Introduzione ai database NoSQL

Ragioni per l'uso dei database:

Dati Persistenti: I database sono utilizzati da lungo tempo per garantire l'archiviazione persistente delle informazioni per le applicazioni software.

Flessibilità: Rispetto al file system del sistema operativo, un database consente un accesso più rapido e facile a piccole porzioni di informazioni.

Concorrenza: Le transazioni nei database forniscono il controllo della concorrenza per le applicazioni aziendali, permettendo a molti utenti di accedere ai dati simultaneamente. Le proprietà ACID (Atomicità, Consistenza, Isolamento e Durabilità) garantiscono che le transazioni siano affidabili e possano essere annullate in caso di errore.

Diagramma che mostra i vantaggi dell'uso dei database.

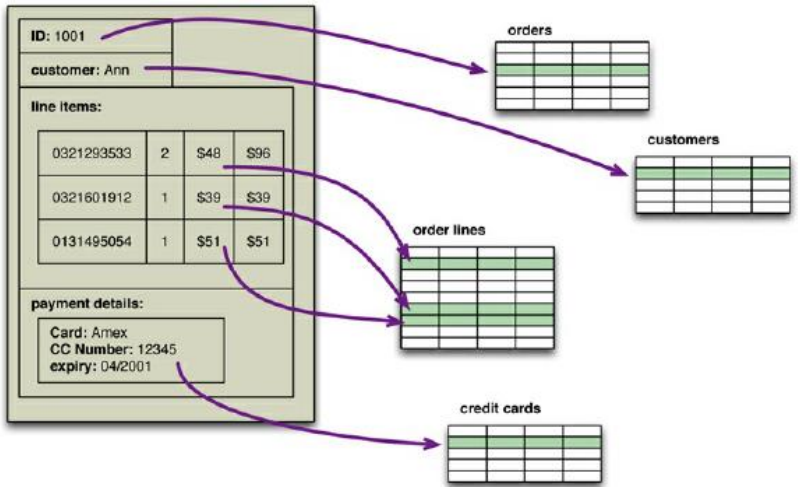
Diagramma che mostra i vantaggi dell'uso dei database.

Integrazione tra applicazioni:

Le applicazioni software devono collaborare e condividere i dati, visibili a tutte le altre applicazioni coinvolte.

Le soluzioni di database relazionali sono state prevalenti per venti anni grazie alla loro capacità di gestire query SQL standard, rendendo le applicazioni portabili.

Diagramma che illustra l'integrazione tra applicazioni.



Mismatch di impedenza:

La differenza tra il modello relazionale e le strutture dati in memoria causa frustrazione negli sviluppatori.

Il modello relazionale richiede dati semplici e non strutturati, mentre le strutture in memoria possono essere più complesse.

Database orientati alle applicazioni:

Un database a cui può accedere una singola applicazione e la cui manutenzione è affidata al team di sviluppo di quella applicazione, mantenendo la struttura dei dati semplice e con maggiore libertà nella scelta del modello di dati.

Applicazioni orientate ai servizi (SOA):

Con l'aumento delle dimensioni e complessità delle applicazioni, i servizi web e SOA hanno permesso una migliore interoperabilità e flessibilità nei protocolli di interazione.

I modelli di dati possono essere più flessibili e ricchi (es. XML e JSON).

L'emergere dei database NoSQL:

I database NoSQL sono nati come movimenti piuttosto che tecnologie specifiche, spesso senza SQL come linguaggio di query.

Offrono una maggiore scalabilità e facilità di programmazione, supportando strutture di dati schemaless o a colonne, chiavi-valore e grafi.

Esempi di database NoSQL.

Esempi di database NoSQL.

Supporto ai cluster:

L'uso di cluster di macchine economiche è aumentato con la bolla dot.com, poiché le soluzioni su mainframe erano costose e meno resilienti.

I database relazionali faticano a funzionare bene sui cluster a causa della difficoltà nel garantire le proprietà ACID e nel gestire la complessità del partizionamento dei dati.

Esempio di cluster support.

Partizionamento e Sharding:

Il partizionamento suddivide tabelle molto grandi in parti più piccole per migliorare le prestazioni delle query.

Lo sharding è una forma di partizionamento orizzontale, replicando lo schema su più server e dividendo i dati basati su una chiave di shard.

Diagramma che mostra partizionamento e sharding.

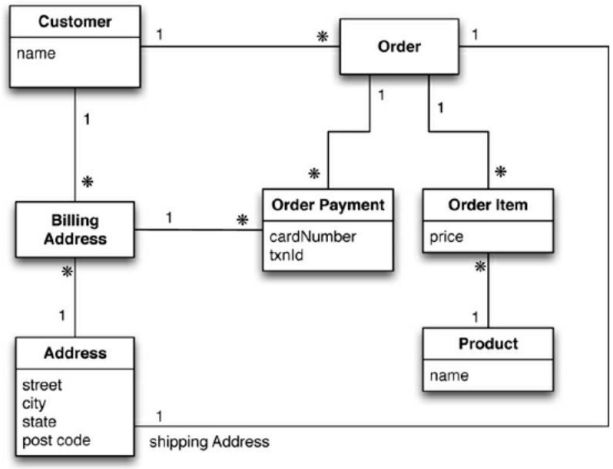


Diagramma che mostra partizionamento e sharding.

Customer

Id	Name
1	Martin

Order

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Diagramma che mostra partizionamento e sharding.

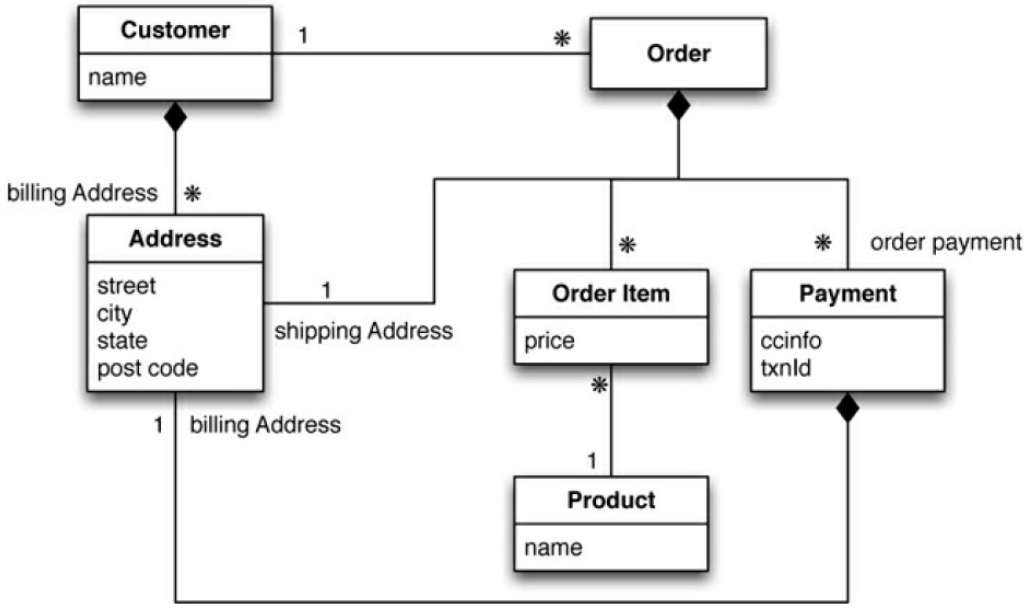


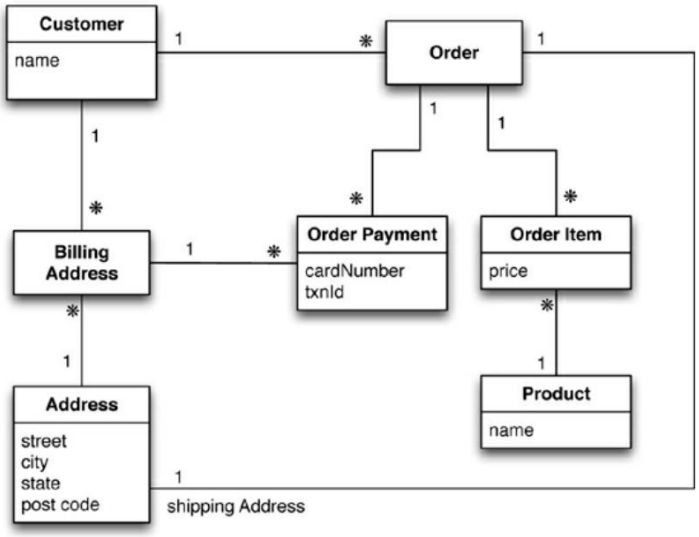
Figure 2.3. An aggregate data model

Modelli di dati nei NoSQL:

I database NoSQL possono essere suddivisi in quattro categorie principali: chiavi-valore, documenti, colonne-famiglia e grafi.

I modelli di dati aggregate orientati raggruppano oggetti da trattare come unità, facilitando il partizionamento e la replica nei cluster.

Tipi di modelli di dati NoSQL.



Tipi di modelli di dati NoSQL.

Customer

Id	Name
1	Martin

Order

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Tipi di modelli di dati NoSQL.

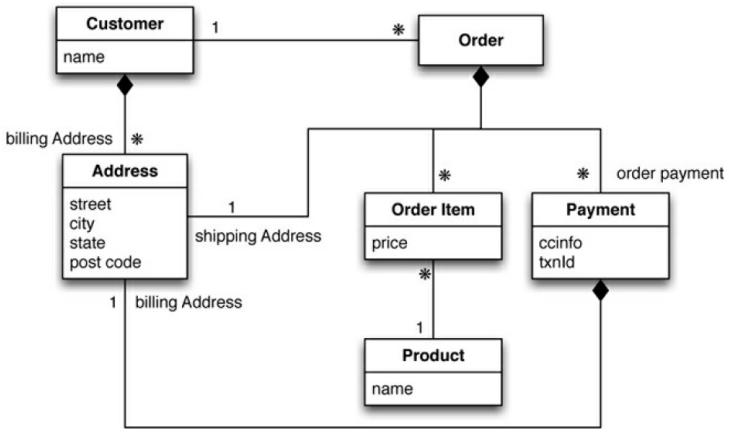


Figure 2.3. An aggregate data model

Esempio di caso d'uso: e-commerce:

L'applicazione di un sito di e-commerce gestisce il catalogo dei prodotti, utenti, ordini, indirizzi di spedizione e dati di pagamento.

Modello di dati relazionale e JSON rappresentato negli esempi forniti.

Esempio di modello di dati e-commerce.

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Esempio di modello di dati e-commerce.

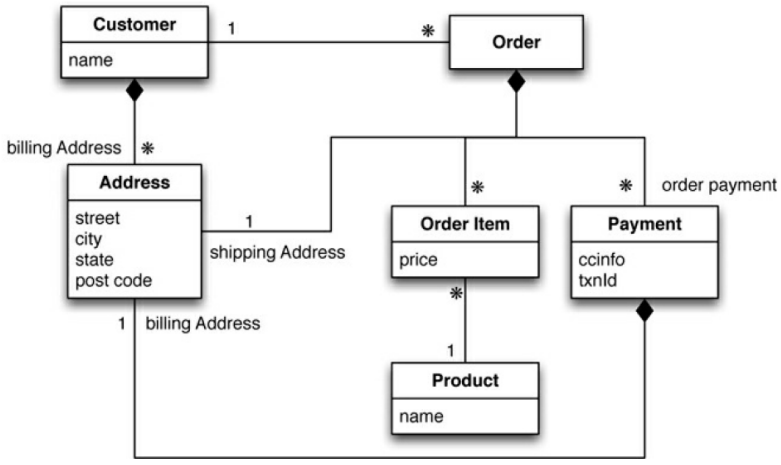


Figure 2.3. An aggregate data model

Esempio di modello di dati e-commerce.

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
```

Esempio di modello di dati e-commerce.

Modelli ignoranti e consapevoli degli aggregate:

Il modello relazionale è aggregate-ignorant, permettendo la manipolazione di combinazioni di righe da diverse tabelle in una singola transazione.

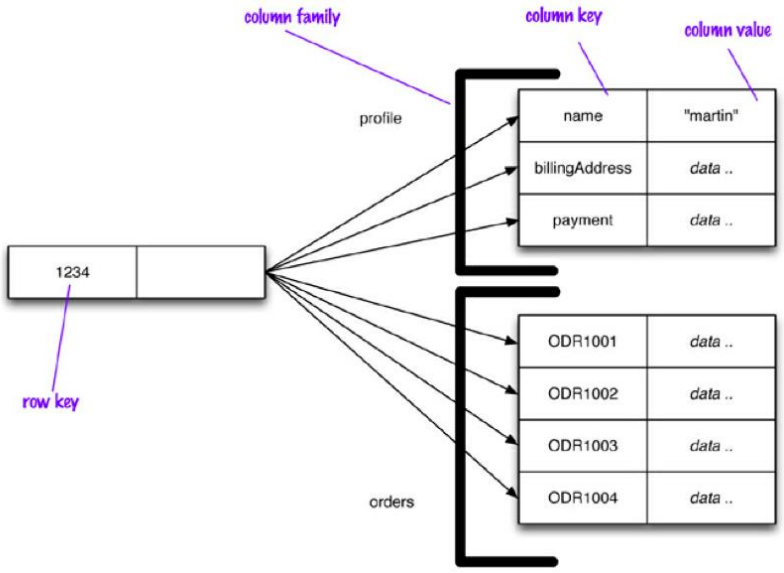
I database orientati agli aggregate garantiscono manipolazioni atomiche solo all'interno dell'aggregate.

Database a colonne-famiglia:

Modello influenzato da BigTable di Google, con struttura a colonne sparse e nessuno schema fisso.

Permette la memorizzazione di gruppi di colonne come unità di base, migliorando le prestazioni di lettura.

Esempio di database a colonne-famiglia.

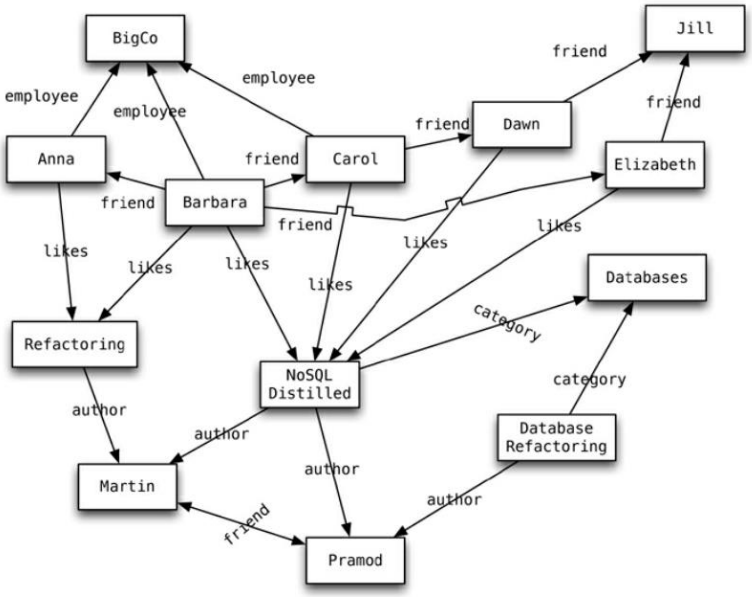


Database a grafi:

Struttura composta da nodi collegati da archi, ideale per catturare dati con relazioni complesse.

Permettono query efficienti sui dati altamente connessi, spostando il lavoro di navigazione dalle query agli inserimenti.

Esempio di database a grafi.



Vantaggi e svantaggi di essere schemaless:

Vantaggi: maggiore libertà nella gestione dei cambiamenti durante lo sviluppo, senza necessità di modificare il database esistente.

Svantaggi: maggiore complessità nello sviluppo del software per analizzare i contenuti degli oggetti, con un'ipotetica struttura implicita nello schema.

Soluzioni agli svantaggi degli schemaless:

1. Incapsulare tutte le interazioni del database all'interno di un'unica applicazione.

2. Delineare chiaramente le diverse aree di un aggregate per l'accesso da parte di diverse applicazioni.

Introduzione ai database NoSQL e modellazione

Vantaggi dei database schemaless:

Modello relazionale:

- Con i database relazionali, bisogna concordare lo schema prima di poter memorizzare i dati.

Modello NoSQL:

- Con i modelli aggregati (documenti, chiavi-valore e colonne-famiglia), si può memorizzare qualsiasi dato associato a una chiave o a una colonna aggiuntiva.

- Nei database a grafo, si ha la libertà di aggiungere nuovi collegamenti e nuove proprietà ai nodi e agli archi.

Vantaggi:

- Maggiore libertà nella gestione dei cambiamenti durante lo sviluppo.

- Possibilità di aggiungere nuove funzionalità durante il progetto e di smettere di memorizzare alcune caratteristiche senza perdere i dati già memorizzati.

Svantaggi dei database schemaless:

- La flessibilità può aumentare la complessità nello sviluppo del software, ad esempio per l'analisi dei contenuti degli oggetti.

- Un database schemaless assume uno schema implicito, ovvero che certi nomi di campo siano presenti e abbiano un significato specifico, spostando lo schema nel codice dell'applicazione.

Esempio di svantaggi dei database schemaless.

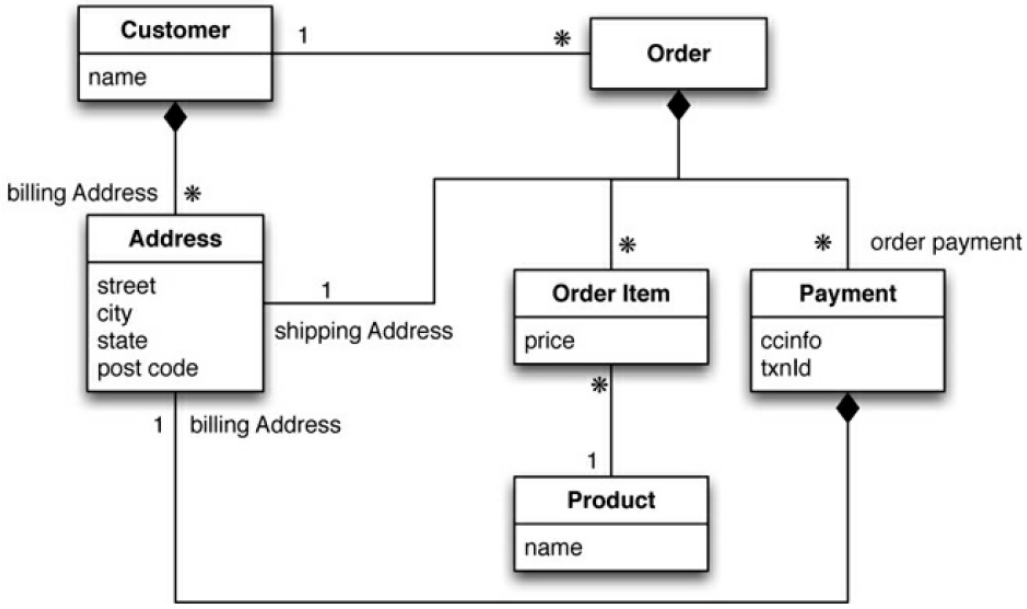


Figure 2.3. An aggregate data model

Esempio di svantaggi dei database schemaless.

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
```

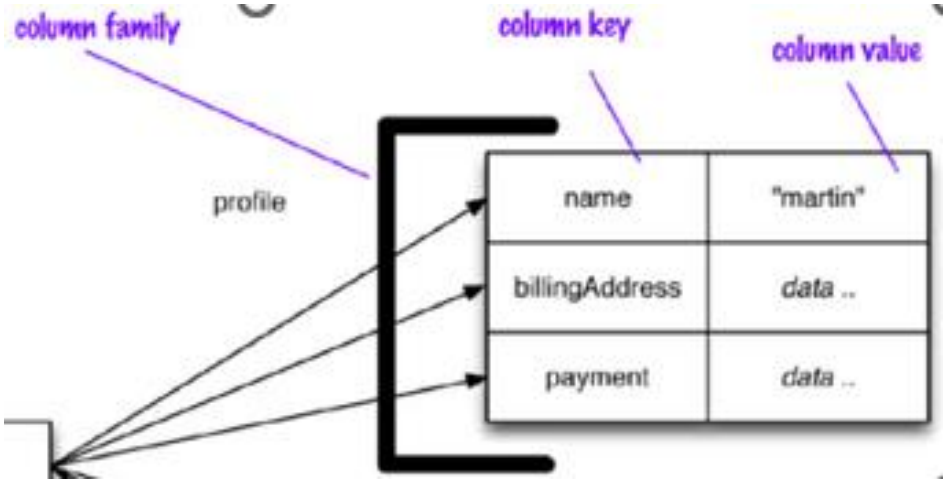
Esempio di svantaggi dei database schemaless.

Soluzioni ai svantaggi dei database schemaless:

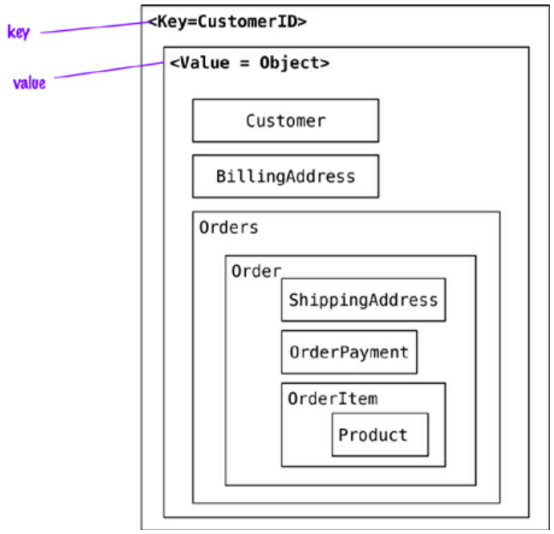
1. Incapsulare tutte le interazioni del database all'interno di un'unica applicazione e integrarla con altre applicazioni utilizzando servizi web.

2. Delineare chiaramente le diverse aree di un aggregate per l'accesso da parte di diverse applicazioni. Tuttavia, cambiare i confini di un aggregate è complesso quanto cambiare schema nei database relazionali.

Soluzioni ai problemi dei database schemaless.



Soluzioni ai problemi dei database schemaless.



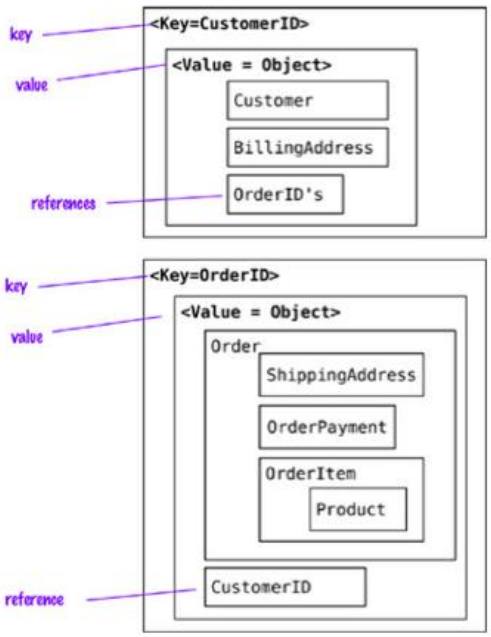
Materialized Views (viste materializzate):

Definizione:

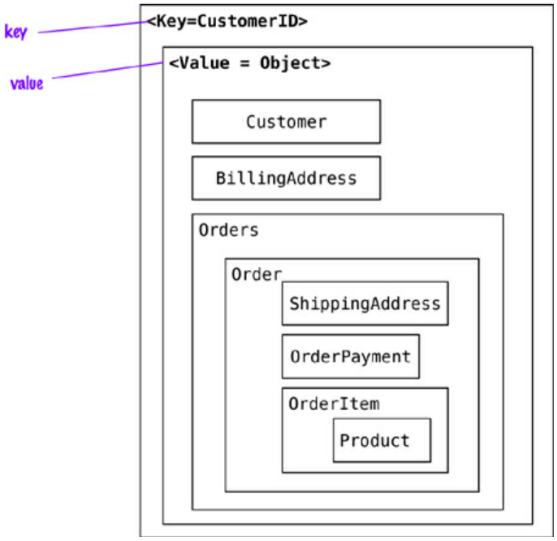
- Una vista è come una tabella relazionale, definita tramite calcoli sulle tabelle di base.

- Le viste materializzate sono calcolate in anticipo e memorizzate su disco, utili per dati letti frequentemente ma che possono rimanere relativamente stabili.

Esempio di vista materializzata.



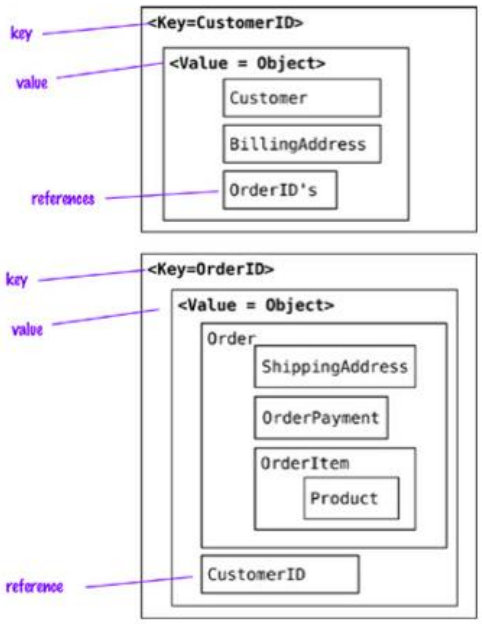
Esempio di vista materializzata.



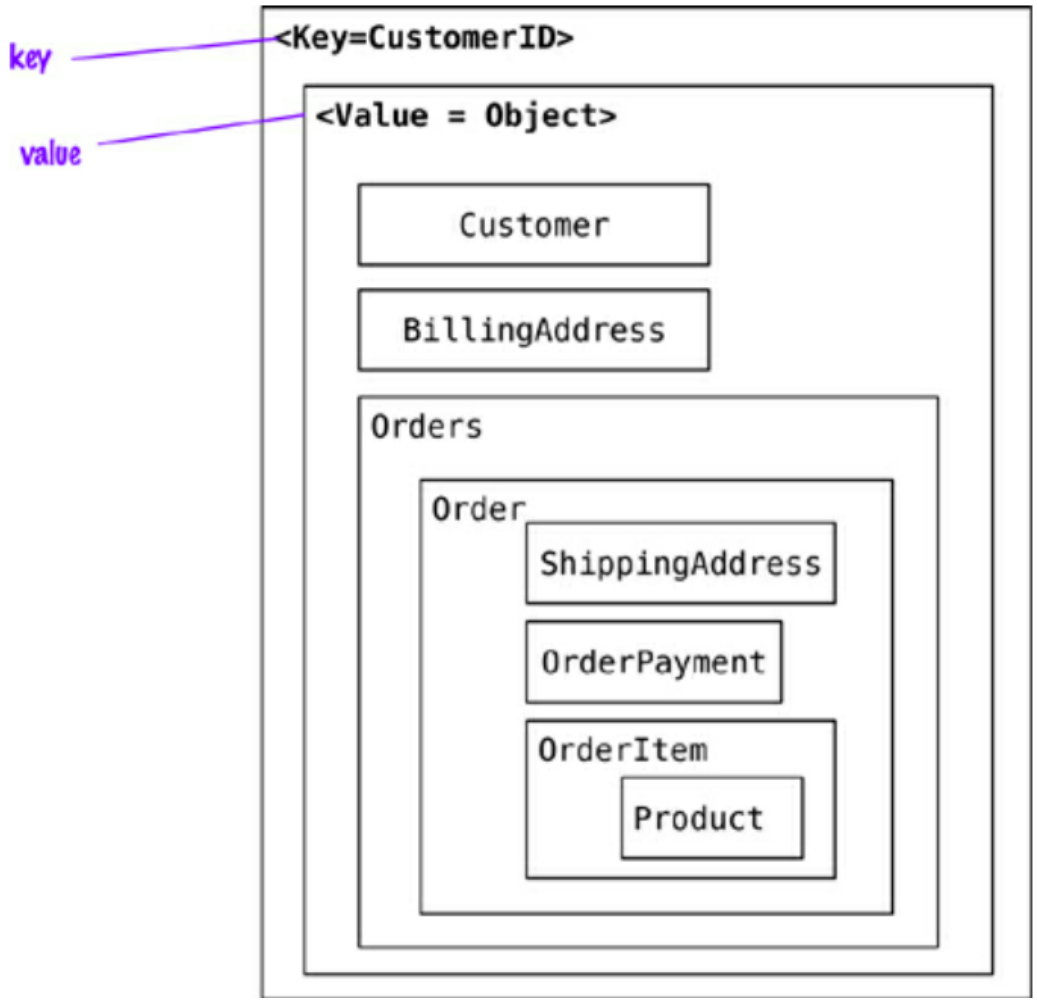
Esempio:

- Per sapere quanto ha venduto un particolare articolo nelle ultime settimane, possiamo eseguire la query una volta e memorizzarne il risultato.

Esempio di query di vendita.



Esempio di query di vendita.



Strategie per mantenere le viste materializzate:

1. Approccio eager: aggiornare la vista materializzata contemporaneamente all'aggiornamento dei dati di base.

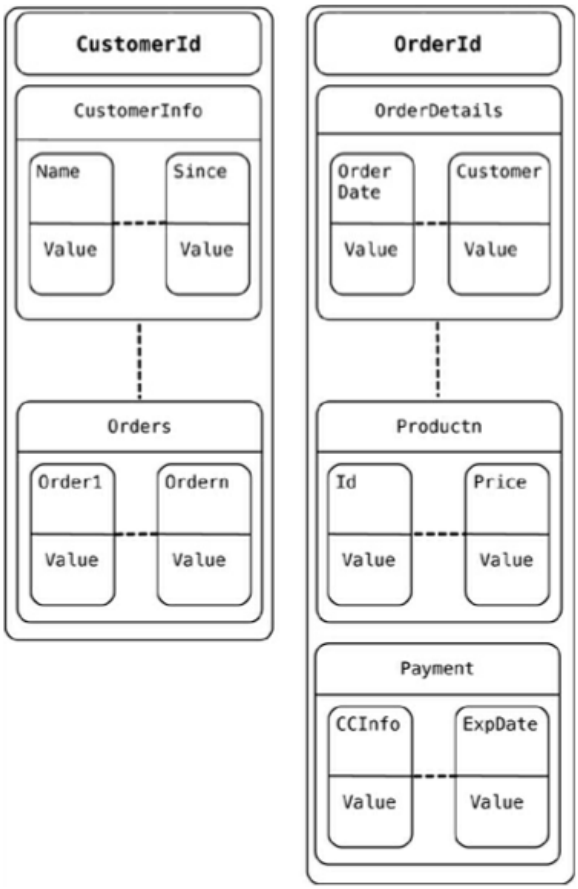
2. Approccio lazy: eseguire job batch per aggiornare le viste materializzate a intervalli regolari.

- Le viste materializzate possono essere utilizzate all'interno dello stesso aggregate, permettendo l'aggiornamento nella stessa operazione atomica.

Strategie di aggiornamento delle viste materializzate.

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Strategie di aggiornamento delle viste materializzate.

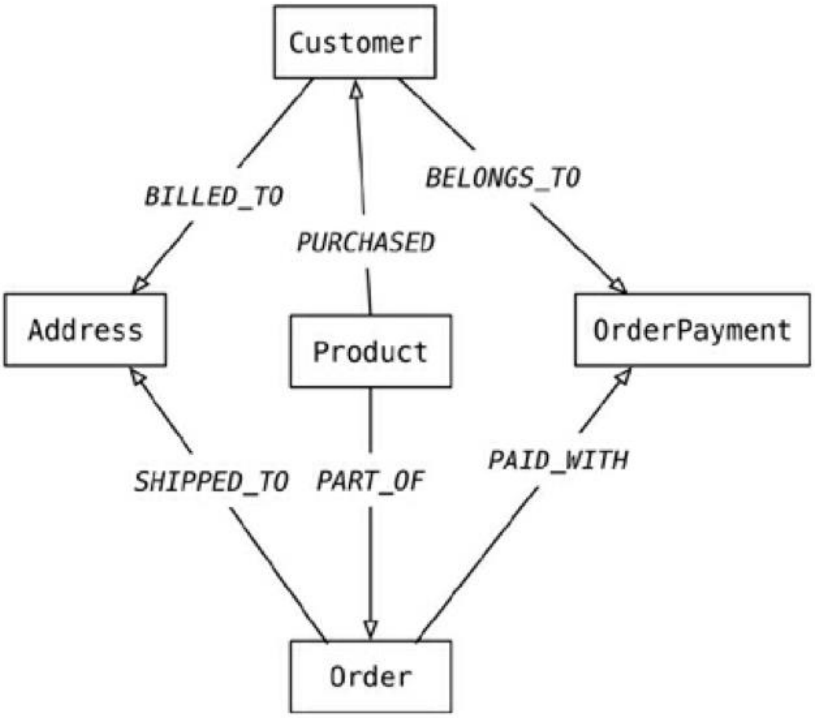


Modellazione per l'accesso ai dati:

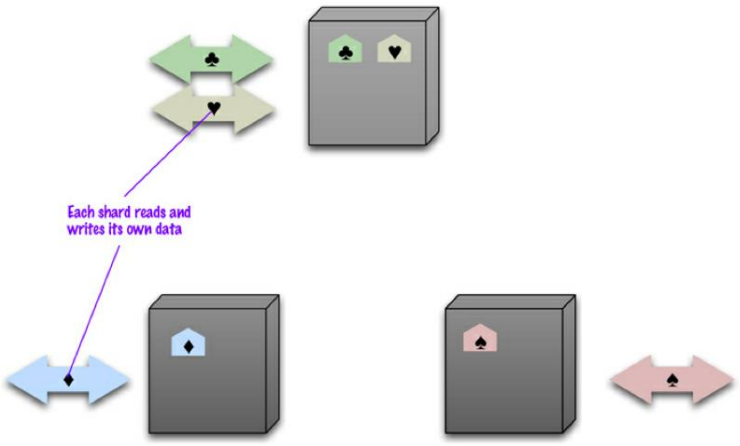
- Quando si modella un aggregate, bisogna considerare come i dati verranno letti.

Esempio: calcolare il numero di ordini per prodotto richiede la lettura di tutti gli ordini. Si può migliorare tenendo riferimenti agli oggetti e utilizzando store a documenti.

Esempio di modellazione per l'accesso ai dati.



Esempio di modellazione per l'accesso ai dati.



Esempio di modellazione per l'accesso ai dati.

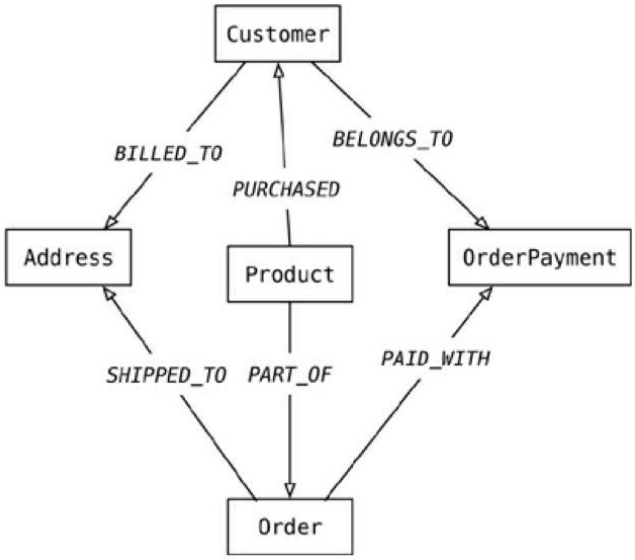
Esempio di modellazione per l'accesso ai dati.

Esempio di modellazione per l'accesso ai dati.

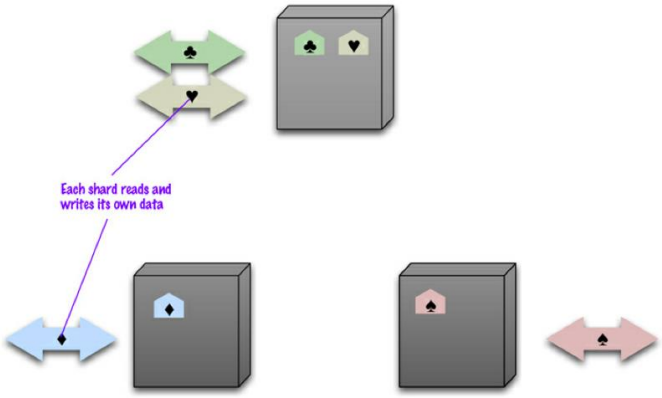
Analisi in tempo reale:

- Gli aggregates possono essere utilizzati per ottenere analisi in tempo reale, riempiendo le informazioni sugli ordini contenenti un determinato prodotto.

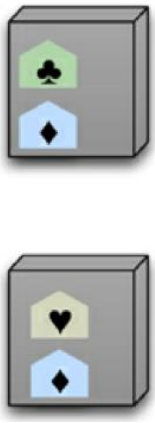
Esempio di analisi in tempo reale.



Esempio di analisi in tempo reale.



Esempio di analisi in tempo reale.



Esempio di analisi in tempo reale.

Esempio di analisi in tempo reale.

Modelli di distribuzione:

- Replicazione e sharding: tecniche ortogonali per distribuire i dati.

- Sharding: divisione dei dati su diversi server, migliorando le prestazioni di lettura e scrittura.

- Replicazione master-slave: replica dei dati su più nodi, con uno come master che gestisce le scritture e altri come slave che gestiscono le letture.

Sharding e Replica: Conflitti nei Database NoSQL

Modelli di distribuzione:

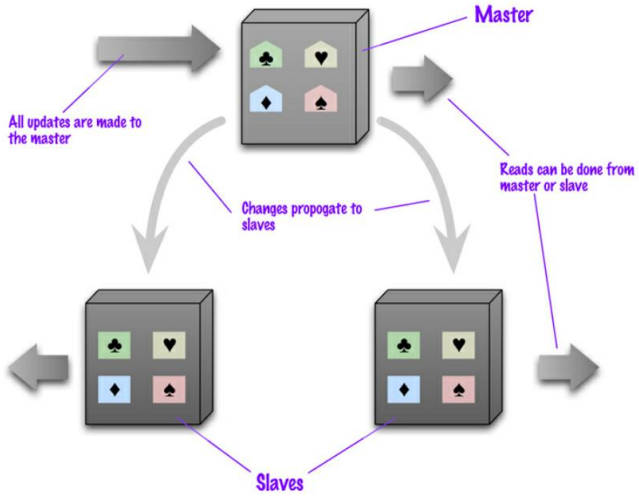
Principali vantaggi del NoSQL:

- Capacità di eseguire database su un grande cluster.
- Gestire grandi quantità di dati.
- Processare un maggiore traffico di lettura o scrittura.
- Gestire rallentamenti o interruzioni di rete.

Distribuzione dei dati:

- Due concetti principali: replicazione e sharding.
- Replicazione: Copia degli stessi dati su più nodi.
- Sharding: Divisione dei dati su nodi diversi.

Diagramma che mostra i modelli di distribuzione.



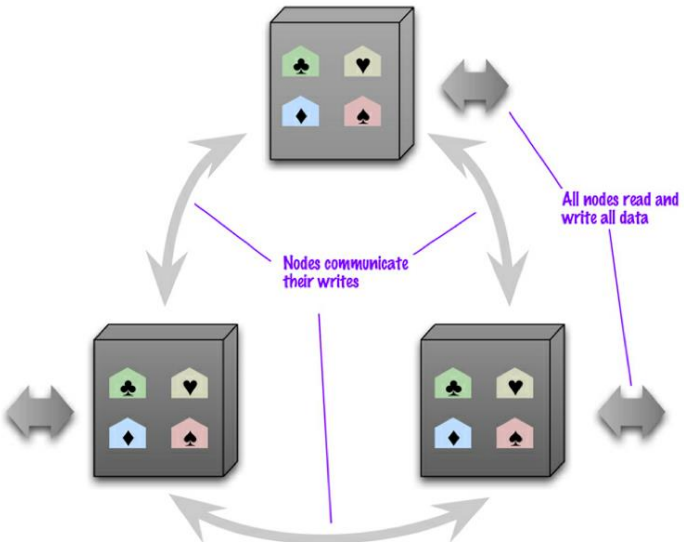
Combinazioni di tecniche di distribuzione:

Replicazione e sharding sono tecniche ortogonali:

- Replicazione:
  - Master-slave: un nodo master gestisce le scritture, i nodi slave le letture.
  - Peer-to-peer: tutti i nodi sono uguali, accettano scritture e letture.

- Sharding: Suddivisione orizzontale dei dati tra più nodi.

Diagramma che mostra la combinazione di tecniche di distribuzione.



Distribuzione su singolo server:

Nessuna distribuzione, configurazione semplice.

Un unico server gestisce tutte le operazioni di lettura e scrittura.

Funziona bene con database a grafo, chiavi-valore e documenti.

Caratteristiche, vantaggi e rischi dello sharding:

Servizi di database con auto-sharding:

- Assegnazione automatica dei dati agli shard e reindirizzamento delle query.
- Migliora le prestazioni di lettura e scrittura.
- Non migliora la resilienza da solo (da combinare con la replicazione).

Replicazione master-slave:

Un nodo master gestisce le scritture, i nodi slave gestiscono le letture.

Vantaggi:

- Resilienza aumentata, il master può essere sostituito da uno slave in caso di guasto.
- I nodi slave possono gestire le letture durante il fallimento del master.

Svantaggi:

- Incoerenza tra i nodi slave, letture non aggiornate.

Replicazione peer-to-peer:

Tutti i nodi sono uguali, accettano scritture e letture.

Vantaggi:

- Gestione dei guasti senza perdere l'accesso ai dati.

- Aggiunta di nodi per migliorare le prestazioni.

Svantaggi:

- Consistenza complicata, rischio di conflitti di scrittura-scrittura.

Combattere i conflitti di scrittura-scrittura:

Approccio pessimista:

- Coordinazione delle repliche per evitare conflitti.

- Richiede il consenso della maggioranza delle repliche.

Approccio ottimista:

- Gestione dei conflitti con politiche di merge.

- Maggiori benefici di prestazioni, ma rischio di scritture incoerenti.

Combinazione di sharding e replicazione:

Utilizzo di master-slave e sharding, con più master per ogni dato.

Uso di peer-to-peer e sharding nei database a colonne-famiglia:

- Fattore di replicazione di 3, ogni shard presente su tre nodi.

- Gestione dei guasti ricostruendo gli shard sui nodi rimanenti.

Varie forme di consistenza:

Consistenza forte: Evita ogni tipo di inconsistenza.

Consistenza eventuale: Le repliche si aggiornano col tempo.

Consistenza lettura-scrittura: Letture coerenti con le scritture effettuate durante una sessione.

Consistenza di sessione: Garanzia di coerenza all'interno di una sessione utente.

Il teorema CAP:

Tre proprietà: Consistenza dei dati, Disponibilità e Tolleranza alle Partizioni.

È possibile ottenere solo due delle tre proprietà contemporaneamente.

Consistenza eventuale: Compromesso tra disponibilità e consistenza.

Sharding e Replica: Conflitti, BASE e MapReduce nei Database NoSQL

Modelli NoSQL:

Categorie principali:

- Chiavi-valore

- Documenti

- Colonne-famiglia

- Grafi

Motivi per utilizzare NoSQL -> distribuzione su cloud:

- Replica: copia dei dati su più server.

- Sharding: distribuzione di dati diversi su più server.

Replica master-slave:

Replica master-slave:

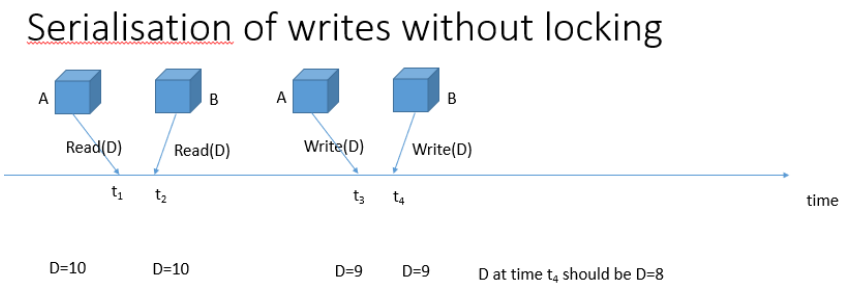
- Un nodo master gestisce le scritture, i nodi slave sincronizzano con il master e gestiscono le letture.

Pro e contro:

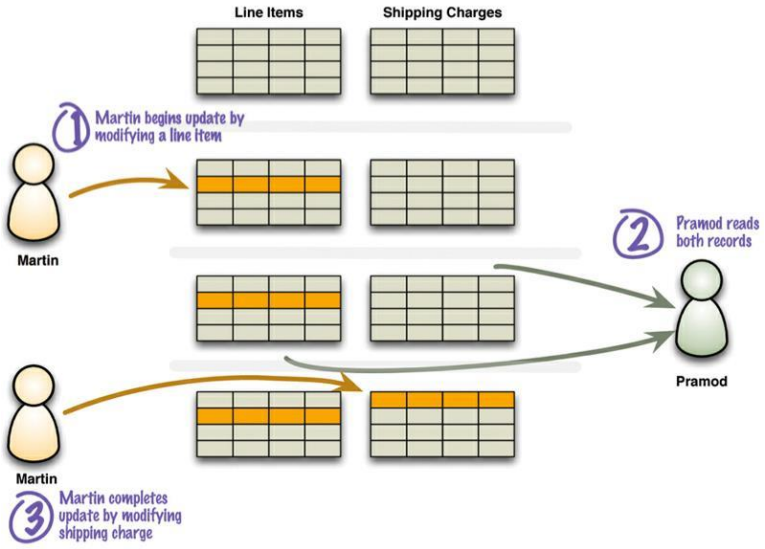
- Pro: scalabilità delle letture, resilienza contro il fallimento di uno slave.

- Contro: non aiuta con la scalabilità delle scritture, il master è un punto di fallimento singolo.

Esempio di replica master-slave.



Esempio di replica master-slave.



Replica peer-to-peer:

Replica peer-to-peer:

- Tutti i nodi accettano scritture, la perdita di un nodo non impedisce l'accesso ai dati.

Pro e contro:

- Pro: facilità di gestione dei guasti, scalabilità.

- Contro: complicazioni di consistenza, rischio di conflitti di scrittura-scrittura.

Combinazione di Sharding e Replica:

Uso combinato di master-slave e sharding:

- Più master, ma ogni dato ha un solo master.

- Configurazione flessibile: nodi master per alcuni dati e slave per altri.

Vari tipi di consistenza:

Consistenza forte: evita ogni tipo di inconsistenza.

Consistenza eventuale: le repliche si aggiornano col tempo.

Consistenza lettura-scrittura: garantisce risposte consistenti alle richieste dei lettori.

Approcci pessimistici o ottimistici:

Approccio pessimistico:

- Prevenzione dei conflitti tramite blocchi di scrittura.

Approccio ottimistico:

- Gestione dei conflitti tramite politiche di merge.

Tradeoff tra sicurezza e vivacità:

Programmazione concorrente: tradeoff tra sicurezza e reattività.

Replica: aumenta la probabilità di conflitti di scrittura-scrittura.

Consistenza nella replica:

Consistenza della replica: garantisce che lo stesso dato abbia lo stesso valore su repliche diverse.

Consistenza eventuale: le repliche si aggiornano alla stessa versione col tempo.

Quorum:

Quorum di scrittura: numero di nodi necessario per confermare una scrittura.

Quorum di lettura: numero di nodi necessario per garantire la versione più aggiornata dei dati.

MapReduce e Cluster di server:

MapReduce: framework algoritmico per l'esecuzione di job in parallelo su più nodi di un cluster.

Funzioni di MapReduce:

- Map: mappa gli oggetti secondo una chiave e li raggruppa.

- Reduce: riduce gli oggetti raggruppati in un singolo valore.

MapReduce nei Database NoSQL

Teorema CAP e consistenza rilassata:

Il Teorema CAP afferma che in un sistema distribuito, è possibile garantire solo due dei tre seguenti attributi allo stesso tempo: Consistenza dei dati, Disponibilità e Tolleranza alle Partizioni.

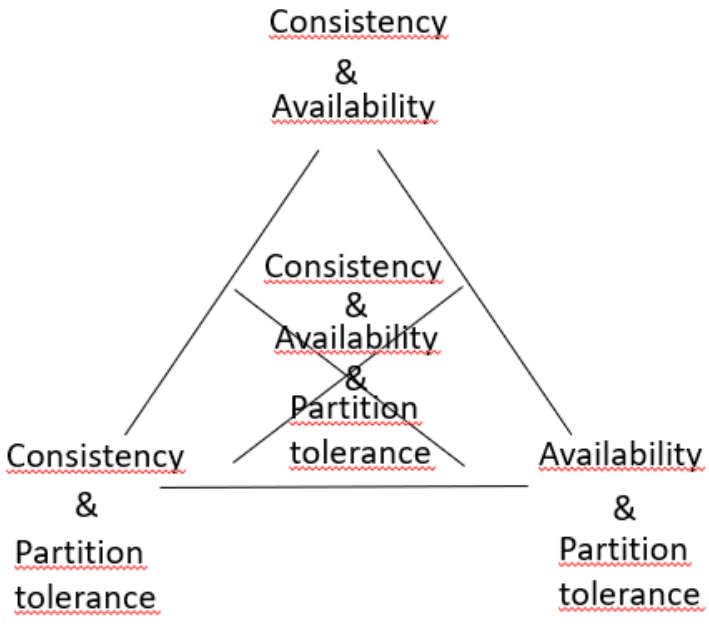
La Consistenza dei dati assicura che tutte le letture da un database distribuito vedano gli ultimi dati scritti.

La Disponibilità garantisce che ogni richiesta di lettura o scrittura riceva una risposta, anche se alcuni nodi del sistema sono offline.

La Tolleranza alle Partizioni significa che il sistema continua a funzionare anche se si verifica una divisione della rete che separa i nodi in più gruppi incapaci di comunicare tra loro.

Esempio: in un sistema di database distribuito, se si verifica una partizione di rete, il sistema deve scegliere tra fornire risposte coerenti o essere disponibile per tutte le richieste.

Immagine che illustra il Teorema CAP e il trade-off tra consistenza, disponibilità e tolleranza alle partizioni.



Tradeoff tra Disponibilità e Latenza:

Nei database NoSQL, invece delle proprietà ACID (Atomicità, Consistenza, Isolamento e Durabilità), spesso si seguono le proprietà BASE (Basicamente Disponibile, Stato Soft, Consistenza Eventuale).

La Consistenza Eventuale significa che, dopo un intervallo di tempo sufficientemente lungo con comunicazione disponibile, tutti i nodi del sistema avranno una visione coerente dei dati.

Esiste un trade-off tra consistenza e latenza: per migliorare la consistenza, è necessario coinvolgere più nodi nell'interazione, aumentando la latenza della risposta.

La disponibilità può essere vista come il limite della latenza che siamo disposti a tollerare: quando la latenza diventa troppo alta, consideriamo i dati come non disponibili.

Durabilità nella replica:

La replica dei dati su più nodi aumenta la durabilità, ma introduce complessità. La durabilità nella replica può fallire quando un nodo elabora un aggiornamento ma fallisce prima che l'aggiornamento sia replicato sugli altri nodi.

Nel modello master-slave, il master gestisce le scritture e le repliche slave gestiscono le letture. Se il master fallisce, le scritture non replicate alle slave andranno perse.

Una soluzione è fare in modo che il master attenda che alcune repliche confermino l'aggiornamento prima di riconoscerlo al client.

Quorum:

Il quorum è un meccanismo per garantire la consistenza in un sistema distribuito. Il quorum di scrittura (W) è il numero di nodi necessario per confermare una scrittura, mentre il quorum di lettura (R) è il numero di nodi necessario per garantire una lettura consistente.

La regola è che W + R deve essere maggiore del numero totale di repliche (N) per garantire la consistenza dei dati.

Ad esempio, se N = 3, per avere una lettura consistente possiamo avere W = 2 e R = 2, così R + W > N.

MapReduce e Cluster di server:

MapReduce è un framework algoritmico per eseguire lavori in parallelo su più nodi di un cluster. Proposto da Google, divide un compito in componenti più piccoli che vengono eseguiti in parallelo sui nodi del cluster.

La funzione Map suddivide i dati in base a una chiave, mentre la funzione Reduce aggrega i risultati della funzione Map per ottenere un risultato finale.

Esempio: per contare il numero di documenti che contengono una certa parola, la funzione Map può contare le occorrenze nei singoli documenti, mentre la funzione Reduce somma i conteggi parziali.

Paradigma di programmazione funzionale:

La programmazione funzionale è un paradigma che tratta l'esecuzione del programma come una serie di valutazioni di funzioni matematiche. Diversamente dalla programmazione imperativa che usa variabili intermedie per rappresentare cambiamenti di stato, la programmazione funzionale garantisce che il risultato di una funzione sia sempre lo stesso per gli stessi parametri.

Questo approccio riduce gli effetti collaterali e rende più facile verificare la correttezza e ottimizzare le funzioni.

Esempio: una funzione matematica che calcola la somma di una lista di numeri restituirà sempre lo stesso risultato indipendentemente da dove e quando viene eseguita.

Cambiamento di paradigma:

Il paradigma classico di elaborazione prevede che il server del database invii i dati al server dell'applicazione per l'elaborazione. Nel paradigma MapReduce, la funzione di calcolo viene inviata ai server dei dati, riducendo la quantità di dati trasferiti sulla rete e mantenendo il più possibile l'elaborazione e i dati sulla stessa macchina.

Questo approccio è particolarmente utile per l'elaborazione di grandi volumi di dati distribuiti su più server.

Implementazioni di MapReduce:

MapReduce è implementato in molti database NoSQL, spesso come parte del progetto Hadoop. Tuttavia, ci sono differenze nei dettagli tra le implementazioni.

Esempi di implementazioni includono Apache Pig, che facilita la scrittura di programmi MapReduce, e Hive, che offre una sintassi SQL-like per definire i programmi MapReduce.

MapReduce incrementale:

Molte computazioni MapReduce richiedono molto tempo, anche con hardware clusterizzato. Gli aggiornamenti incrementali permettono di aggiornare solo i dati che cambiano senza rieseguire l'intera computazione.

Esempio: se i dati di input cambiano, solo le parti della computazione MapReduce che dipendono da quei dati devono essere rieseguite.

Persistenza poliglotta:

La persistenza poliglotta si riferisce all'uso di diverse tecnologie di gestione e archiviazione dei dati in circostanze diverse. Questo approccio consente l'adozione di più modelli di dati e linguaggi di query anche in una singola applicazione.

Esempio: un'applicazione può utilizzare un database relazionale per le transazioni finanziarie e un database NoSQL per la gestione delle sessioni degli utenti.

Key-Value Stores nei Database NoSQL

Introduzione ai database NoSQL:

Bisogni potenziali per NoSQL:

Per capire quale modello di dati è corretto (e il database corrispondente) dobbiamo elencare i bisogni dell'applicazione e le risorse disponibili:

Query flessibili su attributi arbitrari.

Indicizzazione per un recupero rapido dei dati.

Query ad hoc vs. query pianificate.

Schema rigido vs. schema negoziabile.

Partizionamento dei dati; repliche (copie); distribuzione dei dati con hashing.

Ottimizzazione del database per operazioni di lettura/scrittura.

Scalabilità - orizzontale (più server), verticale (potenza di calcolo aggiuntiva ai server esistenti) o mista.

Key-Value Store:

Un Key-Value Store è una semplice tabella hash:

Utilizzata principalmente quando tutto l'accesso al database avviene tramite chiave primaria.

L'applicazione può fornire un ID e un VALORE e memorizzare la coppia.

Se l'ID esiste già, il valore attuale viene sovrascritto.

Osservazioni:

La consistenza è applicabile solo per operazioni su una singola chiave.

Le scritture ottimistiche possono essere eseguite, ma sono molto costose da implementare.

Nei Key-Value Store distribuiti come Riak, viene implementato il modello di consistenza eventuale.

Riak risolve i conflitti di aggiornamento in due modi:

La scrittura più recente vince, e le scritture più vecchie vengono perse.

Vengono restituiti tutti i valori, permettendo al client di risolvere il conflitto.

Tutti i Key-Value Store possono eseguire query tramite chiave. Per eseguire query tramite attributo del valore, non è possibile utilizzare il database.

Alcuni Key-Value Store offrono la possibilità di cercare all'interno del valore, come Riak Search che consente di eseguire query sui dati come con gli indici Lucene.

Molti Key-Value Store scalano utilizzando lo sharding.

Diagramma che illustra il funzionamento di un Key-Value Store.

Riak – un Key-Value Store:

Il valore può essere qualsiasi cosa – testo, JSON, XML, immagini, clip video, ecc.

È tollerante ai guasti dei server (nodi del cluster).

Le query vengono effettuate sul web, tramite URL, intestazioni, ecc.

Le risposte sono fornite tramite interfaccia REST HTTP.

Riak fornisce un'interfaccia REST HTTP, quindi interagiamo con esso tramite lo strumento URL cURL.

In produzione, si utilizza quasi sempre un driver nel proprio linguaggio di programmazione preferito.

Manca il supporto per query ad hoc.

Riak parla web meglio di qualsiasi altro database: è adatto a supportare data center che necessitano di bassa latenza nelle risposte e gestisce la crescita del traffico verso le pagine web.

È difficile eseguire collegamenti logici tra i valori (non ha chiavi esterne).

Esempio di Riak come Key-Value Store.

Querying a cluster node:

Il cluster dei server/nodi è disposto in un anello con tutti i nodi che sono peer.

Tutti i server sono partecipanti uguali nell'anello.

Qualsiasi nodo può essere interrogato tramite REST (REpresentational State Transfer) utilizzando i comandi:

POST (crea),

GET (legge),

PUT (aggiorna),

DELETE (cancella).

L'URL per interrogare il database ha il formato:

http://SERVER:PORT/riak/BUCKET/KEY

Riak divide le classi di chiavi da memorizzare nel database in bucket per evitare collisioni tra chiavi.

Aggiunta di una chiave al database:

Esempio di creazione di un database come dizionario delle parole trovate in un corpus (collezione) di documenti:

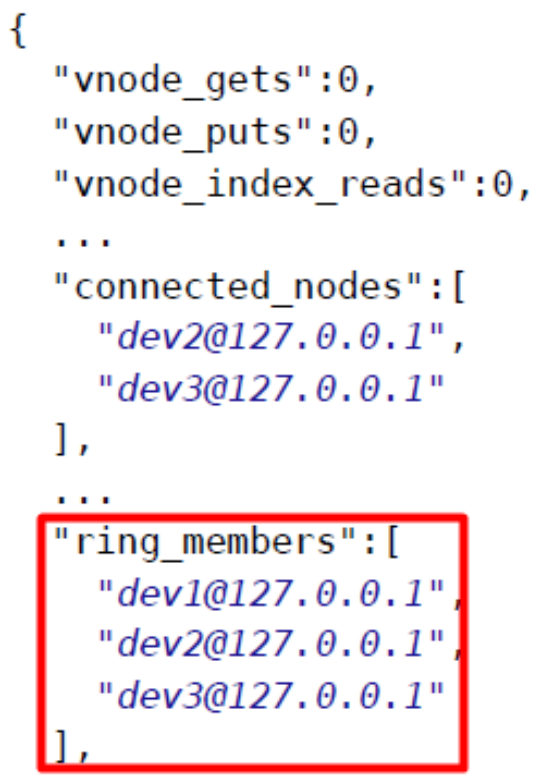
Introduciamo il nome della classe delle chiavi (parole).

Supponiamo di voler memorizzare ogni parola come chiave, associata ai seguenti valori: categoria della parola (nome, verbo, aggettivo, avverbio, pronome, preposizione, congiunzione, articolo, ecc.) e frequenza (conteggio) della parola nel corpus.

Richiediamo un'operazione di PUT:

PUT http://localhost:8091/riak/words/mamma -H "Content-Type: application/json" -d '{"category": "noun", "count": 1}'

Immagine che mostra l'aggiunta di una chiave al database.



Visualizzare i bucket e interagire con essi:

Possiamo visualizzare l'elenco dei bucket creati.

GET http://localhost:8091/riak?buckets=true

Possiamo restituire i risultati impostando il parametro returnbody=true.

PUT http://localhost:8091/riak/animals/polly?returnbody=true -H "Content-Type: application/json" -d '{"nickname": "Sweet Polly Purebred", "breed": "Purebred"}'

Se abbiamo dimenticato alcune delle nostre chiavi in un bucket, possiamo ottenerle tutte con keys=true.

http://localhost:8091/riak/animals?keys=true

Possiamo anche ottenerle come stream con keys=stream, che può essere una scelta più sicura per set di dati enormi.

Riak e HBase nei Database NoSQL

Riak:

Valore nei Key-Value Store:

Il valore può essere qualsiasi cosa: testo, JSON, XML, immagini, clip video, ecc.

È tollerante ai guasti dei server (nodi del cluster).

Il cluster dei server/nodi è disposto in un anello con tutti i nodi che sono peer.

Riak divide le classi di chiavi da memorizzare nel database in bucket per evitare collisioni tra chiavi.

Le query vengono effettuate sul web, tramite URL, intestazioni, ecc. Le risposte sono fornite tramite interfaccia REST HTTP.

Immagine che illustra il funzionamento di Riak.

Link nei Key-Value Store:

I link sono metadati che associano una chiave ad altre chiavi:

Il link è unidirezionale.

È possibile aggiungere più link.

La struttura per definirli è: Link: </riak/bucket/key>; riaktag="tag name"

Esempio di utilizzo dei link:

Bucket=documents, Key=1:

Contenuto del documento: "mamma told me to go home"

Bucket=words, Key=mamma:

Categoria: nome, conteggio: 2

Contiene:

Bucket=documents, Key=2: Contenuto del documento successivo.

Query sui link:

Richiesta di dati collegati al documento n. 1:

GET http://localhost:8091/riak/documents/1/_,_,_

Risposta: bucket con le coppie chiave-valore indicate dai link con i tag "contains" e "successive".

Applicazione di MapReduce in Riak:

Esempio di mappatura:

POST http://localhost:8091/mapred --data @-

{"inputs":[["document","1"],["document","2"],["document","3"]],

"query":[{"map":{"language":"javascript", "source":"function(v) {

var parsed_data = JSON.parse(v.values[0].data);

var data = {};

data[parsed_data.key] = parsed_data["document content"].find("mamma");

return [data];}}}]}

HBase: un database orientato alle colonne:

HBase è stato progettato per:

Grandi quantità di dati.

Essere scalabile.

Gestire molti nodi (non meno di cinque).

Fornisce forti garanzie di consistenza dei dati.

Le tabelle non sono come le relazioni nei database relazionali, le righe non sono come i record relazionali.

Le colonne sono completamente variabili e non vincolate in schema.

Versioning, compressione, garbage collection, e tabelle in-memory.

Introduzione a HBase:

Basato sul concetto di BigTable, inizialmente creato per NLP (Natural Language Processing).

Nato come pacchetto in Apache Hadoop, è diventato autonomo.

È fault-tolerant: i guasti hardware sono comuni nei grandi cluster, ma HBase può recuperare rapidamente dai guasti dei singoli server.

Utilizzato da grandi aziende come Facebook, Twitter, eBay, Meetup, Ning, Yahoo!, StumbleUpon, ecc.

Concetti base per le operazioni CRUD:

La tabella in HBase è una grande mappa di mappe:

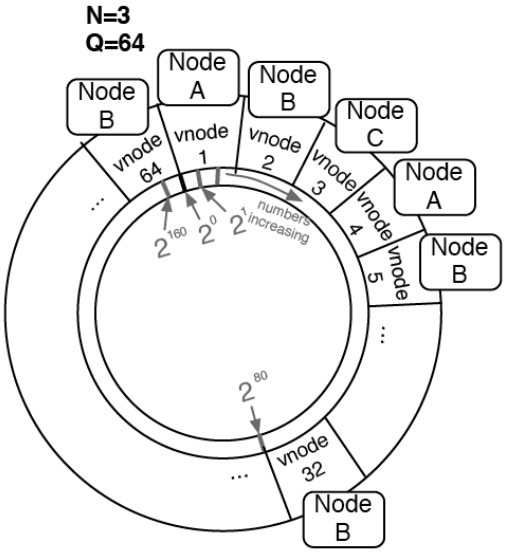
Una mappa è una coppia chiave-valore.

Una riga è una mappa in cui le chiavi sono chiamate colonne e i valori sono array di byte, non interpretati.

Le colonne sono raggruppate in famiglie di colonne.

Ogni colonna ha due parti: nome della famiglia e qualificatore della colonna nel formato family_name:column_qualifier.

Esempio di operazioni CRUD in HBase.



Rigie, chiavi, famiglie, colonne, valori:

Concetti base per le operazioni CRUD:

Una riga è una mappa in cui le chiavi sono chiamate colonne e i valori sono array di byte, non interpretati.

Le colonne sono raggruppate in famiglie di colonne.

Ogni colonna ha due parti: nome della famiglia e qualificatore della colonna nel formato family_name:column_qualifier.

Una tabella per le pagine wiki:

Creare una tabella wiki con la famiglia di colonne 'text'.

In HBase una colonna è specifica di una riga e verrà creata e riempita di valori ad ogni nuova riga.

PUT 'wiki', 'Home', 'text:', 'Welcome to the wiki!'

Esempio di tabella wiki in HBase.

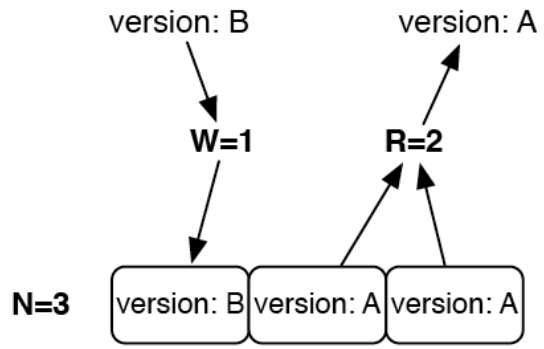


Figure 9—Eventual consistency: W+R <= N

Esempio di tabella wiki in HBase.

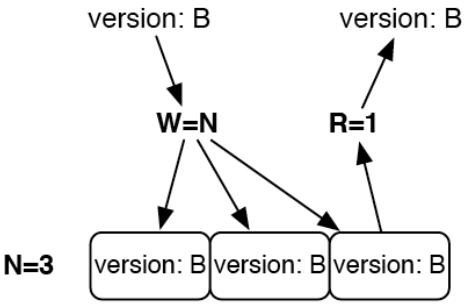


Figure 10—Consistency by writes: W=N, R=1

Approfondimento su Riak e HBase nei Database NoSQL

Riak:

Riak permette di controllare gli aspetti del teorema CAP (Consistenza, Disponibilità, Tolleranza alle Partizioni).

Un nodo che gestisce un'operazione di lettura o scrittura è il coordinatore per essa. Riak controlla le letture e le scritture nei nodi tramite tre valori:

- N: numero di nodi in cui deve replicare un'operazione di scrittura.

- W: numero di nodi su cui un'operazione di scrittura deve avere successo per confermare la scrittura.

- R: numero di nodi necessari per considerare completata con successo un'operazione di lettura.

Riak Vector Clock:

Riak è un semplice database key-value distribuito su un cluster di server. Quando si gestiscono più nodi, possono verificarsi conflitti di dati.

Gli orologi vettoriali vengono utilizzati per mantenere l'ordine degli aggiornamenti key-value in conflitto. Un orologio vettoriale è un token che i sistemi distribuiti come Riak usano per mantenere l'ordine degli aggiornamenti.

Esempio di vettore di versione:

Riak KV è un database key-value con consistenza eventuale che favorisce la disponibilità delle scritture. Consente a più client di scrivere contemporaneamente, potenzialmente sulla stessa chiave.

Riak KV utilizza orologi logici per tracciare la cronologia degli aggiornamenti ai valori e rilevare conflitti di scrittura. Ogni chiave in Riak ha il proprio vettore di versione, che è una struttura dati composta da coppie di attore e contatore.

Risoluzione dei conflitti:

Quando si verifica un conflitto, o concorrenza, deve essere risolto con una scrittura, che diventa un nuovo evento.

Il merge dei vettori di versione viene fornito al client come contesto. Riak memorizza il vettore di versione unito con il nuovo valore come tempo logico più recente.

Vector Clock Properties:

Gli orologi vettoriali permettono di rilevare conflitti e risolverli mantenendo traccia degli aggiornamenti effettuati da ciascun attore. Questa proprietà è fondamentale per garantire la consistenza eventuale in un sistema distribuito.

Esempio pratico:

Un esempio pratico di utilizzo degli orologi vettoriali per risolvere conflitti tra aggiornamenti concorrenti è il seguente: supponiamo di avere due attori A e B che aggiornano lo stesso valore contemporaneamente.

L'orologio vettoriale permette di rilevare che entrambi gli aggiornamenti sono avvenuti e di risolvere il conflitto unendo i valori aggiornati.

Redis:

Redis è un archivio key-value con strutture dati avanzate e la capacità di gestire insiemi di dati. È molto veloce sia nelle letture che nelle scritture.

Redis è stato definito un server di strutture dati. Supporta una coda di blocco e un meccanismo di publish-subscribe. Offre politiche di scadenza, livelli di durabilità e opzioni per la replica dei dati.

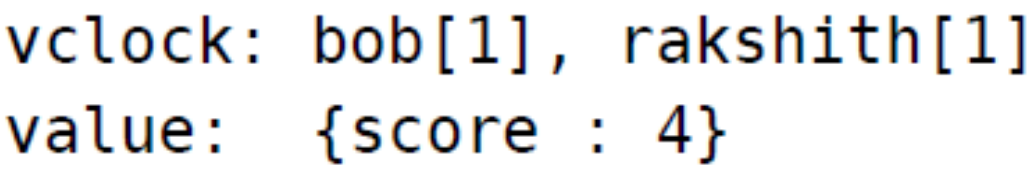
Introduzione a Redis.

```
vclock: bob[1]
value:  {score : 3}
```

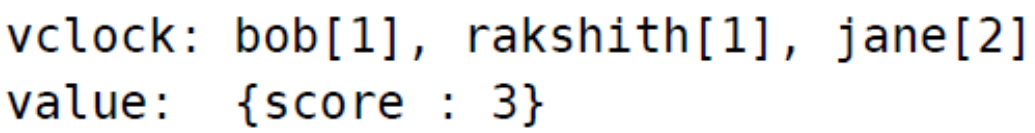
Introduzione a Redis.

```
vclock: bob[1], jane[1]
value:  {score : 2}
```

Introduzione a Redis.



Introduzione a Redis.



Riassunto di Redis:

Redis è un key-value store che offre un ricco set di operazioni su strutture dati. Fornisce opzioni di durabilità che permettono di scambiare velocità per sicurezza dei dati.

Offre replica master-slave per garantire la durabilità. Risiede nella memoria RAM, il che lo rende molto veloce ma può causare perdita di dati se il database si arresta prima di una snapshot.

HBase: un database orientato alle colonne:

HBase è stato progettato per grandi quantità di dati, essere scalabile e gestire molti nodi (non meno di cinque). Fornisce forti garanzie di consistenza dei dati.

Le tabelle in HBase non sono come le relazioni nei database relazionali, le righe non sono come i record relazionali. Le colonne sono completamente variabili e non vincolate in schema.

HBase ha caratteristiche come versioning, compressione, garbage collection e tabelle in-memory.

Concetti base per le operazioni CRUD:

La tabella in HBase è una grande mappa di mappe. Una riga è una mappa in cui le chiavi sono chiamate colonne e i valori sono array di byte, non interpretati.

Le colonne sono raggruppate in famiglie di colonne, ognuna con un nome di famiglia e un qualificatore di colonna nel formato family_name:column_qualifier. Questo permette di avere una struttura dati flessibile e scalabile.

MongoDB nei Database NoSQL

MongoDB: un database orientato ai documenti:

MongoDB è un database orientato ai documenti che consente ai dati di essere persistenti e nidificati. Le query sui dati nidificati avvengono in modo ad hoc.

Non vincola i dati a uno schema predefinito, quindi i documenti possono avere campi o tipi diversi.

È utilizzato da progetti importanti come Foursquare, bit.ly e CERN per memorizzare i dati degli esperimenti del Large Hadron Collider (LHC).

Oggetti in MongoDB in formato JSON:

Per creare un database, utilizziamo il comando `mongo book`, che crea un database chiamato 'book'. Si possono visualizzare gli altri database con `show dbs` e cambiare database con il comando `use`.

La creazione di una collezione di oggetti avviene con l'inserimento del primo oggetto, simile a un bucket nella nomenclatura di Riak. Esempio di inserimento di un oggetto:

```
javascript
    db.towns.insert({
      name: "New York",
      population: 22200000,
      last_census: ISODate('2009-07-31'),
      famous_for: ["statue of liberty", "food"],
      mayor: {
        name: "Michael Bloomberg",
        party: "I"
      }
    })
```

Per visualizzare le collezioni create, utilizziamo i comandi `show collection` e `system.indexes`.

Query in MongoDB utilizzando Javascript:

Per richiedere tutti gli oggetti in una collezione si utilizza `find()`, che consente ai processi client di essere indipendenti nella creazione degli oggetti rispetto agli altri processi distribuiti.

Esempi di query in MongoDB:

javascript
```
typeof db  // "object"
typeof towns  // "object"
typeof db.towns.insert  // "function"
```

Popolazione della collezione `towns` avviene con il comando `db.towns.insert({...})`.

Query per documenti:

Per richiedere un singolo oggetto si utilizza `find` con il parametro `_id`. È possibile specificare i campi da mostrare:

javascript
```
db.towns.find({_id: ObjectId("...")}, {name: 1, _id: 0})
```

Si possono cercare congiunzioni di criteri utilizzando espressioni regolari simili a Perl. Esempio:

javascript
```
db.towns.find({name: /^P/, population: {$lt: 10000}})
```

Gli operatori condizionali seguono il formato `{$op: value}`.

Ricerche avanzate:

Per cercare un intervallo di valori definito dall'utente, costruiamo un oggetto Javascript per la condizione della query. Esempio di ricerca di corrispondenze parziali:

javascript
```
db.towns.find({name: /partial_value/})
```

Per cercare un campo in un oggetto nidificato, utilizziamo `$elemMatch` e altri operatori.

Operazioni di aggiornamento:

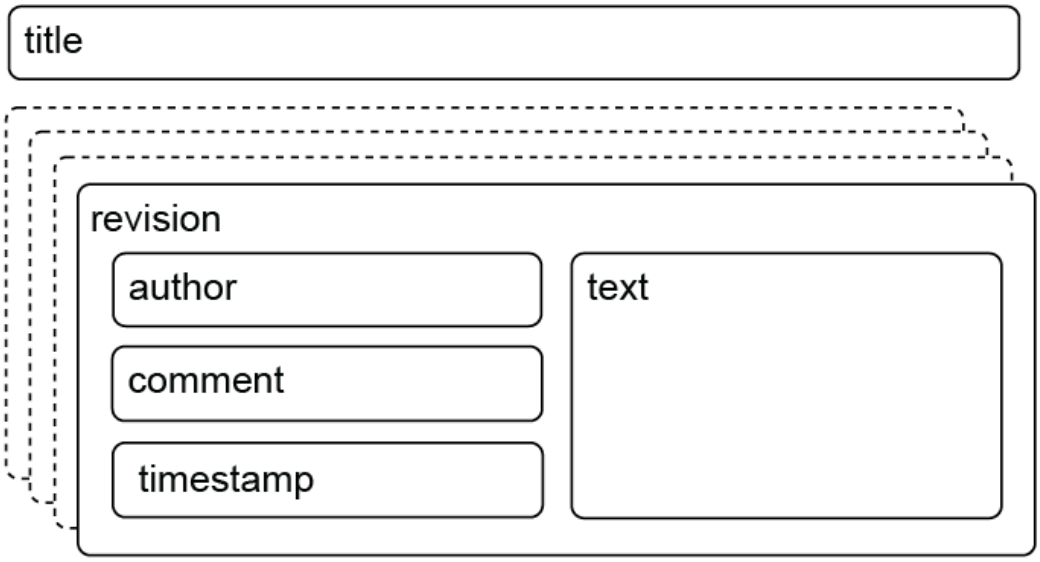
Le operazioni di aggiornamento richiedono due parametri: criteri di ricerca (come in `find()`) e un oggetto con i nuovi valori dei campi o un'operazione di modifica che utilizza l'operatore `$set`.

Esempio di aggiornamento di un oggetto esistente:

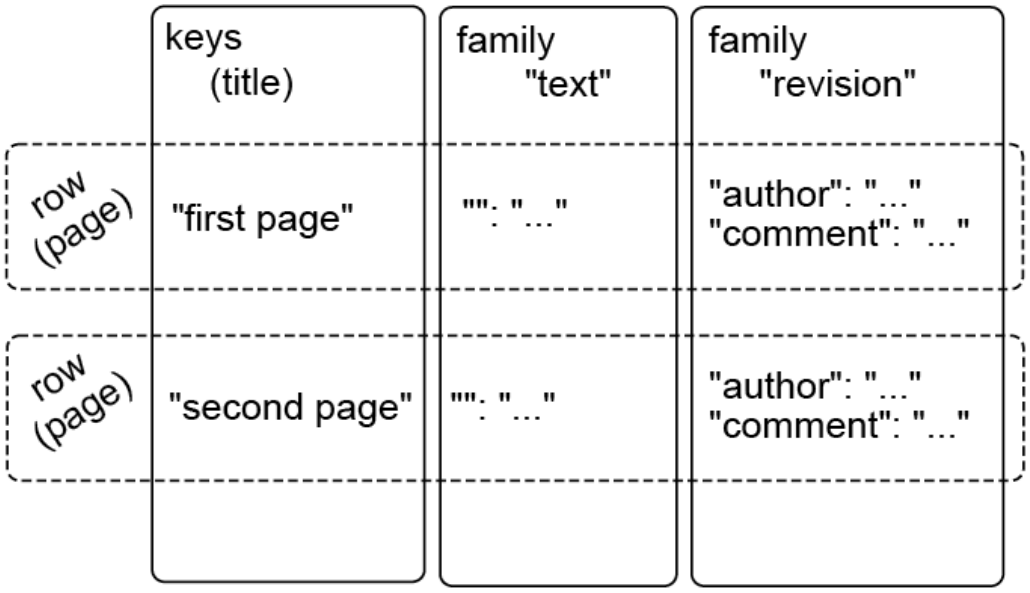
javascript
```
db.towns.update({_id: ObjectId("...")}, {$set: {population: 23000000}})
```

Esistono diversi operatori per la modifica dei documenti esistenti.

Esempi di operazioni di aggiornamento in MongoDB.



Esempi di operazioni di aggiornamento in MongoDB.



Join e riferimenti:

Le operazioni di join non sono efficienti in MongoDB poiché è un database distribuito per natura. È possibile aggiungere riferimenti agli oggetti nelle collezioni con la sintassi:

javascript
```
{$ref: "collection_name", $id: "reference_id"}
```

Indici in MongoDB:

Per impostazione predefinita, MongoDB crea un indice sul campo `_id` utilizzando un B-tree.
Per creare un indice su altri campi, utilizziamo:

javascript
```
db.collection.ensureIndex({property_name: 1}, {unique: true, partialFilterExpression: {property_name_2: condition}})
```

Funzioni aggregate:

Esempio di conteggio dei documenti con una parola specifica:

javascript
```
db.documents.count({'lemma': {$in: ['mamma']}})
```

Esempio di utilizzo delle funzioni di aggregazione in Javascript:

javascript
```
db.documents.group({
  initial: {count: 0},
  reduce: function(documents, output){output.count++;},
  cond: {'lemma': {$in: ['mamma']}},
  key: {'n_words': true}
})
```

Esempio di funzioni aggregate in MongoDB.

hbase/thrift_example.rb

```
$:.push('./gen-rb')
require 'thrift'
require 'hbase'

socket = Thrift::Socket.new( 'localhost', 9090 )
transport = Thrift::BufferedTransport.new( socket )
protocol = Thrift::BinaryProtocol.new( transport )
client = Apache::Hadoop::Hbase::Thrift::Hbase::Client.new( protocol )

transport.open()

client.getTableNames().sort.each do |table|
  puts "#{table}"
  client.getColumnDescriptors( table ).each do |col, desc|
    puts "  #{desc.name}"
    puts "    maxVersions: #{desc.maxVersions}"
    puts "    compression: #{desc.compression}"
    puts "    bloomFilterType: #{desc.bloomFilterType}"
  end
end

transport.close()
```

Esempio di funzioni aggregate in MongoDB.

```
$> ruby thrift_example.rb
links
  from:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
  to:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
wiki
  revision:
    maxVersions: 2147483647
    compression: NONE
    bloomFilterType: NONE
  text:
    maxVersions: 2147483647
    compression: GZ
    bloomFilterType: ROW
```

Stored Procedures:

Esempio di salvataggio di una funzione come procedura memorizzata:

javascript

```
db.system.js.save({
  _id: 'make_the_groups',
  value: function(collection){
```

```
        return collection.group({
          initial: {count: 0},
          reduce: function(documents, output){output.count++;},
          cond: {'lemma': {$in: ['mamma']}},
          key: {'n_words': true}
        });
      }
    })
    db.eval('make_the_groups(db.documents)')
```

MapReduce in MongoDB:

Esempio di utilizzo di `emit()` e `reduce()` per contare le parole in MongoDB:

```
javascript
    map = function(){
      for(var i=0; i<this.lemma.length; i++){
        emit({word: this.lemma[i]}, {count: 1});
      }
    }
    reduce = function(word, doc_list){
      var total = 0;
      for(var i=0; i<doc_list.length; i++){
        total += doc_list[i].count;
      }
      return {count: total};
    }
    db.runCommand({
      mapReduce: 'documents',
      map: map,
      reduce: reduce,
      out: 'documents.word_count'
    })
```

Replica Sets e Sharding:

Un Replica Set in MongoDB è un gruppo di processi `mongod` che forniscono ridondanza e alta disponibilità. Un Replica Set è composto da un membro primario e molti membri secondari che mantengono le informazioni aggiornate tramite un `oplog`.

Il Sharding in MongoDB partiziona la collezione utilizzando la chiave di shard. Esempio di partizionamento per cardinalità della chiave, frequenza e tasso di modifica.

MongoDB: un database orientato ai documenti:

MongoDB è un database orientato ai documenti che consente ai dati di essere persistenti e nidificati. Le query sui dati nidificati avvengono in modo ad hoc.

Non vincola i dati a uno schema predefinito, quindi i documenti possono avere campi o tipi diversi.

È utilizzato da progetti importanti come Foursquare, bit.ly e CERN per memorizzare i dati degli esperimenti del Large Hadron Collider (LHC).

Oggetti in MongoDB in formato JSON:

Per creare un database, utilizziamo il comando `mongo book`, che crea un database chiamato 'book'. Si possono visualizzare gli altri database con `show dbs` e cambiare database con il comando `use`.

La creazione di una collezione di oggetti avviene con l'inserimento del primo oggetto, simile a un bucket nella nomenclatura di Riak. Esempio di inserimento di un oggetto:

```
db.towns.insert({
        name: "New York",
        population: 22200000,
        last_census: ISODate('2009-07-31'),
        famous_for: ["statue of liberty", "food"],
        mayor: {
           name: "Michael Bloomberg",
           party: "I"
        }
    })
```

Per visualizzare le collezioni create, utilizziamo i comandi `show collection` e `system.indexes`.

Query in MongoDB utilizzando Javascript:

Per richiedere tutti gli oggetti in una collezione si utilizza `find()`, che consente ai processi client di essere indipendenti nella creazione degli oggetti rispetto agli altri processi distribuiti.

Esempi di query in MongoDB:

```
typeof db  // "object"
    typeof towns  // "object"
    typeof db.towns.insert  // "function"
```

Popolazione della collezione `towns` avviene con il comando `db.towns.insert({...})`.

Query per documenti:

Per richiedere un singolo oggetto si utilizza `find` con il parametro `_id`. È possibile specificare i campi da mostrare:

db.towns.find({_id: ObjectId("...")}, {name: 1, _id: 0})

Si possono cercare congiunzioni di criteri utilizzando espressioni regolari simili a Perl. Esempio:

db.towns.find({name: /^P/, population: {$lt: 10000}})

Gli operatori condizionali seguono il formato `{$op: value}`.

Ricerche avanzate:

Per cercare un intervallo di valori definito dall'utente, costruiamo un oggetto Javascript per la condizione della query. Esempio di ricerca di corrispondenze parziali:

db.towns.find({name: /partial_value/})

Per cercare un campo in un oggetto nidificato, utilizziamo `$elemMatch` e altri operatori.

Operazioni di aggiornamento:

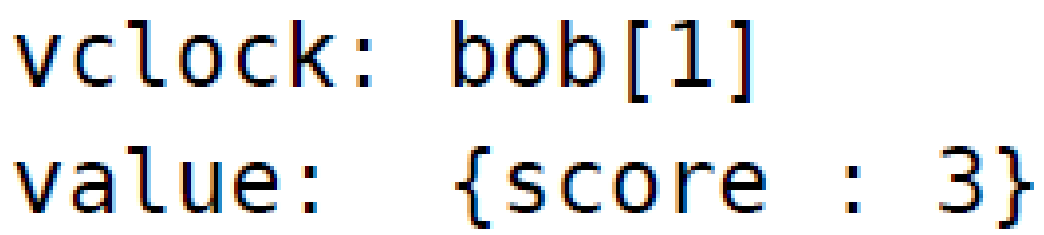
Le operazioni di aggiornamento richiedono due parametri: criteri di ricerca (come in `find()`) e un oggetto con i nuovi valori dei campi o un'operazione di modifica che utilizza l'operatore `$set`.

Esempio di aggiornamento di un oggetto esistente:

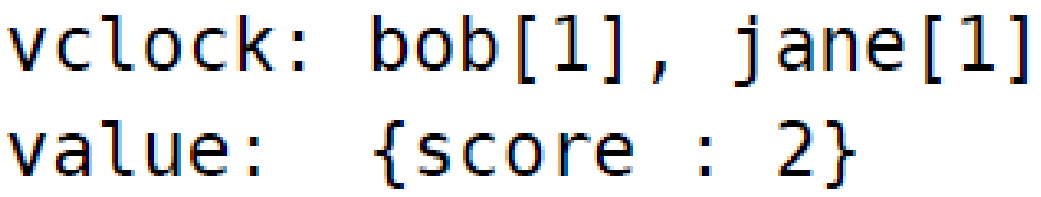
db.towns.update({_id: ObjectId("...")}, {$set: {population: 23000000}})

Esistono diversi operatori per la modifica dei documenti esistenti.

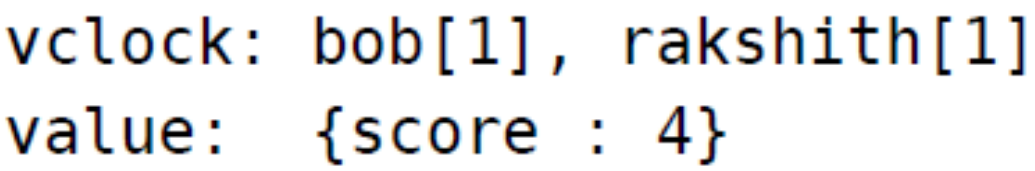
Esempi di operazioni di aggiornamento in MongoDB.



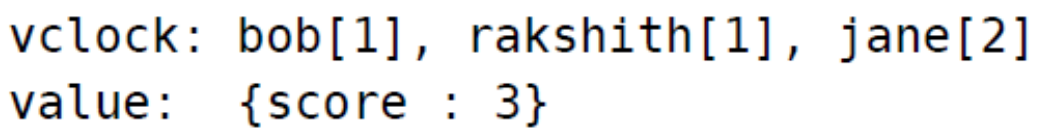
Esempi di operazioni di aggiornamento in MongoDB.



Esempi di operazioni di aggiornamento in MongoDB.



Esempi di operazioni di aggiornamento in MongoDB.



Join e riferimenti:

Le operazioni di join non sono efficienti in MongoDB poiché è un database distribuito per natura. È possibile aggiungere riferimenti agli oggetti nelle collezioni con la sintassi:

{$ref: "collection_name", $id: "reference_id"}

Indici in MongoDB:

Per impostazione predefinita, MongoDB crea un indice sul campo `_id` utilizzando un B-tree. Per creare un indice su altri campi, utilizziamo:

db.collection.ensureIndex({property_name: 1}, {unique: true, partialFilterExpression: {property_name_2: condition}})

Funzioni aggregate:

Esempio di conteggio dei documenti con una parola specifica:

db.documents.count({'lemma': {$in: ['mamma']}})

Esempio di utilizzo delle funzioni di aggregazione in Javascript:

```
db.documents.group({
        initial: {count: 0},
```

```
    reduce: function(documents, output){output.count++;},
    cond: {'lemma': {$in: ['mamma']}},
    key: {'n_words': true}
})
```

Stored Procedures:

Esempio di salvataggio di una funzione come procedura memorizzata:

```
db.system.js.save({
    _id: 'make_the_groups',
    value: function(collection){
      return collection.group({
        initial: {count: 0},
        reduce: function(documents, output){output.count++;},
        cond: {'lemma': {$in: ['mamma']}},
        key: {'n_words': true}
      });
    }
})
db.eval('make_the_groups(db.documents)')
```

MapReduce in MongoDB:

Esempio di utilizzo di `emit()` e `reduce()` per contare le parole in MongoDB:

```
map = function(){
    for(var i=0; i<this.lemma.length; i++){
      emit({word: this.lemma[i]}, {count: 1});
    }
}
reduce = function(word, doc_list){
    var total = 0;
    for(var i=0; i<doc_list.length; i++){
      total += doc_list[i].count;
    }
    return {count: total};
}
db.runCommand({
    mapReduce: 'documents',
    map: map,
    reduce: reduce,
    out: 'documents.word_count'
})
```

Replica Sets e Sharding:

Un Replica Set in MongoDB è un gruppo di processi `mongod` che forniscono ridondanza e alta disponibilità. Un Replica Set è composto da un membro primario e molti membri secondari che mantengono le informazioni aggiornate tramite un `oplog`.

Il Sharding in MongoDB partiziona la collezione utilizzando la chiave di shard. Esempio di partizionamento per cardinalità della chiave, frequenza e tasso di modifica.

Replica Sets e Sharding in MongoDB

Replica Sets e Sharding:

MongoDB costruisce alcune repliche che sono composte da un numero dispari di nodi che mantengono copie dei dati su server diversi.

Ciò permette di essere robusti contro l'indisponibilità temporanea dei nodi e di mantenere la consistenza dei dati.

MongoDB esegue il sharding (partizionamento della collezione per intervallo di valori).

Se alcuni nodi non sono raggiungibili, la rete diventa composta dalla maggioranza dei nodi che possono comunicare tra loro.

Un nodo master viene scelto e i nodi continuano a rispondere alle richieste restituendo i dati più recenti che hanno.

La rete si basa su un file system distribuito: GridFS.

Può eseguire query distribuite, anche su dati geospaziali con `geoNear`.

Immagine che illustra il concetto di Replica Sets e Sharding.

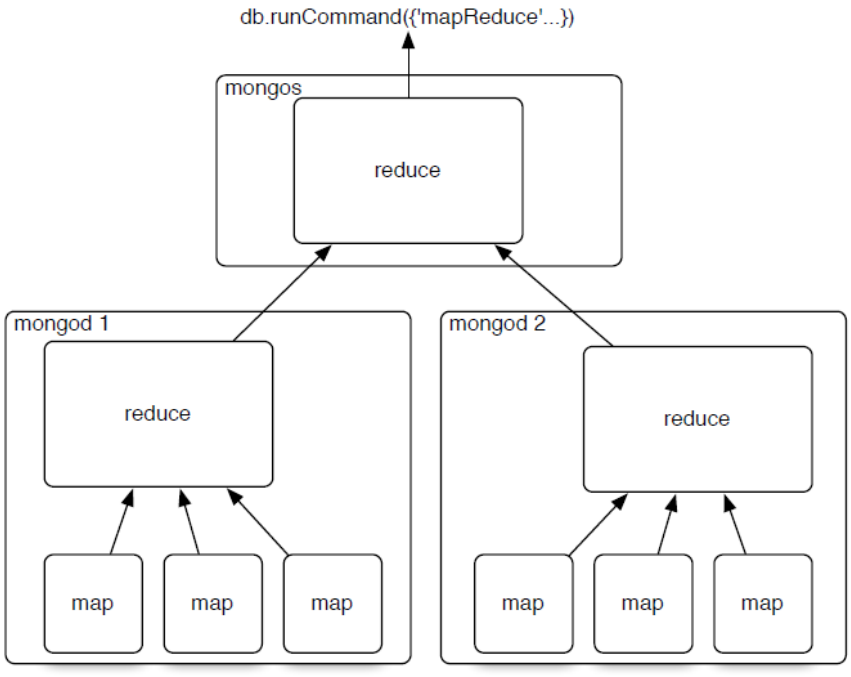


Figure 22—A Mongo map reduce call over two servers

Replica Sets e Shards in un Cluster Shardato:

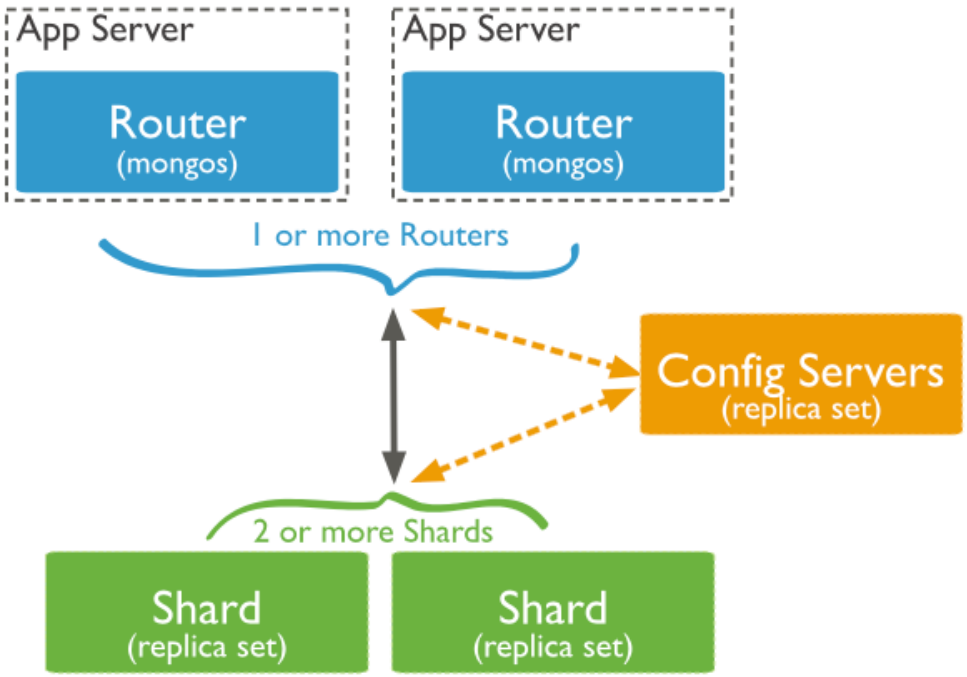
Un replica set in MongoDB è un gruppo di processi `mongod` che forniscono ridondanza e alta disponibilità.

I membri dei replica set sono:

Un membro primario.

Molti membri secondari che mantengono aggiornate le informazioni del primario tramite un `oplog` (registro delle operazioni).

Esempio di Replica Sets e Shards in un Cluster Shardato.



Creazione di un Replica Set:

Esempio di creazione di un Replica Set:

Creare le directory dei dati:

$ mkdir ./mongo1 ./mongo2 ./mongo3

Avviare i server Mongo aggiungendo il flag `replSet` con il nome `book` e specificare le porte:

$ mongod --replSet book --dbpath ./mongo1 --port 27011 --rest
$ mongod --replSet book --dbpath ./mongo2 --port 27012 --rest
$ mongod --replSet book --dbpath ./mongo3 --port 27013 --rest

Inizializzare il replica set eseguendo la funzione `rs.initiate()`:

```
$ mongo localhost:27011
    > rs.initiate({
      _id: 'book',
      members: [
       {_id: 1, host: 'localhost:27011'},
       {_id: 2, host: 'localhost:27012'},
       {_id: 3, host: 'localhost:27013'}
      ]
    })
    > rs.status()
```

Partizionamento dei Documenti in Shards per Chiave:

MongoDB partiziona la collezione usando la chiave di shard:

Il partizionamento può essere basato su:

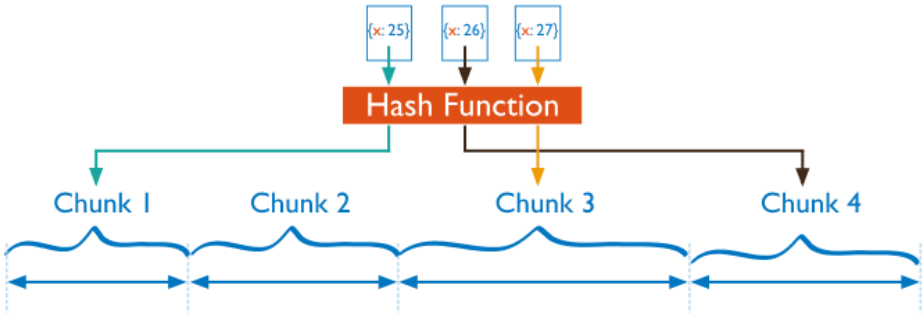
Cardinalità della chiave (numero di valori distinti per la chiave).

Frequenza (numero di oggetti con quel valore di chiave).

Tasso di modifica (numero di richieste di modifica di quella chiave per unità di tempo).

MongoDB utilizza la chiave di shard associata alla collezione per partizionare i dati in chunks.

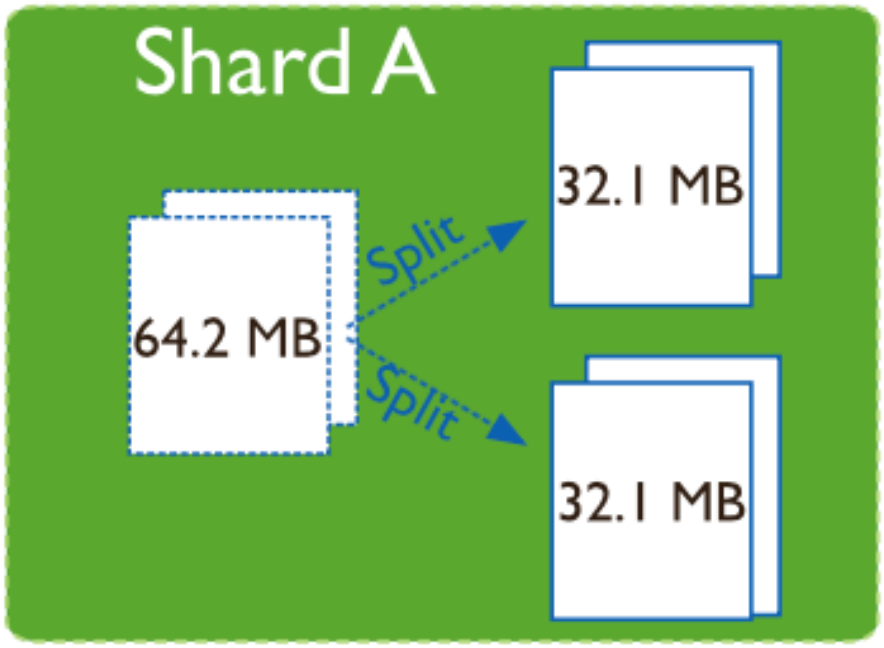
Esempio di partizionamento dei documenti in shards per chiave.



Hashing delle Chiavi di Shard:

MongoDB utilizza l'hashing delle chiavi di shard per distribuire uniformemente i dati tra i nodi.

Esempio di hashing delle chiavi di shard.



Divisione dei Chunks:

La divisione dei chunks è un processo che impedisce ai chunks di crescere troppo.

Quando un chunk supera una dimensione specificata o se il numero di documenti nel chunk supera un massimo per chunk, MongoDB divide il chunk basandosi sui valori della chiave di shard rappresentati nel chunk.

Configurazione dei Dati Shardati:

Esempio di lancio di server mongod senza replica:

```
$ mkdir ./mongo4 ./mongo5
    $ mongod --shardsvr --dbpath ./mongo4 --port 27014
    $ mongod --shardsvr --dbpath ./mongo5 --port 27015
```

Creare un server di configurazione:

```
$ mkdir ./mongoconfig
    $ mongod --configsvr --dbpath ./mongoconfig --port 27016
```

Avviare un server mongos:

```
$ mongos --configdb localhost:27016 --chunkSize 1 --port 27020
```

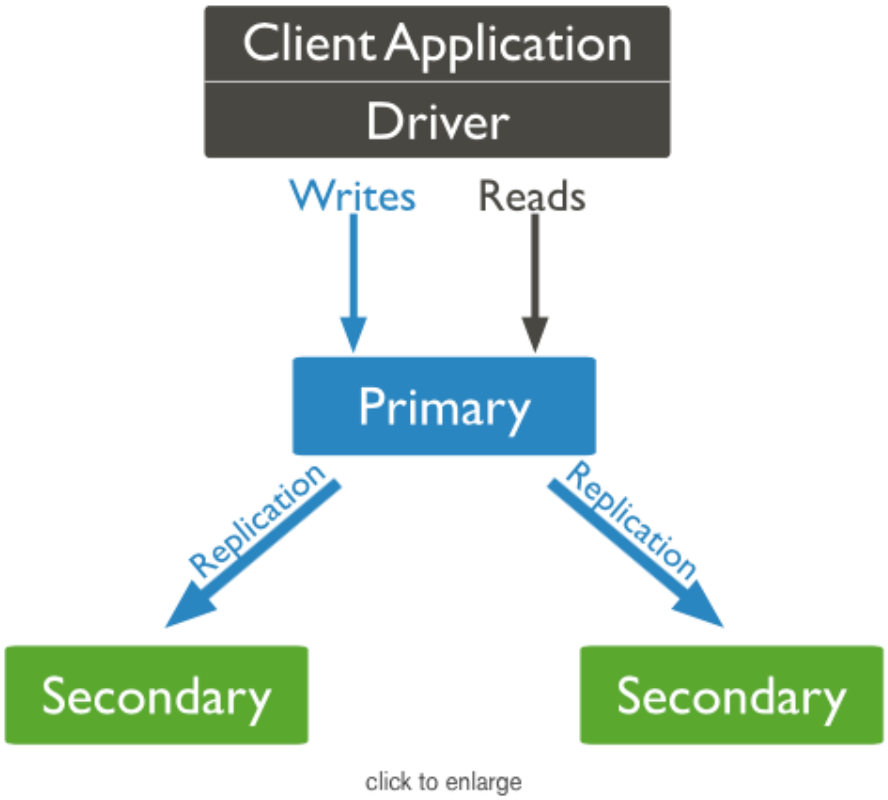
Aggiungere i shard:

```
$ mongo localhost:27020/admin
    > db.runCommand( { addshard : "localhost:27014" } )
    > db.runCommand( { addshard : "localhost:27015" } )
```

Bilanciamento degli Shards:

MongoDB bilancia automaticamente i chunks tra i nodi shard per garantire una distribuzione uniforme dei dati.

Esempio di bilanciamento degli shards in MongoDB.



click to enlarge

Gremlin: un Linguaggio di Query per i Grafi:

Neo4J supporta Gremlin:

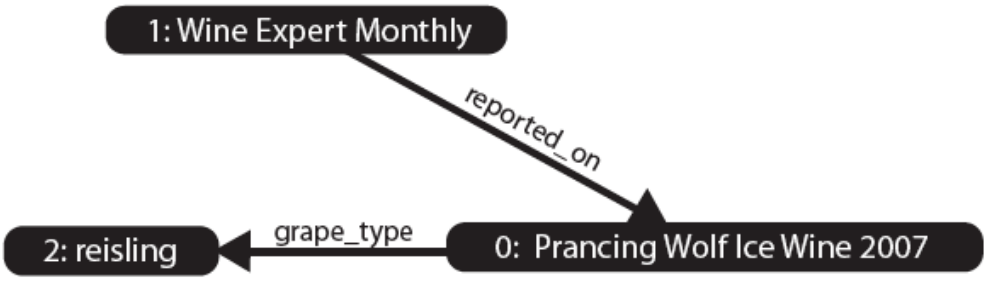
Gremlin è un linguaggio di query per attraversare i grafi.

Simile a jQuery (in JavaScript) e può essere utilizzato anche per navigare documenti HTML.

Esempi di query in Gremlin:

```
g.V().hasLabel('wine').values('name')
    g.E().hasLabel('produced_by').inV().values('name')
```

Esempi di query in Gremlin per Neo4J.



Esempi di query in Gremlin per Neo4J.

```
gremlin> g.V
==>v[0]
==>v[1]
==>v[2]
```

Esempi di query in Gremlin per Neo4J.

```
gremlin> g.E
==> e[0][1-reported_on->0]
==> e[1][0-grape_type->2]
```

Redis: Remote Dictionary Service:

Redis è un archivio key-value con strutture dati avanzate e la capacità di gestire insiemi di dati.

È molto veloce nelle letture ma soprattutto nelle scritture.

Redis è stato definito un server di strutture dati.

Supporta una coda di blocco e un meccanismo di publish-subscribe.

Offre politiche di scadenza, livelli di durabilità e opzioni per la replica dei dati.

Introduzione a Redis.

```
redis 127.0.0.1:6379> SET count 2
OK
redis 127.0.0.1:6379> INCR count
(integer) 3
redis 127.0.0.1:6379> GET count
"3"
```

Introduzione a Redis.

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET prag http://pragprog.com
QUEUED
redis 127.0.0.1:6379> INCR count
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) (integer) 2
```

Comandi per le Strutture Dati:

I comandi per SET iniziano con `S`.

I comandi per Hash iniziano con `H`.

I comandi per l'ordinamento iniziano con `Z`.

I comandi per List iniziano con `L` (Left) o `R` (Right) a seconda della direzione in cui la lista è attraversata.

Progetto NoSQL: Ristrutturazione di una piattaforma web

Riepilogo:

I database possono essere suddivisi in cinque generi principali:

1. Relazionale
2. Key-Value
3. Columnar
4. Document
5. Graph

Relazionale:

I sistemi di gestione di database relazionali (RDBMS) sono basati sulla teoria degli insiemi e implementati come tabelle bidimensionali con righe e colonne.

Enfatizzano il tipo di dati e sono generalmente numerici, stringhe, date e blob non interpretati.

Forniscono estensioni come array o cubi.

Key-Value:

I database Key-Value (KV) sono i più semplici, mappano chiavi semplici a valori complessi come una grande tabella hash.

Elevata flessibilità di implementazione.

Le ricerche hash sono veloci e facilmente distribuite.

Columnar:

I database Columnar (orientati alle colonne) condividono somiglianze con i database KV e RDBMS.

Valori raggruppati in colonne anziché righe.

Esempio classico: HBase.

Document:

I database orientati ai documenti consentono campi variabili e oggetti nidificati.

Rappresentazione comune in JSON, utilizzato da MongoDB e CouchDB.

Facili da shardare e replicare.

Graph:

I database orientati ai grafi si concentrano sull'interrelazione dei dati piuttosto che sui valori.

Ideali per motori di raccomandazione, liste di controllo accessi e dati geografici.

Esempio di progetto:

Ristrutturazione della piattaforma utilizzando NoSQL:

Gestione degli ordini di più utenti.

Ogni ordine può contenere più prodotti.

L'utente può modificare l'ordine finché non è completato.

Il management vuole conoscere le performance dei prodotti.

Esempio di progetto NoSQL.

Modello utente:

L'utente fa il login e gli vengono suggeriti prodotti in base ai dati e alle relazioni.

Gli utenti possono mettere prodotti nel carrello e i prodotti hanno una disponibilità.

Sessioni utente gestite con chiavi-valore.

Modello utente per la gestione degli ordini.

Problema della gestione del carrello:

Gestione del carrello utente durante la sessione.

Verifica disponibilità prodotti nel carrello al ripristino della sessione.

Notifica all'utente in caso di indisponibilità di prodotti.

Descrizione del problema della gestione del carrello.

Modello di dati:

Utente: ID, sessione.

Prodotto: ID, disponibilità, catalogo.

Ordine: ID utente, sessione, lista prodotti con quantità.

Durata della sessione.

Modello di dati per la gestione degli ordini.

Scelta di Riak:

Sessioni con ID e oggetti, recupero immediato.

Prodotti con ID univoci e complessità variabili.

Ordini referenziati nell'oggetto sessione finché non chiusi.

Motivazioni per la scelta di Riak.

Salvataggio dello stato degli ordini:

Quando una sessione si chiude:

Salvataggio dei carrelli in due collezioni: venduti e ordini aperti.

Funzionamento e modello dei dati descritti.

Procedura per il salvataggio dello stato degli ordini.

Modello Document Database:

Shopping Center con magazzini distribuiti globalmente:

Partizionamento dei prodotti per ID-store.

Sharding dei dati per magazzini diversi ma simili.

Durata dei documenti modellata.

Modello Document Database per la gestione dei magazzini.

Analisi delle performance dei prodotti:

Monitoraggio dei carrelli per informazioni sui prodotti richiesti:

Suggerimenti basati sull'area dell'utente.

Aggiornamento giornaliero degli ordini in un database HBase o relazionale per analytics.

Analisi delle performance dei prodotti e suggerimenti.

Raccomandazioni ai clienti:

Raccomandazioni basate sugli interessi degli utenti:

Utilizzo di un Graph DB per osservare gli interessi degli utenti nell'area o tra gli amici.

Sistema di raccomandazioni ai clienti.