UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

Relazione Finale
Studio e sperimentazione di basi di dati NoSQL in memory per sensoristica

Relatore: Prof. Michele Melchiori

Laureando:
Matteo Rizzo
Matricola n. 727499

Anno Accademico 2021/2022

Ringraziamenti

Introduzione

Analisi database noSQL chiave-valore, in particolare *REDIS* progettazione concettuale/logica database chiave-valore

Indice

Capitolo 1

DataBase NoSQL

I database *NoSQL*, che sta per *not only SQL*, sono database non tabellari che archiviano i dati in maniera completamente differente dai classici relazionali. Le caratteristiche principali sono la progettazione specifica per carichi elevati e il supporto nativo per la scalabilitá orizzontale, la tolleranza agli errori e la memorizzazione dei dati in modo denormalizzato. Infatti ogni elemento viene archiviato singolarmente con una chiave univoca, e la coerenza dei dati non viene garantita. Questa impostazione fornisce un approccio molto piú flessibile alla memorizzazione dei dati rispetto a un database relazionale, un controllo migliore e una maggiore semplicitá nelle applicazioni.

1.1 Perche é nato?

A partire dagli anni 2000 si é passati da un modello in cui le persone principali dell'IT erano sistemisti ad un modello in cui le persone principali sono diventate gli sviluppatori. Tale passaggio ha comportato la nascita di database NoSQL che sono fortemente orientati agli sviluppatori ed allo sviluppo Agile. Inoltre i dati si sono trasformati passando dai classici strutturati a dati non strutturati (di differenti dimensioni, semistrutturati, polimorfici..) che non permettevano di definire un modello relazionale organico e cosí i database NoSQL sono diventati estremamente popolari perché permettono di lavorare principalmente con dati non strutturati anche di enormi dimensioni.

1.2 Perché utilizzare un database *NoSQL*?

I database NoSQL sono una soluzione ideale per molte applicazioni moderne, quali dispositivi mobili, Web e videogiochi che richiedono strutture dati flessibili, scalabili, con prestazioni elevate ed altamente funzionali.

1.2.1 Vantaggi

I principali vantaggi sono:

- **Schemaless**: vengono offerti schemi flessibile che consentono uno sviluppo piú veloce. Quindi é una soluzione ideale per i dati semi-strutturati e non strutturati. É possibile arricchire le applicazioni di nuovi dati e informazioni senza dover sottostare ad una rigida struttura;

- **Scalabilitá**: grazie alla semplicitá vi é la possibilitá di *scalare in orizzontale* in maniera estremamente efficiente. Infatti, si predilige l'utilizzo di cluster con molti nodi distribuiti, rispetto all'utilizzo di server centralizzati. Inoltre, vi é la possibilitá di aggiungere nodi a caldo in maniera completamente trasparente per l'utente finale;

- **Elevate Prestazioni**: grazie alla mancanza di operazioni di aggregazione dei dati("join") ed anche grazie all'introduzione di semplificazioni, come il mancato supporto delle transazioni ACID, si ha una elevata velocitá computazionale;

- **Riduzione dei tempi di sviluppo**: grazie alla definizione di logiche di lettura dati molto piú semplici rispetto a quelle da scrivere con database relazionali.

1.2.2 Limiti

La semplicitá di questi database comporta anche degli svantaggi:

- **Integritá:** mancando i controlli fondamentali sull'integritá dei dati, il compito ricade totalmente sull'applicativo che dialoga con il database;

- **Scrittura**: problema strettamente collegato all'integritá, poiché ogni volta che si deve aggiornare un dato ridondato in piú entitá diventa necessario aggiornare il dato su tutte le entitá in cui é stato duplicato; questo di fatto tende ad aumentare i tempi di sviluppo, anche se le operazioni di lettura sono notevolmente semplificate;

- **Standard universale**: ogni database ha il proprio metodo di storing ed accesso ai dati, ne deriva che vi é una mancanza di uno standard universale come SQL. Quindi, il passaggio da un database ad un altro puó richiedere alcuni cambi piú o meno radicali da apportare all'applicativo;

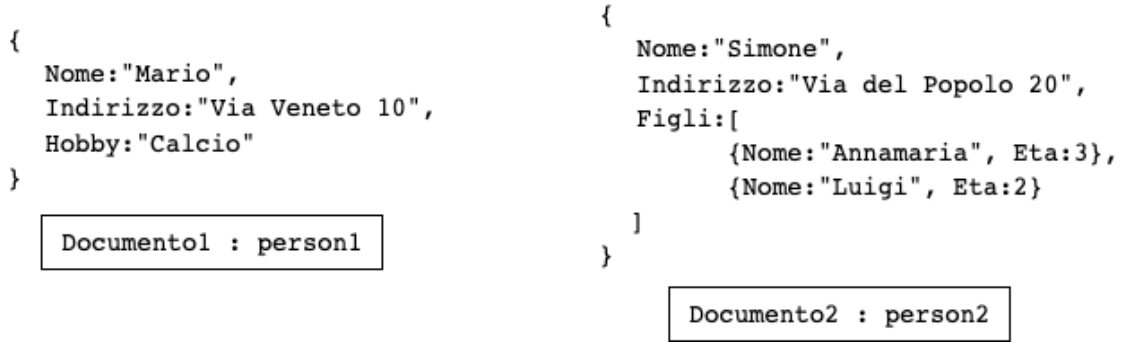
- **Espressivitá**: problema legato ai linguaggi di interrogazione, infatti risulta essere meno espressivo rispetto a SQL nella grande maggioranza dei casi, portando ad una maggiore difficoltá di correzione massiva (data fixing), report ed export dei dati.

1.3 Tipologie di *NoSQL*

Tipologie principali di database *NoSQL*:

- **documentali**: la rappresentazione dei dati é affidata a strutture simili ad oggetti, dette *documenti*, ognuno dei quali possiede un certo numero di proprietá che rappresentano le informazioni. Viene creata una semplice coppia, a una chiave viene assegnato un documento specifico, e in questo documento, il quale puó essere formattato in vari modi (XML, JSON, YAML ...) si possono trovare le informazioni. La nozione di schema é dinamica, ogni documento puó contenere appunto dei campi diversi. Questa flessibilitá puó essere particolarmente utile per la modellazione dei dati in cui le strutture possono cambiare da un record all'altro, ad esempio nei dati polimorfici. Inoltre, diventa piú semplice l'evoluzione di un'applicazione durante il suo ciclo di vita, ad esempio nel caso in cui vadano aggiunti nuovi campi.

  Tra gli esempi di maggiore interesse vi sono: MongoDB, Azure CosmosDB, Apache CouchDB.

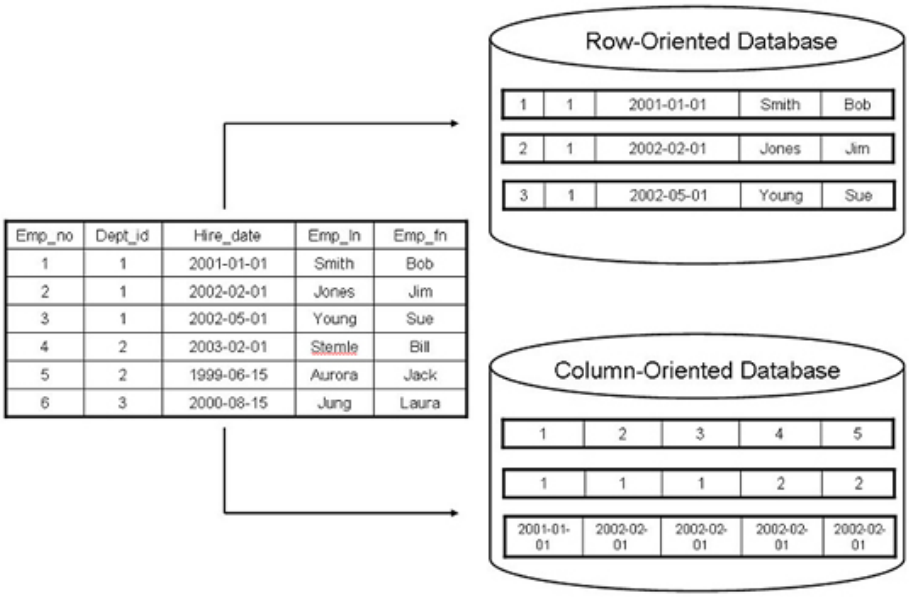


- **key-value**: i dati vengono immagazzinati mediante un semplice metodo chiave-valore. Una chiave rappresenta un identificatore univoco. Le chiavi e i valori possono essere qualsiasi cosa, da un oggetto semplice ad articolati oggetti composti. (Questo tipo di base di dati é oggetto di tesi e quindi verrá sviluppato il suo concetto nel corso dei prossimi capitoli.)

  Tra gli esempi di maggiore interesse vi sono: **Redis**, MemCached.

- **colonnari**: i dati vengono archiviati per colonne, anziché per righe come avviene nei database relazionali classici. Queste colonne vengono raccolte per formare dei sottogruppi. Le chiavi e i nomi delle colonne di questo tipo di database non sono fissi. Ogni colonna é memorizzata separatamente. Se sono presenti colonne simili, vengono unite in famiglie di colonne ed ogni famiglia viene archiviata separatamente dalle altre su un "file" diverso. Questa tipologia di database viene utilizzata quando é necessario un modello di dati di grandi dimensioni. Estremamente utili per i data warehouse, oppure quando sono necessarie prestazioni elevate o la gestione di query intensive.

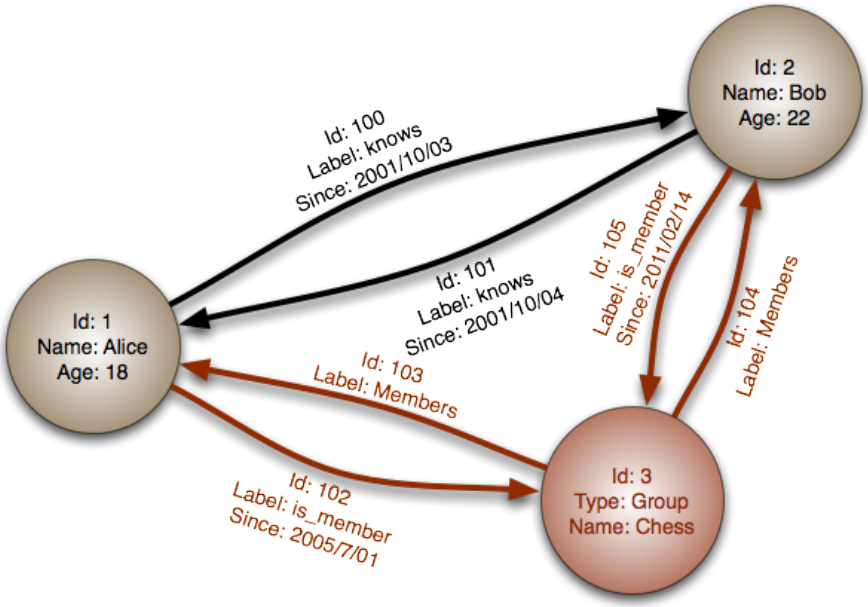
  Tra gli esempi di maggiore interesse vi sono: HBase, Cassandra, Vertica.



- **a grafo**: progettati appositamente per l'archiviazione e la navigazione di relazioni. Le relazioni rivestono un ruolo chiave e buona parte del valore di questi database deriva proprio dalla loro presenza. Vengono utilizzati i *nodi* per archiviare le entitá di dati e gli *archi* per archiviare le relazioni tra entitá. Le

relazioni che un nodo puó avere sono illimitate. In questo tipo di database attraversare collegamenti o relazioni é molto veloce perché le relazioni tra i nodi non vengono elaborate al momento della query, ma sono giá presenti nel database. I casi d'uso piú tipici sono i Social Network, motori di raccomandazioni e rilevamento di frodi, ovvero in tutti quegli ambiti dove é necessario creare molte relazioni tra dati ed eseguire rapidamente query su di esse.

Tra gli esempi di maggiore interesse vi sono: Neo4J, Titan.



Capitolo 2

Progettazione Concettuale/Logica

Sebbene i database *NoSQL* vengano definiti *schemaless*, la progettazione dell'organizzazione dei dati richiede di prendere decisioni significative. Infatti, i dati persistenti delle applicazioni hanno un impatto sui principali requisiti di qualitá che devono essere soddisfatti in un'applicazione vera e propria (scalabilitá, prestazioni, coerenza). Il mondo NoSQL é altamente eterogeneo, quindi questa attivitá di progettazione di solito si basa su pratiche e linee guida da seguire in base al sistema adoperato. Peró, sono stati studiati diversi approcci che vogliono generalizzare il problema di progettazione che é alla base di ogni sistema di persistenza.

NoAM é uno dei principali strumenti di modellazione astratto per database NoSQL. Grazie ad esso riusciamo a definire una progettazione che é indipendente dal sistema specifico in cui viene usata. Viene utilizzato un modello dei dati intermedio e astratto, che, a sua volta, viene utilizzato per rappresentare i dati dell'applicazione come raccolte di oggetti aggregati.

2.1 Modellazione NoAM → NoSQL abstract model

La metodologia *NoAM* é composta da:

- Modellazione Concettuale e design degli Aggregati
- Partizionamento degli Aggregati e modellazione NoSQL
- Implementazione

2.1.1 Modellazione Concettuale e design degli Aggregati

Riguarda la vera e propria progettazione del modello di dominio, e comporta l'identificazione delle diverse classi di aggregati necessari in un'applicazione.

Cos'é un aggregato? Un aggregato é una porzione di dati correlati, con una struttura piú o meno complessa ed ha un identificatore univoco. Gli aggregati regolano anche la distribuzione dei dati, infatti per supportare la scalabilitá sono distribuiti tra i nodi di un sistema; ogni oggetto aggregato si trova su un singolo nodo.

Sono possibili diversi approcci per identificare classi di aggregati per una particolare applicazione. L'approccio Domain-Driven Design(*DDD*), attraverso il quale viene generato un diagramma UML delle classi, é guidato dai casi d'uso, ovvero dai requisiti funzionali, e da esigenze di scalabilitá e coerenza all'interno dell'aggregato. Si procede nel modo seguente:

- I dati persistenti di un'applicazione sono modellati in termini di entitá, oggetti di valore e relazioni. Un'entitá é un oggetto persistente che ha un'esistenza indipendente ed é caratterizzata da un identificatore univoco, mentre un oggetto di valore é caratterizzato appunto da un suo valore senza un proprio identificatore

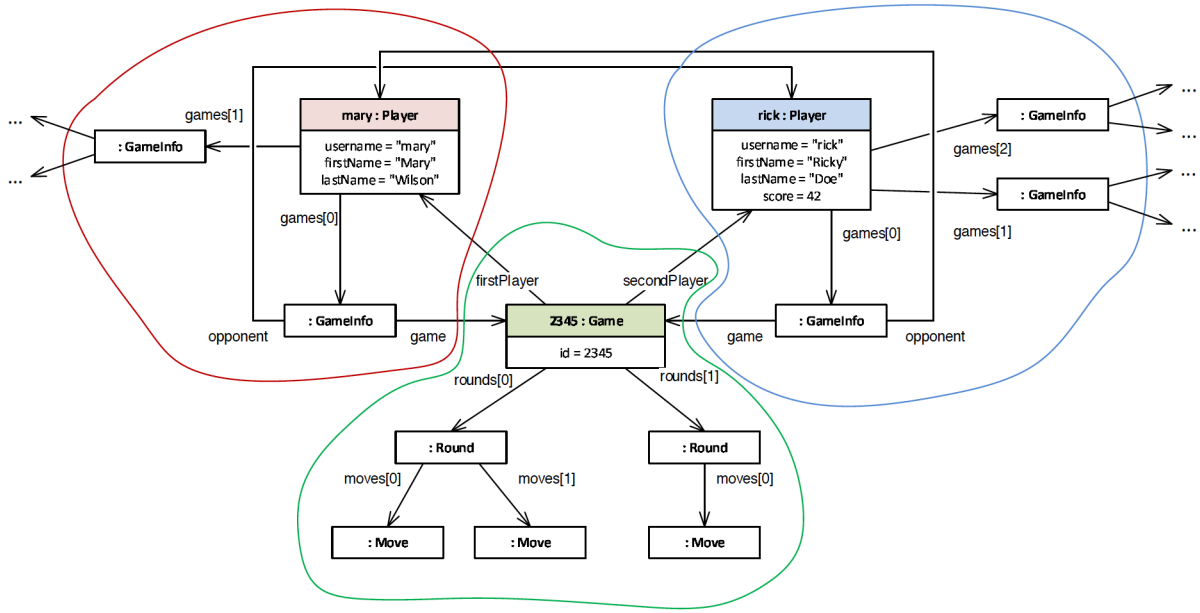
- Entitá e oggetti di valore vengono raggruppati in *aggregati*. Un aggregato ha un'entitá come radice e puó contenere molti oggetti di valore.

A causa delle loro caratteristiche, la progettazione degli aggregati comporta un compromesso per quanto riguarda la loro granularitá. Infatti:

- Gli aggregati dovrebbero essere abbastanza grandi per poter includere tutti i dati coinvolti da certi vincoli di integritá.

- Gli aggregati dovrebbero essere i piú piccoli possibile, in quanto dimensioni ridotte consentono di soddisfare requisiti di prestazioni e scalabilitá.

Preso come esempio un dominio in cui vanno salvati in modo persistente dati su giocatori e giochi



Nella figura sopra riportata l'oggetto con lo scomparto superiore colorato é un'entitá, altrimenti é un oggetto di valore. La linea chiusa denota il confine di un aggregato. Pertanto avremo due classi aggregate principali: Player e Game.
Quindi, per costruire un'aggregato si parte sempre da un'entitá principale. Le freccie uscenti indicano la composizione dell'entitá e, ricorsivamente, degli oggetti di valore. Se consideriamo l'entitá `mary:Player`, essa sará composta, oltre che dai suoi attributi, da:
due oggetti di valore, rispettivamente `games[0]` e `games[1]`, a sua volta `games[0]` é formato dall'oggetto di valore `:GameInfo` cosí composto:
il valore di `game` sará `2345:Game`, il valore di `opponent` sará composto da `rick:Player`.
Stesso procedimento verrá fatto per `games[1]`.
Bisognerá ripetere il procedimento per ogni entitá presente nel diagramma.

2.1.2 Partizionamento degli Aggregati e modellazione NoSQL

In questa fase viene utilizzato *NoAM* come modello intermedio tra gli aggregati e i database NoSQL, quindi potrebbe essere visto come un equivalente della *progettazione concettuale* fatta nei database relazionali. Il modello NoAM → modello di dati astratti, ha il compito di sfruttare i punti in comune dei vari modelli di dati, ma introduce anche astrazioni per bilanciare le variazioni che vi sono tra diversi modelli di NoSQL.

Da questa trasformazione si ottengono due strutture con diverse granularitá:

- `blocco`, unitá di dimensione maggiore, ha massima consistenza
- `entry`, unitá di dimensione minore, che permette l'accesso ai dati

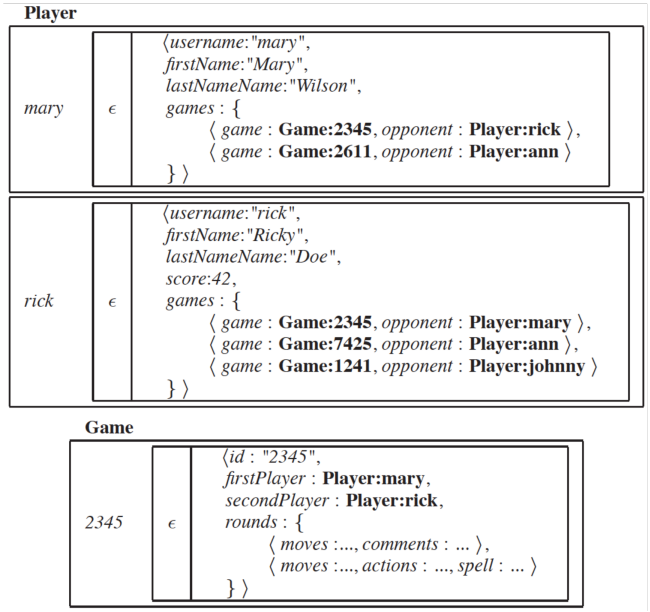
Con riferimento in particolare ai database chiave-valore una `entry` corrisponde a una coppia chiave-valore, mentre un `blocco` corrisponde a un gruppo di coppie chiave-valore correlate tra loro.

Quindi, si ha che:
Un `database` é un insieme di `collections`;
Ogni `collection` ha un nome distinto; Una `collection` é un insieme di `blocchi`;
Ogni `blocco` all'interno della `collection` é identificato da una chiave di blocco, che deve essere univoca; Un `blocco` é un insieme non vuoto di `entries`;
Ogni `entry` é composta da una coppia chiave-valore, la quale é univoca all'interno del blocco, il valore puó essere anche complesso.

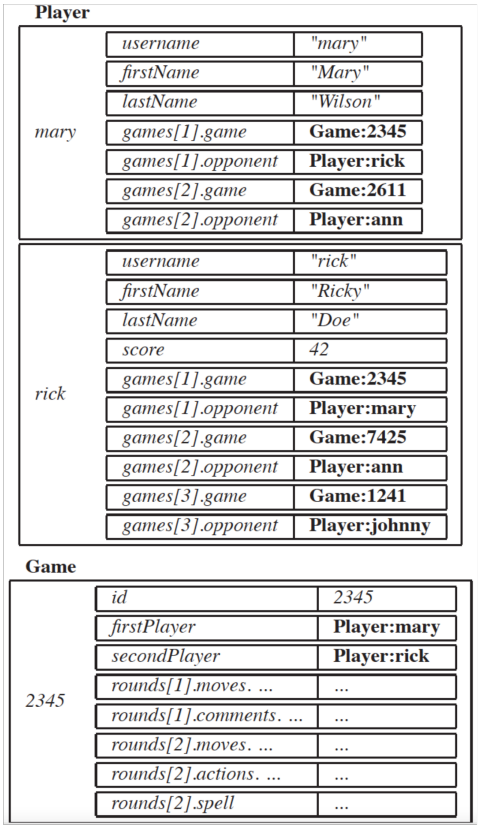
Per effettuare il passaggio da aggregati a modellazione NoSQL, ogni classe di aggregati viene rappresentata da una `collection` ed ogni singolo aggregato viene rappresentato da un `blocco`.
Un paragone puó essere fatto con la programmazione ad oggetti, in cui una `collection` é una classe, ed un `blocco` é l'istanza di quella classe.

Vi sono due modalitá principali per rappresentare gli aggregati:

- *Entry per Aggregate Object (EAO)*: Rappresenta ogni aggregato utilizzando una singola `entry`, la chiave della `entry` é vuota, il valore contiene l'intero aggregato. Metodologia dedicata maggiormente ai database documentali.
- *Entry per Atomic Value (EAV)*: Rappresenta ogni aggregato per mezzo di piú `entry`; Nella fase successiva, ovvero quella di implementazione, le chiavi di ogni `entry` verranno utilizzate insieme al nome della collection ed alla chiave di blocco per creare una chiave con un nome strutturato e rendere piú semplice la ricerca; i valori di ogni `entry` sono atomici.
  Questa metodologia é proprio quella che rispecchia in modo piú appropriato i database key-value.



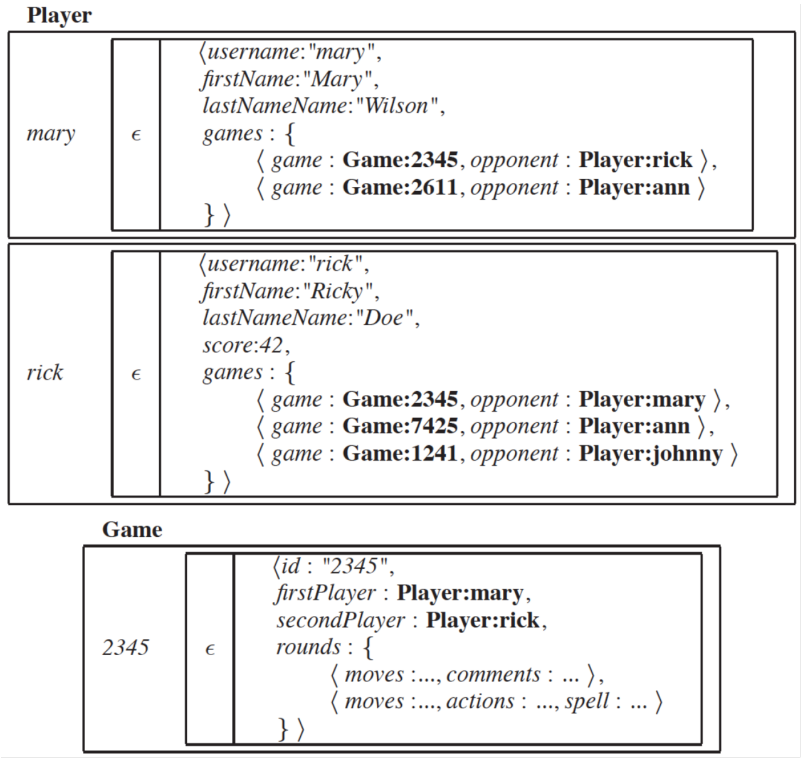
EAO



EAV

Nella figura *EAO* si ha per ogni `blocco` *Player* o *Game* una singola `entry` con chiave vuota, valore strutturato in modo complesso
Nella figura *EAV* si ha per ogni `blocco` piú `entry` in modo da avere valori atomici al suo interno

Ovviamente queste due sono le rappresentazioni piú estreme che si possono ottenere.
É possibile adottare una strategia di rappresentazione intermedia, denominata *Entry per Top level Field (ETF)*, in cui viene utilizzata una `entry` distinta per ogni campo di livello superiore, quindi si ha una sorta di struttura mista.



ETF

Se dovessimo aver bisogno di una rappresentazione degli aggregati ancora piú flessibile é possibile effettuarne il *partizionamento*, ovvero si possono raggruppare `entry` che vengono accedute insieme, oppure separare certe `entry` per avere dei valori meno complessi.

2.1.3 Implementazione

Consiste nel tradurre i modelli NoAM ottenuti nella fase precedente in strutture corrette per il dbms specifico che stiamo utilizzando. Per quanto riguarda i database chiave-valore si utilizzerá una coppia chiave-valore per ogni `entry` ottenuta nella struttura precedente.
In base alle scelte di progetto si decide quale modello NoAM utilizzare (EAO/EAV/ETF); bisogna decidere che livello di complessitá vogliamo avere su chiavi e valori. Se vogliamo ottenere dei valori semplici, rappresentabili semplicemente con dei tipi atomici, utilizzeremo il metodo *EAV*, questo, peró, comporterá una maggiore complessitá delle chiavi. Mentre, se vogliamo chiavi molto semplici dovremo utiliz-

zare *EAO*, peró otterremo dei valori strutturati e complessi, quindi, dovremo anche confrontarci con il dbms di cui disponiamo per verificare se saranno presenti delle strutture dati adatte a rappresentare dei valori con un certo livello di complessitá, cosa non sempre presente nei database key-value.

EAO

key (/major/key/-/minor/key)	*value*
/Player/mary/-	*{ username: "mary", firstName: "Mary", ... }*

EAV

key (/major/key/-/minor/key)	*value*
/Player/mary/-/username	*mary*
/Player/mary/-/firstName	*Mary*
/Player/mary/-/lastName	*Wilson*
/Player/mary/-/games/1/game	**"Game:2345"**
/Player/mary/-/games/1/opponent	**"Player:rick"**
/Player/mary/-/games/2/game	**"Game:2611"**
/Player/mary/-/games/2/opponent	**"Player:ann"**

ETF

key (/major/key/-/minor/key)	*value*
/Player/mary/-/username	*mary*
/Player/mary/-/firstName	*Mary*
/Player/mary/-/lastName	*Wilson*
/Player/mary/-/games	*[{ ... }, { ... }]*

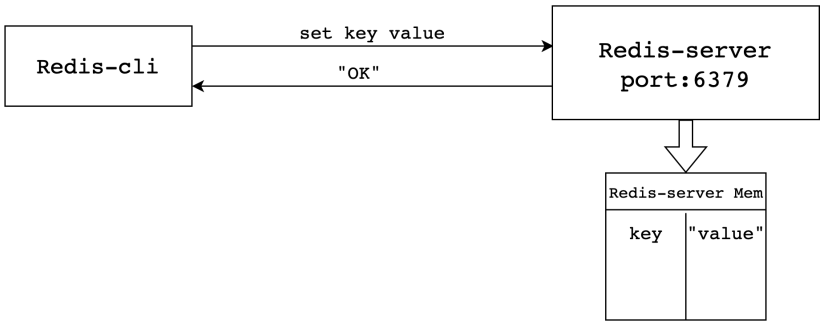
La figura *EAV* é quella che rispecchia maggiormente un database key-value classico con valori atomici. Si puó notare che le chiavi hanno una struttura gerarchica, si ha una larga somiglianza con il directory service dei sistemi operativi; questo viene fatto principalmente per facilitare la ricerca dei valori nelle interrogazioni.

Capitolo 3

Redis → Remote Dictionary Server

Redis, acronimo di *Remote Dictionary Server*, é un archivio dati veloce, open source, in memoria e di tipo chiave-valore(**dbms NoSQL**). Si basa su una struttura a dizionario: ogni valore immagazzinato é abbinato ad una chiave univoca che ne permette il recupero. É stato sviluppato nel linguaggio di programmazione C, e funziona principalmente con sistemi unix based, non esiste un supporto ufficiale per Windows. Redis si basa su un modello client-server, infatti i programmi esterni dialogano con il server Redis utilizzando un socket TCP e un protocollo specifico di tipo request-response. Il client invia una richiesta al server attendendo la risposta sul socket ed il server elabora il comando e invia la risposta al client.

```
matteorizzo@MBP-di-matteo ~ % redis-cli
127.0.0.1:6379> set key value
OK
127.0.0.1:6379> get key
"value"
```



3.1 Caratteristiche

3.1.1 Strutture Dati

Una caratteristica di Redis é mettere a disposizione una grande varietá di tipi di dati associabili alle chiavi, infatti il valore archiviato in corrispondenza di una certa chiave puó essere molto differente da un tipo semplice come la stringa ed il valore stesso puó addirittura rappresentare una struttura dati. Inoltre vi é una grandissima possibilitá di manipolazione grazie all'elevato numero di funzioni presenti.
I principali tipi di dato disponibili sono:

- **Stringhe**: é il tipo piú semplice, vengono memorizzate sequenze di byte, inclusi testo, oggetti serializzati e array binari; sono spesso usati per la memorizzazione nella cache;

- **Liste**:rappresentano un elenco di stringhe indicizzate in base all'ordine di inserimento nella struttura. Possono essere modificate con inserimenti in testa o in coda. Vi é la possibilitá di trattare una lista come una coda (First In First Out) tramite il comando di inserimento `LPUSH` e il comando di prelievo `RPOP` oppure puó essere trattata come una pila (First In Last Out) tramite i rispettivi comandi `LPUSH` e `LPOP`;

- **Set**: é una raccolta non ordinata di stringhe univoche(sono chiamate *membri* del set); quindi vi é la possiblitá di utilizzare questa struttura dati per tenere traccia degli elementi univoci, rappresentare relazioni o eseguire operazioni di insiemi comuni come intersezioni, unioni e differenze;

- **Hash**: sono oggetti strutturati come raccolte di coppie campo(chiave)-valore. Possono essere utilizzati per rappresentare oggetti di base e per memorizzare raggruppamenti di contatori;

- **SortedSet**: sono una versione modificata dei Set. Sono anch'essi insiemi di stringhe che non ammettono duplicati ma, in piú, includono un valore detto `score` associato ad ogni elemento, in base al quale é possibile ordinare in senso ascendento o discendente i valori dell'insieme.

Associati a queste strutture dati vi sono comandi specifici dedicati ad ognuna, per avere maggiori informazioni rimando al manuale

3.1.2 Confronto con *proprietá ACID*

Nelle basi di dati relazionali ogni transazione gode delle proprietá ACID. Nei database NoSQL non é vera questa affermazione, in questa sezione si vogliono mettere in evidenza quali proprietá vengono soddisfatte da Redis e quali no.

Innanzitutto, bisogna vedere in che modo sono gestite le transazioni in Redis: i comandi utilizzati per le transazioni sono quattro:

- `MULTI`: contrassegna l'inizio di un blocco di transazione, i comandi successivi verranno accodati per l'esecuzione
- `EXEC`: esegue tutti i comandi precedentemente accodati in una transazione e ripristina lo stato di connessione normale;
- `DISCARD`: svuota tutti i comandi precedentemente accodati in una transazione e ripristina lo stato di connessione normale;
- `WATCH`: contrassegna le chiavi fornite con un certo valore per eseguire un controllo condizionale al momento dell'esecuzione di una transazione (serve per la gestione di lock, ovvero controllo della concorrenza)

Quindi una transazione viene eseguita in questo modo:

1. inviamo il comando `MULTI`. Redis risponde `OK`;
2. digitiamo i comandi che devono far parte della transazione. Redis risponde `QUEUED`, ovvero il comando non viene eseguito istantaneamente ma viene messo in coda;
3. conclusione della transazione: si puó scegliere se eseguire tutti i comandi con `EXEC` oppure annullare la transazione con `DISCARD`.

Di seguito riporto un esempio utilizzando `redis-cli` con la struttura dati lista; i comandi per gestire le liste in questo esempio sono 2: `LPUSH`: comando per inserire un singolo elemento nella lista; `LRANGE`: comando per ottenere tutti i valori presenti nella lista

```
> MULTI
  OK
(TX)> LPUSH listaNumeri 3
      QUEUED
(TX)> LPUSH listaNumeri 10
      QUEUED
(TX)> LPUSH listaNumeri 34
      QUEUED
```

```
(TX)> LPUSH listaNumeri 45
      QUEUED
(TX)> EXEC
      1) (integer) 1
      2) (integer) 2
      3) (integer) 3
      4) (integer) 4

> LRANGE listaNumeri 0 -1
  1) "45"
  2) "34"
  3) "10"
  4) "3"
```

Si puó notare come l'inserimento di tutti i valori nella lista avvenga dopo il comando `EXEC`.

Quali proprietá ACID implementa Redis?

- **Atomicitá**: Redis puó avere due livelli di atomicitá:
  - singola operazione: ovvero ogni singola richiesta da parte del client viene eseguita in maniera atomica dal server;
  - transazione con operazioni multiple: come illustrato sopra con i comandi appositi;
- **Isolamento**: Tutti i comandi in una transazione vengono serializzati ed eseguiti in sequenza. Una richiesta inviata da un altro client non sará mai soddisfatta nel bel mezzo dell'esecuzione di una transazione. Ció garantisce che i comandi vengano eseguiti come un'unica operazione isolata.
  Se vogliamo avere un isolamento multi-transazionale, vi é un meccanismo che riesce a fornire delle garanzie, in cui viene fatta una sorta di operazione di check-and-set. Questo meccanisco utilizza il comando `WATCH` definito precedentemente. Le chiavi, su cui viene definito watch, vengono continuamente monitorate per eventuali modifiche; se anche una sola chiave monitorata da `WATCH` viene modificata prima della `EXEC`, l'intera transazione verrá abortita.

  Consideriamo un esempio in pseudo-codice in cui si deve aumentare il valore di una chiave di 1.

```
num = GET sampleKey
num = num + 1
SET sampleKey num
```

i comandi mostrati sopra funzioneranno senza problemi purché sia presente un solo utente che esegue l'operazione in un determinato momento.
Il problema si verifica nel caso in cui ci siano piú utenti che tentano di aumentare il valore della chiave contemporaneamente. Possiamo eliminare questo potenziale problema di race condition utilizzando il comando `WATCH` nel modo seguente:

```
1 WATCH sampleKey
2 num = GET sampleKey
3 num = num + 1
4 MULTI
5 SET sampleKey num
6 EXEC
```

Con questa implementazione, se si dovesse verificare una race condition ed un client modifica il valore di `sampleKey` tra il nostro `WATCH` e `EXEC`, la transazione verrá interrotta. Avremo bisogno di ripetere la transazione quando la race condition non sará piú presente.
Quindi questo é un modo efficace per ottenere un buon livello di isolamento nel caso di transazioni multiple.

- **consistenza**: I vincoli di integritá sono dei concetti relazionali, quindi é difficile fare un collegamento con un database di questo tipo. L'unica chiave che esiste é quella primaria e deve essere univoca; l'integrita referenziale non é mantenuta da Redis stesso e deve essere gestita dalle applicazioni client.

- **persistenza(durability)**: l'efficienza di Redis é dovuta in buona parte al suo modo di gestire questa proprietá. É un database in memoria ma con possibilitá di essere persistente su disco, quindi rappresenta un compromesso in cui si ottengono velocitá di scrittura e lettura molto elevate con la limitazione di avere un set di dati non piú grande della memoria. Questo database mette a disposizione la possibilitá di scegliere tra diversi meccanismi offrendo l'opportunitá di salvare database totalmente su disco oppure no.
I meccanismi, che verranno illustrati di seguito, sono:

  - **RDB**
  - **AOF**
  - **Database in Memory**

RDB → Redis Database File Questo tipo di persistenza esegue snapshot del set di dati a intervalli specificati. Viene prodotto come risultato un file

compatto, pertanto agevole da salvare su qualsiasi tipo di supporto. Inoltre, il recupero dei dati all'avvio del server Redis é molto efficiente. Il salvataggio dei dati viene eseguito su file ad intervalli di tempo e non con continuitá, quindi questo potrebbe essere un punto a sfavore nel caso di crash del sistema tra uno snapshot ed un altro con conseguente perdita dei dati. Conviene utilizzare questo tipo di persistenza quando si richiede un salvataggio meno oneroso per il server e si ha particolare interesse ad avere un backup piú comodo.

AOF → Append Only File é un meccanismo di persistenza che consente al server Redis di tenere traccia e registrare ogni comando eseguito dal server. Quindi vi é un file di log dove vengono aggiunti i comandi ogni volta che vengono eseguiti. Questo registro di comandi puó essere riprodotto all'avvio del server, ricreando il database al suo stato originale. Il vantaggio é che basandosi su un log scritto continuamente, non vi é il rischio di incorrere in perdite in caso di crash. Inoltre, il formato dei file che vengono prodotti da questa modalitá permette un recupero piú semplice in caso di corruzione. Peró, i file AOF risultano meno compatti e piú voluminosi rispetto a quelli in formato RDB e da ció consegue un ripristino del database meno rapido all'avvio. Questo meccanismo viene utilizzato quando la principale preoccupazione é la perdita di dati.

É possibile utilizzare contemporaneamente AOF e RDB, e durante il ripristino del database verrá preferito l'utilizzo di file AOF per la loro maggiore completezza.

Database In Memory é possibile rinunciare ad entrambi i meccanismi definiti precedentemente per dare vita ad un database in memory, risultando molto piú efficiente, poiché non deve piú occuparsi dei salvataggi su disco, e puó essere utilizzato per immagazzinare dati ad uso temporaneo la cui perdita non risulterebbe irreparabile per il sistema.
Redis puó anche essere utilizzato come memoria cache, in cui viene fissata la quantitá massima di memoria utilizzabile. Quando questa sará colma, i dati piú vecchi verranno eliminato con una politica LRU o LFU.

Come configurare i diversi livelli di persistenza?

Per fare ció bisogna accedere al file di configurazione del server Redis andando a modificare/cancellare dei parametri e riavviando il server, oppure digitando `CONFIG SET ...` da CLI con la possibilitá di avere un effetto immediato sulle modifiche

apportate alla configurazione del server senza doverlo riavviare.
Verrá illustrato un esempio di modifica del file di configurazione, denominato `redis.conf`.

RDB é l'impostazione predefinita. In particolare sono giá impostati i seguenti parametri di default:

```
save 900 1
save 300 10
```

Ció significa che viene eseguito uno snapshot dopo 900 secondi se vi é almeno 1 modifica al set di dati e dopo 300 secondi se vi sono almeno 10 modifiche al set di dati. Di conseguenza, se si ha la necessitá di avere intervalli aggiuntivi o diversi é molto semplice andare a modificarli aggiungendo o togliendo questi parametri predefiniti.

Al momento del salvataggio verrá visualizzato un messaggio di questo tipo nella CLI del server:

```
10 changes in 300 seconds. Saving...
Background saving started by pid ...
DB saved on disk
```

La strategia AOF viene configurata mediante due parole chiave:
`appendonly`, che se impostato a `yes` attiva AOF;
`appendfilename`, che specifica il nome del file in cui verranno salvate le operazioni.

Per avere un *database in memory* é sufficiente includere questi comandi:

```
save ""
appendonly no
```

3.1.3 Ambiti di utilizzo

Redis é estremamente flessibile e grazie alla sua efficienza puó essere applicato a casi d'uso molto diversi tra loro:

- **Analisi in tempo reale**: viene utilizzato come datastore in memoria per acquisire, elaborare e analizzare dati in tempo reale con latenze molto basse, infatti puó essere utilizzato in modo estremamente efficace in ambito **IoT**. Si puó immaginare una rete di sensori che invia dati in maniera continua, questi dati vanno memorizzati ed elaborati con una bassa latenza. Queste sono proprio le caratteristiche che un dbms come Redis offre. Verrá analizzato un caso reale nel capitolo 5, in cui verrá mostrato un modo in cui Redis puó essere utilizzato proprio con questo particolare caso d'uso;

- **Chat, messaggistica e code**: grazie alle strutture dati che offre, come le liste e le hash, e strutture dati aggiuntive come pub/sub (in cui vi sono diversi publisher e subscriber) puó essere utilizzato per la messaggistica. Infatti vengono offerte prestazioni elevate per chat e flussi di commenti in tempo reale;

- **Classifiche di videogiochi**: é un servizio molto utilizzato per la creazione di classifiche in tempo reale. Infatti, é sufficiente utilizzare la struttura dati SortedSet per ottenere un elenco ordinato in base ai punteggi degli utenti. In questo modo, la classifica viene aggiornata simultaneamente alla variazione dei punteggi dei giocatori. Inoltre, SortedSet potrebbe venire utilizzata anche per gestire serie temporali utilizzando timestamp come punteggio;

- **Memorizzazione**: soluzione molto utilizzata nel caso in cui occorre memorizzare e gestire dati di sessione per applicazioni su Internet, ad esempio profili utente, credenziali, stati di sessione e personalizzazioni specifiche per ciascun utente. Inoltre, é ideale anche per lo streaming di contenuti multimediali in tempo reale, in particolare per memorizzare metadati di profili utente e cronologie di visualizzazione, informazioni di autenticazione per milioni di utenti e file manifest con cui permettere la distribuzione di contenuti a milioni di utenti contemporaneamente;

- **Dati Geospaziali**: viene offerta una struttura per gestire dati geospaziali reali, chiamata `Geospatial`, su vasta scala e con la massima rapiditá. Praticamente é una SortedSet con uno score che viene calcolato in base alle coordinate che vengono assegnate ad un certo membro. Vengono offerti diversi comandi, tra cui `GEOADD` che permette di aggiungere elementi ad un indice geospaziale. Infatti, se consideriamo un esempio in cui stiamo tracciando un gruppo di auto basterá fare nel modo seguente:

```
> GEOADD auto -115.17087 36.12360 auto-p1
> GEOADD auto -115.171971 36.120609 auto-p2
```

  Per aggiornare la posizione dell'auto andrá eseguito un nuovo `GEOADD` sulla stessa auto.
  É possibile determinare la distanza in metri tra diverse auto con il comando `GEODIST`:

```
> GEODIST auto auto-p1 auto-p2
  "347.0365"
```

  Quindi, si puó vedere da questo semplice esempio il grande potenziale che ha questa struttura dati.

- **Machine Learning**: le moderne applicazioni basate sui dati necessitano di apprendimento automatico e quindi devono analizzare ed elaborare in modo rapido grandissime quantitá di dati di vario genere e con varie frequenze di aggiornamento. Queste caratteristiche si adattano perfettamente a Redis.

3.1.4 sistema distribuito

Redis supporta una distribuzione di vari processi server su piú macchine con la possibilitá di collaborazione tra di loro. Oltre alle soluzioni proprietarie fatte su misura in base al caso d'uso specifico, Redis offre la possibilitá di implementare lato server un sistema distribuito in modo piuttosto semplice ed efficiente.
vi sono due tipologie di sistemi distribuiti che é possibile implementare:

- **Redis Master-Slave**

- **Redis Cluster**

Per sistemi estremamente avanzati, vi é la possibilitá di fondere queste due modalitá.

Redis Master-Slave

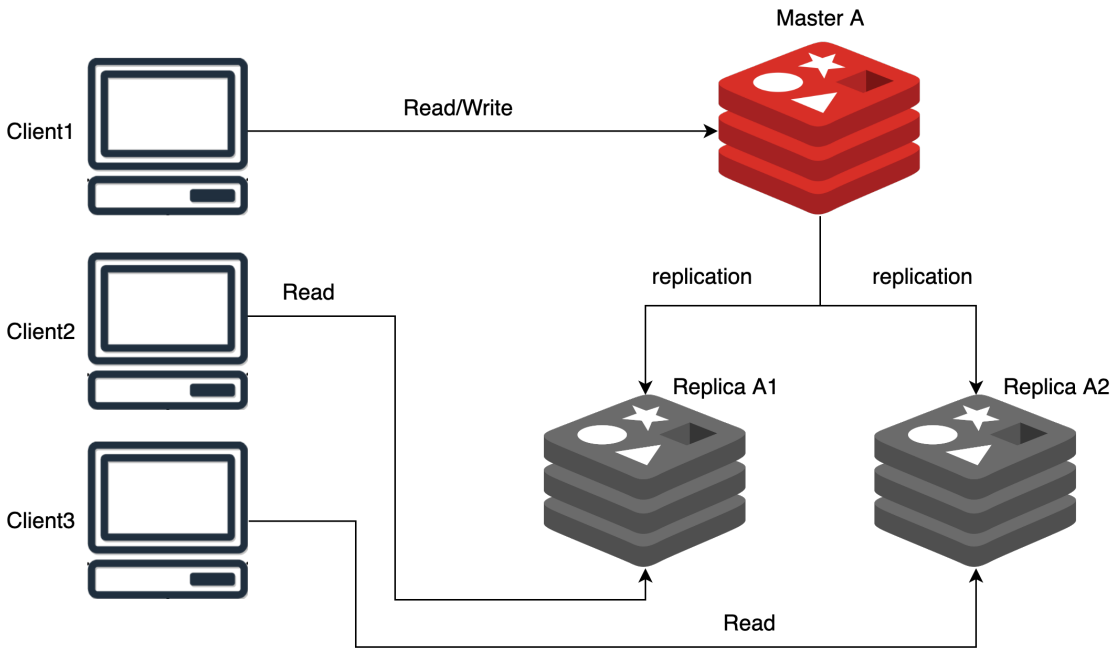
Redis puó implementare un'architettura master-slave, ovvero un noto paradigma informatico in cui un dispositivo o processo (il master) controlla o coordina piú dispositivi o processi subordinati (gli slave).

il server Redis puó essere eseguito in due modalitá:

- Modalitá Master (Redis Master);

- Modalitá Slave (Redis Slave o Redis Replica).

Redis Master funge da interfaccia con il mondo esterno, gestendo tutte le richieste scrittura esterne e puó gestire anche quelle di lettura; ogni volta che viene apportata una modifica al database master, la modifica viene propagata ai database slave collegati al master.



La propagazione della replica puó avvenire in due modi:

- sincrona: le modifiche ai database slave avvengono apportate istantaneamente;
- asincrona: le modifiche vengono apportate solo a distanza di tempo, é l'impostazione predefinita poiché si adatta alla maggior parte dei casi d'uso di Redis.

La replica master-slave é in gran parte non bloccante, il che significa che il database master puó continuare a funzionare mentre i database slave sincronizzano i dati. I Redis Slave saranno in grado di gestire le query utilizzando la versione non aggiornata del database, tranne che per un breve periodo durante il quale vengono caricati i nuovi dati.

Come avviene il failover nel caso di guasto di Redis Master?
vi sono due scelte:

1. aggiungi una nuova macchina come Redis Master;
2. rendi qualsiasi Redis Slave esistente come nuovo Redis Master.

Il problema con l'approccio **1** é che nel momento in cui aggiungiamo una nuova macchina che fa da Master e questa sincronizzerá tutti i dati sui vari Slaves perderemo

tutti i dati.
L'approccio **2** é quello migliore perché lo slave esistente avrá giá tutti i dati e una volta che lo avremo impostato in modalitá Master, esso replicherá/sincronizzerá i dati su tutti gli Slaves, il che significa che non avremo una perdita di dati, o comunque avremo una minima perdita di dati nel caso in cui il guasto del Master si sia verificato prima della sincronizzazione con gli slaves.

Invece, nel caso di crash di Redis Slave le sue richieste di lettura saranno semplicemente sostituite da altri Redis Slave.

Teorema CAP

Formulato da Eric Brewer nel 1998, afferma che se i dati sono in un sistema i distribuito tra i nodi di una rete, solo due delle seguenti proprietá possono essere soddisfatte contemporaneamente:

- **Consistency**: le operazioni di lettura restituiscono il dato aggiornato, ovvero proveniente dall'ultima scrittura dello stesso, oppure un messaggio d'errore. Non bisogna confonderla con la consistenza delle proprietá ACID, infatti questa fa riferimento al valore piú aggiornato di un certo dato, mentre nelle proprietá ACID si fa riferimento al rispetto dei vincoli d'integritá.

- **Availability**: l'accesso ai dati é sempre garantito, non si ricevono mai messaggi d'errore, ma i dati restituiti non sono necessariamente consistenti, ovvero potrebbero non coincidere con quelli utilizzati nell'ultima operazione di scrittura degli stessi.

- **Partition Tolerance**: il sistema continua a funzionare nonostante alcuni messaggi vengano persi o subiscano rallentamenti nelle comunicazioni tra i nodi della rete.

É un teorema che viene utilizzato per analizzare le dinamiche dei sistemi informativi che possiedono due o piú archivi di dati posti su nodi distinti della rete. Quindi rientra perfettamente in questa categoria Redis.

Redis é un sistema **AP**, ovvero non fornisce una forte coerenza.

Per quale motivo?

Quando Redis Master riceve una richiesta di scrittura da un cliente:

1. Esegue la richiesta del cliente e manda un ack di conferma;

2. Redis Master replica la richiesta di scrittura su uno o piú slave.

Redis Master non attende il completamento della replica sugli slave, ma esegue prima la richiesta del client. Se supponiamo che il guasto del Master avvenga prima del completamento della replica agli Slave avremo una perdita di coerenza.

Potremmo pensare di migliorare la coerenza con la replica sincrona, forzando prima il Redis Master a replicare e poi mandare l'ack di conferma al client, ma questo riduce pesantemente le prestazioni di scrittura e comunque non si ha una garanzia di consistenza. Infatti, potrebbe verificarsi uno scenario in cui uno slave non ha ricevuto la scrittura e viene promosso a Master.

Redis Cluster

Uno dei piú grandi limiti di Redis, essendo un database in-memory, é la limitazione della memoria che un'istanza di Redis puó avere, in quanto tutto il set di dati archiviato non deve mai superare la dimensione massima.

Vi é la possiblitá di dividere i dati in diverse istanze eseguite su piú server, in modo che ogni istanza contenga solo un sottoinsieme delle chiave; tutto questo avviene tramite l'operazione di **sharding**, ovvero il partizionamento orizzontale.

Vi sono dei compromessi da considerare: suddividendo i dati in molte istanze, nasce il problema della ricerca delle chiavi, quindi i dati devono essere partizionati seguendo alcune regole coerenti.
Sono possibili diverse implementazioni per il partizionamento dei dati:

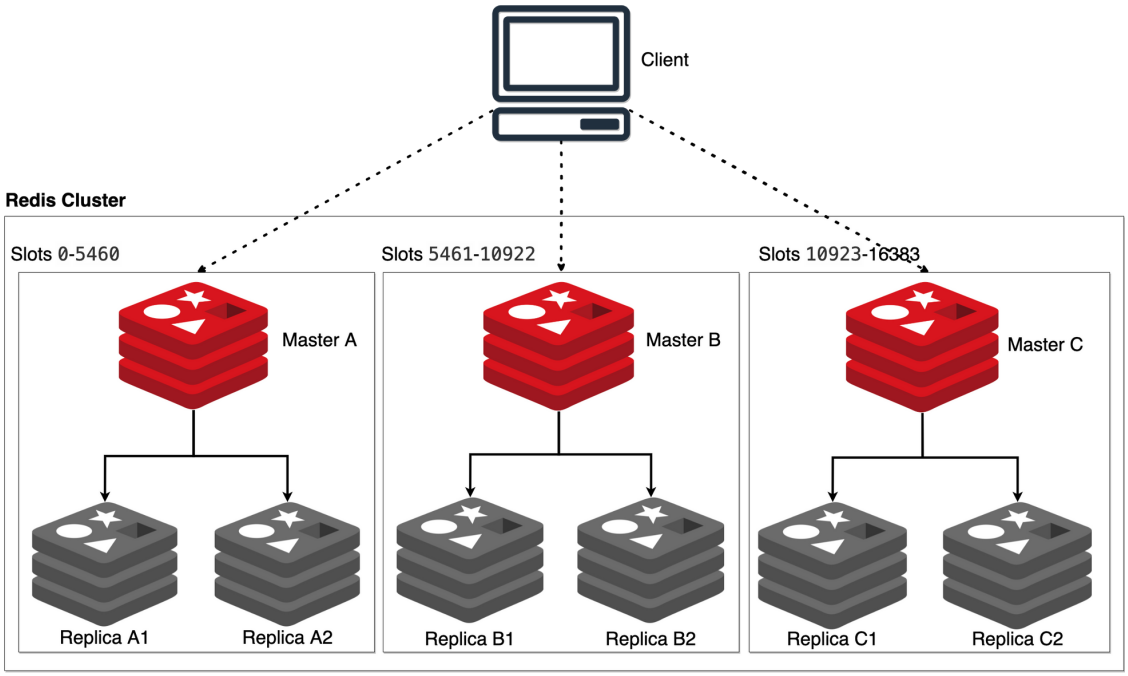
- **partizionamento lato client**: i client selezionano direttamente l'istanza corretta per scrivere o leggere una determinata chiave;

- **partizionamento assistito da proxy**: i client inviano le richieste a un proxy che supporta il protocollo Redis, invece di inviare le richieste direttamente alle istanze Redis corrette. Il proxy si assicurerá di inoltrare le richieste alle istanze corrette in base allo schema di partizionamento configurato e invierá le risposte ai client. Il limite principale di questa implementazione é che il proxy puó diventare un "collo di bottiglia", poiché tutte le richieste e risposte passano per esso.

- **instradamento della query**: i client inviano la query a un'istanza Redis casuale e l'istanza si assicurerá di inoltrare la query a quella corretta. Il client successivamente dovrá essere reindirizzato all'istanza corretta.

In realtá, Redis Cluster utilizza una **forma ibrida di instradamento della query**, in cui il nodo casuale che viene interrogato non inoltra la query al nodo corretto, ma reindirizza solo il client. I client alla fine ottengono una mappatura completa di quali nodi servono quali sottochiavi e possono contattare direttamente i nodi corretti.

Lo sharding dei dati in molte istanze non risolve il problema della sicurezza dei dati e della ridondanza. Se una delle istanze muore a causa di un guasto hardware e non si hanno backup da cui ripristinare i dati, tutti i dati di quella istanza saranno persi. Per ridurre questo problema si potrebbero utilizzare diverse tecniche, come la replicazione su disco oppure l'architettura master-slave.
Inoltre, le transazioni relative a piú chiavi distribuite su piú istanze non sono supportate, poiché richiederebbero lo spostamento dei dati tra diversi nodi, con conseguente calo delle prestazioni.



La figura sopra riportata mostra un sistema avanzato in cui si ha una combinazione di Redis Cluster e Redis Master-Slave.

Capitolo 4

Interrogazioni Redis

Le capacitá piú essenziali per il lavoro su database sono saperli popolare di informazioni in modo coerente e interrogare in modo intelligente, per ricavare l'informazione che si sta cercando in mezzo a volumi di dati che possono essere immensi. Quindi, per raggiungere questo obiettivo adoperiamo un linguaggio per l'interrogazione dei database, il quale serve per formulare delle query, in modo adeguato alla struttura delle informazioni. Nei database di tipo relazionale é presente il linguaggio SQL (Structured Query Language), il quale é comune a tutti i dbms di questo tipo, quindi é diventato negli anni un vero e proprio standard.

nei NoSQL vi é un linguaggio standard da adoperare?

i NoSQL, di conseguenza anche Redis, si caratterizzano proprio da schemi non fissi, che possono variare in modo dinamico e, soprattutto, non vi sono vincoli referenziali che possono associare diverse tabelle all'interno del database (quindi non vi sono operazioni di join). Per memorizzare e ricavare dati, i quali possono essere strutturati, semi-strutturati, non strutturati e polimorfici, un dbms noSQL utilizza un'ampia gamma di tecnologie proprietarie in grado di fare questo. Quindi, é compito del produttore del dbms di dover mettere a disposizione un insieme di comandi che permettano di sfruttare pienamente le potenzialitá del NoSQL da lui fornito. Ne deriva che il linguaggio SQL non é supportato; sono disponibili delle interfaccie di comunicazione per i principali linguaggi di programmazione che vengono offerte direttamente al programmatore, con le quali si riesce a gestire il dialogo con il dbms.

Sulla documentazione di Redis é presente un elenco completo di tutte le API disponibili per i principali linguaggi, tramite le quali si riesce a comunicare con il server in questione.

Il modo piú rapido e semplice per interagire con il server é da linea di comando, infatti esiste una libreria chiamata `redis-cli`, con la quale viene semplificato notevolmente il lavoro di hacking. In questo capitolo per fare un confronto tra comandi Redis e SQL adopereremo questa libreria.

Sappiamo che il linguaggio SQL é diviso in due sezioni principali:

- Data Definition Language
- Data Manipulation Language

Per quanto riguarda il Data Definition Language, a differenza di un database relazionale, non abbiamo la possibilitá di definire domini, tabelle, vincoli sulle strutture e cosí via; non avremo a disposizione nessuna operazione di questo tipo.
Redis essendo un key-value é composto da tuple, quindi, viene creato un legame tra chiave e valore al momento dell'aggiunta di una chiave; inoltre, il tipo del valore associato sará definito al momento dell'aggiunta.

4.1 Data Manipulation Language

Come nel linguaggio SQL si hanno delle operazioni di query, di modifica e addirittura anche comandi transazionali.

4.1.1 Comandi di Modifica

come in SQL i comandi di modifica sono quelle istruzioni che permettono:

- inserimento;
- cancellazione;
- modifica dei valori associati a delle chiavi.

Come esempio di confronto adoperiamo un database che rappresenta i profili di utenti. Un utente é composto da: nome utente, nome, cognome, password, data di creazione.

In Redis avremo una tupla cosí composta:
chiave → **utente:**nomeutente

valore → tipo **hash**, in cui saranno contenute le informazioni
I comandi che verranno analizzati sono associati al tipo Hash, in quanto Redis dedica dei comandi specifici per ogni struttura dati.

Nel database relazionale avremo un record composto da tutti gli attributi sopra definiti, la chiave primaria sará rappresentata dallo username;

L'**INSERIMENTO** di un record con linguaggio SQL viene eseguito in questo modo:

```
1 INSERT INTO Utenti (username, nome, cognome, password, dataCreazione)
2 VALUES ('matteo00', 'matteo', 'rizzo', 'mypsw', '12092020')
```

Analizziamo una possibile aggiunta in Redis: Se utilizziamo una Hash come valore, il comando di cui abbiamo bisogno é `HSET`, il quale viene utilizzato per aggiungere una nuova tupla.

```
1 >HSET utente:matteo00 nome matteo cognome rizzo password mypsw dataCreazione
      12092020
2 (integer) 4
```

Si puó notare che in Redis non abbiamo definito uno schema fisso della hash, ma siamo noi al momento della creazione della tupla a definire le chiavi e i valori che compongono la hash, infatti potremo avere utenti con una hash con schema non coerente alle altre, questo ha, ovviamente, i suoi vantaggi e svantaggi.

Per la **CANCELLAZIONE** di un record in SQL facciamo:

```
1 DELETE FROM Utenti WHERE username = 'matteo00'
```

Mentre in Redis dobbiamo eliminare una chiave; questo comando non dipende dal tipo di valore utilizzato, ma é un comando comune a tutti i tipi, il comando si chiama `DEL`.

```
1 >DEL utente:matteo00
2 (integer) 1
```

In Redis questo approccio é poco utilizzato, solitamente viene impostata una scadenza alle chiavi, ovvero dopo un certo periodo le chiavi vengono eliminate automaticamente. Per fare questo si utilizza il comando `EXPIRE`, in cui viene passato come parametro la chiave, a cui va impostata la scadenza definendola in secondi.

```
1 >EXPIRE utente:matteo00 60
2 (integer) 1
```

Il valore 60 indica i secondi di tempo dopo i quali la chiave deve essere cancellata.

Per **MODIFICARE** i valori degli attributi di uno o piú record tramite SQL:

```
UPDATE Utenti
SET password = 'newpsw'
WHERE username = 'matteo00'
```

In Redis non vi é un comando specifico per la modifica dei valori delle tuple, ma viene utilizzato `HSET` passando come parametri del comando i campi che vogliamo modificare con il loro nuovo valore:

```
>HSET utente:matteo00 password newpsw
(integer) 4
```

Cosí facendo abbiamo modificato il valore associato alla password di `utente:matteo00`, il valore passa da `"mypsw"` a `"newpsw"`.

4.1.2 Comandi di Query

In SQL le interrogazioni hanno una struttura `select-from-where`, che puó essere estesa in diversi modi. Sono comandi che possono anche avere una certa complessitá semantica, questo é dovuto principalmente dal fatto che si possono eseguire dei join tra diverse tabelle presenti nel database.
Questo non avviene in Redis, infatti le interrogazioni di una base di dati chiave-valore non possono avere una complessitá paragonabile a quella SQL.

Per fare una selezione di un record in SQL:

```
SELECT *
FROM Utenti
Where username = 'matteo00'
```

Mentre in Redis per selezionare il valore di una certa tupla, nel caso in cui sia di tipo Hash, viene messo a disposizione `HGETALL`, che ha il compito di mostrare tutto il contenuto associato ad una certa chiave.

```
> HGETALL utente:matteo00
1) "nome"
2) "luca"
3) "cognome"
4) "rizzo"
5) "password"
6) "mypsw"
7) "dataCreazione"
8) "12092020"
```

Inoltre, viene messo a disposizione il comando `HGET` per ottenere solo il valore di un campo tra quelli presenti nella Hash, questo viene fatto per permettere agli sviluppatori di risparmiare operazioni di trasformazione delle strutture all'interno del proprio software.

```
> HGET utente:matteo00 nome
"matteo"
```

Vi é anche la possibilitá di fare una sorta di proiezione sulle chiavi utilizzando il comando `KEYS`, il quale restituisce tutte le chiavi che corrispondono al pattern passato come parametro.

```
> KEYS utente:*
1) "utente:user1"
2) "utente:user2"
3) "utente:user3"
4) "utente:matteo00"
```

Comando che viene ampiamente utilizzato perché é possibile filtrare su chiavi che hanno un prefisso particolare.

Per quanto riguarda i comandi transazionali, sono giá stati definiti nel capitolo precedente nella sezione del confronto con le proprietá ACID.

4.2 Store procedure

In SQL vi é la possibilitá di definire procedure che vengono memorizzate nella base di dati come parte dello schema ed eseguite dal dbms, ovvero con una modalitá opposta rispetto all'esecuzione dei client. Queste procedure svolgono una specifica attivitá di manipolazione dei dati.
Anche Redis offre un meccanismo analogo: viene consentito ai client di caricare ed eseguire script lato server. Il vantaggio é la grande efficienza che si ottiene nella lettura e scrittura dei dati poché gli script memorizzati sono eseguiti completamente server-side. É garantita l'esecuzione atomica dello script e durante la sua esecuzione tutte le attivitá del server vengono bloccate. É supportato un unico motore di scripting, ovvero l'interprete `LUA`, che é un particolare linguaggio di scripting.

Ci sono diverse modalitá lato client per invocare/caricare gli script:

- Il comando `EVAL` serve per eseguire direttamente uno script; vanno forniti diversi argomenti:
  Il primo argomento é una stringa che consiste nel codice sorgente LUA vero e proprio;
  il secondo argomento indica il numero di chiavi che verranno passate nei parametri successivi (quindi é un numero);
  a partire dal terzo argomento si indicano i nomi delle chiavi di Redis.

Un esempio di codice puó essere il seguente:

```
1 > EVAL "return {KEYS[1], KEYS[2]}" 2 key1 key2
2 1) "key1"
3 2) "key2"
```

Con questo comando eseguiamo gli script dinamicamente, perché stiamo appunto generando lo script durante il runtime dell'applicazione.

- É possibile caricare uno script nella cache del server Redis tramite il comando `SCRIPT LOAD` e fornendo il suo codice sorgente; in questo modo il server non esegue lo script, ma lo compila e lo carica nella sua cache e restituisce al client un codice, chiamato **SHA1 digest**, che punta direttamente allo script. Successivamente lo script viene eseguito invocando il comando `EVALSHA` e passandogli come primo argomento il SHA1 digest generato in precedenza e partendo dal secondo argomento si ripetono quelli di `EVAL`.

```
1 > SCRIPT LOAD "return 'example of script load!'"
2 "c664a3bf70bd1d45c4284ffebb65a6f2299bfc9f"  \\SHA1 digest
3 >EVALSHA "c664a3bf70bd1d45c4284ffebb65a6f2299bfc9f" 0
4 "example of script load!"
```

La cache degli script Redis é sempre volatile, non é considerato come parte del database e non é persistente, quindi gli script memorizzati sono temporanei e il contenuto della cache puó essere perso in qualsiasi momento.

Capitolo 5

Caso Concreto - IoT sensore Temperatura

implementazione librerie Redis in linguaggi di programmazione Parlare di Jedis, libreria Java.

Riportare esempi di codice che implementa DataBase.
Descrivere progetto IoT fatto con sensore di temperatura DHT11. utilizzata scheda ESP32 che comunica con database e manda dati a redis server di rilevazione temperatura ogni 5 secondi.
inserire parti di codice sketch arduino per far vedere come comunica con database.

Illustrare software fatto con Java che preleva il dataset da redis server e lo elabora creando un grafico che si aggiorna in tempo reale.

Bibliografia

Bibliografia