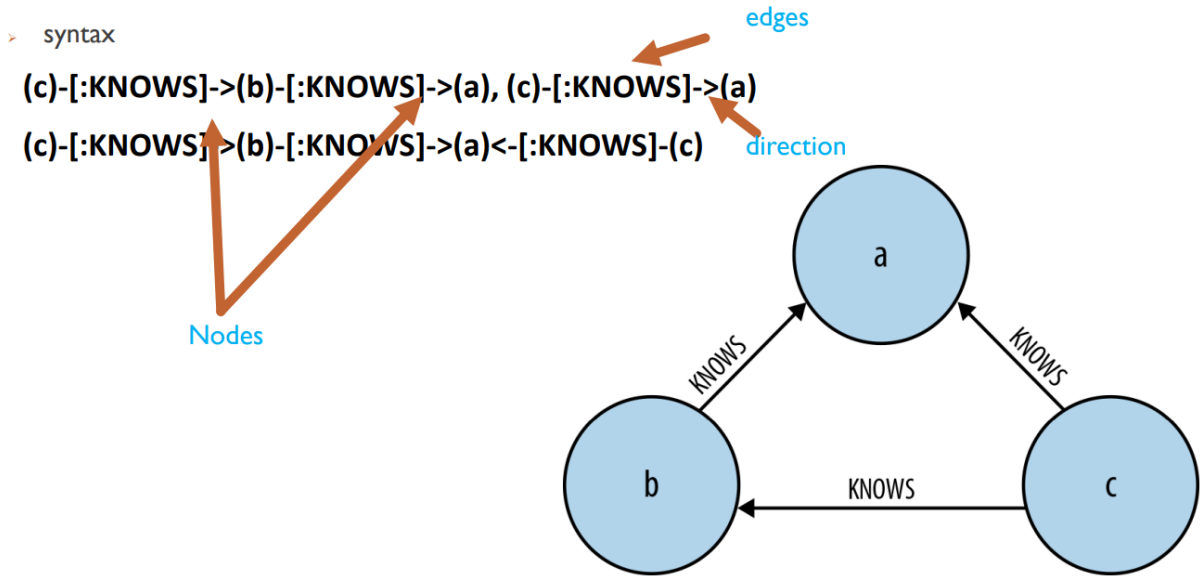
Lezione 7 08/04/2024

Query languages

Come posso interrogare un grafo tramite una query testuale? Il linguaggio usato si chiama **cypher**. Può fare delle operazioni di aggregazione, è un linguaggio orientato ai grafi.

Il motore va a trovare tutti i sottografi che matchano la query.



il primo è “c conosce a”, il secondo “a è conosciuto da c”.

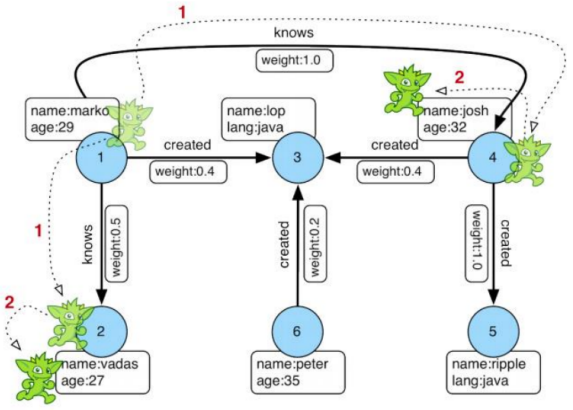
Query cypher:

```
MATCH (c:user)
WHERE (c)-[:KNOWS]->(b)-[:KNOWS]->(a), (c)-[:KNOWS]->(a), c.user=“Michael”
RETURN a, b
```

Il risultato di una query è un sottografo o una tabella (nel caso avessi ritornato il nome del nodo a e il nome del nodo b al posto che ritornare i due nodi direttamente).

Un altro linguaggio è **gramlin**, a differenza di cypher, qui vado a spiegare il percorso che deve fare l’ipotetico “folletto” nel grafo.

- g.V().has('name','marko').out('knows').values('name')

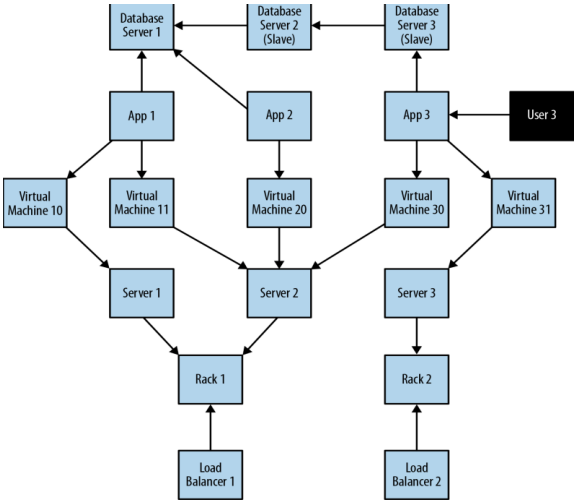


Immaginiamo di istanziare i folletti, che possono andare in giro seguendo gli archi di uscita che si chiamano “knows”, duplicando il folletto.

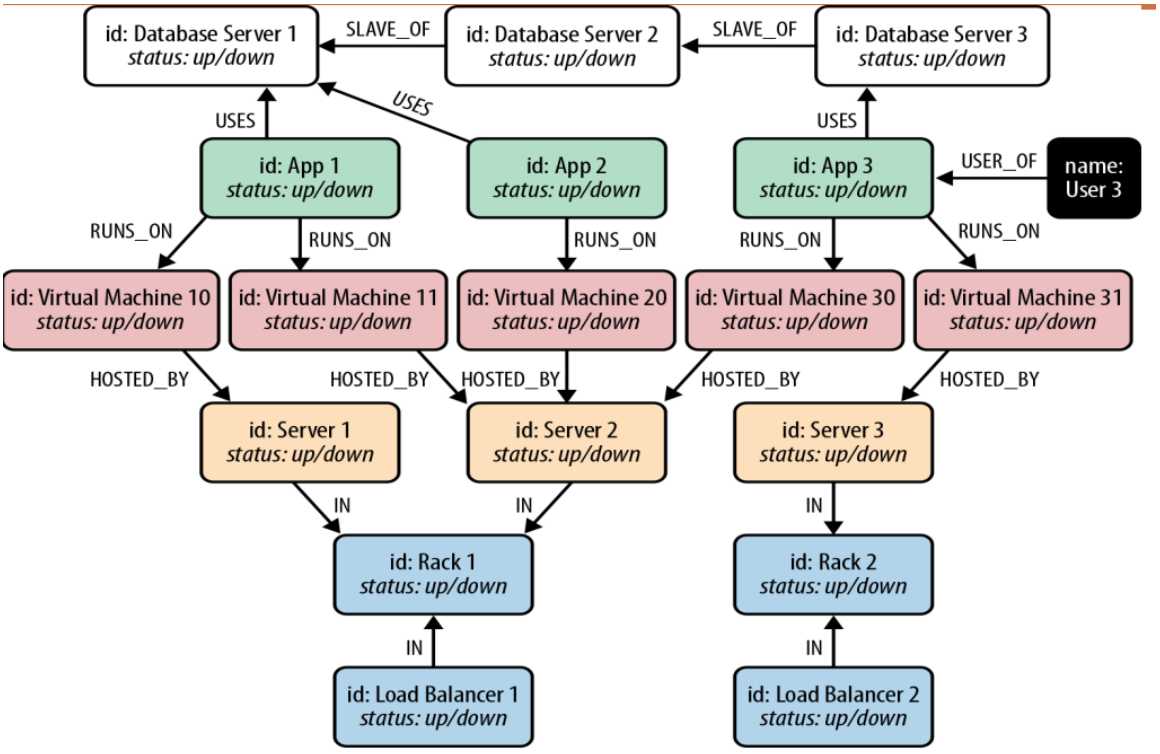
Comparazione tra modelli relazionali e a grafo

Per esempio se parliamo di una server farm.

Un attributo comune è “up” o “down”.



Questa è la stessa rappresentazione relazionale ma come un grafo:



- User 3 notify that the application does not work.
- The app or any other asset related to it (database, virtual machine, server, rack, load balace) can be down
- Find any asset whose status is down for user 3

```
START user=node:users(name = 'User 3')
MATCH (user)-[*1..5]-(asset)
WHERE asset.status! = 'down'
RETURN DISTINCT asset
```

Questo è uno dei vantaggi di modellare come un grafo.

Database poliglotti

Sono database che sanno parlare più lingue.

ArangoDB

è un database che supporta key value, document based, graph.

Le collezioni sono l'equivalente di una tabella relazionale. Viene salvato tutto in json. Quindi i key value saranno dei documenti chiave-valore.

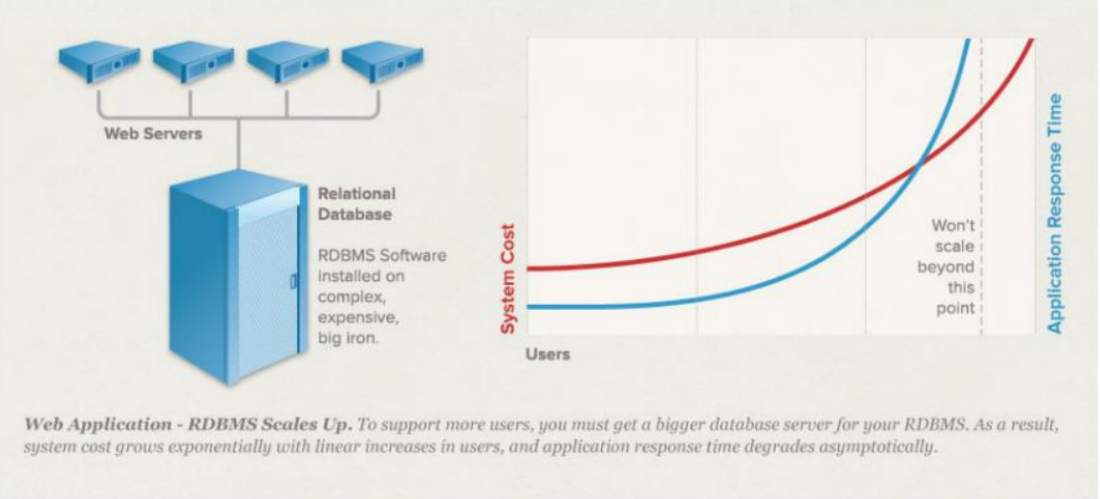
Distributed NOSQL

Il problema fondamentale, che è uno dei motivi per cui c'è stata la nascita dei modelli non relazionali, è la scalabilità.

Abbiamo parlato di scenari dove il modello non relazionale non è efficiente. I RDBMS possono scalare verticalmente facilmente (aumento di potenza alle risorse attuali), ma non orizzontalmente (incremento delle risorse per dividere il carico). Quindi la scalabilità verticale ha un limite fisico, mentre quella orizzontale tecnicamente no.

C'è un altro problema più teorico. I sistemi distribuiti hanno un limite di scalabilità.

> You cannot add servers forever



Web Application - RDBMS Scales Up. *To support more users, you must get a bigger database server for your RDBMS. As a result, system cost grows exponentially with linear increases in users, and application response time degrades asymptotically.*

Per garantire i protocolli come il two phase commit, dopo un certo punto non funzionano più, perché devono richiedere l'ok a tutti i nodi, etc.

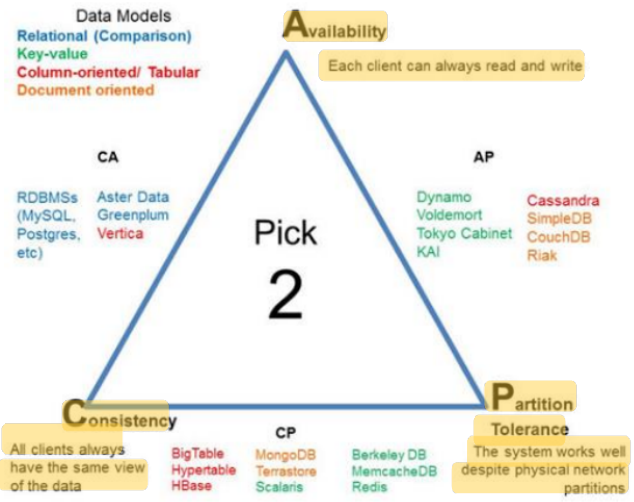
I sistemi noSQL non hanno lo schema, e sono stati implementati usando un approccio radicalmente diverso, basato su due approcci fondamentali:

- **CAP theorem**: **Consistency** (ad un certo momento, tutti i nodi vedono la stessa versione dei dati), **Availability** (tutti i nodi sono sempre raggiungibili), **Partition tolerance** (il sistema funziona anche se c'è un partizionamento della rete). Un sistema distribuito può soddisfare 2 di questi requisiti contemporaneamente, ma non tutti e 3. Per esempio i social sacrificano la consistenza.

I RDBMS sono CA. I noSQL sono CP o AP.

CP: i dati sono coerenti ma il dbms non funziona 24/7.

AP: a volte i dati non sono consistenti.



- **Base**:
  - Basic Availability: i sistemi cercano di garantire la disponibilità.
  - Soft state: la consistenza dei dati non è garantita (abbandona i requirements ACID).
  - Eventual consistency: ad un certo punto i dati convergeranno in uno stato consistente, ma questo è delayed, non immediato. Quindi prima della convergenza potrebbe ritornare dei valori diversi.

Bisogna capire quale piattaforma è quella più efficace in base alle esigenze.

- Key-Value Stores (redis)
- Column Family Stores (HBASE, CASSANDRA)
- Document Databases (mongoDB)
- Graph Databases

Redis

è un key value, i dati possono essere replicati, non frammentati.

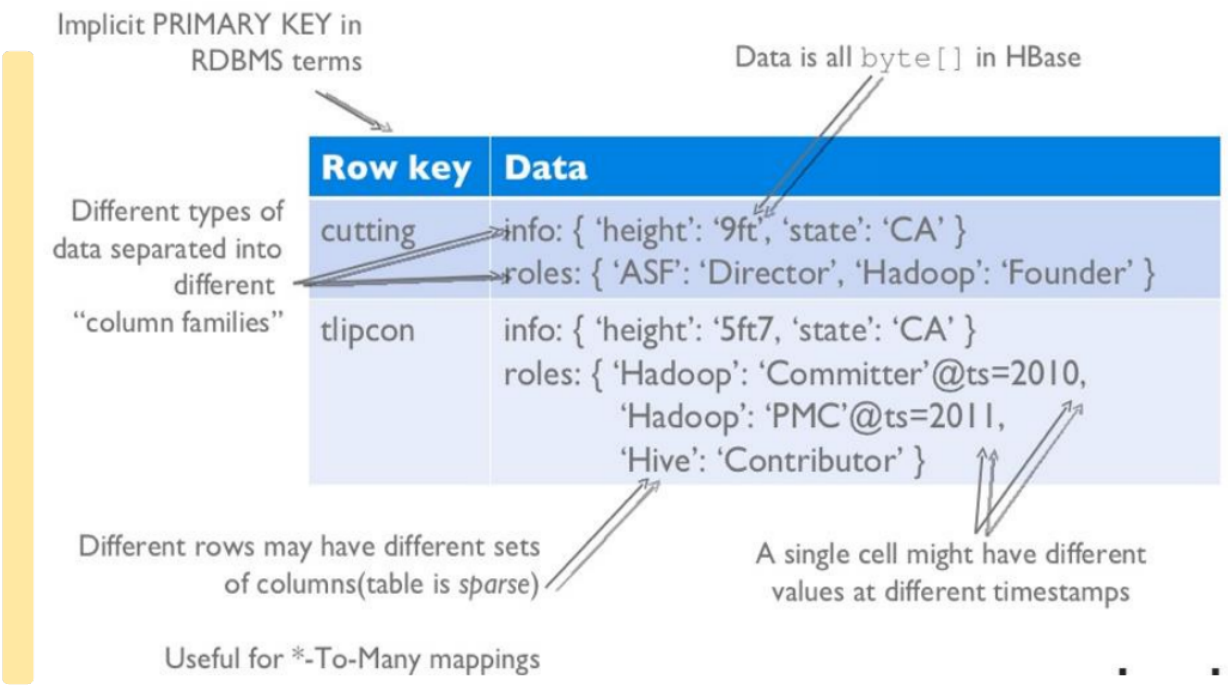
Viene fatto l'hashing, perchè oltre ad essere veloce, hanno una logica che rende possibile facendo una query su un nodo, identificare il routing per raggiungere gli altri nodi.

Si può fare una distribuzione su cluster diversi.

Tutto questo avviene in modo trasparente rispetto all'utente.

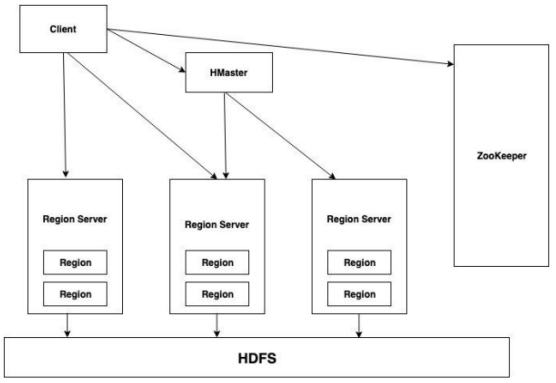
HBase

C'è il concetto di column family. (google)



è un modello master-slave. C'è l'HBaseMaster, e i HRegionServer (slaves).

Il client fa una query sul master, che gli dice dove sono i dati, e poi il client esegue la query sugli slave (region server).



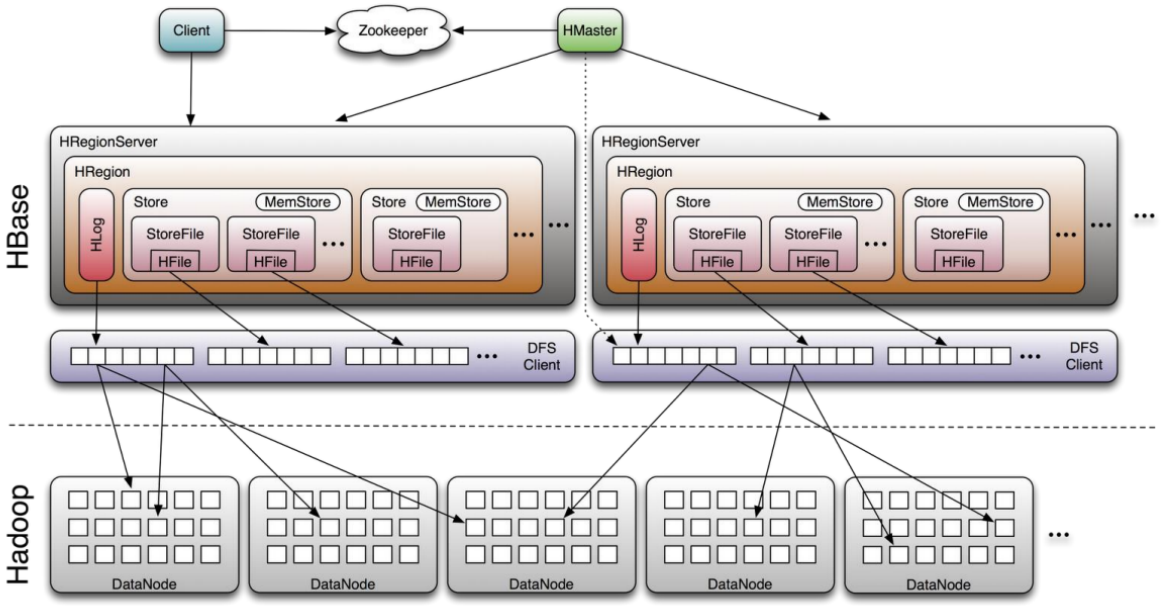
La regione è la distribuzione logica dei dati, ogni table viene suddivisa tra le column family in regioni diverse. Queste regioni possono essere memorizzati su diversi region server (slave, diverse macchine).

Il master conosce dove ogni pezzo della tabella si trova. Quando il client fa una query verso una particolare regione, il master scopre qual è lo slave (region server) e reindirizza il client.

Il client scrive sul master (sul file di log), il file di log viene passato agli slave.

Quindi il master è responsabile del collegamento degli slave.

Il master si accorge dei nodi che smettono di funzionare e redistribuisce i dati nel caso accade. Di solito c'è un numero minimo di repliche di 3.



Il master coordina tutto, e quindi è il single point of failure. Quindi spesso viene replicato.

C'è un personaggio nuovo, zookeeper, è un software che coordina la comunicazione tra master e client, e di riconoscere i fallimenti degli slaves, e gestirli. Sia il master che gli slaves si registrano con lo zookeeper.

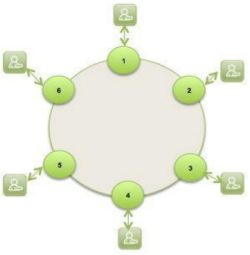
Questo garantisce una scalabilità orizzontale, i server possono aumentare quanto voglio, l'unico limite è la capacità del master di gestirli.

Cassandra

Come modello dati è simile a HBase. Ci sono le column family. Mentre Hbase è CP (quindi se un nodo casca, non garantisce la disponibilità), Cassandra è AP (non considera importante la consistenza).

Ha un'architettura peer to peer. L'idea è che non c'è un nodo che sa tutto, tutti i nodi sono uguali e sono collegati in maniera da garantire la disponibilità e il partizionamento.

Ogni nodo cassandra è inserito in un anallo, i nodi sono ordinati rispetto al valore della chiave di hashing.

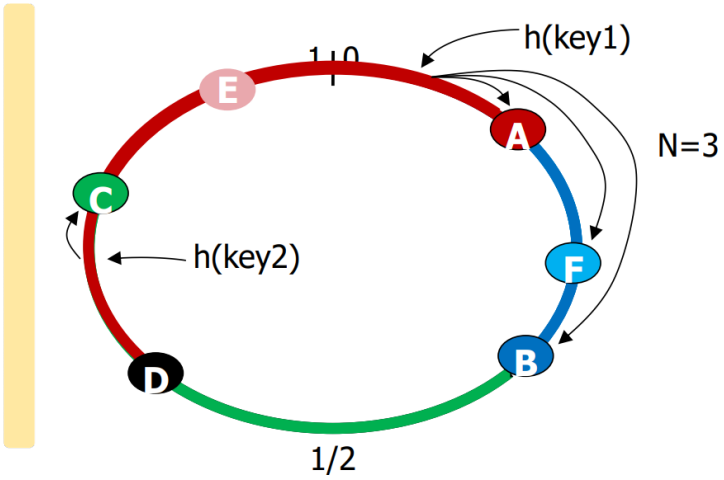


I nodi possono essere aggiunti e rimossi senza alcun downtime.

è usato da facebook.

Non c'è single point of failure, i dati sono replicati (3 volte almeno) e in caso di fallimento, i dati verranno recuperati.

Immaginiamo un anello (spazio di indirizzamento visto come un anello, ovvero la funzione di hash da 0 a 1), creo 3 nodi A B C. Con cassandra inserisco una chiave, in base alla funzione di hashing il nodo viene messo nel punto giusto dell'anello, tra altri nodi.



Quindi se aggiungo un nuovo nodo D, e poi aggiungo un dato (h(key2)) questo va su C, i dati su C che dovrebbero essere su D verranno duplicati o trasferiti. Quindi quando aggiungo un nuovo nodo se ne preoccupa solo il prossimo nodo a valle (al massimo anche quello precedente), il resto della rete no.

Si replica in avanti, se faccio una replica a 3 nodi, allora ogni volta che scrivo su A, scrivo anche su F e B.

Come faccio a scoprire quando A è morto (dato che non c'è un master che sa tutto)? Si utilizza il protocollo di gossip: il nodo A parla con il suo vicino, dicendogli la sua situazione, quindi F parla con il prossimo, dicendo che sta bene e che anche A sta bene.

Quindi per scoprire che un nodo non è più disponibile, si usa il meccanismo di **Accrual Failure Detector**, non si da un valore 0-1 per la disponibilità, ma si assegna un valore di probabilità. Quando mando un messaggio di gossip

conosco anche le condizioni del traffico, so quanto ci metto a comunicare, se un nodo rallenta nella comunicazione allora la probabilità del fault aumenta. Una volta settata la soglia, questa è assegnata al tempo medio di comunicazione, per evitare di assegnare sistemi morti quando non lo sono (se setto un timeout troppo basso).

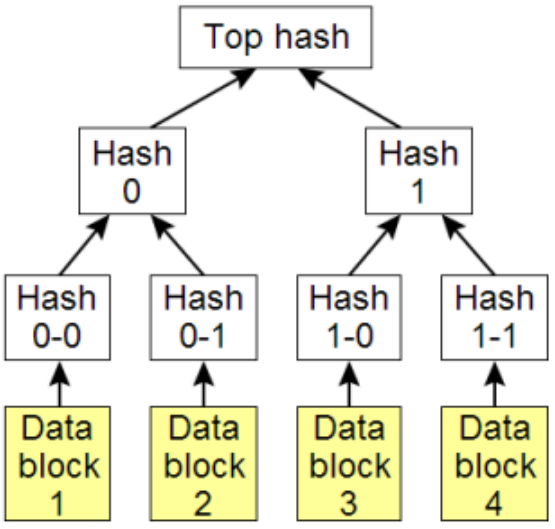
Come fa il singolo nodo a salvare i dati? Ogni singolo nodo lavora così: quando ha un'operazione di scrittura, scrive sul file di log e poi scrive in memoria. In questo modo le operazioni interrotte possono essere recuperate tramite il log.

La sintassi delle query sembra un SQL, ma visto che la consistenza non è garantita, noi possiamo scegliere i livelli di consistenza. Per esempio se scrivo consistency one, la scrittura viene terminata quando il nodo scrive. Però prima che il nodo replica il dato, il dato potrebbe essere perso. Questo è scelto nelle query. Se invece metto consistency quorum, allora la metà + 1 delle replicheе devono aver dato l'ok. Altrimenti devo scegliere consistency all. Questo è l'eventually consistency.

Cosa succede se un nodo muore quando deve essere scritto un dato sul nodo? Il nodo aspetta e continua a riprovare a scrivere. Se il nodo non si riprende, dopo un po' ci si dimentica del dato (perchè la consistenza non c'è).

Come si cancellano i dati? I record sono segnati come cancellati, come se fosse il cestino, i dati sono cancellati durante operazioni di grande compattazione o tramite un timer configurabile.

I nodi nei protocolli di gossip si parlano, e dicono ai vicini anche che dati hanno, di modo che un nodo si può accorgere di non avere una copia dei dati. Viene usato un albero di hashing crittografico chiamato **markle tree** dove sostanzialmente si prendono i dati, si costruisce una funzione di hashing sui dati, su queste funzioni di hashing si costruisce un'altra funzione di hashing e così via. Il nodo passa al nodo successivo solo il top hash, se questi non sono uguali si scende



l'albero per trovare la sorgente del problema.

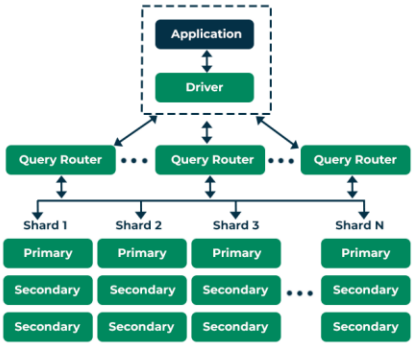
Questo sistema evita l'anti entropia, assicurandosi della sincronizzazione tra i nodi.

Per quanto riguarda la lettura questa può essere su un solo dato (consistency one), oppure any o quorum.

Chi scrive e chi legge può usare politiche diverse. Non esiste il concetto di transazione.

MongoDB

MongoDB è CP, consistenza al partizionamento, non garantisce che è sempre disponibile. Architettura master slave.



Ha 2 concetti fondamentali: ogni dato è sia replicato che frammentato. In ogni shard c'è il pezzo primario di un dato e delle copie secondarie di altri. Gli slave sono istanze diverse di mongodb, il master è un router che riceve la richiesta e indirizza i dati, come HBase.

Le repliche possono avere vari stati.

- Ogni nodo può assumere uno di questi stati
  - STARTUP: un nodo che non è membro effettivo di nessun replica set.
  - PRIMARY: l'unico nodo nel replica set che accetterà le operazioni di lettura.
  - SECONDARY: un nodo che di effettuare la sola replicazione dei dati. (può essere promosso a PRIMARY)
  - RECOVERING: un nodo che sta effettuando una qualche operazione di recupero dei dati. (può essere promosso a PRIMARY)
  - STARTUP2: un nodo che è appena entrato nel set. (può essere eletto a PRIMARY)
  - ARBITER: un nodo non atto alla replicazione dei dati con lo scopo di prendere parte alle elezioni per promuovere i nodi a PRIMARY. (può essere eletto a PRIMARY)
  - DOWN/OFFLINE: un nodo irraggiungibile.
  - ROLLBACK: un nodo che sta svolgendo un rollback per cui sarà inutilizzabile per le letture. (può essere promosso a PRIMARY)

Se il master cade qualcuno viene promosso, gli slave sono numerati da 1 a n, quello col numero più basso viene promosso.

Se il nodo primario è riconosciuto come down, le repliche effettuano un'elezione e tipicamente votano quello con la versione più aggiornata dei dati.

Il meccanismo di elezione funziona così: ogni replicaSet se non può comunicare con il nodo padre comunica con i fratelli, a questo punto partono le elezioni, si segue la gerarchia in base all'ordine con cui sono numerate le repliche, quello con proprietà più alta è quello che chiama le elezioni. Un replica set può avere al massimo 50 membri di cui solo 7 votanti. Il secondario che viene promosso diventa il primario.

Come funziona la scrittura? Durante un'operazione di scrittura è possibile impostare un'opzione chiamata writeConcern per indicare al sistema il comportamento per le replica set, ovvero quanti nodi devono scrivere prima di ricevere l'ok della scrittura.

- W=0 nessuna certezza dell'inserimento
- W=1 scrittura sul nodo primario
- W=n scrittura su almeno n nodi (a maggioranza)
- W=majority almeno la meta più uno dei nodi deve avere scritto prima di avere conferma dell'operazione

Esiste anche un timeout dopo che l'operazione viene annullata.

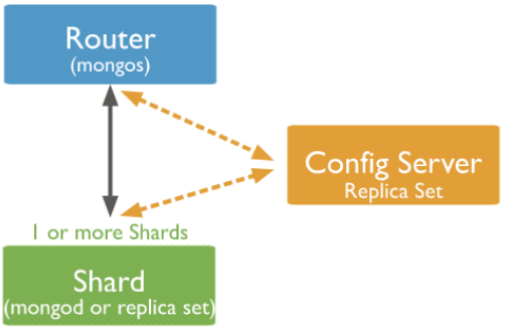
Anche mongodb usa il logging per annotare l'intenzione di scrivere, prima di scrivere.

La frammentazione è automatica. Una volta che ho preparato i chunk devo metterli sui nodi, esiste un programma mongos che legge da un file di configurazione tutte le repliche disponibili e le ridirige a tutti i nodi.

Ruoli

- Mongos: L'entry point del cluster, tutte le query verranno eseguite su questo processo che, in base alle operazioni,
  - target query
  - broadcast query
- Config Server: I server che ospitano i file di configurazione del cluster, necessari al sistema per avere informazioni sulla posizione dei documenti nei vari shard.
- MongoDB richiede che sia gli Shard sia i Config Server siano configurati su dei Replica Set così da garantirne la disponibilità in caso di guasti e non avere un unico point of failure.

Architettura



Lo shard primario contiene tutte le collezioni ed eventualmente un secondo shard contiene le copie.

Per la lettura c'è lo stesso meccanismo del readConcern. Local significa che leggi il valore sul primario o sul più vicino, non viene effettuata una verifica

sugli altri nodi. Available legge i dati sui nodi secondari, che possono essere più vicini al client. Majority legge quando la metà più uno delle repliche ha ricevuto un ack in scrittura sul dato.

In MongoDB la scrittura avviene sempre sul nodo primario, e poi sulle repliche tramite il file di log, mentre le letture si fanno su qualunque nodo.

MongoDB non garantisce la disponibilità, perchè se sto scrivendo e sto trasferendo i dati e ho un problema e quindi non riesco a replicare tutti i dati, il sistema non è più disponibile finchè finchè tutte le repliche sono state scritte. Quindi garantisco la consistenza e perdo la disponibilità.

Se casca un nodo, devo ricostruire gli indici e le copie, e quindi il sistema non è disponibile.

Dalla versione 4 mongodb gestisce le transazioni. Ovvero si possono modificare due json diversi, sapendo che alla fine o entrambi sono modificati o nessuno dei due.