa sul modello BSP.

Sembra quasi di progettare protocolli di rete.

Esegue computazioni locali simili a MapReduce.

In **input** abbiamo un grafo diretto, se non lo è lo posso simulare sostituendo ogni arco con due archi orientati in direzioni opposte.

Ogni vertice ha un ID univoco e un valore assegnato dall'utente.

Ogni arco ha un ID univoco e un valore assegnato dall'utente.

Anche in questo modello, ogni iterazione prende il nome di *superstep*.

In TLAV però si lavora direttamente con l'input.

Non serve sapere quanti processori ci sono (come in MapReduce).

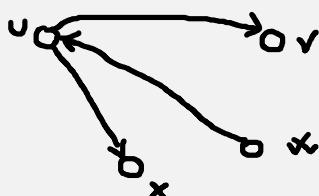
Un grafo in input su una rete di processori.

In un superstep, ogni vertice può inviare messaggi ai suoi vicini (adiacenti) tramite i suoi archi uscenti o direttamente a un vertice di cui conosce l'ID.

A ogni messaggio è associato un valore definito dall'utente.

I **messaggi** che vengono inviati durante il superstep  $i$  sono disponibili come input a partire dal superstep  $i + 1$  (il successivo). In poche parole si accumulano, in modo da garantire la sincronizzazione. Posso anche non accumularli e decidere di inviarli direttamente senza sincronizzazione ma il sistema diventa così più complesso e meno conveniente per cluster di grandi dimensioni.

Durante un superstep, ogni vertice riceve in input i messaggi che gli sono stati inviati durante il superstep precedente ed esegue una **funzione di calcolo** locale basata sul suo valore corrente e sul valore del messaggio ricevuto. I messaggi vengono elaborati con questa compute function che idealmente in tutte le esecuzioni viene eseguita in parallelo. Le funzioni map e reduce del MapReduce qui sono le compute function.



Con l'utilizzo della funzione compute un vertice può:

- Cambiare il suo valore.
- Cambiare i valori dei suoi archi uscenti.
- Inviare nuovi messaggi.
- Alterare la topologia del grafo (aggiungendo o rimuovendo archi o vertici).

All'inizio, tutti i vertici partono come attivi.

Ognuno guardando il suo stato può disattivarsi.

Se però al superstep successivo riceve un messaggio da un altro nodo si riattiva automaticamente. L'algoritmo **termina** quando tutti i vertici sono inattivi e non ci sono messaggi in transito.

**L'output** dell'algoritmo è spesso lo stesso grafo ma con valori diversi.

BSP esegue un mapping 1:1 (1 processore  $p$ , 1 nodo  $n$ )

TLAV esegue un mapping 1:molti (1 processore  $p$ , più nodi  $n$ )

Il mapping di TLAV porta a dover bilanciare i nodi sui vari processori.

Sia  $G$  un grafo con  $n$  vertici e  $m$  archi in input a un cluster di  $p$  processori, con  $n \gg p$ . Il grafo  $G$  è partizionato casualmente in  $p$  parti bilanciate, una per processore (ognuna contenente circa  $n/p$  vertici).  $h(v.id) \bmod p$

Ad ogni superstep, ogni processore esegue la compute function sui suoi vertici. La transizione tra supersteps consecutivi è gestita tramite barriere di sincronizzazione. Ho vincoli sia sul numero dei messaggi che sulla loro dimensione.

Ogni messaggio dovrebbe essere piccolo:  $O(\log n)$ .

*Limiti di banda:* messaggi tra vertici dello stesso processore sono meno costosi che tra vertici di processori diversi.

Il numero max di messaggi inviati ad ogni superstep dovrebbe essere ordine degli archi  $O(m)$ .

Questo implicherebbe che il costo totale per comunicazione su  $S$  supersteps è ordine di  $H \in O(m * s)$ .

Limiti di memoria: ogni vertice deve archiviare poche informazioni  $O(n)$  vertici  $O(m)$  archi. Il singolo vertice difficilmente ha grandi operazioni.

Questo implica che il tempo di calcolo totale su  $S$  supersteps è:  $W \in O(m * S)$ .

Se questi vincoli sono soddisfatti, l'unico parametro che resta da ottimizzare è il numero di supersteps. Spesso abbiamo  $S \in O(n)$ .

Il modello TLAV è spesso utilizzato insieme a Information Diffusion → Social Network.

## Capitolo 5 – Apache Giraph

Framework per elaborazione distribuita, versione 1.3.0.

Fa parte dell'ecosistema Hadoop ed è utile per l'analisi di un ampio set di dati connessi su un cluster di macchine.

È progettato per l'esecuzione di algoritmi ad alta intensità di calcolo e che elaborano il grafo in input in modo iterativo.

In particolare, Giraph API implementa il paradigma TLAV.

### Architettura

Giraph è basata su 3 servizi di rete: masters, workers, e coordinators.

Il grafo in input è partizionato dal **master** che successivamente assegna tali parti a vari workers tramite il coordinatore (una o più parti per worker). Monitora anche la loro salute. Anche in questo caso ci sono dei master in standby pronti ad intervenire nel caso di malfunzionamento del primo master.

Il master può eseguire dei piccoli codici globali.

Ogni **worker** invoca il metodo compute sui suoi vertici.

Le transizioni tra superstep sono coordinate dal master.

Le connessioni tra worker sono invece gestite dal coordinatore.

I workers utilizzano dei **checkpoint** per evitare di dover rieseguire il lavoro da zero nel caso di un fallimento di un'iterazione. Salvano uno snapshot ogni tot iterazioni e possono ripartire dalla più recente.

I **coordinatori** offrono servizi di locazione, comunicazione, sincronizzazione, configurazioni distribuite e nominano i servizi di registrazione.

Un esempio di coordinatore è **ZooKeeper**, scelta standard nell'ecosistema Hadoop.

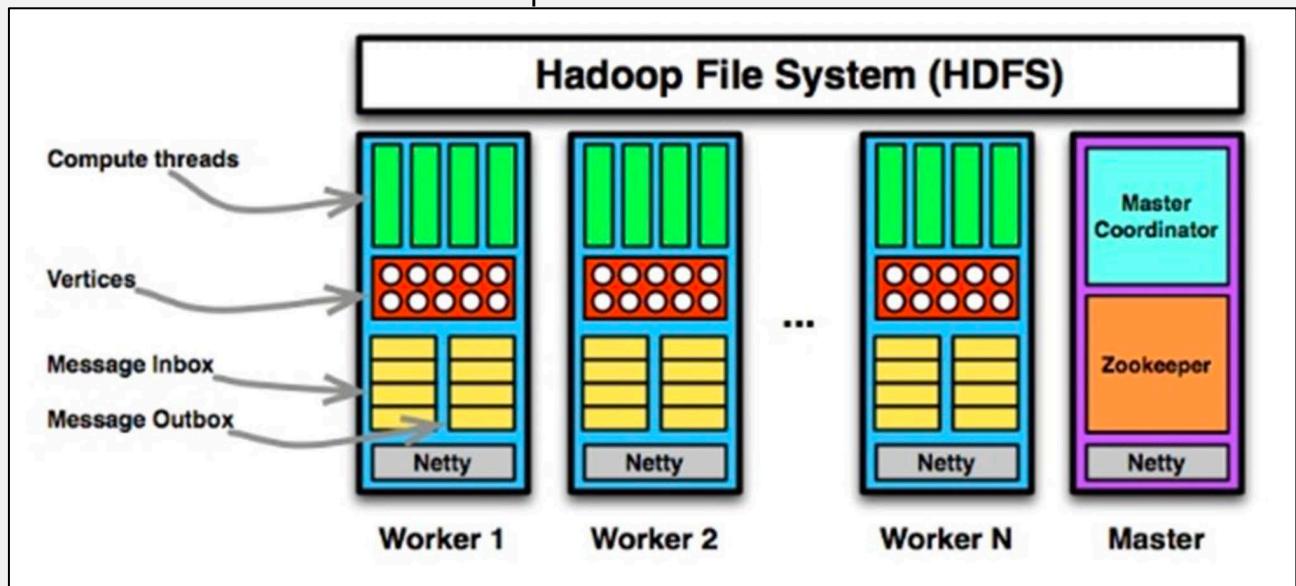
Può stare sia nel master che a parte.

Un insieme di nodi eseguono un servizio di coordinazione chiamato **ensemble**, se non c'è viene creato ogni volta.

**Netty:** librerie di semplificazione di scrittura di rete. Framework client-server per lo sviluppo di applicazioni Java per le comunicazioni telematiche.

È asincrono, orientato ad eventi e usato per semplificare i server dei socket TCP e UDP.

Per avere un traffico regolare di rete si utilizzano dei **buffer** sia in input che in output. Possiamo definire delle classi di computazione diversi.



Da un punto di vista di alto livello, il grafo di input può essere descritto in due modi:

- Rappresentazione basata sui vertici: raccolta di vertici (lista di adiacenza).

La classe da estendere si chiama `VertexInputFormat`.

- Rappresentazione basata sugli archi: raccolta di archi (non è possibile fornire informazioni sui vertici). La classe da estendere si chiama `EdgeInputFormat`.

Entrambe le classi hanno le rispettive rappresentazioni per l'output.

L'API Giraph offre funzionalità avanzate per ottimizzare i calcoli come: Aggregators, MasterCompute, Combiners, Partitioners.

Un **aggregatore** è una funzione globale con la quale i vertici possono inviare dei valori (non sappiamo l'ordine con cui tali valori arrivano). Durante ogni superstep l'aggregatore aggrega i valori ricevuti. Il risultato è disponibile al superstep successivo.

Gli aggregatori possono essere di 2 tipologie: **regolari**, ovvero aggregatori che durano 1 superstep e alla fine di esso il loro valore viene resettato, e **persistenti**, ovvero aggregatori il cui valore non viene mai resettato (esempio: contatore totale dei messaggi). Prima di utilizzare qualsiasi aggregatore esso deve registrarsi col Master.

La classe **MasterCompute** è utile nel caso in cui vogliamo iniettare del codice globale.

Un **combinatore** è una funzione che appunto combina i vari messaggi inviati a un vertice. È utile per ridurre la quantità di messaggi inviati tra superstep.

Non ci sono garanzie su quante volte una funzione combiner viene eseguita.

Questa funzione è utile quando ad esempio un reducer non può partire perché magari si sta aspettando la fine di un mapper. Allora intanto quello che può fare è eseguire una funzione combiner che gli permette di combinare e ridurre i messaggi.

Se ad esempio il programma esegue delle somme, con la funzione combiner intanto è possibile sommare i valori che si hanno finora e poi inviare un unico messaggio al superstep successivo.

A livello di partizione, un buon algoritmo dovrebbe creare partizioni veloci, bilanciate (in termini di archi e/o vertici) e che minimizzi il numero di archi tra partizioni diverse (o massimizzi il numero di archi nella stessa partizione).

Meglio pochi archi tra nodi di partizioni differenti.

La funzione di base di Giraph per il partizionamento è quella di una funzione hash che in modo veloce distribuisce i nodi tra le partizioni in modo bilanciato.

$$h(k) \bmod p = x \in [0, p - 1]$$

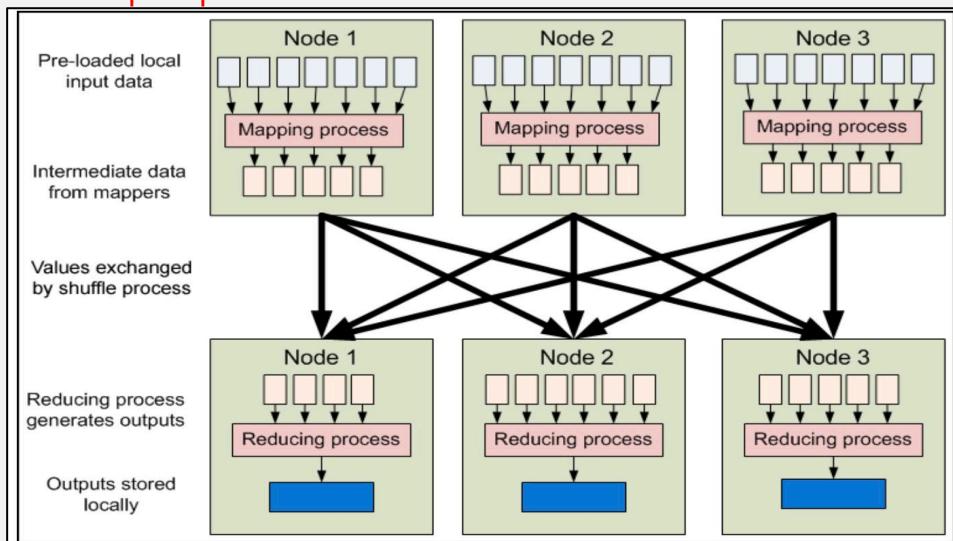
Tuttavia, questa soluzione non è ottima rispetto al numero di archi tra partizioni diverse.

Una possibile alternativa, più intelligente, è quella di partizionare i nodi in base a come sono disposti gli altri. Esempio dei vertici colorati.

Inizialmente tutti i nodi sono neutri e inizio a colorarli sulla base del numero di colori (partizioni) che ho, quando seleziono un nodo guardo il colore dei nodi vicini in modo da assegnarli un colore uguale per evitare di aumentare le comunicazioni tra partizioni diverse. Banalmente questa funzione converge colorando i vertici tutti dello stesso colore. Si può evitare inserendo una soglia per ogni colore, in modo tale che quando seleziono un vertice, se il colore a cui è più vicino ha raggiunto o sta per raggiungere il suo tetto massimo, lo farò uscire più difficilmente (moneta pesata con percentuale regolata in base al raggiungimento o meno della soglia).

## Capitolo 6 – Apache Spark

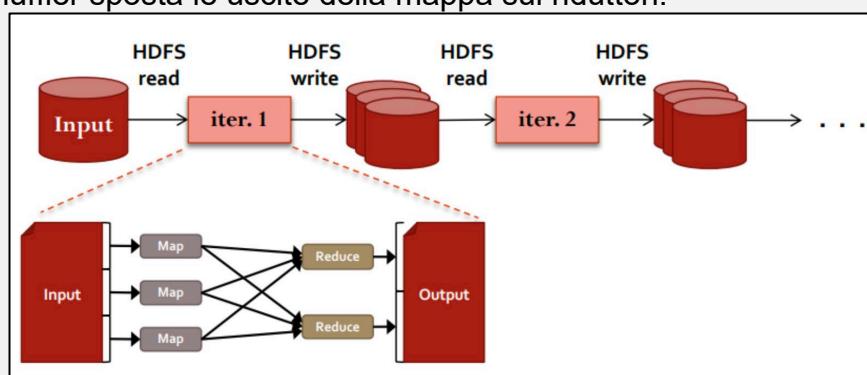
### Limitazioni di Hadoop MapReduce



Quando i risultati intermedi dei mapper vengono inviati ai reducer, c'è una comunicazione sulla rete.

Il partitioner determina a quale partizione andrà una data coppia chiave-valore.

Il predefinito calcola un valore hash per la chiave e assegna la partizione in base a questo risultato. Lo shuffler sposta le uscite della mappa sui riduttori.



Ridurre il costo di questi step porta alla drastica riduzione del costo in termini di computazione.

A volte un contributo al rallentamento viene dato anche dal disco, dall'elevato numero di comunicazioni, dalla serializzazione.

Questo rende scomodo l'utilizzo di Hadoop MapReduce per applicazioni iterative come machine learning e data mining.

## Architettura Spark

È un motore di esecuzione, ha bisogno di uno stack software sotto per funzionare bene.

Cerca di utilizzare più la memoria RAM che le altre memorie.

Ha un ampio set di API di alto livello. È un framework per il calcolo distribuito.

È pienamente compatibile con Hadoop: può leggere e scrivere su qualsiasi sistema basato su Hadoop come HDFS e può girare al di sopra del cluster Hadoop.

Le applicazioni vengono eseguite come insiemi indipendenti di processi su un cluster, coordinati dall'oggetto **SparkContext** nel programma principale (il programma driver).

Per essere eseguito su un cluster, SparkContext può connettersi a diversi tipi di gestori di cluster (come **YARN**), che allocano risorse tra le applicazioni.

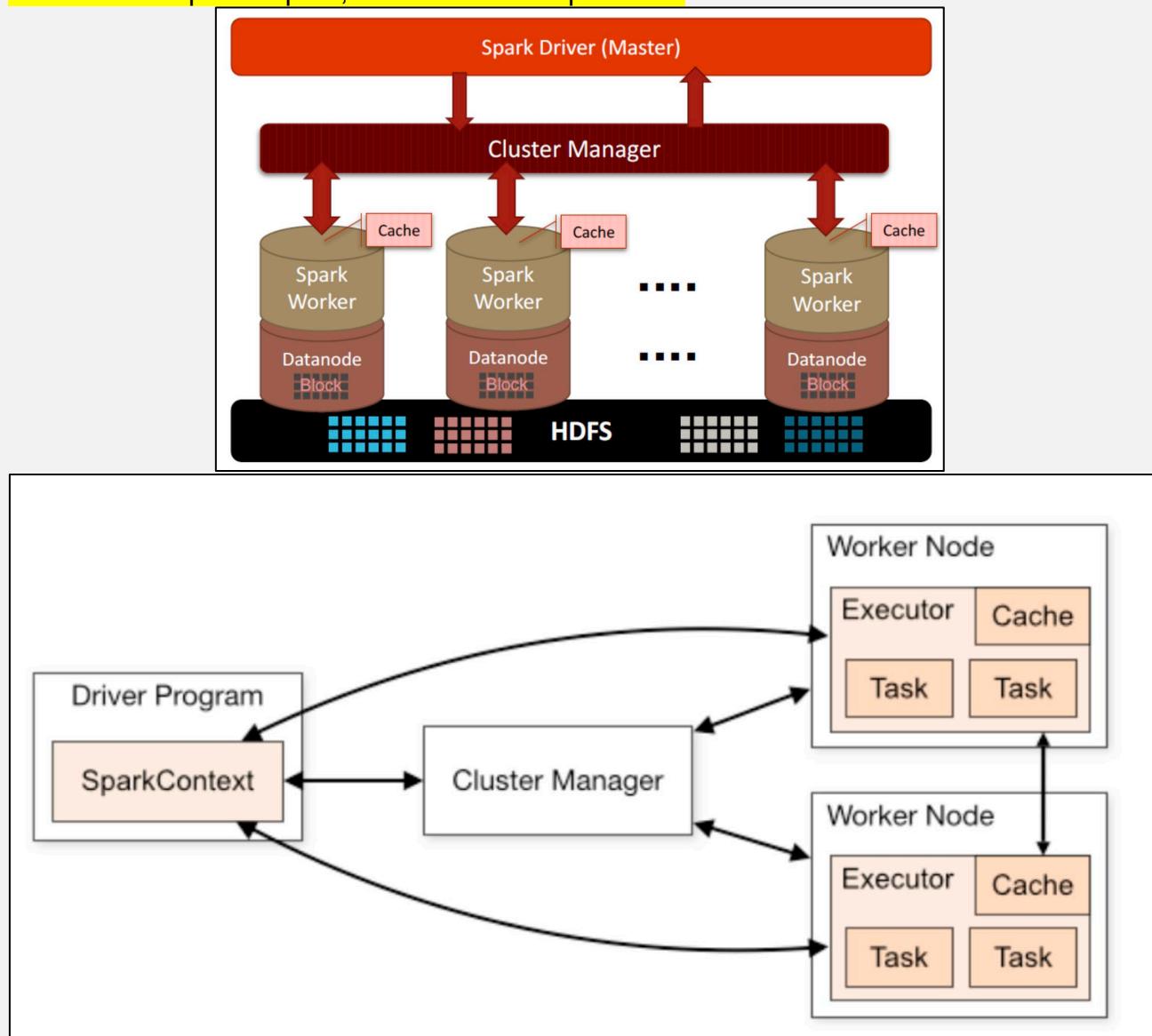
Una volta connesso, SparkContext acquisisce gli esecutori sui nodi del cluster, che eseguono i calcoli e archiviano i dati.

Successivamente, invia il codice dell'applicazione agli esecutori.

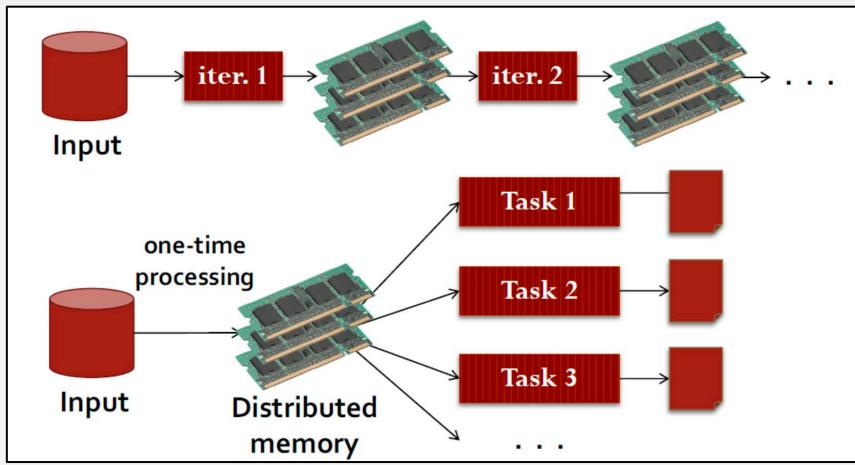
Infine, invia i task agli esecutori.

Ogni applicazione ottiene i propri processi di esecutori, che rimangono attivi per la durata dell'intera applicazione ed eseguono attività in più thread.

Se uso Hadoop con Spark, YARN resta sempre attivo.

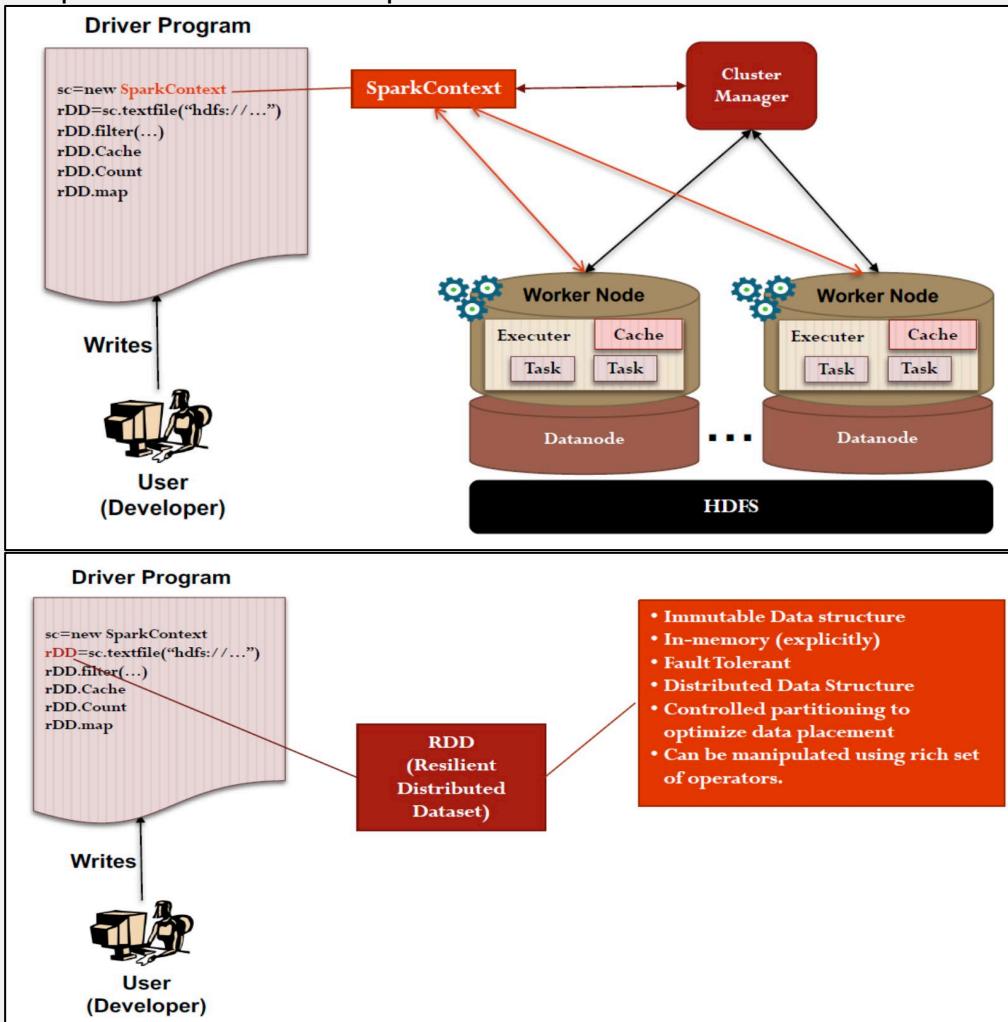


Spark è basato sulla velocità: molta RAM e SSD come memoria secondaria.



### Modello di programmazione Spark

Il programmatore scrive direttamente il programma (e non solo le funzioni come in Map Reduce) che poi viene distribuito su più basso livello.



**RDD** = collezione di dati che sono partizionati (con ridondanza) tra i nodi del cluster Spark.  
È l'API di Spark (di basso livello) che processa dati in forma distribuita.

Possono essere manipolati attraverso vari parallelvi operativi.

Vengono ricreati automaticamente in caso di guasto.

Possono persistere in memoria, su disco o in entrambi.

Possono essere partizionati per controllare l'elaborazione parallela.

Generalmente al fine dell'operazione la RDD viene eliminata ma si può comunque scegliere di rendere persistente andando ad aumentare la computazione.

Le RDD sono composte da quattro parti:

- 1) **Partitions**: parti atomiche del dataset. Come lo divido per eseguirlo.
- 2) **Dependencies**: ci spiega come siamo arrivati ad ottenere la RDD, in questo modo posso tornare all'indietro nel caso di fallimenti/ricreazioni.
- 3) **Function**: funzioni che hanno permesso di arrivare all'RDD.
- 4) **Metadata**: schema di partizioni dove sono piazzate le varie parti.

**Trasformazioni**: sono operazioni su un RDD che restituiscono oggetti RDD o raccolte di RDD. L'input resta immutabile, ne viene restituito uno nuovo, non una modifica. L'input in base a cosa abbiamo deciso o viene cancellato al termine delle trasformazioni o resta.

Esempi di trasformazioni possono essere:

- Map (func): mapping 1:1.
- Filter (func): ci restituisce true (la porto a RDD di output) o false (la filtro).
- Distinct([numTasks]): restituisce nuova RDD senza duplicati.

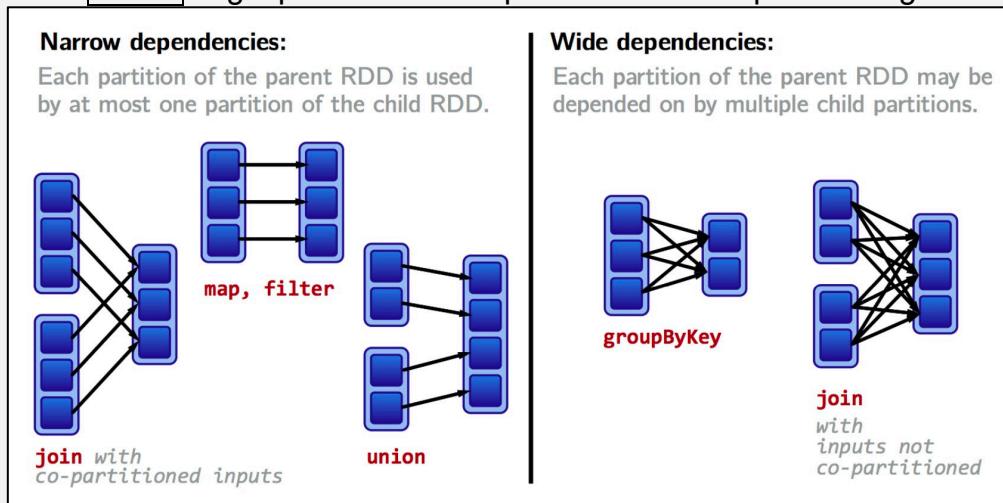
Le trasformazioni sono pigre, ovvero non vengono eseguite immediatamente, ma solo quando un'azione richiede la restituzione di un risultato al driver.

Finché non chiedo il risultato Spark aspetta.

Quando lo chiedo ottimizzo (più trasformazioni ho meglio è, inutile fare un output per ogni trasformazione, aspetta che gli dico tutto prima).

Le trasformazioni sono di due tipologie:

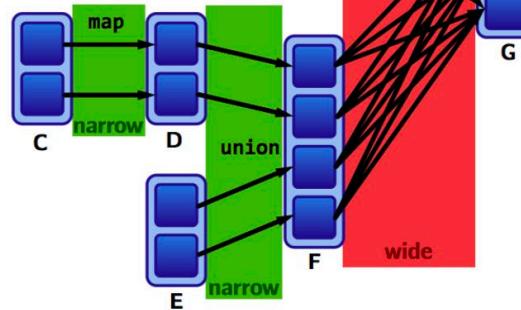
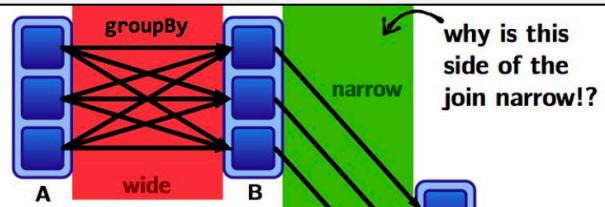
- **Narrow**: non richiede la fase di shuffle sulla rete. Compila tutto in uno **stage** senza comunicare con altri. **1:1**. Ogni partizione RDD padre è usata al più da una RDD figlio.
- **Wide**: richiede la fase di shuffle sulla rete. Devo prima eseguire le trasformazioni Narrow. **1:molti**. Ogni partizione RDD padre è usata da più RDD figlie.



Let's visualize an example program and its dependencies.

**Wide transformations:**  
groupBy, join

**Narrow transformations:**  
map, union, join 



La trasformazione di join da B a G è Narrow perché groupByKey partiziona già le chiavi e le posiziona in modo appropriato in B dopo lo shuffle.

Pertanto operazioni come il join a volte possono essere Narrow e talvolta Wide.

### Azioni

Sono tutte quelle operazioni che ritornano dei valori, visibili in qualche modo.

Le azioni fanno sì che i dati vengano restituiti al driver o salvati nell'output, nonché il recupero dei dati e l'esecuzione di tutte le trasformazioni sugli RDD. Fino a quando non viene eseguita alcuna azione, non si accede nemmeno ai dati da elaborare. Solo le azioni possono concretizzare l'intero processo con dati reali.

Alcuni esempi sono le funzioni che restituiscono qualcosa, come ad esempio:

- `Count()`: ritorna il numero di elementi nel dataset.
- `Reduce(func)`: aggrega elementi del dataset attraverso una funzione specifica, la quale richiede due argomenti e ne restituisce uno.
- `Collect()`: restituisce tutti gli elementi del dataset come un array.
- `Take(n)`: ritorna un array con i primi n elementi del dataset.

### Operazioni

Persistenti: per salvare in memoria cache dataset per operazioni future, grazie a questo settaggio le operazioni future sono molto più veloci.

Le RDD possono essere rese persistenti attraverso i metodi `persist()` o `cache()`.

- La cache di Spark è fault-tolerant (a tolleranza di errore): se una partizione di un RDD viene persa, verrà automaticamente ricalcolata utilizzando le trasformazioni che l'hanno creata originariamente.

Utilizzando `persist` si può specificare il livello di archiviazione per la persistenza di un RDD.

L'uso del metodo `cache()` in realtà è solo una scorciatoia per il livello di archiviazione predefinito, che è `MEMORY_ONLY`.

Altri possibili `StorageLevel` per la persistenza sono: `MEMORY_ONLY`, `MEMORY_AND_DISK`, `MEMORY_ONLY_SER`, `MEMORY_AND_DISK_SER`, `DISK_ONLY`, `MEMORY_ONLY_2`, ...

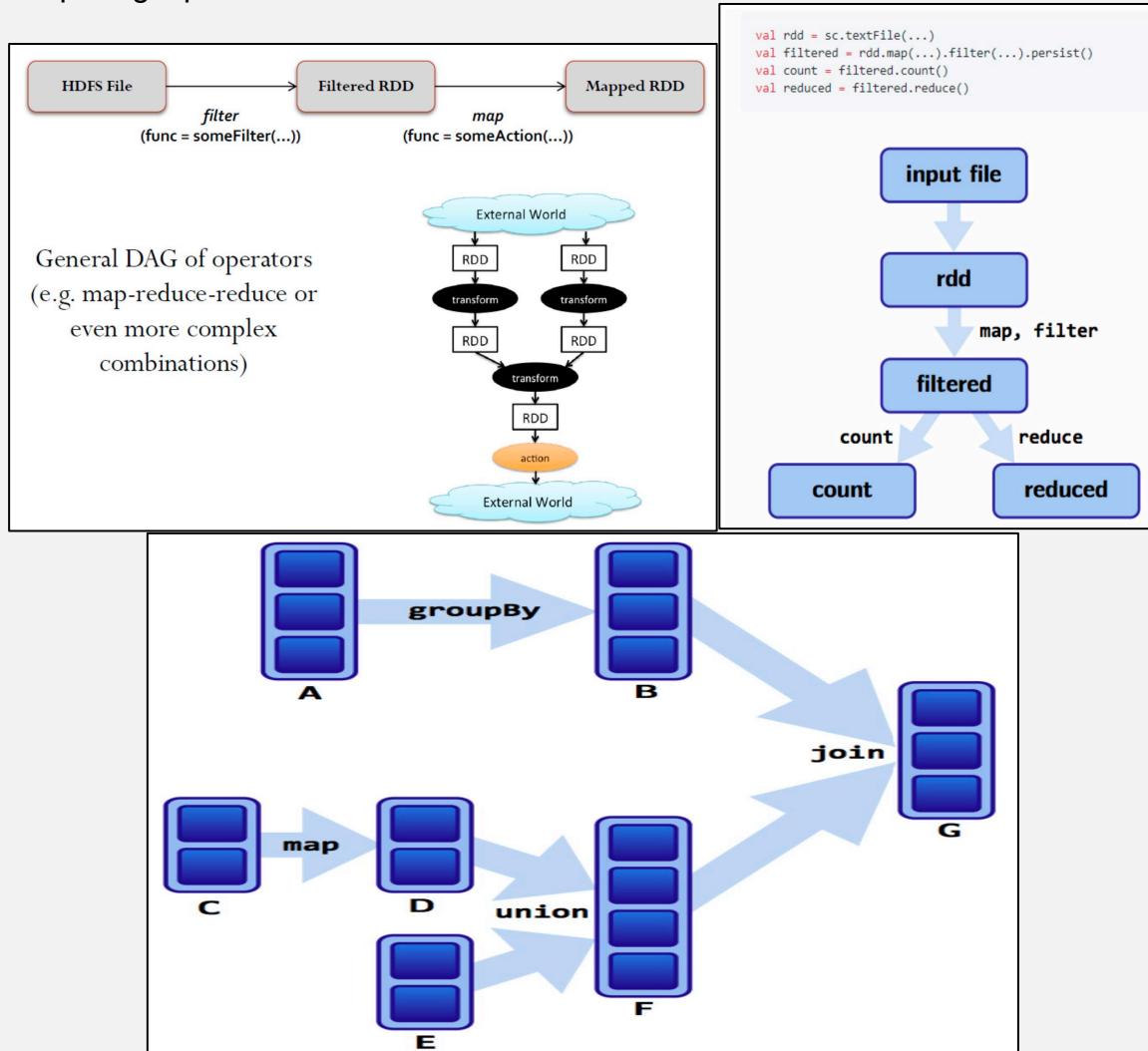
Quale livello di archiviazione è il migliore?

Bisogna cercare di tenere in memoria il più possibile. La serializzazione rende gli oggetti molto più efficienti in termini di spazio.

Cercare di non riversare su disco a meno che le funzioni che hanno elaborato i dataset non siano costose. Utilizzare la replica solo se si desidera la tolleranza agli errori perché la persistenza da sola non garantisce la robustezza. Comunque la robustezza in esecuzione ci interessa poco in questo corso.

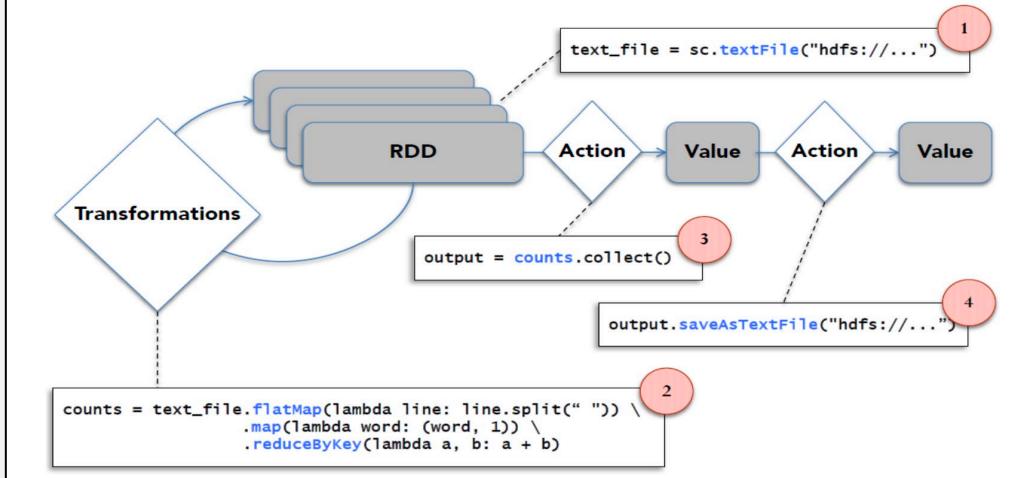
### DAG (Directed Acyclic Graph)

La creazione di RDD, le loro trasformazioni ed azioni generano un DAG di operatori. Il DAG è compilato in stage e ogni stage è eseguito come una serie di task. Un task per ogni partizione.



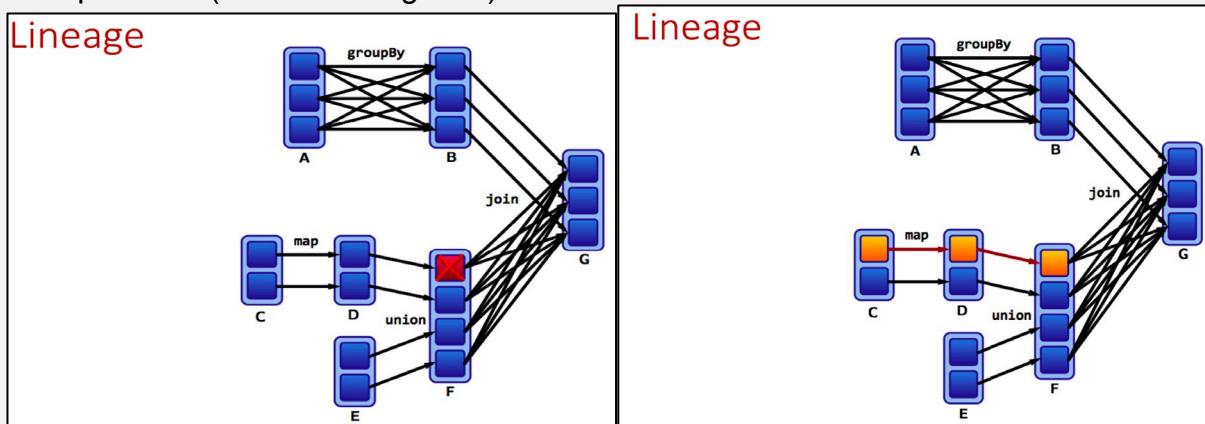
Gli stage sono sequenze di RDD che non hanno una fase di shuffle in mezzo (Narrow). Spark crea un task per ogni partizione nel nuovo RDD.

# Conceptual representation



Ogni RDD ha un **lineage** (lignaggio) (piano di esecuzione logico), ovvero un grafo composto da tutti gli RDD padre che lo hanno generato.

Questo grafo contiene le informazioni che vengono utilizzate per ricalcolare i dati persi in caso di problemi (tolleranza ai guasti).



Torno indietro nel processo fino a quando non torno su una partizione persistente da cui posso ripartire e rieseguire le successive funzioni.

Il caso peggiore sarebbe quello di dover tornare indietro fino all'input e rieseguire tutto.

## Capitolo 7 – NoSQL

### Breve richiamo SQL

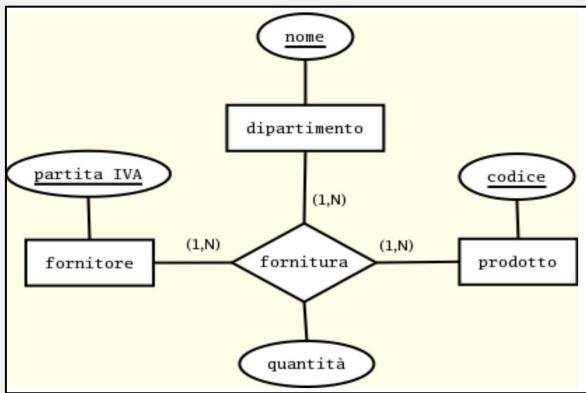
Rappresentazione logica del dataset (struttura e relazioni).

**Chiave primaria:** identifica univocamente.

**Chiave esterna:** attributi che identificano e mettono in relazione una tabella con altre.

**Tuple:** righe o record.

Un modello di dati è spesso descritto utilizzando alcune notazioni grafiche (ad es. diagrammi ER, diagrammi UML).



**fornitore(partitaIVA)**

**dipartimento(nome)**

**prodotto(codice)**

**fornitura(fornitore, dipartimento, prodotto, quantità)**

**Transazione:** unità di lavoro effettuata all'interno del DBMS.

È un'operazione complessa che può coinvolgere più query e tabelle ma è da considerarsi come un'unica operazione (come fare un'unica query).

Se dopo averla eseguita non ci sono errori allora si fa il **commit** e si conclude.

Se dopo averla eseguita ci sono errori allora si fa il **rollback** ovvero si torna indietro a prima della transazione.

Gli RDBMS garantiscono **4** proprietà per ogni transazione, dette **ACID**:

- 1) **Atomicità**: ogni transazione è indivisibile, si esegue tutta o niente. Ci semplifica la logica della nostra applicazione. Non riusciamo a vedere un momento in cui la transazione è in atto, la vediamo solo quando ha finito. Se fallisce una parte fallisce il tutto, ci previene update a metà.
  - 2) **Consistenza**: validità dei dati scritti. Garantisce la consistenza di tutti i vincoli definiti nel nostro schema. Non garantisce però l'assenza di malfunzionamenti. Quando possibile è bene garantire vincoli a livello di schema (database). Qualsiasi transazione porterà il database da uno stato valido a un altro.
  - 3) **Isolamento**: permette di eseguire le transazioni in modo sequenziale. Se eseguo operazioni di sola lettura posso eseguirle anche più in parallelo. Si possono ottenere diversi livelli di isolamento. Il più vincolato (e sicuro) è il **serializable**: effetto di percepire le transazioni una dopo l'altra. Un lost update è un aggiornamento di cui non si vede l'effetto, magari sovrascritto subito da un'altra operazione.
  - 4) **2PL protocol**: quando una transazione vuole accedere a un oggetto/tabella deve metterci un lucchetto, se ci riesce può farci ciò che vuole. Generalmente il lucchetto è esclusivo in scrittura e condiviso in lettura. Se 1 sta scrivendo e 2 vuole scrivere sullo stesso oggetto deve aspettare che finisca 1. Ci sono due fasi, nella prima vengono richiesti ed assegnati i lock, nella seconda vengono rilasciati. Potrebbero verificarsi deadlock (in informatica, lo stallo o deadlock indica una situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa.), in tal caso si terminano forzatamente.
  - 4) **Durabilità**: garantisce che una transazione rimane consistente anche in caso di crash, salti di corrente, ecc. Molti RDBMS implementano la durabilità scrivendo le transazioni in un registro (log) che può essere rielaborato per ricreare lo stato del sistema subito prima di qualsiasi errore successivo.
- Una transazione viene considerata confermata solo dopo che è stata inserita nel registro. Ci sono comunque casi irrecuperabili, se brucia la sala server sono cazzi.

**Impedance mismatch**: differenza tra modello relazionale e struttura dati in memoria.

Scollatura tra DB e applicazione. Risolvibile con interfacce software.

Ogni applicazione può memorizzare i dati in modo indipendente.

L'architettura orientata ai servizi (SOA) è un modo di progettare l'interoperabilità in cui i servizi sono forniti dalle applicazioni attraverso dei protocolli di comunicazione.

Ad esempio i servizi web possono comunicare tramite HTTP(S) scambiandosi strutture dati rappresentate in XML o JSON.

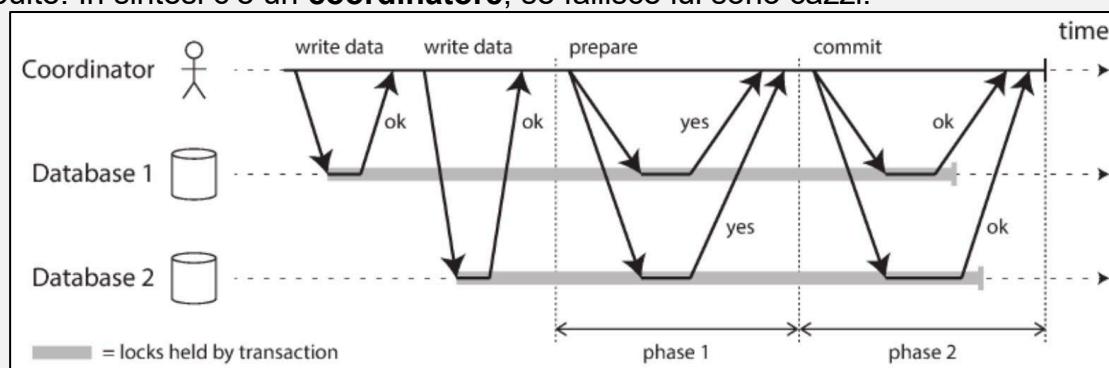
La complessità viene trasferita alla progettazione di API.

Così si ha molta più potenza e flessibilità rispetto a SQL.

### Come implemento le transazioni in un sistema distribuito?

Innanzitutto, non possiamo sfruttare le proprietà ACID perché abbiamo più database.

Si passa dal protocollo 2PL al **protocollo 2PC** per ottenere l'atomicità nelle transazioni distribuite. In sintesi c'è un **coordinatore**, se fallisce lui sono caZZI.



Il più grande **svantaggio** del protocollo 2PC è che si tratta di un **protocollo di blocco**.

Dopo che un partecipante ha inviato un messaggio di accordo al coordinatore, si bloccherà fino alla ricezione di un commit o di un rollback.

Una possibile soluzione è applicare il Saga Pattern a livello di applicazione.

- Una **saga** è una **sequenza di transazioni locali**. Ogni transazione locale aggiorna il database e pubblica un messaggio o un evento per attivare la successiva transazione locale. Se una transazione locale fallisce perché viola una regola aziendale, la saga esegue una serie di transazioni compensative che annullano le modifiche apportate dalle precedenti transazioni locali.

Il **vantaggio** principale consiste nell'eliminare lo svantaggio precedente, ovvero che le risorse non sono bloccate durante una transazione.

Ci sono due modi di coordinare le saghe:

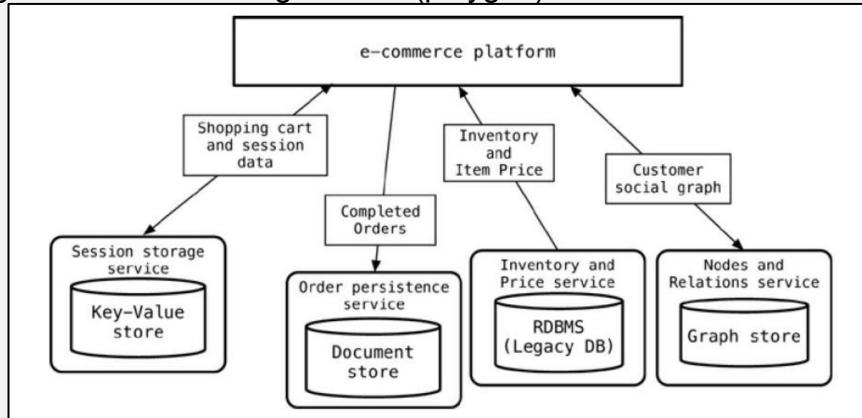
- **Coreografia**: ogni transazione pubblica un evento e avvia la fase successiva. I messaggi sono garantiti dalla presenza di un broker.
- **Orchestrazione**: un orchestratore (oggetto) indica ai partecipanti quali transazioni locali eseguire. La logica è più semplice ma sono possibili colli di bottiglia.

## Database NoSQL (Not Only SQL)

Serie di tecnologie che derivano da quello originale SQL. È un movimento.

Lavorano con dati non strutturati. Schemi più flessibili o assenti.

Parla tante lingue a livello di storage di dati (polyglot).



Sono progettati per girare su cluster (non vale per graph DB).

## Aggregato

Collezione di dati appartenenti a un dominio specifico legati tra loro da una radice.

Ci interessa perché abbiamo a che fare con grandi moli di dati.

Un aggregato **rappresenta un'unità per operazioni atomiche**, i dati di un aggregato sono sullo stesso nodo.

La **radice** dell'aggregato **garantisce la coerenza delle modifiche** apportate all'interno dell'aggregato impedendo agli oggetti esterni di contenere riferimenti ai suoi membri.

Per questo motivo ci da poche soluzioni per relazioni intra-aggregato.

Se ad esempio volessimo sommare due aggregati in uno dovremmo implementare una nostra logica perché non c'è supporto.

Si dice che i DB key-value, documenti e column-family siano **aggregate-oriented**, nel senso che sono progettati per funzionare con gli aggregati.

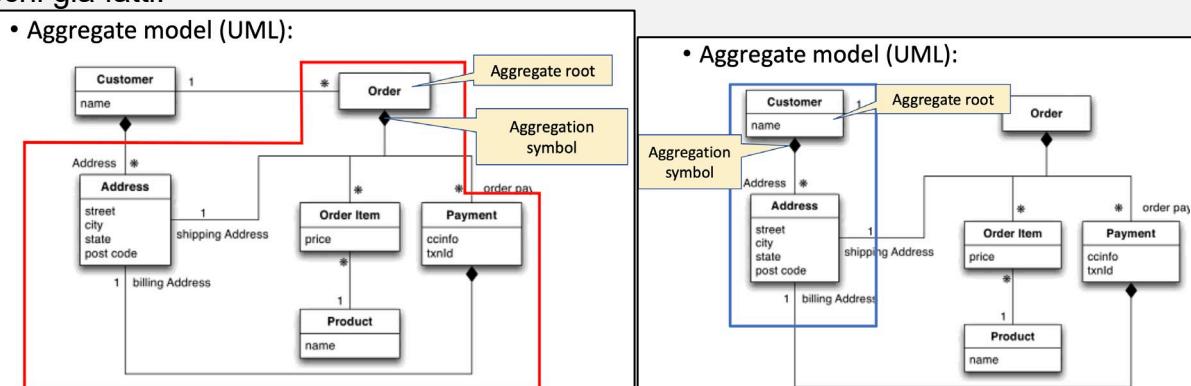
Funzionano meglio quando la maggior parte delle interazioni con i dati avviene con lo stesso aggregato. Ad esempio quando c'è bisogno di ottenere un ordine di un e-commerce con tutti i suoi dettagli, è meglio memorizzare l'intero ordine come oggetto aggregato.

Se il concetto di aggregato non è presente in un sistema allora esso è aggregate-ignorant. Ad esempio, i DB relazionali e i DB Graph.

Comunque gli aggregati sono utili anche se ci impediscono di fatto di fare il join.

Abbiamo ridondanza ma ci semplifica di molto la logica dell'applicazione.

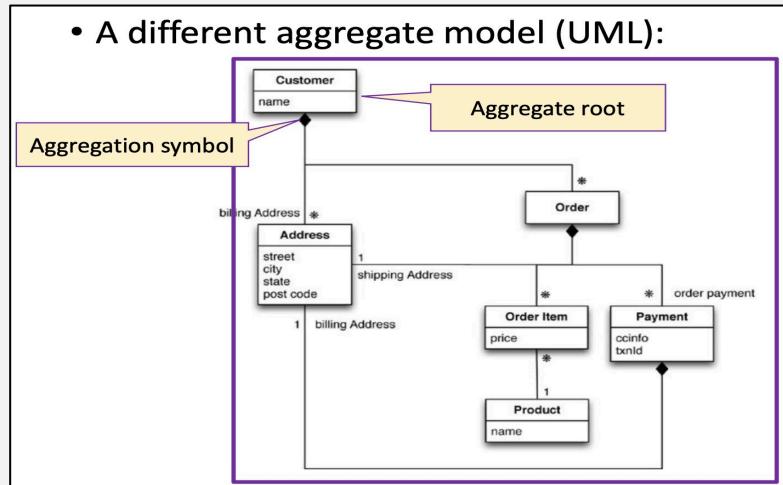
Ad esempio se cambia il prezzo di un ordine non vado a modificare il valore in ordini vecchi già fatti.



Ne abbiamo bisogno perché se abbiamo frammentato i nostri dati (ad esempio un nodo mantiene gli ordini e un nodo mantiene gli indirizzi) è difficile fare operazioni veloci perché abbiamo latenza elevata a causa dell'interrogazione di nodi sparsi in giro.

Con gli aggregati posso fare tutte le operazioni che mi servono in un unico cluster.

Dovremmo ridurre al minimo il numero di aggregati a cui accediamo durante l'interazione dei dati.



Quando si opera su un cluster, è necessario ridurre al minimo il numero di nodi da interrogare durante la raccolta dei dati.

Definendo esplicitamente gli aggregati, diamo al DB una visione importante di quali informazioni dovrebbero essere memorizzate insieme: **tutti i dati riguardanti un aggregato dovrebbero essere posti sullo stesso nodo**.

Ogni volta che eseguiamo un'operazione logica, 1 unità è 1 aggregato.

Progettiamo il nostro DB avendo in mente quali query andremo a utilizzare.

Se lavoro sempre con tutti i dati posso creare un unico grande aggregato.

**Più è grande l'aggregato più il server si scalda perché il payload della richiesta aumenta.**  
È sempre una questione di trade-off.

Bisogna valutare anche la qualità della connessione.

Il nostro scopo è quello di realizzare applicazioni efficienti.

Tipicamente un aggregato ci indica quali dati devono essere salvati insieme e che non devono essere spezzati. Ho garanzie su un singolo aggregato ma non su multipli.

Nel modello <key, value> key è la radice dell'aggregato, per ogni key mantengo un valore ovvero l'aggregato stesso. Può essere qualsiasi cosa.

Nel modello Document si può memorizzare solo documenti JSON ma grazie a questo ci fornisce un metodo di interrogazione.

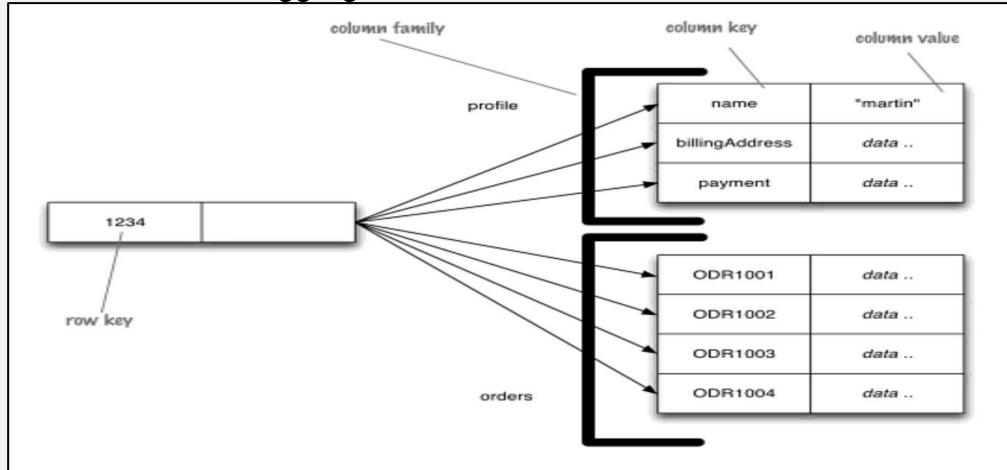
## Column-family

I database Column-family **hanno una struttura aggregata a due livelli**:

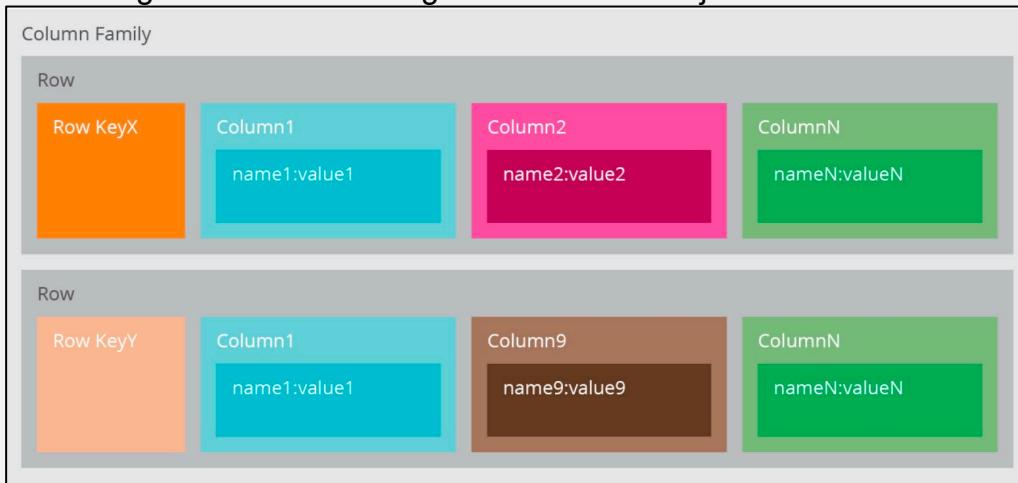
- Nel 1° livello abbiamo un simil <key, value> dove la chiave è l'identificatore **riga**.
- Il 2° livello prende il nome di **colonne**, organizzate in **famiglie**. Ogni colonna fa parte di una sola famiglia. Le colonne sono dinamiche, se ad esempio rappresentano degli ordini chi fa più ordini ha più colonne. Si suppone che le colonne della stessa famiglia siano accessibili insieme.

Anche la visualizzazione può essere di due tipi:

- **Vista orientata alle righe:** ogni riga è un aggregato, le column-family sono blocchi di dati all'interno di un aggregato.



- **Vista orientata alle colonne:** ogni famiglia di colonne definisce un tipo di record con righe per ogni record. È possibile visualizzare una riga come unione di record in tutte le famiglie di colonne. Le righe sono come dei join di record.



**Vantaggio:** possiamo memorizzare dati come liste che aumentano o diminuiscono in base alle necessità. Le colonne all'interno di una famiglia possono essere ordinate in base alle loro chiavi. Il database conosce il raggruppamento delle colonne, utilizza queste informazioni per l'archiviazione e il comportamento di accesso.

## Relazioni

Non possiamo fare operazioni tra aggregati diversi.

Gli aggregati sono utili per mettere insieme i dati a cui si accede comunemente in modo congiunto. I singoli aggregati sono atomici ma gli aggregati multipli non lo sono.

### Come gestire le relazioni tra aggregati?

Il modo più semplice per fornire tale collegamento consiste nell'utilizzare un ID per ciascun elemento nell'aggregato a cui si potrebbe fare riferimento.

Questo non è visibile al database.

Possono esserci **materialized view** ovvero query memorizzate nella cache.

Questo è un aspetto centrale per i database orientati agli aggregati poiché alcune query potrebbero non adattarsi alla struttura aggregata.

Ci sono due approcci:

**Eager**: da usare se servono subito gli aggiornamenti.

Questo approccio va bene quando abbiamo letture più frequenti e poche scritture.

**Lazy**: gli aggiornamenti vengono eseguiti a intervalli regolari.

È utile quando gli aggiornamenti dei dati non sono critici per l'azienda.

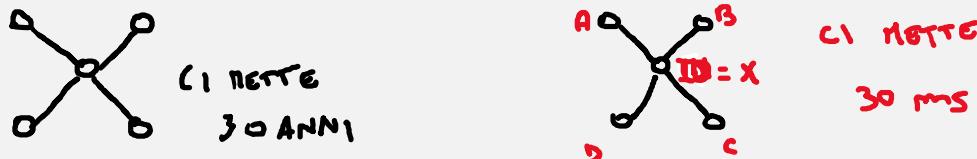
Ad esempio se si analizzano i dati il giorno dopo si fa un update ogni notte.

Nel caso in cui abbiamo dati fortemente correlati tra loro i database orientati agli aggregati diventano scomodi, anche quelli relazionali perché si farebbero molti join costosi.

Si utilizzano quindi i database Graphs.

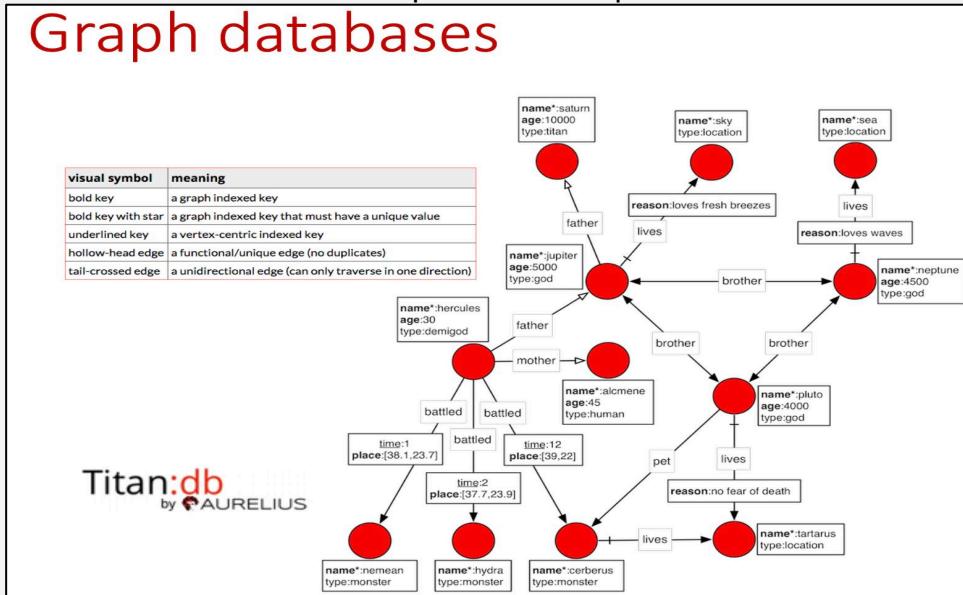
Gli archi diventano dei puntatori tra nodi, la relazione tra nodi è proprio visibile come dato.

Infatti nodi e archi possono essere dotati di etichette e proprietà.



I database a grafo sono ideali per modellare dati su larga scala con relazioni complesse come social network, preferenze di prodotto, ecc.

Però l'esecuzione su un cluster è molto più difficile in questo caso.



Le relazioni vengono ricostruite attraverso attraversamenti veloci.

**Le prestazioni delle query sono più importanti della loro velocità di inserimento.**

Difficilmente supportano la frammentazione dei dati, al massimo se ci sono repliche intere distribuite su più cluster. Generalmente le proprietà ACID sono garantite sulle transazioni tra nodi e archi.

## Capitolo 8 – Modelli Distribuiti E Problemi Di Consistenza

Un database distribuito è una raccolta di più database logicamente interconnessi situati nei nodi di un sistema distribuito.

Un sistema di gestione di database distribuiti (DBMS distribuito, DDBMS) è quindi il sistema software che permette la gestione del database distribuito e rende la distribuzione trasparente agli utenti.

Tipicamente distinguiamo tre tipi di DDBMS:

- **Geograficamente distribuito**: interconnesso da reti geografiche caratterizzate da lunghe latenze dei messaggi e tassi di errore elevati.
- **Unica sede**: sono costituiti da cluster in un data center, consentendo scambi molto più veloci che portano a latenze dei messaggi più brevi e tassi di errore molto bassi.
- **Irido**: “più uniche sedi connesse tra loro”.

Noi studieremo unica sede e irido.

Ci concentreremo su sistemi NoSQL perché sono scalabili, tolleranti agli errori, disponibili e supportano diversi modelli di dati con differenti linguaggi che differiscono da SQL.

Tutto questo a volte a scapito della consistenza.

Distribuiamo o quando la macchina che abbiamo a disposizione non ce la fa o quando la nostra applicazione è intrinsecamente distribuita.

Per avere un DB sul cluster abbiamo bisogno di un sistema di distribuzione.

Come distribuiamo i dati sui vari nodi?

I database orientati agli aggregati facilitano la distribuzione dei dati, poiché il meccanismo di distribuzione deve spostare solo l'aggregato senza preoccuparsi dei dati correlati (in quanto sono tutti contenuti nell'aggregato!). Ci dicono cosa si deve tenere unito.

Esistono due opzioni (non esclusive) per la distribuzione dei dati:

- **Sharding**: distribuisce i diversi dati su più nodi in modo che ciascun nodo funga da singola origine per un sottoinsieme di dati.
- **Replication**: copia i dati su più nodi in modo che ogni bit di dati possa essere trovato in più posizioni e la perdita di essi non precluda l'annullamento.

### Sharding

Dati distribuiti su più nodi, ogni nodo è uno shard (frammento).

Possiamo vedere lo shard come una parte del DB affidata a un cluster.

Questo approccio migliora la scalabilità orizzontale attraverso il bilanciamento del carico.

+ nodi + volume di dati, bilancio e scalo.

Posso frammentare il DB come voglio l'importante è che non si spezza un aggregato.

I dati a cui generalmente si accede insieme devono essere posizionati sullo stesso nodo.

Gli aggregati sono un'unità naturale di sharding.

Se perdiamo un nodo perdiamo le sue informazioni e questo è un problema, per evitarlo si ricorre alla replicazione (si replicano i vari nodi).

I possibili utilizzi sono: partizioni bilanciate in termini di dati/traffico, supporto di intervalli di query (dammi le query dal 2000 al 2002) e semplicità di instradamento delle query.

Se instrado le query casualmente (**Random Sharding**) garantisco il bilanciamento ma complico gli altri due possibili utilizzi.

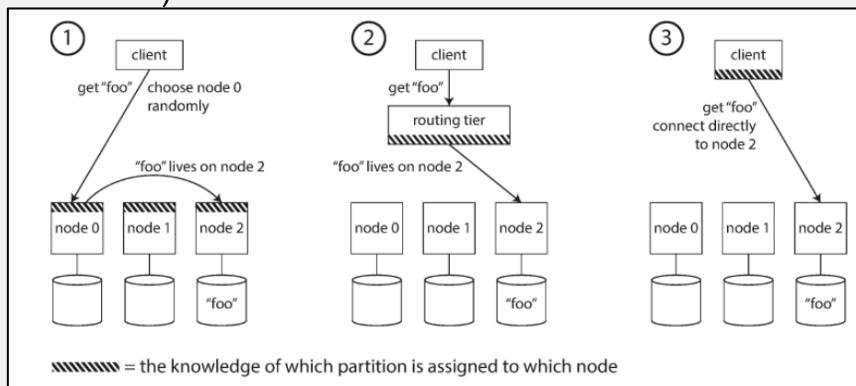
Se instrado le query in modo funzionale (**Sharding by features**) garantisco il supporto di intervalli di query e la semplicità di instradamento delle query ma non il bilanciamento.

Ad esempio Ebay mette su un frammento/shard/nodo i commenti di tot film e su un frammento/shard/nodo le recensioni. Se ci sono 100 recensioni e 10000 commenti si fa 100 e 10000 senza bilanciare nulla.

Se instrado le query basandomi su chiavi ordinate (**Sharding by key range**) ogni frammento possiede tutte le chiavi di uno specifico intervallo. Garantisco il supporto di intervalli di query e la semplicità di instradamento delle query ma non il bilanciamento. Versione modificata è la **Sharding by hashed key** dove la chiave viene prima sottoposta ad hashing utilizzando una funzione hash unidirezionale e ogni frammento possiede tutte le chiavi in un intervallo specifico di hash. garantisco il bilanciamento e la semplicità di instradamento delle query ma non il supporto di intervalli di query. Per migliorare quest'ultimo aspetto è possibile concatenare più indici e solo il primo indice viene sottoposto a hash mentre gli altri vengono ordinati per le query di intervallo. I confini dello shard possono essere scelti in modo pseudocasuale per migliorare il ribilanciamento.

### Come fa un client a sapere a quale nodo chiedere alcuni dati?

- 1- Il client chiede a qualsiasi nodo e poi il nodo instrada la richiesta al nodo di destinazione (quello giusto).
- 2- Il client instrada tutte le richieste livello per livello automaticamente.
- 3- Il client è a conoscenza dello schema di partizionamento e dell'assegnazione dei nodi (non conviene).



### Cosa succede quando le partizioni vengono ribilanciate?

È possibile utilizzare protocolli ad hoc per mantenere il mapping dei nodi come fa ad esempio ZooKeeper, oppure utilizzare protocolli speciali come ad esempio il protocollo Gossip (nessuno sa da chi è partita l'informazione ma la sanno tutti) per diffondere qualsiasi modifica nello stato del cluster.

Gli stati di mantenimento dei cluster devono essere leggeri.

## Replicazione

*La replicazione dei dati va a migliorare aspetti come la latenza e la disponibilità.*

L'utilizzo e la gestione delle repliche dipendono dal caso.

Avere **repliche vicino agli utenti** ad esempio ha come **obiettivo l'abbassamento della latenza**. **Più repliche ci sono per ogni dato più aumenta la produttività in lettura.**

Se un nodo fallisce le altre repliche possono gestire le richieste in lettura/scrittura che aveva tale nodo e quindi aumentare la disponibilità.

Immuni: Datacenter a Milano, basta perché è solo per l'Italia. Poi replica i dati.

Dall'altra parte della medaglia, **l'utilizzo di repliche porta alla condivisione delle informazioni tra di esse** per garantire la coerenza/consistenza.

Bisogna mantenere le repliche allineate. Sennò si incorre in **problemi di consistenza**, uno dei principali problemi che affliggono i sistemi distribuiti.

Per **inconsistenza** intendiamo la gestione di un dato da parte di più entità senza la sincronizzazione. La sincronizzazione garantisce la consistenza: accedi sempre alla versione più aggiornata (recente). Trade-off tra latenza e consistenza fondamentale.

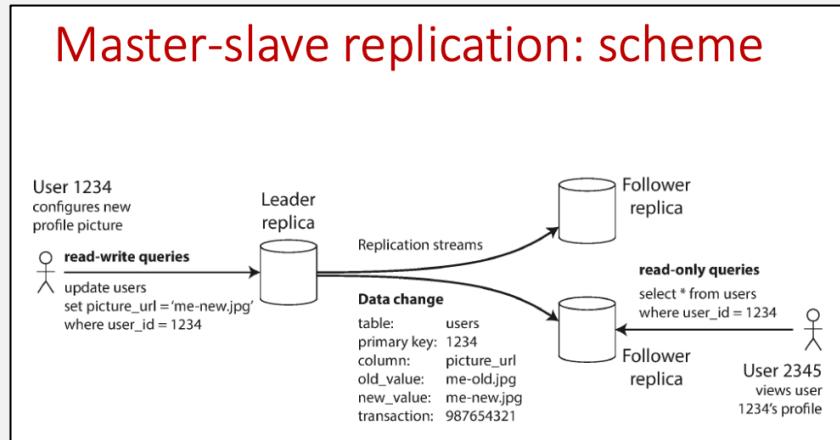
Nei modelli distribuiti la replicazione può essere utilizzata con lo Sharding oppure da sola.

Per garantire la sincronizzazione è possibile utilizzare uno **schema Master-Slave**.

Il master è la sorgente dei dati ed è l'unico che può eseguire operazioni di scrittura (update) mentre gli slave possono sincronizzarsi col master per aggiornare le loro informazioni (possono eseguire solo operazioni di lettura).

Un processo di replicazione aggiorna gli slave con il master.

Lo schema Master-Slave è detto anche Leader-Follower.



Il processo di replica può essere sincrono o asincrono.

La replica sincrona impone la coerenza ma richiede che tutti i leader lavorino e rispondano rapidamente. La replica asincrona potrebbe non garantire la durabilità e potrebbe portare a letture incoerenti. I modelli misti offrono buoni compromessi.

Possiamo gestire più letture aggiungendo più slave e garantendo che le richieste di lettura siano gestite direttamente da essi.

Questo approccio richiede che la maggior parte degli slave utilizzi la replica asincrona.

In base a quale nodo slave l'utente fa la richiesta di lettura potrebbe non leggere il dato più aggiornato immediatamente se ancora non è arrivato l'aggiornamento.

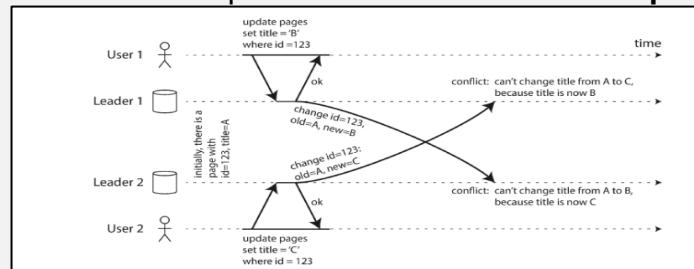
Se modifico la foto profilo di FaceBook dall'Italia non è detto che in America compaia subito mentre un mio parente in Italia la vede subito. **Conflitto di tipo Read-Read**.

Per i dataset con traffico di scrittura intenso, il master rappresenta un collo di bottiglia (si ricordi che HDFS è progettato per scrivere poco e leggere molto).

Se abbiamo un solo master infatti in caso di suo fallimento gli slave possono ancora gestire richieste di lettura degli utenti ma non si può più effettuare una nuova operazione di scrittura. **In termini di resilienza a livello di scrittura il master è un single-point-of-failure**.

Per questo motivo è possibile utilizzare un nodo master secondario che può prendere le sue veci. Oppure utilizzare più nodi master in generale e garantire una maggior velocità di processamento delle operazioni (se ci sono più master se uno deve fare una cosa la può fare subito tanto ci sono altri che fanno il resto).

Avere più master ci riporta al problema precedente, bisogna stare attenti che nessuno di essi modifichi lo stesso dato contemporaneamente. **Conflitto di tipo Write-Write**.



I master possono coordinarsi per evitare conflitti, ad esempio la maggioranza dev'essere d'accordo su un'operazione di scrittura. Oppure le scritture in conflitto possono talvolta essere unite insieme (ad esempio, modifiche apportate a parti diverse di un documento).

Per garantire la sincronizzazione è possibile utilizzare uno **schema Peer-to-Peer**. Fondamentalmente, tutti i nodi sono peer e accettano sia operazioni di lettura che operazioni di scrittura. È uno schema simmetrico che garantisce la scalabilità. Rischio di inconsistenza sempre dietro l'angolo. Ho conflitti sia di tipo Read-Read che Write-Write.

### Problemi di consistenza

Read-Read: 2 client vedono due versioni diverse dello stesso dato.

Read-Write: 1 client legge un dato mentre 1 lo modifica.

Write-Write: 2 client aggiornano lo stesso dato contemporaneamente.

Si affrontano come visto in precedenza coi vari schemi possibili.

Perdite di aggiornamenti e operazioni contemporanee vanno gestiti.

### Tecniche basate su Quorum

Ci danno degli strumenti per rendere il nostro sistema o più performante o più consistente. Non risolvono tutto ma sono delle tecniche molto utilizzate nei sistemi distribuiti.

**Essenzialmente il quorum è il minimo numero di voti che un'operazione distribuita deve ottenere per poter essere eseguita.**

Si fanno votare le diverse repliche per decidere se una modifica dev'essere accettata o no.

Se un dato ha replica 3, tutte e 3 possono votare per accettare l'operazione oppure no.

Indichiamo con **N** il numero di repliche dello stesso dato (quindi è un numero piccolo).

Indichiamo con **R** il quorum da raggiungere per la lettura.

Indichiamo con **W** il quorum da raggiungere per la scrittura.

**R** e **W** sono i voti minimi per accettare operazioni di lettura e di scrittura.

Possono anche essere lo stesso valore ma da un punto di vista teorico è possibile assegnare due valori diversi.

Se il valore di **R + W** è proprio **N** non ho problemi di conflitti Write-Write perché se ho due operazioni di write diverse allora sicuramente ci saranno due repliche che si votano contro e si blocca il processo.

Se una replica si perde posso abbassare di 1 il valore del quorum.

Tuttavia avere un quorum esattamente pari a N significa che devo avere un ACK da parte di tutte le repliche e quindi ho perso un po' il concetto di calcolo distribuito.

I valori ideali sono: **R + W > N**

In questo modo evito conflitti di tipo Read-Read.

Ho almeno una replica che ha accettato sia un'operazione di lettura che di scrittura.

Troverò sempre almeno una replica che ha il valore più aggiornato.

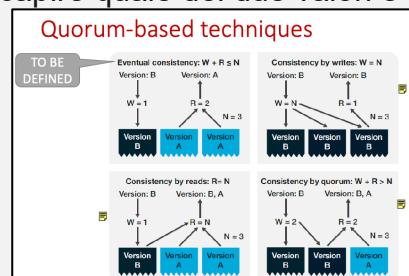
Esempio: 3 repliche, quindi **N = 3**. I client sono 2.

Supponendo di avere 2 client che fanno operazioni di lettura, uno è fortunato e gli restituiscono un valore consistente e aggiornato perché ha interrogato due repliche che hanno lo stesso valore. **R + W = 4 > 3**

L'altro client gli arriva due risposte inconsistenti quindi vede due valori diversi.

Il client a quel punto può evitare la scrittura o comunque si rende conto che c'è un problema.

Può anche essere in grado di capire quale dei due valori è il più recente.



Ponendo  $R + W > N$  risolve conflitti RR e RW.

Ponendo  $W > N / 2$  risolve conflitti WW.

In un cluster di grandi dimensioni con un numero di nodi molto maggiore di  $N$ , una partizione di rete potrebbe non consentire a un client di raggiungere un quorum.

- Invece di restituire errori a tutte le richieste di scrittura, alcuni sistemi utilizzano un quorum sciutto (**sloppy quorum**), ovvero i nodi raggiungibili possono confermare un'operazione di scrittura anche se non fanno parte degli  $N$  nodi originali.

Significa incaricare un'altra replica facendola passare per quella non raggiungibile e facendosi dare una risposta. Chiaramente è superficiale.

Non blocca l'operazione ma raggiunge il quorum con un voto fasullo.

- Quando le  $N$  repliche originali diventano disponibili, ricevono le nuove scritture dai nodi che le hanno temporaneamente accettate per loro conto (**hinted handoff**).

La replica se torna operativa viene aggiornata di quello che è successo da chi l'ha sostituita.

### Orologi inaffidabili

Nei sistemi distribuiti si punta ad un hardware comune ma che insieme fa la forza.

Server con buon rapporto qualità prezzo, senza spendere troppo.

Quindi non ci aspettiamo dentro orologi atomici e neanche ricevitori GPS che ci danno precisioni molto elevate.

Periodicamente tutti i computer si sincronizzano attraverso un server con orologio ad alta precisione comunicandolo attraverso un protocollo di rete e si riallineano.

Non è banale ci vogliono protocolli ad hoc.

Li devo allineare una volta al giorno ad esempio.

Quindi il timestamp su due valori non è detto sia affidabile e veritiero in generale.

Virtualizzare l'orologio al quarzo è sconveniente, non offre una soluzione ottima.

Questo perché accumula ritardi importanti.

### Vettore Dei Tempi

Nei sistemi distribuiti, vari eventi possono verificarsi in momenti diversi a causa di processi concorrenti e gli orologi con orari effettivi sono difficili da sincronizzare nelle reti globali.

Le versioni vettoriali possono essere utilizzate per portare un po' di ordine in questi eventi.

Si accontentano nel dirci qual è l'ordine giusto tra due eventi diversi.

Non ci danno un orario assoluto ma almeno ci ordinano in modo giusto gli eventi o ci dicono quando c'è conflitto.

Vettore con  $N$  componenti o contatori  $C_i$  con  $i = 1, \dots, N$  ( $N$  componenti = #repliche).

Ogni replica  $R_i$  ha un vettore dei tempi  $V_i$ .

$V_i = [C_1, \dots, C_N]$ .

Ogni replica ha associato un vettore con 3 componenti se abbiamo ad esempio 3 repliche.

Abbiamo un vettore con 3 valori ciascuno di essi riguarda i 3 componenti delle 3 repliche.

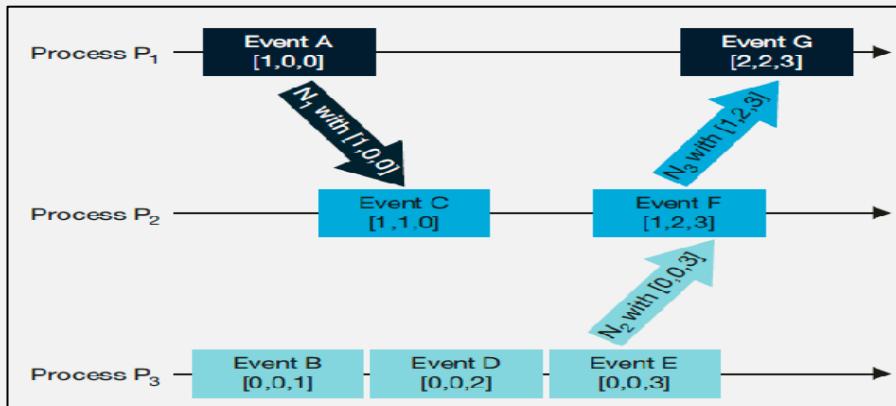
Inizialmente si parte con tutti zero,  $V_i = [0, \dots, 0]$ .

In ogni messaggio la replica mittente invia il proprio vettore temporale  $V_i$  alle repliche destinate.

Ogni volta che una replica riceve un messaggio prende la sua componente  $C_i$  nel suo vettore  $V_i$  e la incrementa di 1.

Quando una replica  $i$  riceve un messaggio da un'altra replica  $t$ , unisce il suo vettore e quello ricevuto da  $t$  componente per componente tenendo il valore più alto.

Se il vettore della replica  $t$  è più alto del suo, allora  $t$  è una replica più aggiornata di  $i$ .



Se X è  $[0, 0, 1]$

Se Y è  $[1, 0, 2]$

Allora  $Y > X$  perché è maggiore o uguale in tutte le componenti.

Se X è  $[0, 0, 1]$

Se Y è  $[1, 1, 0]$

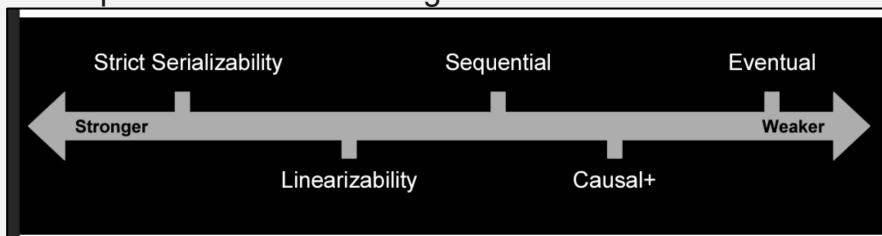
Allora non posso dire niente perché non tutte le componenti di X o di Y sono maggiori o uguali dell'altro. **C'è un conflitto.**

## Capitolo 9 – The CAP Theorem And Beyond

### Consistency models

Una storia è una collezione di operazioni.

Un sistema distribuito supporta un determinato modello di coerenza se ogni storia è conforme a un corrispondente insieme di regole.



### Strict serializability

Modello utilizzato nei database relazionali.

Significa che le operazioni si sono verificate in un certo ordine, coerente con l'ordine in tempo reale di tali operazioni.

Se l'operazione A viene completata prima dell'inizio dell'operazione B, allora A dovrebbe apparire davanti a B nell'ordine di serializzazione.

Le operazioni (transazioni) possono comportare diverse operazioni primitive eseguite in ordine. Le operazioni primitive avvengono atomicamente: l'operazione secondaria di una transazione non sembra intercalare (modificare il numero o l'andamento degli elementi di una serie o successione) con le operazioni secondarie di altre transazioni.

### Linearizability

Rispetto allo strict serializability che devono sempre essere in maniera serializzata.

Lavoriamo con singole operazioni (primitive).

Perdo la transazione ma continuò ad avere un clock real time.

Se ripeto la stessa storia due volte ottengo sempre lo stesso risultato.

È possibile comunque avere dei piccoli ritardi.

Ogni operazione sembra avvenire atomicamente, in un certo ordine, coerente con l'ordine in tempo reale di tali operazioni.

Se l'operazione A viene completata prima dell'inizio dell'operazione B, logicamente B dovrebbe avere effetto dopo A.

Qualsiasi lettura restituirà sempre il valore scritto più di recente per qualsiasi dato.

Fornisce un ordine totale.

N1	W(X) -> a	
N2	R(X) -> a	R(X) -> b
N3		W(X) -> b

Su schemi multi-master e peer-to-peer potrebbero esserci problemi.

Complicata da ottenere anche se uso il quorum perché all'atto pratico ci sono problemi.

### Sequential consistency

Non mi interessa stabilire un ordine cronologico assoluto delle operazioni ma è sufficiente che le varie repliche si accordino su uno specifico ordine.

A prescindere dal nodo dove mi collego ottengo sempre lo stesso aggiornamento.

L'unico vincolo che voglio è che se ci sono operazioni sullo stesso nodo quelle operazioni devono avvenire in maniera corretta.

Voglio che le due operazioni rimangano nell'ordine giusto.

Se ad esempio N1 scrive A ed N2 scrive B prima di A e C dopo A mi deve assicurare che a prescindere B stia prima di A e C dopo di A. Non ci interessa il tempo esatto.

Stabilità la sequenza l'ordine dev'essere consistente. Ovvero si deve rispettare.

Le operazioni sullo stesso nodo devono quindi mantenere lo stesso ordine.

In questo modo mi svincolo dal problema di avere un clock globale.

Per contro lo svantaggio principale è che le storie diventano non deterministiche nel senso che se provo ad eseguire la stessa storia in tempi diversi l'ordinamento sequenziale potrebbe essere diverso.

### Casual Consistency

Più vicino alle applicazioni.

Non ci interessa l'ordinamento tra operazioni concorrenti e non ci interessa nemmeno l'ordine in un certo nodo.

Ci interessa solo mantenere un ordinamento di causalità.

Ovvero se uno evento A causa un evento B l'ordine di causalità deve rimanere.

Princípio di causa effetto.

Come lo stabilisco?

Uno dei modelli più utilizzati è il **read-your-writes** ovvero che ciascun programma vede immediatamente i propri aggiornamenti nell'ordine in cui li ha effettuati.

Un client deve vedere l'ordine corretto degli aggiornamenti.

Un programma se fa un'operazione di scrittura e poi la legge deve vedere immediatamente quella scrittura.

Ci sono tanti modi per garantirlo, ad esempio posso farlo mandando dei bookmark in cui si dice "questo dato è il mio" e quando si legge si richiede quel dato al sistema con quel bookmark.

Finché non gli arriva un valore con quel bookmark non glielo restituisce.

Oppure forzo un programma a non cambiare mai replica durante il suo funzionamento.

Una sessione di 15 minuti su FaceBook non è detto che siamo collegati allo stesso nodo.

Per motivo di bilanciamento potremmo switchare sempre da un nodo all'altro.

## Eventual consistency

Idea: accetto che il sistema sia inconsistente ma chiedo che non duri oltre un certo lasso temporale.

Abbiamo una **finestra di inconsistenza temporale** che accettiamo ma entro quella finestra il sistema deve tornare consistente.

Con il quorum si ha questa sorta di finestra: poi quando tutti si aggiornano essa termina.

Una finestra di inconsistenza temporale dipende da diversi fattori come il carico sul sistema, i ritardi di comunicazione e il numero di repliche.

Nel funzionamento normale dovrebbe essere solo una frazione di secondo.

Se il sistema sta funzionando vicino alla sua capacità o se c'è un problema con la rete, ecc. la finestra potrebbe aumentare fino a diversi secondi o addirittura minuti.

Un sistema può essere read-your-writes in base all'applicazione e eventualmente in base al sistema.

In FaceBook significa che se scrivo un commento su FaceBook la mia applicazione me lo mostra immediatamente anche dopo che la riapro subito.

Mentre un'altra persona potrebbe non vederlo subito perché rispetto a lui il read-your-writes non vale e lo vede magari dopo tot secondi perché passa la finestra di inconsistenza.

## The CAP theorem

Ci dice che date le 3 proprietà fondamentali del sistema distribuito, ovvero:

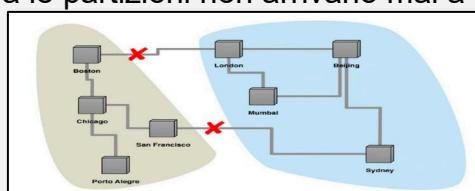
- **Consistenza**: tutti i nodi vedono lo stesso valore di un dato nello stesso momento. A prescindere dal modello di consistenza utilizzato (ignoriamo i modelli deboli).
- **Disponibilità**: se uno o più nodi falliscono, gli altri possono continuare a lavorare.
- **Tolleranza alla Partizione**: il sistema continua a funzionare anche a seguito di una partizione della rete, se ad esempio una parte diventa inattiva o non disponibile per qualche motivo il sistema rimanente continua a lavorare.

Solo 2 proprietà su 3 possono essere garantite contemporaneamente, a scelta del sistema in base al suo funzionamento e/o scopo.

Una **partizione della rete** è un particolare tipo di fallimento di comunicazione e si verifica quando la rete stessa viene divisa in sottoinsiemi (di nodi) e i nodi di una partizione/sottoinsieme non riescono a comunicare con i nodi di un'altra partizione.

Fino a quando tale problema esiste, tutte le modifiche effettuate in qualsiasi dato da parte di un sottoinsieme di nodi saranno invisibili ai nodi delle altre partizioni.

Questo perché i messaggi tra le partizioni non arrivano mai a destinazione.



In base alle 2 proprietà che vengono scelte per il sistema si hanno le seguenti combinazioni:

- **AP (availability, partition tolerance)**: il sistema è disponibile in quanto garantisce sempre una risposta al client in tempi ragionevoli e tollerante alla partizione in quanto in caso di malfunzionamenti di alcuni nodi continua ad operare. Tuttavia, è inconsistente (*manca la C*) se dovesse mancare la comunicazione. La comunicazione è l'unico modo per ottenere la consistenza, una possibile soluzione è quella di aspettare un intervallo di tempo prima di dare la risposta ai client, sperando che in questo intervallo la comunicazione riprenda. Tuttavia, essendo il

sistema disponibile ad un certo punto deve per forza rispondere (il WAIT è limitato) e se il problema non è stato risolto il client ottiene l'ultimo valore aggiornato dal nodo con cui sta comunicando (c'è quindi inconsistenza perché non si ha la garanzia che il valore dato al client sia globalmente il più aggiornato in quanto non c'è stata comunicazione tra i nodi).

- **CA (consistency, availability)**: il sistema è disponibile in quanto garantisce sempre una risposta al client in tempi ragionevoli e consistente ovvero i nodi comunicano tra di loro prima di restituire un'informazione richiesta dal client. Tuttavia, non è tollerante alla partizione (*manca la P*) e se dovesse esserci un problema nella rete che fa saltare la comunicazione il sistema smette di funzionare. Una possibile soluzione è che i client ricevano un messaggio di errore. Il sistema è in questo modo disponibile (nel senso che comunque comunica ai client che il sistema ha problemi) e consistente perché non viene restituita l'informazione fino a quando i nodi non tornano a comunicare per aggiornarsi e accertarsi di quale sia l'ultimo valore del dato/dei dati in questione. Il problema della P, tuttavia, rimane perché la perdita di una parte (un nodo) ha interrotto tutto.
- **CP (consistency, partition tolerance)**: il sistema è consistente in quanto comunica ai client valori sicuramente aggiornati in quanto i nodi comunicano tra di loro prima di restituire un'informazione richiesta dai client e tollerante alla partizione in quanto in caso di malfunzionamenti di alcuni nodi continua ad operare. Tuttavia, non è disponibile. **Indisponibilità = alta latenza**. Un sistema si dice indisponibile se impiega un tempo non ragionevole per restituire una risposta. Se la comunicazione tra i nodi salta essi prima di rispondere al client attendono il ripristino della rete. Una possibile soluzione potrebbe essere stabilire che si confermino sempre le scritture mentre le letture rimangono in attesa (o viceversa). Il sistema resta comunque non disponibile perché in caso di una o l'altra operazione i client devono attendere (per quanto tempo?). Un'altra possibile soluzione è aspettare un tot di tempo, se si risolve il problema bene altrimenti il nodo passa l'informazione che sapeva fino a quel momento. È una scommessa. In questo modo è vero che ottengo sia la disponibilità che la partition tolerance ma perdo la consistenza! Si tornerebbe al modello AP.

Si è dimostrato che in base alle scelte che facciamo riusciamo solo a mantenere 2 delle 3 proprietà contemporaneamente in un sistema.

Questo ci porta ad un ragionamento: nei sistemi distribuiti la P dev'essere mantenuta obbligatoriamente. Non possiamo rinunciare alla P.

Questo perché errori con disconnessione ci sono sempre e sono frequenti.

Si stima che in un mese per sei volte mediamente si hanno interi rack disconnessi dal sistema distribuito.

Impossibile rinunciare alla P sennò ogni volta si blocca tutto.

Quindi o si rinuncia alla disponibilità (A) o alla consistenza (C).

**È questo il vero teorema CAP.**

Scegliere la C rispetto alla A significa restituire un errore quando il timer impostato finisce. Scegliere la A rispetto alla C significa rispondere sempre a qualsiasi query anche se non si ha la certezza che la risposta data sia la più aggiornata.

Si noti che comunque il teorema CAP non significa che bisogna rinunciare completamente a una delle due proprietà (A o C), il trade-off è ancora possibile.

Il problema è sempre capire la nostra soluzione che tipo di latenza offre.

Ma anche quando qualcosa non funziona possiamo offrire un **trade-off**.

Dobbiamo pensare a uno slider dove noi mettiamo la barra nel mezzo.

Un sistema va progettato in modo da poter garantire il giusto compromesso tra consistenza e disponibilità.

Se pensiamo a Booking, in caso di problemi, è meglio far prenotare due volte la stessa camera a due clienti diversi e poi scusarsi con uno dei due (disponibilità) piuttosto che annullare entrambe e non incassare 1€ (consistenza).

**Nei sistemi distribuiti troviamo le proprietà BASE** (possiamo vederle come ACID lasche (flessibili), comunque non è una sostituzione). Esse sono:

- 1) **Basic Availability**: sono sistemi disponibili per il 99% dei casi. È una disponibilità basic perché ci possono essere parti indisponibili. Si ottengono con la replicazione. Se usiamo il quorum potremmo avere disponibilità diversa tra lettura e scrittura.
- 2) **Soft State**: sono sistemi in cui lo stato dei dati può cambiare nel tempo (per questo soft), anche senza effettuare operazioni di modifica (conseguenza dell'eventual consistency).
- 3) **Eventual Consistency**: sono sistemi in cui il database converge in uno stato consistente entro una certa finestra di inconsistenza (se ci sono ulteriori aggiornamenti la finestra si allarga sempre di più).

**Il nostro obiettivo non è avere un sistema perfetto ma un sistema che funzioni.**

In quanto sistemi distribuiti non possiamo aspettarci che 100 computer funzionino in modo perfetto come se fossero solo 1.

Ryanair non offre di scegliere il posto (se non a pagamento). Oltre a farci i soldi gli semplifica la vita. Non hanno posti specifici da assegnare ma una quantità da riempire. Quindi all'inizio è meglio per loro essere disponibili e non consistenti poiché l'obiettivo è quello di finire i posti. All'inizio su 300 posti si fa prenotare a raffica.

Quando rimangono pochi posti, tipo sotto il 20% (soglia), iniziano a stare attenti: il sistema rallenta favorendo la consistenza a scapito della disponibilità.

Se rimangono solo 10 posti non possono lasciar prenotare 20/30 posti.

Un sistema potrebbe non avere sempre gli stessi requisiti in tutte le sue funzionalità in termini di consistenza e disponibilità.

Il sistema può essere segmentato in componenti differenti sulla base delle sue funzionalità (garantito già se utilizziamo sistemi a microservizi).

Se segmento sulla base di questi componenti funzionali posso assegnare livelli diversi di disponibilità e consistenza così ognuno ha ciò di cui a bisogno e il sistema globalmente non è né disponibile né consistente. È una media tra tutti i microservizi.

Ogni operazione può avere il suo valore di quorum e quindi il suo trade-off A - C.

Dobbiamo progettare bene il nostro sistema.

Più il sistema è complesso più bisogna stare attenti.

Esempi: Amazon, Ryanair.

## The PACELC model

Indichiamo con latenza di un sistema il tempo di risposta ad una query.

Indichiamo con disponibilità la garanzia di risposta entro un certo intervallo di tempo.

Oggi si parla di sistemi ad alta disponibilità, ovvero a bassa latenza. Sistemi reattivi.

Quando tutto il sistema funziona bene (maggior parte del tempo) vogliamo bassa latenza.

Se ho problemi gravi posso accettare una latenza più alta.

Nell'informatica teorica, il teorema PACELC è un'estensione del teorema CAP. Asserisce che nel caso di tolleranza di partizione (P) in un sistema informatico distribuito si deve scegliere tra disponibilità (A) e consistenza (C) (come per il teorema CAP), ma altrimenti (E), anche quando il sistema è in esecuzione normalmente in assenza di partizioni, si deve scegliere tra latenza (L) e consistenza (C).

Alta disponibilità → repliche a nastro e geo-repliche (repliche in base all'area geografica in cui un utente si trova).

Qualsiasi forma di inconsistenza porta a una comunicazione con le repliche e quindi → aumenta la latenza. In generale, comunque, anche senza problemi più repliche esistono e più bisogna comunicare intrinsecamente.

Alta Disponibilità → Replicazione → Consistenza → Latenza.

Non appena un sistema distribuito replica i dati, si verifica un compromesso tra coerenza e latenza.

Ora che "sappiamo tutto", **come replicare bene?** Ci sono tre possibili metodi.

### Caso 1

Gli aggiornamenti dei dati vengono inviati a tutte le repliche contemporaneamente.

- **Senza preelaborazione:** ogni replica potrebbe scegliere un ordine diverso in cui applicare una serie di aggiornamenti simultanei, il che potrebbe causare problemi di consistenza.
- **Con livello di preelaborazione:** le repliche utilizzano un protocollo di accordo per decidere l'ordine delle operazioni; quindi, tutte le repliche concorderanno sull'ordine in cui elaborare gli aggiornamenti. Tuttavia, questo porta chiaramente ad avere una maggiore latenza.

### Caso 2

Il sistema aggiorna prima un nodo master concordato.

Architettura di tipo master-slaves. Il master come aggiorna gli slaves?

**In modo sincrono** (con quorum), il master aspetta che tutte le repliche abbiano ricevuto tutti gli aggiornamenti.

**In modo asincrono**, il master risponde subito senza aspettare che gli slaves si aggiornino. A questo punto dipende dalla lettura. Se le letture avvengono solo su master (quindi le altre repliche sono solo di backup) allora non ci sono problemi di consistenza, ma la latenza aumenta. Se invece le letture possono avvenire su tutti i nodi allora chiaramente la latenza è molto bassa ma posso avere problemi di consistenza (non ci sono garanzie che tutte le repliche si siano aggiornate ne ho garanzie sulla propagazione degli update).

**In modo ibrido** con quorum pari a un sottoinsieme delle repliche così alcune sono sincronizzate e altre sono asincrone. In base a dove si legge si ha una situazione diversa.

### Caso 3

Il sistema aggiorna prima un singolo nodo scelto arbitrariamente (basandosi ad esempio sulla distanza geografica o carico di lavoro bilanciato) che poi propaga gli update agli altri. Dipende molto dal modo in cui trattiamo gli aggiornamenti.

La latenza va sempre tenuta in considerazione, in base ad essa ci sono trade-off diversi.

## PACELC

Se sono nel caso della partizione di rete mi chiedo come il sistema gestisce A e C.

Altrimenti, qual è il trade-off che scelgo tra latenza e consistenza?

Il modello PACELC rappresenta una descrizione più completa per quanto riguarda il concetto dei potenziali trade-off per i sistemi distribuiti.

Si tenga presente che il compromesso latenza/consistenza (ELC) si applica solo ai sistemi che replicano i dati. In caso contrario, il sistema soffre di problemi di disponibilità su qualsiasi tipo di errore o nodo sovraccaricato.

**Sistemi PA/EL:** se si verifica una partizione, rinunciano alla consistenza in cambio della disponibilità mentre in condizioni di normale funzionamento rinunciano alla consistenza per una latenza inferiore. L'eliminazione di entrambe le C rende il design più semplice. Tuttavia, questi sistemi hanno impostazioni regolabili dall'utente per alterare il compromesso ELC, ad esempio, aumentando R + W, ottengono maggiore consistenza a scapito della latenza.

Esempi: Cassandra, Riak.

**Sistemi PC/EC:** sacrificano latenza e disponibilità per mantenere la consistenza.

**Sistemi PA/EC:** in caso di partizione rinunciano alla consistenza in favore della disponibilità. Esempio: MongoDB.

**Sistemi PC/EL:** in caso di partizione favoriscono la consistenza mentre in casi normali favoriscono la latenza/disponibilità. Esempio: Yahoo!

## Capitolo 10 – Key-Value Stores

I key-value store sono dei database orientati agli aggregati realizzati per gestire coppie chiave-valore. Ogni coppia chiave-valore è considerata come un aggregato.

Il database chiave-valore si basa su una tabella con solo due colonne: una contiene la chiave (identificatore univoco), l'altra contiene un valore. Un valore può assumere forme diverse: sono quindi possibili valori molto semplici come stringhe o numeri interi, ma anche oggetti complessi possono comparire come valori nel database. Ad esempio, anche un documento può assumere la funzione o il ruolo del valore, in quel caso si parla dunque di un database orientato ai documenti. Anche i riferimenti ai file possono essere inseriti nel database. Non abbiamo modo di accedere ad un valore se non grazie alla sua chiave, questo garantisce performance elevate e un funzionamento semplice.

Viene utilizzato il termine store per marcare il fatto che non sono veri e propri database ma strutture dati di dimensioni minori.

Le operazioni di base che un client può eseguire sono:

- **Get:** ottenere il valore di una specifica chiave.
- **Put:** inserire uno specifico valore per una specifica chiave.
- **Delete:** eliminare una coppia chiave-valore.

Esempio di store chiave-valore: Riak.

## RIAK

Key-value store distribuito. Fornisce diverse API per tutte le esigenze.

Lo spazio delle chiavi può essere frammentato in **buckets** (secchi, dentro metto le chiavi).

Ogni bucket è uno spazio delle chiavi che può essere configurato autonomamente.

Abbiamo il vantaggio di poter avere configurazioni personalizzabili per ciascun bucket.

Se ho due buckets con stessa chiave ma valori diversi essi non collidono.

Di solito, un bucket contiene tutti aggregati dello stesso tipo.

Posso utilizzare un bucket come cache della mia applicazione. Posso immaginare dei dati che vengono immagazzinati durante una sessione in questo bucket e poi vengano eliminati a sessione conclusa. Esempio, visivo, di bucket → Possiamo avere gruppi di bucket dello stesso tipo.

Possiamo usare lo stesso nome su più bucket diversi purché siano di diversa tipologia.

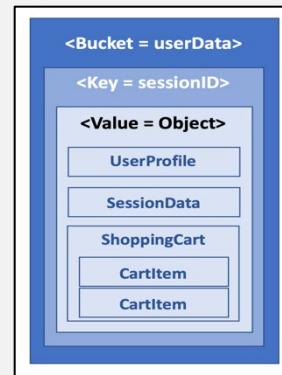
Riak utilizza un'architettura peer-to-peer (o multi-master, stessa cosa) combinando replicazione e sharding.

Essendo peer-to-peer tutti i nodi accettano letture e scritture.

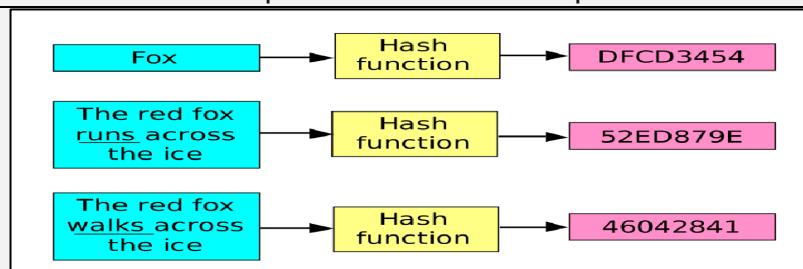
Quindi si predilige più la disponibilità che la consistenza.

La capacità di storage di Riak deve scalare in funzione del numero dei nodi nel cluster.

Utilizza lo sharding basato sul consistent hashing della chiave favorendo, come detto in precedenza, il bilanciamento del carico dei dati sui vari nodi.



Per **Hash Functions** si intendono tutte quelle funzioni che prendono (mappano) dei dati in input di dimensione arbitraria e restituiscono un valore in output con una dimensione fissa. I valori in output che restituiscono queste hash functions prendono il nome di **hash values**.



Per definirsi tale, un hash function dev'essere deterministica (ovvero se calcolo due valori l'hash sulla stessa chiave deve uscire lo stesso valore).

Inoltre, dovrebbe mappare gli input previsti nel modo più uniforme possibile nel suo intervallo di output, ovvero ogni valore hash nell'intervallo di output dovrebbe essere generato con all'incirca la stessa probabilità.

Inoltre, deve evitare che input simili diano luogo ad output simili e questo può ottenerlo grazie all'utilizzo di scrum, ovvero mescolare i bit per garantire randomicità dell'output.

La dimensione del range in output dev'essere parametrica alla dimensione dell'input.

Più aumenta l'output, più chiavi ha a disposizione.

Esempio: ho 128 bit → ho  $2^{128}$  bit possibili valori di hash.

Evitare che due chiavi collidano, ovvero che diano luogo allo stesso hash.

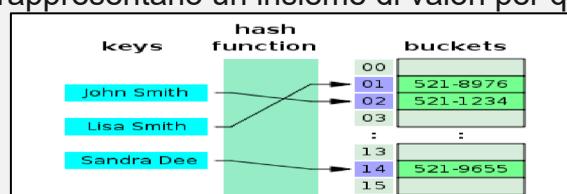
Le hash functions possono essere utilizzate nelle **hash tables**: ho un dizionario, il dizionario ha un termine (chiave) e la sua spiegazione (valore).

Non memorizzano le chiavi perché possono essere molto diverse fra loro ma vanno a memorizzare l'hash delle chiavi che garantiscono le proprietà sopra citate: chiavi a dimensione fissata e uniformemente distribuite nel mio dominio.

Prendo la chiave la passo ad una funzione e il risultato diventa l'indice del dizionario.

Il valore resta il valore e ovviamente quello che succede è che sono soggetto a collisioni ovvero è possibile che valori uguali diano luogo alla stessa chiave.

A quel punto vado a memorizzare o un unico valore basandomi su una certa regola o un insieme di collisioni che rappresentano un insieme di valori per quella chiave.



In una tabella hash distribuita le coppie chiave-valore sono memorizzate in modo tale che ogni nodo partecipante può recuperare il valore della chiave.

Abbiamo il problema di partizionare le varie chiavi della tabella nei nodi del cluster.

Lo *sharding by hashed key* può essere effettuato facendo il modulo  $N$  (numero dei nodi) alla fine dell'hash: deterministico, bilanciato e tutto ok.

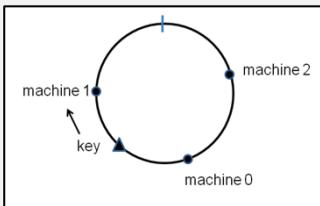
$$p_{i,j} = \text{hash}(\text{bucket}_i, \text{key}_i) \bmod N$$

Il problema è che se cambia il numero di nodi alcune chiavi devono essere riassegnate.

Quindi molte coppie chiave-valore si dovranno spostare. Questo rischia di far crollare le performance e chiaramente non va bene. Senza contare che generiamo noi stessi un grande traffico dentro al nostro cluster.

**La tecnica del consistent hashing [versione generale]** risolve questo problema: devo immaginare il mio spazio delle chiavi come un anello. Piazzo poi su questo anello i nodi del mio cluster, lo posso fare in maniera random. Quando arriva una coppia chiave-valore, calcolo l'hash del bucket più la chiave:  $h(b + k)$ . Corrisponde a un punto sull'anello. Per sapere a quale nodo assegnare la coppia cammino sull'anello finché non trovo un nodo. Il primo nodo sarà il possessore di questa coppia chiave-valore.

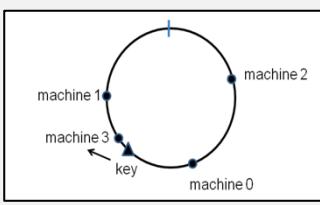
Ogni nodo quindi si prende un pezzo dell'anello.



Supponendo che una macchina\_3 si aggiunga in corso, allora la macchina\_1 non gestirà più tutto quel pezzo ma se lo dividerà con la macchina\_3. Quindi un blocco di coppie chiave-valore passa dalla macchina\_1 alla macchina\_3.

In questo modo lo controllo bene.

So che un pezzo del mio ring viene assegnato a una nuova macchina.



Questa cosa risolve il problema ma non è ancora sufficiente.

**Assegnare in modo randomico ha il problema di non far campionare bene sull'anello.**

Il secondo problema è che **non sto tenendo conto della capacità effettiva dei nodi**, in questo modo li assumo tutti uguali ma in realtà potrei avere nodi più potenti e nodi più deboli. Quindi non sto tenendo conto neanche dell'eterogeneità.

**Soluzione ottima:** non mappo un nodo del cluster su una singola parte del ring ma a più parti dell'anello. Quindi ad ogni nodo reale del cluster vengono associati dei **nodi virtuali** i quali vengono mappati nel ring.

In questo modo, ogni nodo è come se ha tanti punti nel ring. I vantaggi sono che se un nodo diventa non disponibile (problemi, manutenzione, ecc.) allora il carico di tale nodo non viene passato tutto a un altro nodo a tante macchine diverse, così nessuna soffre troppo. Con lo stesso ragionamento, se il nodo torna disponibile o se ne viene aggiunto un altro esso prenderà un po' il carico da tutte le macchine e non solo da una.

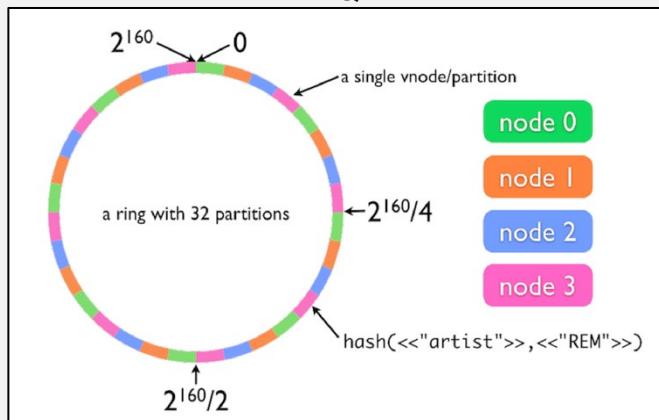
Il numero di nodi virtuali di cui un nodo è responsabile può essere deciso in base alla sua capacità, tenendo conto dell'eterogeneità nell'infrastruttura fisica.

Chiaramente, un nodo con scarsa capacità avrà pochi nodi virtuali sul ring.

## Consistent hashing in Riak

Riak usa consistent hashing SHA-based per organizzare la memorizzazione dei suoi dati e la replicazione di essi. Il ring è un key-space di 160 bit. I nodi virtuali, chiamati **vnodes**, si occupano dello sharding dell'anello di Riak. **Ad ogni elemento viene associata una chiave, essa passa per la funzione di hash, trova un punto nel ring e infine viene associata al primo vnode che la reclama.** Riak usa frammenti della stessa dimensione, assegnati secondo lo scheduler **Round Robin**.

**Ad ogni nodo vengono assegnati Q/S vnodes**, dove con S indichiamo il numero dei nodi nel sistema e con Q indichiamo il numero totale di partizioni.



Scheduler: Round Robin.

Se il nodo verde sparisce, i frammenti (shard) degli altri si ingrandiscono e i suoi vicini, arancione e viola, si prendono un po' ciascuno il verde.

Più furbo rispetto all'hash normale.  
Bilanciamento ottimo, funziona bene,  
performance consistenti in caso di  
ribilanciamento del cluster (perdita o  
aggiunta di nodi).

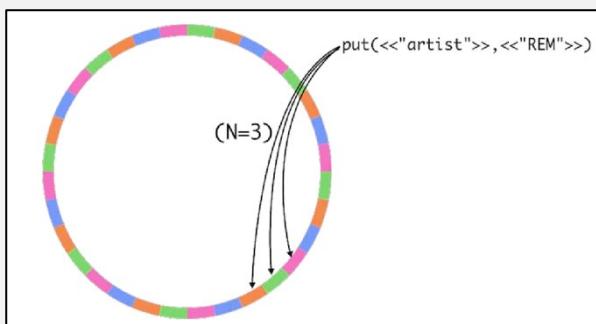
## Lo stato del ring è condiviso nell'intero cluster tramite il Gossip Protocol.

Ogni nodo parla con un certo numero di altri nodi estratti a caso e con essi comunica i cambiamenti di cui lui è a conoscenza. Quando un nodo subisce cambiamenti lo annuncia. Protocollo iterativo, ogni nodo aggiunge agli altri nodi le informazioni nuove che ha scoperto, ma non a tutti per evitare overhead, solo a un certo numero.

Dopo un po' converge e tutti i nodi sono venuti a conoscenza dei cambiamenti.

Occorre evitare il rapporto n-n e occorre evitare di lasciare indietro nodi che magari sono assenti durante gli aggiornamenti. Aggiornati in seguito ma aggiornati.

Poiché ogni nodo è a conoscenza dello stato dell'anello, un client può inviare una query a qualsiasi nodo, esso poi la inoltrerà al nodo/i giusto/i.



Per quanto riguarda la **replicazione**, essa funziona che se N è il mio fattore di replicazione (default N = 3) quello che succede è che quando si inserisce un dato in un nodo, essendo a replica 3, il dato viene **posto anche** in altri 2 **nodi vicini**.

## Come vengono eseguite le operazioni in Riak?

Le operazioni di scrittura utilizzano un quorum.

Se utilizzo W come quorum di scrittura significa che posso tollerare una perdita di nodi fino a un massimo di N-W.

Se ad esempio  $N = 3$  e  $W = 2$ , posso perdere al massimo 1 nodo.

Se ne perdo 2 non sono più in grado di effettuare operazioni in scrittura.

Se utilizzo  $W > N / 2$  vado incontro a un conflitto di tipo write-write.

In tal caso, Riak mette a disposizione due possibili soluzioni per risolvere il problema del conflitto (entrambe le opzioni possono essere configurate quando viene creato un bucket):

- La scrittura più recente vince: basata sulle **versioni vettoriali**, è la **strategia predefinita**. Le versioni vettoriali possono imporre la coerenza causale.
  - Quando i conflitti non possono essere risolti, vengono create più copie dello stesso oggetto (fratelli).
- Siblings: memorizza tutti i conflitti in scrittura. Al client verrà poi chiesto di risolvere il conflitto utilizzando logiche specifiche di business, scegliendo una scrittura.

*Le operazioni di lettura utilizzano un quorum.*

Come per la scrittura abbiamo un read quorum.

Se il client riceve R risposte diverse va a guardare la più recente attraverso il **vettore delle versioni**. Inoltre esegue operazioni di **read repair** ovvero va ad aggiornare i valori nelle repliche che non dispongono del valore più aggiornato.

Se impostiamo **R + W > N** dovremmo avere la garanzia che ogni lettura riuscita veda l'ultimo valore scritto con successo.

In generale, Riak permette di modificare R e W in base alle situazioni.

Il valore di N può variare in base al bucket.

In base alle necessità può anche decidere se utilizzare un quorum oppure no.

*Riak usa sloppy quorums e hinted handoff* in modo da avere una disponibilità in scrittura anche in assenza di repliche appartenenti al quorum e affida l'incarico ad altre repliche.

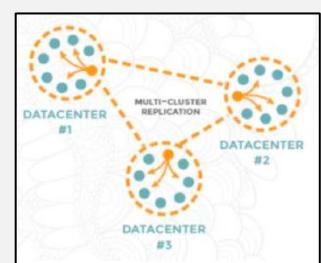
Poi il risultato del voto viene comunicato alle repliche che sono le vere designate.

Per quanto riguarda *cluster geo-distribuiti*, in Riak c'è un cluster che funge da primario e altri che fungono da secondari, su quest'ultimi vengono replicati i dati.

Questo per ragioni di disaster recovery (ripristino di emergenza).

Cluster diversi si trovano spesso in regioni o paesi diversi.

Se il cluster primario diventa inattivo, un cluster secondario può assumere automaticamente il ruolo di primario.



## Capitolo 11 – Document Databases

I document database sono database orientati agli aggregati realizzati per gestire documenti. Ogni documento è considerato come un singolo aggregato.

Un documento può essere di vario tipo, come XML, JSON, ecc.

Un document database è fondamentalmente un key-value store tale che il valore sia esaminabile.

I documenti sono auto-descrittivi, ciascuno di loro definisce il proprio schema.

Documenti differenti posso avere differenti schemi.

Un documento può avere ricorsivamente dentro di lui altri documenti, per questo motivo può essere rappresentato come una struttura dati gerarchica ad albero.

Le operazioni base che un client può eseguire sono:

- **Create**: creare un nuovo documento.
- **Retrieve**: recuperare un insieme di documenti basandosi sul loro contenuto.
- **Update**: aggiornare un documento.
- **Delete**: eliminare un documento.
- **Organize**: organizzare documenti secondo una collezione, metadati, directory, ecc.

```
{
  firstName: "Bob",
  address: "5 Oak St.",
  hobby: "sailing"
}
```

## MongoDB

MongoDB è uno dei più famosi database orientati ai documenti.

Un **record** in MongoDB è un **documento BSON** (simile al JSON).

BSON è una rappresentazione binaria di documenti JSON.



Il valore di un campo può essere qualsiasi tipo di dato BSON, compresi altri documenti, array e array di documenti. Il nome del campo `_id` è riservato come chiave primaria ed il suo valore deve essere unico nella collezione, immutabile e non di tipo array.

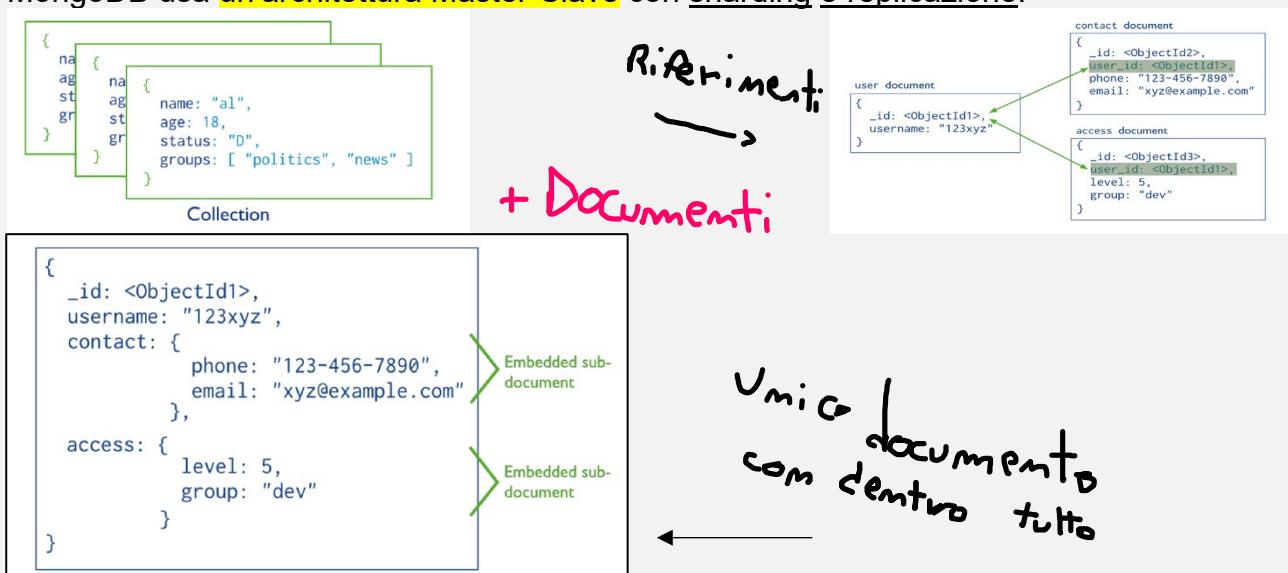
Il massimo peso di un documento è limitata a pochi MB per evitare di utilizzare un'eccessiva quantità di RAM e per evitare un eccessivo consumo di banda durante un suo eventuale trasferimento.

I documenti possono essere organizzati in collezioni all'interno dello stesso database.

Le **relazioni tra documenti** possono essere ottenute attraverso **riferimenti** e **documenti embeddati** (unico documento con dentro tutto).

Le **operazioni sono atomiche a livello di documento**.

MongoDB usa un'architettura Master-Slave con sharding e replicazione.



Progettare gli aggregati giusti per la nostra applicazione. Conviene creare un unico documento o farne molti? Questo dipende da come la nostra applicazione accede ai dati. Ad esempio, utilizzando i riferimenti per modellare le relazioni si ha una riduzione della ridondanza mentre utilizzando i documenti annidati abbiamo una migliore performance delle query (soprattutto nelle relazioni 1:1, 1:N).

Comprendere e delimitare i confini di un aggregato è fondamentale per raggiungere il giusto grado di performance.

**Relazione 1:1, differenza tra tipo riferimento e tipo embedded**



## Relazione 1:N, differenza tra tipo riferimento e tipo embedded

<pre>{   "_id": "joe",   "name": "Joe Bookreader",   "addresses": [012658974, 012658985] }  {   "_id": 012658974,   "street": "123 Fake Street",   "city": "Faketon",   "state": "MA" }  {   "_id": 012658985,   "street": "1 Some Other Street",   "city": "Boston",   "state": "MA" }</pre>	<pre>{   "_id": "joe",   "name": "Joe Bookreader",   "addresses": [     {       "street": "123 Fake Street",       "city": "Faketon",       "state": "MA"     },     {       "street": "1 Some Other Street",       "city": "Boston",       "state": "MA"     }   ] }</pre>
---	---

La **Mongo shell** è un'interfaccia JavaScript iterativa con la quale è possibile interagire con MongoDB. Può essere utilizzata infatti per interrogare e aggiornare i dati così come eseguire operazioni di amministrazione.

```
use myNewDatabase
db.myCollection.insert( { x: 1 } );
```

Tutte le operazioni in MongoDB sono atomiche a livello di un singolo documento.

Le **operazioni di inserimento** hanno come destinazione una singola collezione.

È possibile inserire uno o più documenti.

```
db.users.insertOne(
  {
    name: "sue",
    age: 26,
    status: "pending"
  }
)
```

Le **operazioni di lettura** recuperano i documenti da una singola collezione.

```
db.users.find(
  { age: { $gt: 18 } },
  { name: 1, address: 1 }
).limit(5)
```

Le **operazioni di update** modificano documenti esistenti in una collezione.

È possibile aggiornare uno o più documenti in base al match con la query.

```
db.users.updateMany(
  { age: { $lt: 18 } },
  { $set: { status: "reject" } }
)
```

Le **operazioni di delete** rimuovono documenti da una collezione. È possibile eliminare uno o più documenti in base al match con la query.

```
db.users.deleteMany(
  { status: "reject" }
)
```

Le query supportano campi, intervalli, ricerche di espressioni regolari.

Le query possono restituire dei campi specifici dei documenti e includere anche delle funzioni JavaScript definite dall'utente.

I campi in un documento possono essere indicizzati con indici primari e secondari.

Map-Reduce può essere utilizzato per l'elaborazione batch di dati e operazioni di aggregazione. Senza aver bisogno di un cluster Hadoop.

Il paradigma Map-Reduce è ristretto al campo JSON.

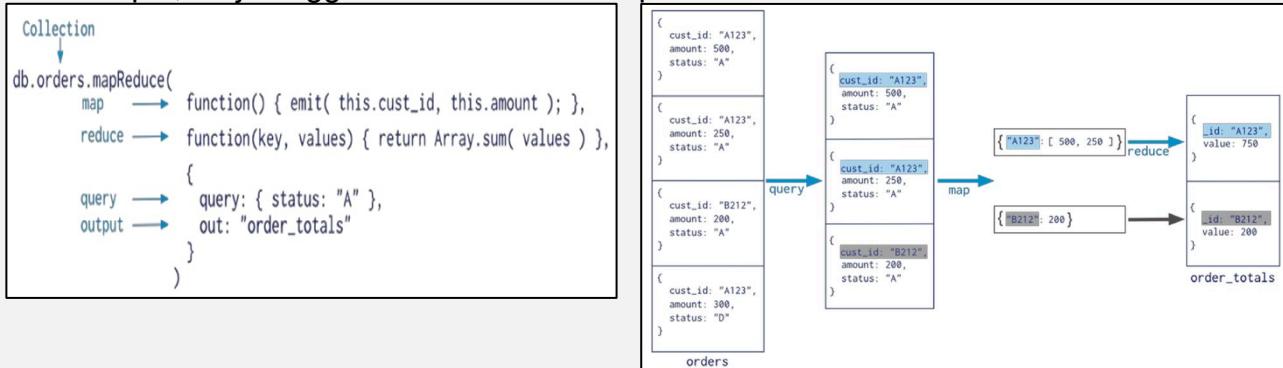
Le operazioni Map-Reduce hanno due fasi:

La fase di Map elabora ogni documento di input ed emette uno o più oggetti per essi.

La fase di Reduce combina l'output dell'operazione di Map.

Map-Reduce può specificare una condizione di una query per selezionare dei documenti di input nonché ordinare e limitare i risultati.

È possibile costruire delle visite materializzate di una collezione per poi fare delle analisi. Ad esempio, un job aggiorna i dati di notte e poi ci lavora un ufficio di analisi dati.



### Replicazione + Frammentazione

MongoDB usa un'architettura Master-Slave e combina sharding e replicazione.

Per quanto riguarda le repliche, un **replica set** è un gruppo di processi diversi (nodi del cluster) che mantengono due o più copie dello stesso documento.

Ogni replica set ha una replica primaria (Master) ed una o più repliche secondarie (Slaves). Per evitare conflitti e garantire la consistenza, **tutte le scritture e le letture vengono eseguite sulla replica primaria per impostazione predefinita**.

Di default abbiamo il massimo della consistenza perché di fatto abbiamo un unico nodo che risponde alle richieste.

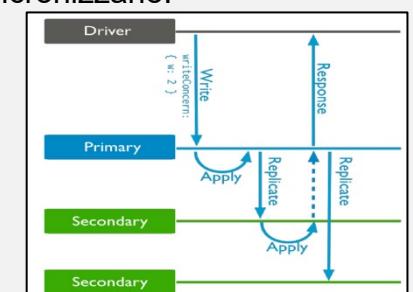
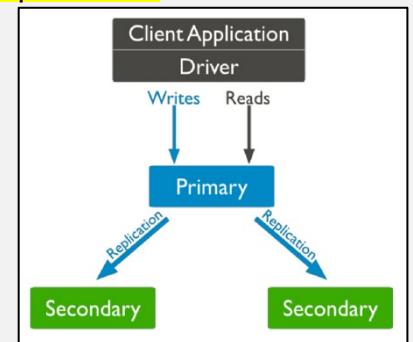
Questo porta ad avere una discreta latenza.

Per ridurla, le operazioni di lettura si possono prediligere sugli slave.

Il master scrive in un log nel quale gli slave vanno a leggere e si sincronizzano.

*La replica primaria opera con le repliche secondarie utilizzando un protocollo basato su quorum.*

Il **quorum di scrittura** specifica il numero di repliche (inclusa la primaria) che devono riconoscere qualsiasi operazione di scrittura. Una volta raggiunto il numero di repliche da attendere (replica sincrona), le altre operazioni di replica continueranno in modo asincrono. Questo garantisce una migliore durabilità delle applicazioni.



Per impostazione predefinita, i client leggono dalla replica primaria.

Tuttavia, per ridurre la latenza, i client possono specificare una preferenza di lettura per inviare le operazioni di lettura alle repliche secondarie. In questo caso però è da considerare che parte della replicazione è asincrona; quindi, un'operazione di lettura potrebbe restituire dati non aggiornati se legge in una replica asincrona non aggiornata.

MongoDB favorisce la consistenza, anche quando il master non funziona (**PC/EC**).

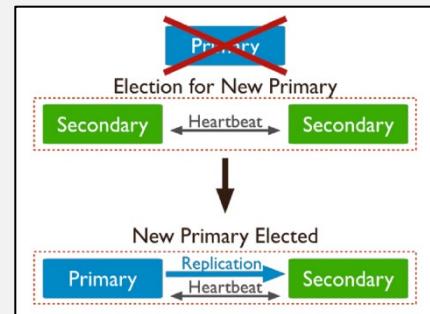
È possibile però settare altre preferenze, ad esempio posso favorire la disponibilità quando il master non funziona (**PA/EC**).

È possibile anche favorire sempre la disponibilità (la bassa latenza) facendo sì che il client legga sempre dalla prima replica che gli risponde (**PA/EL**).

## Fallimento Automatico

Se il Master non è raggiungibile per più di 10 secondi allora gli altri membri attraverso una votazione a maggioranza possono eleggere un nuovo Master.

Generalmente questa procedura dura per 1 minuto al massimo.



MongoDB può spartire i dati e distribuirli in più database, chiamati **shards**.

Uno shard viene distribuito come set di repliche per garantire la replica dei dati.

Supponendo di avere un dataset diviso in tre parti, ogni parte è una replica set con il suo nodo primario e slaves. Un nodo sarà primary soltanto per il suo shard e non di tutto il db. Ogni documento deve avere la propria **shard key**.

MongoDB partiziona i frammenti in **chunks** (blocchi).

*Ogni blocco ha un limite inferiore inclusivo e un limite superiore esclusivo.*

La shard key può essere utilizzata direttamente o indirettamente.

MongoDB migra i chunk nei replica set tramite lo **sharded cluster balancer**.

Il sistema di bilanciamento tenta di raggiungere un equilibrio uniforme dei blocchi in tutti gli shard nel cluster. Ciascun frammento può essere configurato come set di repliche.

Il **Ranged Sharding** comporta la suddivisione dei dati in intervalli basati sulla shard key.

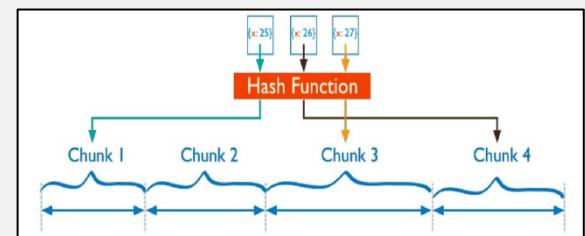
Ogni chunk è assegnato a un range sulla base della sua shard key.

Un range di shard keys che hanno valori molto vicini (limiti inferiori e superiori) sono più probabili a risiedere nello stesso chunk.

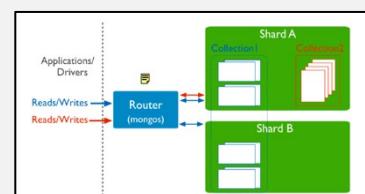
Come sempre, questa scelta di prendere direttamente il range delle chiavi e dividere in maniera diretta equi spaziata (mod N in questo caso con N = #shard keys) in modo che ogni chunk finisce nel suo range non è ottima.

Questo perché potrebbe comportare di avere tutto il traffico su un solo chunk.

Come sempre, utilizzando la versione hashing, in questo caso chiamata Hashed Sharding, il problema viene risolto. Calcola l'hash dello shard key, ad ogni chunk viene assegnato un intervallo basato sull'hash. Garantisce una distribuzione uniforme grazie all'hash delle chiavi e non si accumula tutto su un unico chunk.

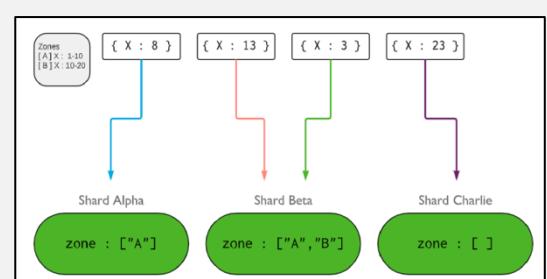


I client si connettono a dei router per interagire con qualsiasi collezione nel cluster frammentato.



Per cluster geo distribuiti intendiamo cluster che prevedono più datacenter, di base divisi in zone geografiche diverse.

Ogni zona può essere associata con uno o più shards nel cluster.



Caso di utilizzo: Internet of Things.

## Capitolo 12 – Column-Family Stores

Un column-family store è un database orientato agli aggregati progettato per gestire tabelle hash multidimensionali **sparse**.

I dati sono conservati in righe, ciascuna con la propria **row key**.

Le column families sono gruppi di dati collegati che spesso vengono acceduti insieme, similmente alle tabelle nei database relazionali (ma con diverse differenze).

Ad esempio, per un cliente potremmo pensare di avere una column family per il suo profilo e una column family per i suoi ordini.

Ogni riga ha un insieme ordinato di **colonne** attaccate.

Ogni colonna consiste in **coppie nome-valore**, dove **il nome si comporta come la chiave** per quella colonna.

*Ogni colonna è solitamente dotata di un timestamp utile per dare una scadenza ai dati, risolvere conflitti, gestire dati obsoleti, ecc.*

Righe diverse possono avere colonne diverse, anche all'interno della stessa famiglia di colonne.

Le colonne possono essere aggiunte a qualsiasi riga in qualsiasi momento.

Una famiglia di colonne contenente solo colonne semplici è chiamata **standard column family**.

Il valore di una colonna può essere un map di colonne, in quel caso quella colonna viene chiamata **super column**. Analogamente, una column family contenente una super columns prende il nome di **super column family**.

Un **keyspace** è un contenitore di famiglie di colonne standard e super. Nell'esempio, una super colonna è **address** perché il suo valore è un altro insieme di valori (name e value).



```
//super column family
{
  //row
  name: "billing:martin-fowler",
  value: {
    address: {
      name: "address:default",
      value: {
        fullName: "Martin Fowler",
        street: "100 N. Main Street",
      }
    },
    billing: {
      name: "billing:default",
      value: {
        creditCard: "8888-8888-8888",
        expDate: "12/2016"
      }
    }
  }
  //row
  ...
}
```

### Apache Cassandra

Il focus di Cassandra è quello della scalabilità, più di qualsiasi altro software visto finora. Il modello di dati Cassandra può essere descritto come un archivio di righe **partizionato**, in cui i dati vengono archiviati in tabelle hash multidimensionali **sparse**.

Per partizionato si intende che ogni riga dispone di una chiave univoca che la rende unica e che permette l'accesso ai suoi dati, nonché la distribuzione delle righe lungo più nodi del cluster (sharding). Le righe sono assegnate all'interno del cluster secondo un qualche algoritmo di sharding (**consistent hashing**).

Per sparse intendiamo che per ogni riga è possibile avere una o più colonne, ma ogni riga non ha bisogno di avere tutte le stesse colonne di altre righe simili (cioè, non c'è bisogno di valori nulli). Se una riga non ha valori su una colonna, quest'ultima non esiste proprio. Un tipo diverso di memorizzazione dei dati rispetto ai RDBMS.

Poiché i dati vengono archiviati in ogni colonna, essi vengono archiviati come entità separate di una tabella hash.

I valori delle colonne vengono archiviati in base a un ordinamento coerente.

**CQL** è un linguaggio di interrogazione che supporta comandi simili a SQL (il join non è possibile) e consente di definire tabelle (column family) con uno schema.

CQL salva i dati in **tabelle** (per esempio, column family) con uno schema e queste tabelle saranno poi raggruppate in **keyspaces**.

• CREATE KEYSPACE statement:

```
CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name
WITH options
```

• Example:

```
CREATE KEYSPACE Excelsior WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3}
AND durable_writes = true;
```

• CREATE TABLE

```
CREATE TABLE users (
    id text PRIMARY KEY,
    name text,
    favs map<text, text>
);
INSERT INTO users (id, name, favs) VALUES ('jsmith',
'John Smith', { 'fruit' : 'Apple', 'band' : 'Beatles'
});
UPDATE users SET favs = { 'fruit' : 'Banana' }
WHERE id = 'jsmith';
```

Un map è un insieme (ordinato) di coppie chiave-valore, dove le chiavi sono univoche e la mappatura è ordinata in base alle chiavi.

Un set è una collezione (ordinata) di valori univoci.

Una list è una collezione (ordinata) di valori non univoci in cui gli elementi sono ordinati per posizione.

CQL supporta la definizione di tipi definiti dall'utente.

Per impostazione predefinita, CQL consente solo query SELECT che non implicano il "filtraggio" lato server, ovvero query in cui sappiamo che tutti i record letti verranno restituiti nell'insieme di risultati.

Il ragionamento è che quelle query "non filtranti" hanno prestazioni prevedibili nel senso che verranno eseguite in un tempo proporzionale alla quantità di dati restituiti dalla query (che può essere controllata tramite LIMIT).

L'opzione ALLOW FILTERING consente le query che richiedono il filtraggio e che pertanto potrebbero avere prestazioni imprevedibili.

Anche una query che seleziona pochi record può presentare prestazioni che dipendono dalla quantità totale di dati archiviati nel cluster.

```
SELECT name FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM
users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
    AND time > '2011-02-03'
    AND time <= '2012-01-01'

SELECT COUNT (*) AS user_count FROM users;
```

Esempi di SELECT

Andiamo ad analizzare i pattern di accesso ai dati, ovvero quali sono le query fondamentali dell'applicazione. Una query deve accedere ad una sola tabella, la quale deve chiaramente contenere informazioni sufficienti. La stessa informazione può apparire più volte a causa della ridondanza introdotta nei dati.

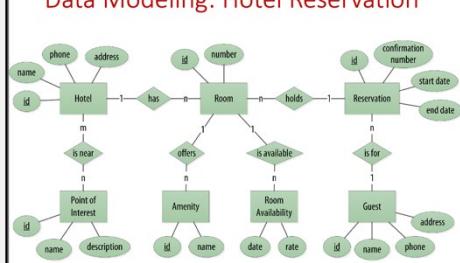
I dati sono modellati attorno query specifiche.

Le query sono progettate al meglio per accedere a una singola tabella.

Il join non è supportato in Cassandra, quindi tutti i campi obbligatori (colonne) devono essere raggruppati in un'unica tabella.

Poiché ogni query è supportata da una tabella, i dati vengono duplicati su più tabelle in un processo noto come denormalizzazione.

Data Modeling: Hotel Reservation



Approccio basato sulle query: il design dell'interfaccia utente può essere molto utile per iniziare a identificare le query. In base ai dati occorre aggiornare l'applicazione per gestire eventuali cambiamenti di utilizzo della stessa da parte dell'utente finale. Impostando un pattern di accesso ai dati, conosco l'ordine delle query.

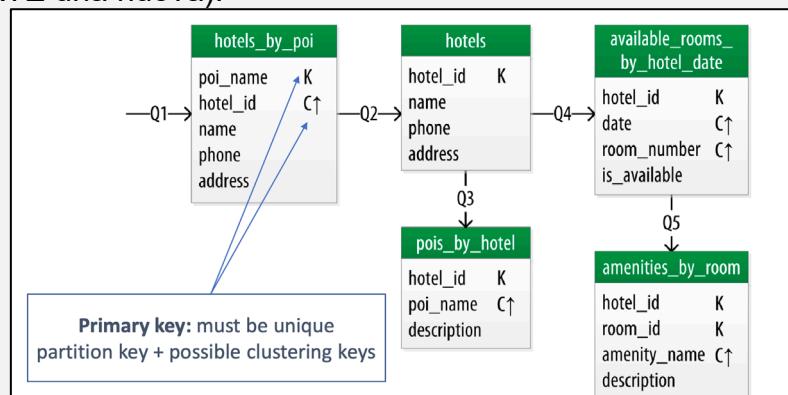
È possibile creare un modello logico contenente una tabella per ogni query, acquisendo entità e relazioni dal modello concettuale. K è la chiave che Cassandra utilizza per partizione le righe, significa che se due righe hanno la stessa chiave K esse staranno sullo stesso nodo, non vengono frammentate perché durante un'interrogazione si sa già che bisogna accedere a tutte le righe che hanno tale chiave K; quindi, possibilmente tutte le righe che si cercano devono stare su un nodo così non si coinvolgono altri nodi.

La chiave C viene utilizzata per ordinare le righe all'interno della stessa partizione. Viene fatto per comodità ed efficienza.

La chiave primaria è composta da K ed eventualmente anche da C.

Privilegia la lettura rispetto alla scrittura.

Si punta anche a mantenere il DB pulito (se ad esempio occorre fare un UPDATE di un'informazione, potrebbe non essere possibile ma si induce a fare un DELETE della stessa e a CREATE una nuova).



### Replication + Sharding

Cassandra utilizza **un'architettura peer-to-peer** (multi-master) combinando replicazione e sharding. Cassandra partiziona i dati nel cluster attraverso il **consistent hashing** e preservando un certo ordine attraverso una funzione di hash.

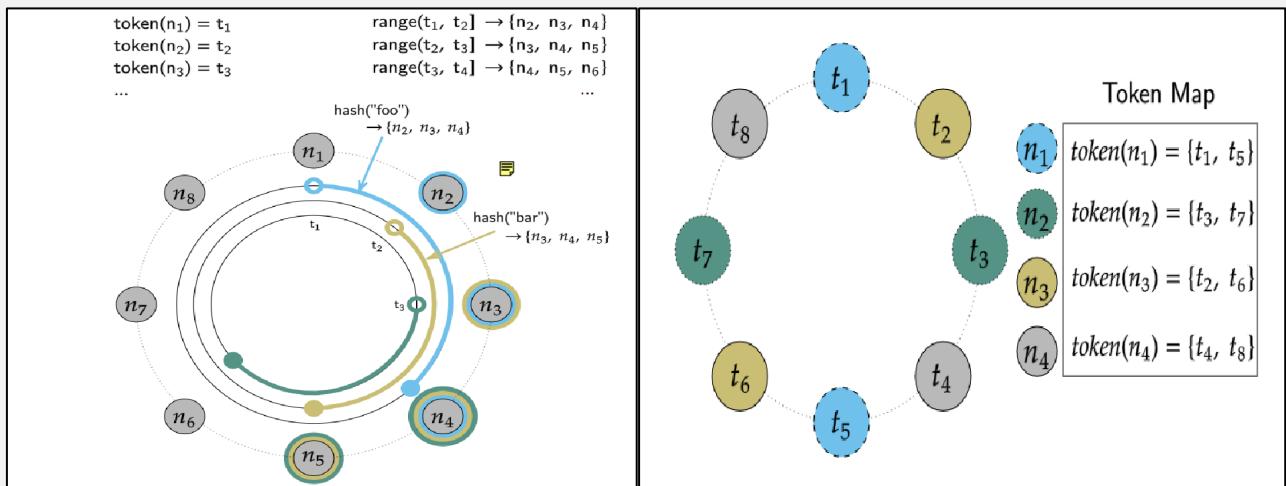
Il vantaggio principale del consistent hashing è che la rimozione o l'aggiunta di un nodo influisce solo sui suoi vicini immediati mentre gli altri nodi rimangono inalterati.

In questo modo Cassandra mantiene sempre la possibilità di fare query basate sul range.

## Recap sul Consistent Hashing

Il range in output di una funzione di hash viene trattato come uno spazio circolare (anello). Ad ogni nodo del cluster viene assegnato un valore di questo spazio che rappresenta quindi la sua posizione nell'anello.

Ogni elemento di dati identificato da una chiave viene assegnato a un nodo eseguendo l'hashing della chiave dell'elemento di dati per ottenere la sua posizione sull'anello, si percorre poi l'anello in senso orario e appena viene trovato il primo nodo si assegna ad esso. Pertanto, ogni nodo è responsabile di una zona che va da lui fino al suo nodo predecessore. Così come Riak, anche Cassandra per ottenere un buon bilanciamento sull'anello utilizza i nodi virtuali; ovvero, ad ogni nodo vengono assegnate più posizioni sull'anello. Più nodi virtuali ci sono più i range sono piccoli, in questo modo è difficile che un nodo si becchi solo zone fredde.



In questo esempio, tutto quello che cade nel range blu, per via del fattore di replicazione, può essere gestito dai nodi  $n_1, n_2, n_3, n_4$ .

Stesso discorso, il range giallo può essere gestito dai nodi  $n_2, n_3, n_4, n_5$ .

**I token sono i nodi virtuali.** Grazie ai quali appunto un nodo gestisce più zone.

Cassandra "sente" sul cluster e aumenta o diminuisce di piccoli range automaticamente in base al traffico che effettivamente c'è sul cluster. Se "sente" che una parte di range è fredda allora lo allarga (così prende più carico), se "sente" che è caldo allora lo restringe (così prende meno carico).

Così come Riak, Cassandra fa utilizzo del [Gossip Protocol](#) per condividere lo stato del cluster tra i nodi.

Per quanto riguarda la replicazione, ogni dato può essere replicato su  $N$  nodi, dove  $N$  è il fattore di replicazione che può essere configurato per un keyspace (cioè, database).

Ogni chiave viene assegnata a un nodo coordinatore in base al consistent hashing e il coordinatore è responsabile della replica dei dati che rientrano nel suo intervallo.

Per la replicazione Cassandra utilizza due possibili strategie: [semplice](#) e [avanzata](#).

- **Semplice**: i dati vengono replicati selezionando gli  $N-1$  nodi successori al coordinatore.
- **Avanzata**: si tiene conto della topologia del cluster. Se il numero di rack è maggiore o uguale al fattore di replica per un data center, è garantito che ciascuna replica verrà scelta da un rack diverso. In caso contrario, ogni rack conterrà almeno una replica, ma alcuni rack potrebbero contenere più di una.

Cassandra gestisce il **trade-off tra consistenza e disponibilità** attraverso i **livelli di consistenza**. Si può impostare per ogni operazione ed essenzialmente specifica quante repliche devono rispondere per considerare un'operazione un successo (confermarla) (read/write quorum):

- 1 / 2 / 3: devono rispondere una, due o tre repliche.
- Quorum: deve rispondere la maggioranza ( $(N / 2) + 1$ ) delle repliche.
- All: devono rispondere tutte le repliche.

Per cluster geo-distribuiti i principali livelli di consistenza sono:

- LOCAL\_QUORUM: la maggior parte delle repliche nel data center locale (qualunque sia il data center in cui si trova il coordinatore) deve rispondere. Latenza bassa, disponibilità normale, consistenza da vedere.
- EACH\_QUORUM: la maggior parte delle repliche in ogni datacenter deve rispondere. Latenza alta, scarsa disponibilità, consistenza massima.
- LOCAL\_ONE: solo una singola replica deve rispondere. Ciò garantisce inoltre che le richieste di lettura non vengano inviate alle repliche in un data center remoto. Latenza normale, disponibilità normale, consistenza alta.

In Cassandra, le richieste di lettura e scrittura vengono generalmente **instradate** dal nodo coordinatore verso gli altri nodi del cluster. Determina le repliche in base alla chiave.

Per le operazioni di scrittura di solito le richieste vengono inoltrate a tutti i nodi senza badare al livello di consistenza impostato, quest'ultimo semplicemente poi controlla quante risposte il coordinatore deve aspettare prima di rispondere al client.

Per le operazioni di lettura invece il coordinatore in genere invia solo le richieste a un numero sufficiente di repliche tali da soddisfare il livello di coerenza.

Per le operazioni di lettura il coordinatore chiede il dato a una sola replica. Le altre repliche che gli mancano per raggiungere il quorum gli inviano solo l'hash-bit.

Serve solo l'hash-bit per vedere se il dato è uguale a tutti, non c'è bisogno che ogni replica mandi tutta l'intera informazione. Questo riduce le comunicazioni. Se tutte le repliche hanno lo stesso hash-bit allora è tutto ok, altrimenti c'è un problema di consistenza.

In caso di problemi di consistenza (se l'hash-bit è diverso) allora il coordinatore chiede tutto il dato a tutte le repliche, capisce il più aggiornato sulla base del timestamp che c'è su di essi e poi di conseguenza aggiorna il dato più recente su tutte le altre repliche.

Per le operazioni di scrittura, **l'ultima scrittura vince**, basata sul timestamp.

Tutti gli aggiornamenti che entrano nel sistema lo fanno con un timestamp fornito da un orologio del client o, in assenza di un timestamp fornito dal client, dall'orologio del nodo coordinatore. Quando arriva una richiesta di scrittura i dati vengono prima registrati in un **commit log** e poi in una struttura in memoria chiamata **memtable**.

Le scritture vengono ordinate e archiviate periodicamente in strutture immutabili dette **SSTable**. Se i dati cambiano creo una nuova **SSTable**, non si sovrascrive.

Una scrittura è atomica a livello di riga. Quando lavoro su righe diverse non posso considerare la struttura atomica. **Se un nodo cade il commit log viene usato per aggiornare il nodo caduto**. Chiaramente più nodi ci sono più aumenta la capacità di scrittura.

Cassandra attiva l'**hinted handoff** (passaggio di consegne) nel caso in cui una o più repliche non dovessero rispondere durante un'operazione di scrittura.

Non fa uso invece dello **sloppy quorum**: la replica che ha preso le scritture per lei non può partecipare al quorum. Fa le sue veci, l'hinted handoff resta valido ma la replica vice non può votare, si ha così una consistenza maggiore.

## Capitolo 13 – Graph Databases

Rientra nei database NOSQL. Le entità del nostro dominio le rappresentiamo attraverso i nodi di un grafo. Quelle che sono le relazioni invece le rappresentiamo in maniera esplicita con degli archi.

Possiamo avere delle proprietà che andiamo ad associare ai nodi e agli archi per arricchirli con delle informazioni. Ad esempio, i nodi possono avere un nome.

Gli archi sono tipicamente orientati, da una sorgente a una destinazione.

Quello che faccio con un graph database è visitare il grafo.

Le relazioni sono esplicite.

Fare queste relazioni rispetto ai database relazionali è molto più veloce e semplice.

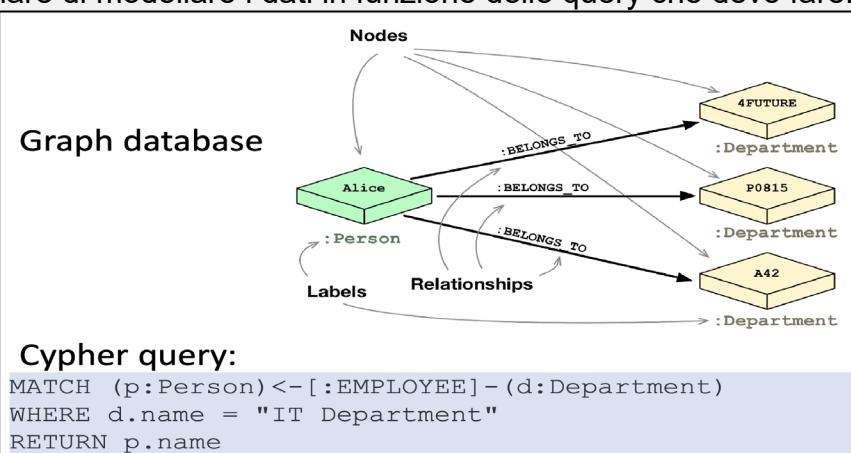
È difficile fare degli aggregati in questa famiglia di database.

I database chiave-valore che siano davvero chiave-valore, documenti o colonne semplificano la gestione delle relazioni utilizzando gli aggregati.

**I Graph Databases non offrono supporto agli aggregati.**

Come per Cassandra, bisogna andare a vedere quali sono le query più importanti.

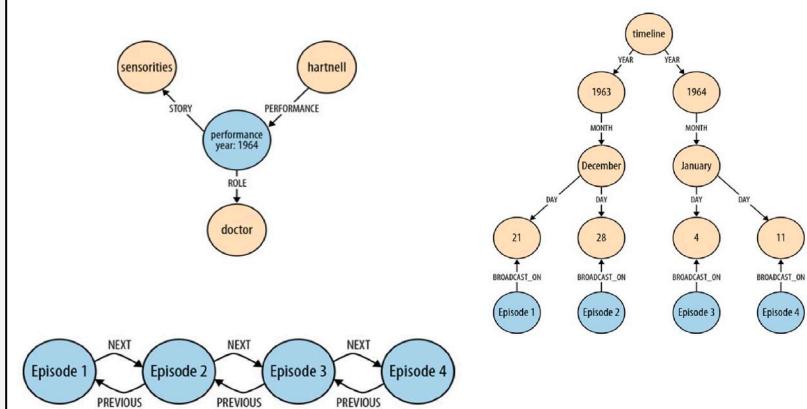
Posso immaginare di modellare i dati in funzione delle query che devo fare.



Per sfruttare gli attraversamenti veloci nei database a grafo, i nodi possono anche rappresentare fatti, azioni, episodi.

Posso anche creare archi che permettono di andare da un episodio all'altro se so che la mia applicazione è sequenziale. Andando avanti o indietro.

Many possible choices...



Il punto di forza di questo database è la navigazione tra queste relazioni.

**Neo4J** usa il **property graph model** per rappresentare i dati: è un grafo diretto con etichette sui vertici e sugli archi. Più vertici possono avere archi multipli.

Abbiamo le **entità** che sono i nodi (vertici) e le relazioni (archi).

Abbiamo i **cammini** chiamati path.

Abbiamo i **token** che possono essere label, tipo di relazione, chiave di proprietà.

Abbiamo le **proprietà** che possono essere delle coppie chiave-valore.

**Un'identità** ha un id univoco e comparabile.

Possiamo assegnare delle proprietà ad ogni entità, ognuna delle quali è identificata univocamente dalla **chiave di proprietà**.

Un nodo è l'identità base del grafo.

A un nodo può essere assegnato un insieme di **etichette univoche**.

Ogni nodo ha archi (relazioni) sia entranti che uscenti.

Una **relazione** è un'entità che **codifica esattamente una connessione diretta tra due nodi**.

È sia entrante che uscente, in base a dove si iniziare a guardare.

Ad ogni relazione è assegnata esattamente un tipo.

Un **camino** in un grafo rappresenta una visita dentro il grafo, ovvero una sequenza alternata obbligatoria di nodi e archi.

Un cammino deve sempre iniziare e finire su un nodo.

Il cammino più corto è quello che non include nessun arco, ovvero un vertice A che va nel vertice A, ovvero un cammino vuoto, ovvero nessuna relazione.

Il più lungo invece è uguale al numero massimo di relazioni nel grafo.

I **token** sono stringhe non vuote.

Un'etichetta è un token assegnato solo a un nodo.

Un tipo di relazione è un token assegnato solo alle relazioni.

Una chiave di proprietà è un token assegnato solo alle proprietà.

Le transazioni in un graph database sono delle operazioni che modificano (aggiungono o rimuovono). Sono garantite le proprietà ACID.

Non si può eliminare un nodo senza eliminare anche gli archi ad esso collegati.

**Chyper** è il linguaggio di query che usiamo su nod4j.

Utilizziamo le query per inserire, eliminare o aggiornare i nodi.

Non abbiamo uno schema. Il grafo auto descrive lo schema stesso.

Abbiamo una clausola MATCH usata per ottenere i dati dal grafo.

Abbiamo una clausola WHERE usata per impostare dei vincoli.

Abbiamo una clausola RETURN usata per dire cosa deve restituire.

Non si possono fare operazioni di JOIN.

Tanto le relazioni sono esplicite nel database.

Neo4J non supporta lo sharding.

Garantisce le proprietà ACID su un singolo nodo.

L'architettura di Node4J è master-slave di tipo multi-master.

Ogni nodo è sia Core Server che Read Server.

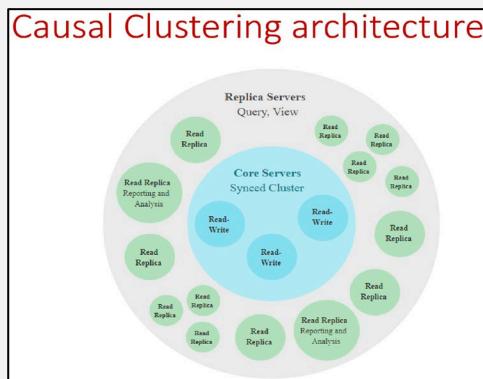
Tutti i server mantengono l'intera copia del database (senza sharding).

Aumentando la replicazione si aumenta l'affidabilità del grafo.

I Core Server accettano sia letture che scritture mentre i Replica Server solo le letture.

I Core Server sono sempre sincronizzati tra loro, quelli Replica in base a delle politiche.

Avere tanti Core Server mi dà la garanzia che fino quando la maggioranza è attiva il cluster funziona.



È possibile scalare le repliche: le posso aumentare se mi serve tanto traffico in lettura e viceversa.

**Causal consistency:** siccome la lettura può avvenire su repliche non ancora sincronizzate garantisco comunque al client il modello **read your writes** usando un bookmark. Ovvero è sicuro di leggere almeno quello che scrive.

Per questo si chiama Causal Clustering architecture.

Per garantire i requisiti di sicurezza i Core Server elaborano le operazioni di scrittura attraverso un quorum. Quindi, è possibile utilizzare il cluster finche  $(N / 2) + 1$  è attiva.

Se vogliamo una tolleranza pari a F occorre usare un numero di Core Server pari a  $2F + 1$ . In modo tale che se ne perdo F Core Server, con  $2F + 1$  ho ancora la metà + 1.

Se non ho più il quorum in termini di Core il cluster funziona ma solo in lettura.

I Read Server processano operazioni solo in lettura e quindi danno scalabilità in termini di lettura. Sono aggiornate in maniera asincrona: periodicamente i Replica Server andranno a chiedere a un Core Server quali sono le ultime transazioni leggendo un transaction log, il Core Server gliene dà e così si aggiornano in maniera asincrona.

Tuttavia, può esse già vecchio di qualche secondo.

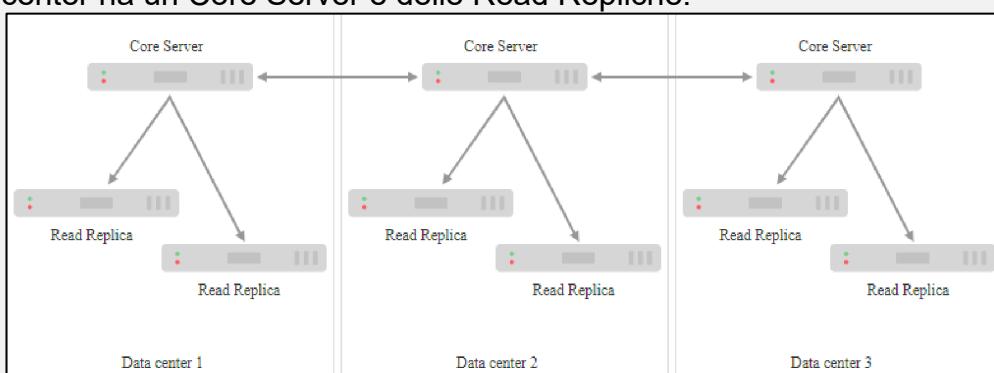
Posso avere un numero di read server alto quanto voglio, non uso neanche il quorum.

Vengono usati come risorsa se servono li metto sennò no.

Non hanno nessun impatto sul cluster.

Possiamo avere anche cluster geo distribuiti.

Ogni datacenter ha un Core Server e delle Read Repliche.



Quello in foto è detto omogeneo: per le scritture ho una latenza alta perché devo aspettare degli ACK però è molto robusto perché anche se perdo un intero datacenter posso ancora scrivere. Se ne ho 3 come in foto ne posso perdere 1 ( $N / 2 + 1$  garantito).

È possibile anche avere un solo Core Server per più datacenter.

Posso anche avere cluster con sviluppi eterogenei con Read Server da una parte e un Core Server in un altro adiacente.

Non è robusta la perdita di un intero datacenter, se salta il Core Server è finita.  
Mettere Core Server dove non servono è una fatica inutile.

Anche nelle versioni geo distribuite si può assumere che ogni client legga almeno le proprie scritture: quando il client scrive il Core Server gli restituisce un bookmark. Quando fa delle letture il client lo chiede e la Read Repliche lo restituisce solo se dispone di quel bookmark, sennò lo chiede ad altri.  
Si ottiene così una catena causale.

## Capitolo 14 – Apache Kafka

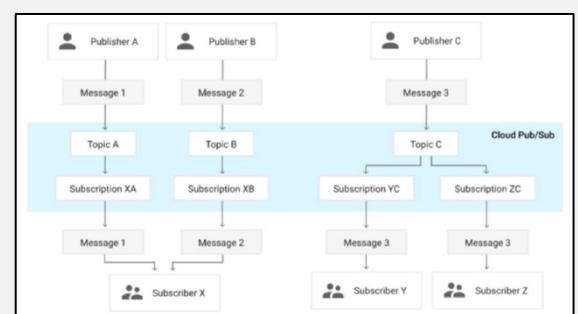
Utilizza un'architettura di tipo **publish-subscribe**.

I mittenti di messaggi, chiamati **publishers**, non programmano i messaggi da inviare direttamente a destinatari specifici, chiamati **subscribers**, ma li inviano invece a specifiche classi **senza alcuna conoscenza** dei sottoscrittori.

I subscribers esprimono interesse per una o più classi e ricevono solo i messaggi di interesse, **senza alcuna conoscenza** dei publishers.

Il processo di selezione dei messaggi per la ricezione e l'elaborazione è chiamato **filtraggio**, può essere di due tipologie:

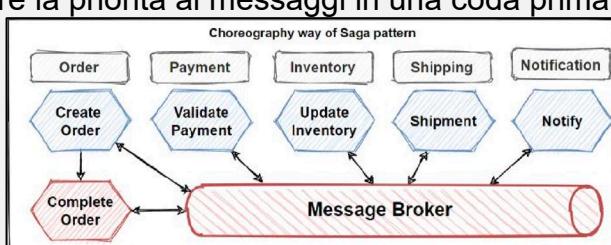
- **Topic-based**: i messaggi vengono pubblicati su dei topic. Gli abbonati in un sistema basato su topic riceveranno tutti i messaggi pubblicati nei topic a cui si iscrivono.
- **Content-based**: i messaggi vengono consegnati a un abbonato solo se gli attributi o il contenuto di tali messaggi corrispondono ai vincoli definiti dall'abbonato. È più flessibile, non devo definire dei canali logici per forza (i topic). Però in compenso è più complesso.



In molti sistemi publish-subscribe, gli editori inviano messaggi a un broker di messaggi intermedio e gli abbonati registrano le sottoscrizioni con quel broker, lasciando che il broker esegua il filtraggio. Offrono una scalabilità al sistema memorizzando le informazioni del publisher e facendo forward sui subscriber interessati.

Il broker esegue normalmente una funzione di archiviazione e inoltro per instradare i messaggi dagli editori agli abbonati.

Inoltre, il broker può dare la priorità ai messaggi in una coda prima dell'instradamento.



L'architettura a microservizi si sposa con questa tecnologia perfettamente.

Le applicazioni ad alta scalabilità seguono deployment a microservizi.

Un microservizio si registra in un message broker e quando arrivano degli eventi legati a determinati topic (ad esempio un acquisto) possono leggere tali messaggi e comunicare quando finiscono le loro conseguenti azioni.

In questo modo il sistema è asincrono e altamente scalabile.

### Event streaming

A fornire degli eventi possono essere molteplici sorgenti (database, sensori, ecc.).

Gli event streams sono dei dati che arrivano in tempo reale da una sorgente.

Lo streaming di eventi è la pratica di:

- Acquisizione di flussi di eventi.
- Memorizzazione duratura dei flussi di eventi per un successivo recupero.
- Manipolazione, elaborazione e reazione ai flussi di eventi in tempo reale e retrospettivamente.
- Indirizzare i flussi di eventi a diverse tecnologie di destinazione secondo necessità.

Esempi di streaming di eventi possono essere:

- Monitorare i pazienti ricoverati in ospedale e prevedere i cambiamenti delle condizioni per garantire un trattamento tempestivo in caso di emergenza.
- Acquisire e analizzare continuamente i dati dei sensori da dispositivi IoT o altre apparecchiature, ad esempio nelle fabbriche e nei parchi eolici.
- Per tracciare e monitorare auto, camion, flotte e spedizioni in tempo reale, come nella logistica e nell'industria automobilistica (esempio, Uber).

### Apache Kafka

Implementa streaming di eventi attraverso un sistema publish-subscribe che include import / export continua da altri sistemi.

Lo store degli eventi viene conservato fin quando occorrono i dati per processarli quando serve, anche in modo retrospettivo.

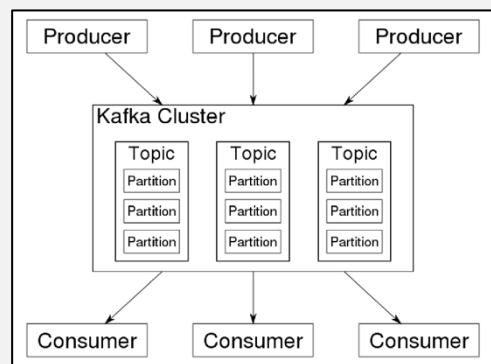
È scalabile, tollerante agli errori, elastico.

**Funzionamento:** Kafka archivia i messaggi (coppie chiave-valore) che possono provenire da tanti **producers**. I dati possono essere partizionati in partizioni con differenti topics.

I messaggi restano ordinati secondo il tempo in cui sono stati scritti, sono infatti equipaggiati con un **timestamp**.

I **consumer** sono tutti quei processi che leggono i messaggi dalle varie partizioni.

I **topic** sono essenzialmente i nodi del cluster.



### Architettura

Abbiamo tre tipologie di server:

- **Storage Layer:** server che hanno la funzione di conservare i dati, detti **brokers**.
- **Kafka Connect:** server che hanno la funzione di succhiare i dati per poi passarli ai Kafka server che li memorizzano nelle loro partizioni.

- Client: possiamo considerarli come dei pezzi di software che implementiamo nelle nostre applicazioni e che ci permettono di usare Kafka.

Quindi abbiamo tre attori fondamentali: Kafka Connect che invia automaticamente i dati dentro al cluster Kafka, i broker e il client ci permette di interagire con Kafka.

Sono tutti e tre dei server.

Kafka usa ZooKeeper per la gestione del suo cluster in quanto esso permette una configurazione dinamica. Inoltre, lo utilizza per notificare al produttore e al consumatore la presenza o il fallimento di qualsiasi broker nel sistema Kafka.

Un **evento** registra il fatto che "è successo qualcosa" nel mondo o in un sistema specifico. Concettualmente, un evento ha una chiave, un valore, un timestamp e intestazioni di metadati facoltative.

Event key: "Alice"  
 Event value: "Made a payment of \$200 to Bob"  
 Event timestamp: "Jun. 25, 2020 at 2:06 p.m.".

I producer sono applicazioni client che scrivono eventi, i consumer sono anch'essi applicazioni client che consumano eventi. Possiamo essere entrambi.

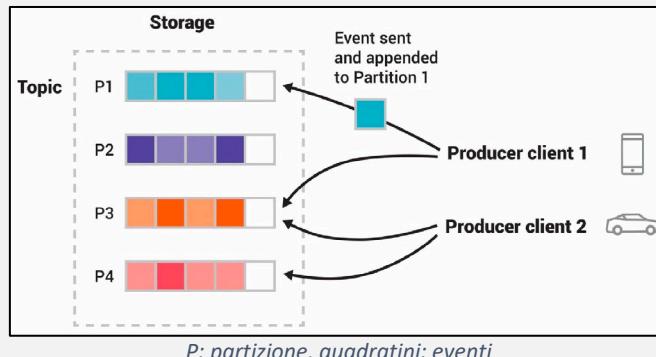
Sono completamente scollegati.

I **topic** possono avere zero, uno o tanti producer e zero, uno o tanti consumer.

In Kafka gli eventi possono essere letti tante volte, non vengono eliminati dopo una singola lettura. Possiamo comunque scegliere di eliminare tutti gli eventi dopo un tot intervallo di tempo in modo da avere uno storage di dimensioni costante.

**Sharding**: i topic sono partizionati, il che significa che un topic è distribuito su un numero di buckets situati su diversi broker Kafka.

- Gli eventi con la stessa chiave (ad esempio, un ID cliente) vengono scritti nella stessa partizione e Kafka garantisce che qualsiasi consumatore di una determinata topic-partizione leggerà sempre gli eventi di quella partizione esattamente nello stesso ordine in cui sono stati scritti.



**Replication**: ogni topic può essere replicato, anche tra aree geografiche o data center, in modo che siano sempre presenti più broker che dispongono di una copia dei dati.

- Un'impostazione di produzione comune è un fattore di replica pari a 3.
- Questa replica viene eseguita a livello di topic-partitions, ovvero Kafka prende intere partizioni e le replica.

API principali: Java. Caso d'uso: Uber.

