UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

# Relazione Finale/Tesi di laurea
# Studio e analisi di una base di dati NoSQL per la sensoristica

**Relatore:** Chiar.mo Prof. Michele Melchiori

Laureando:
Matteo Rizzo
Matricola n. 727499

---

Anno Accademico 2021/2022

# Sommario

# Ringraziamenti

# Indice

# Introduzione

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

# 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 *NoSQL*?

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 dovrei 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.

- `Flessibilitá`: 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 dei dati;

- `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.

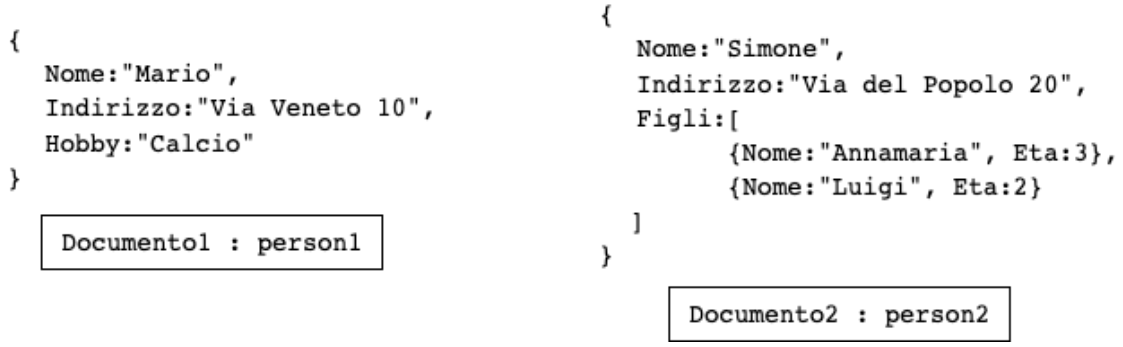
- `Altamente funzionali`: non vi é piú un linguaggio generale (SQL) come nei database relazionali, ma vi sono API in base al database specifico che si va ad utilizzare.

## 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.

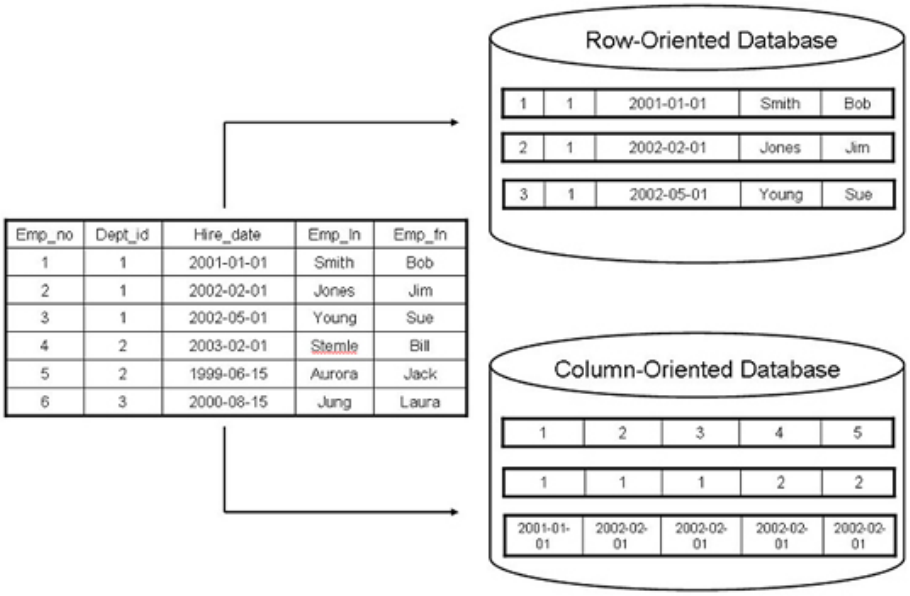



- `key-value`: i dati vengono immagazzinati mediante un semplice metodo chiave-valore. Una chiave rappresenta un identificatore univoco. Le chiavi e i valori possono essere qualsiasi cosa, da un oggetto semplice ad articolati oggetti composti. (Questo tipo di base di dati é oggetto di tesi e quindi verrá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

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

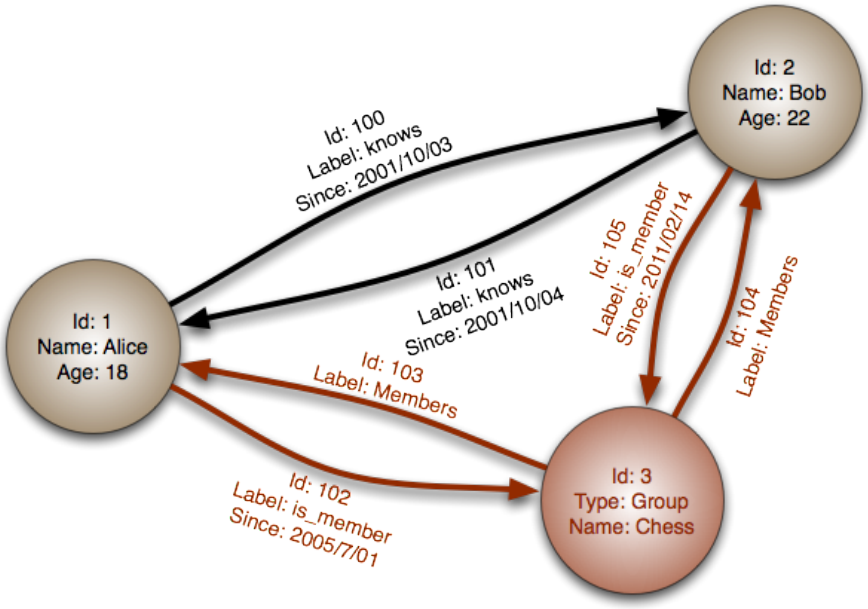
- `colonnari`: i dati vengono archiviati per colonne, anziché per righe come avviene nei database relazionali classici. Queste colonne vengono raccolte per formare dei sottogruppi. Le chiavi e i nomi delle colonne di questo tipo di database non sono fissi. Ogni colonna é memorizzata separatamente. Se 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.


Id: 2
Name: Bob
Age: 22
Id: 100
Label: knows
Since: 2001/10/03
Id: 101
Label: knows
Since: 2001/10/04
Id: 105
Label: is_member
Since: 2011/02/14
Id: 104
Label: Members
Id: 1
Name: Alice
Age: 18
Id: 103
Label: Members
Id: 102
Label: is_member
Since: 2005/7/01
Id: 3
Type: Group
Name: Chess

# Capitolo 2

# Progettazione Concettuale/Logica

Sebbene i database *NoSQL* vengono 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 selezionato. 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. 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

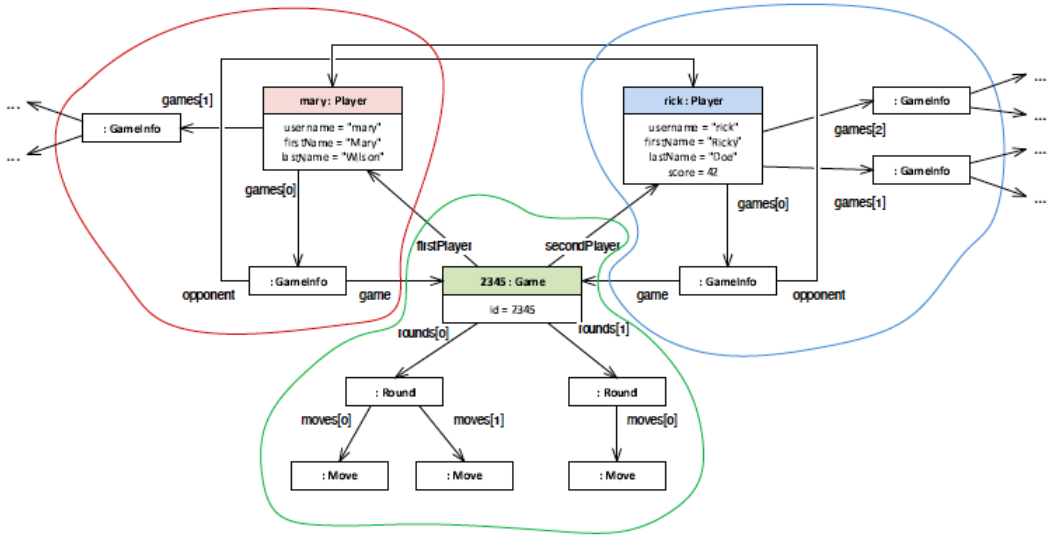
- Entitá e oggetti di valore vengono raggruppati in *aggregati*. Un aggregato ha un'entitá come radice e puó contenere molti oggetti di valore.

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.

### 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. Vi sono due nozioni distinte di *unitá* di accesso ai dati:

- `blocco`, unitá di dimensione maggiore, ha massima consistenza
- `entry`, unitá di dimensione minore, che permetto 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.

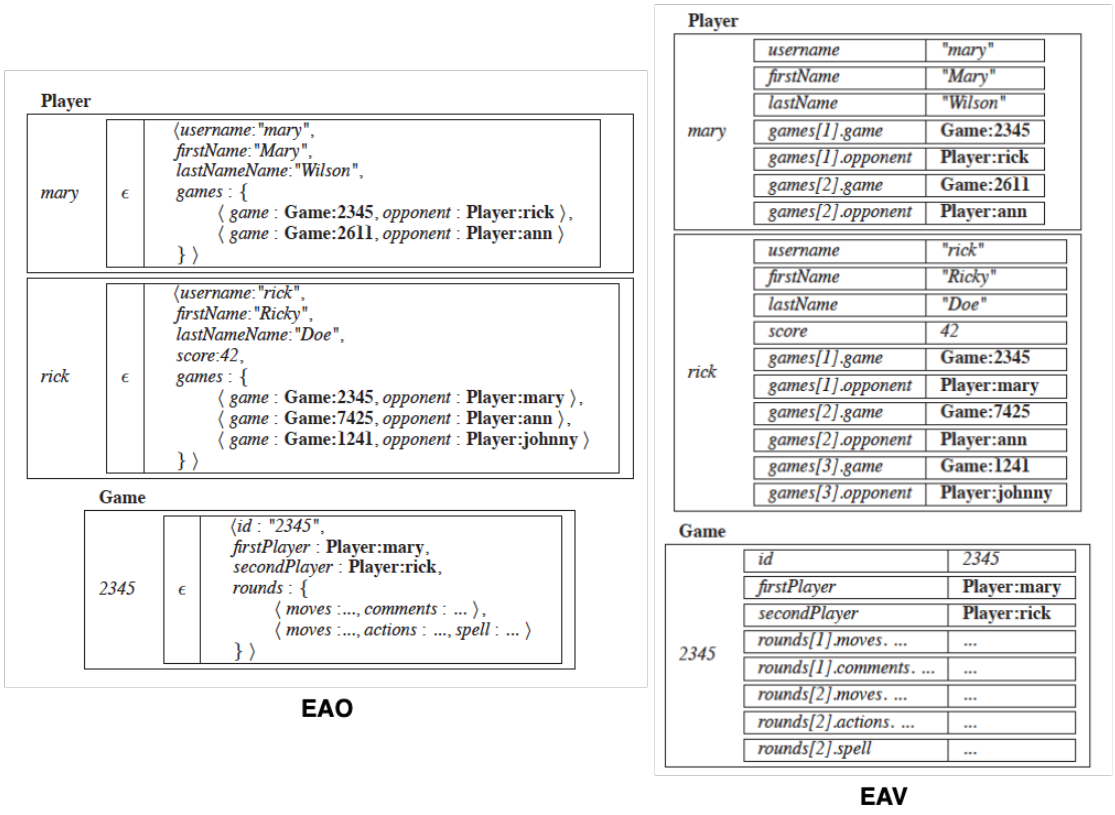
Quindi, si ha che: Un `database` é un insieme di `collections`. Ogni `collection` ha un nome distinto; Una `collection` é un insieme di `blocchi`, ogni `blocco` all'interno della `collection` é identificato da una chiave di blocco, che deve essere

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`.

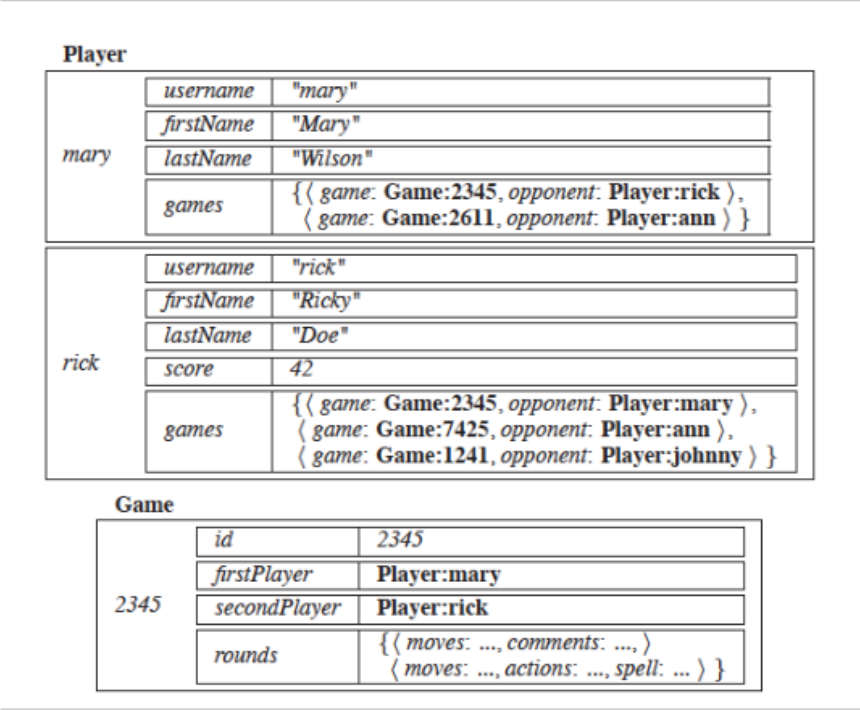
Vi sono due modalitá principali per rappresentare gli aggregati:

- *Entry per Aggregate Object (EAO)*: Rappresenta ogni aggregato utilizzando una singola `entry`, la chiave della `entry` é vuota, il valore contiene l'intero aggregato.
- *Entry per Atomic Value (EAV)*: Rappresenta ogni aggregato per mezzo di piú `entry`, le chiavi di ogni `entry` devono rappresentare il percorso per accedere al valore di un certo componente(quindi possono esserci anche chiavi con nomi strutturati a piú livelli), i valori di ogni `entry` sono atomici.



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.

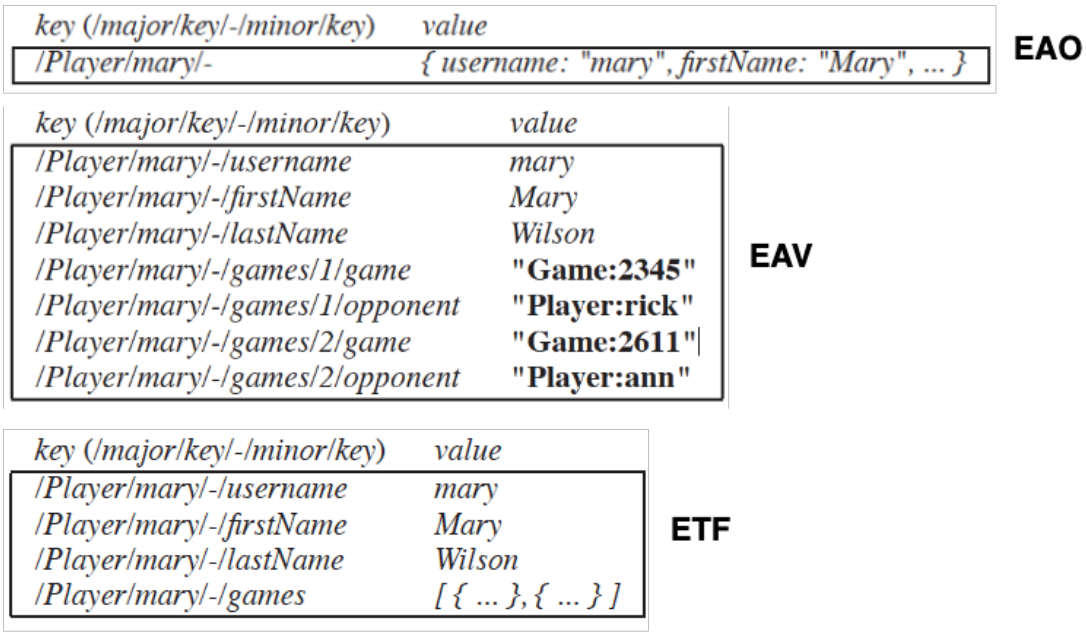



**ETF**

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 database 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), praticamente bisogna decidere che livello di complessitá vogliamo avere sulle chiavi e sui valori.

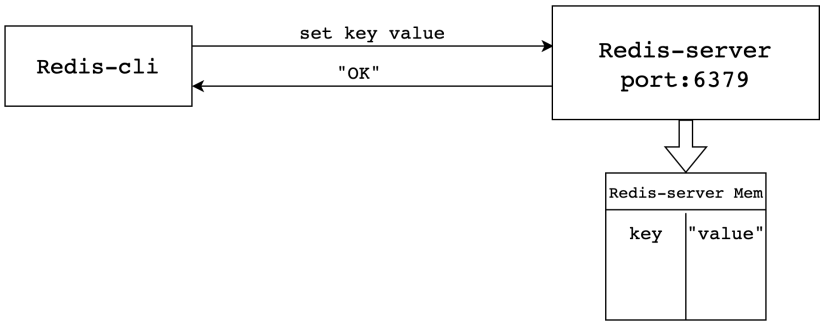


# Capitolo 3

# 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. Inoltre, é possibile interagire con il server anche da linea di comando con un programma chiamato `redis-cli`, grazie ad esso viene semplificato notevolmente il lavoro di hacking con il sistema.

```
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.

Ovviamente associati a queste strutture dati vi sono molti comandi specifici per ognuna, analizzare tutti i comandi non ha molto senso, quindi ogni volta che verranno incontrati dei comandi particolari verranno illustrati in quel momento.

### 3.1.2 *proprietá ACID*

Nelle basi di dati relazionali ogni transazione gode delle proprietá ACID. in questa sezione l'obiettivo é mettere in evidenza quali proprietá vengono verificate da questo dbms, se c'é la possibilitá di abilitare/disabilitare certe proprieta e cosí via.

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, quindi i comandi per gestire le liste in questo esempio sono 2: `LPUSH` comando per inserire un singolo elemento nella lista e `LRANGE` comando per ottenere tutti i valori presenti nella lista

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> LPUSH listaNumeri 3
QUEUED
127.0.0.1:6379(TX)> LPUSH listaNumeri 10
QUEUED
127.0.0.1:6379(TX)> LPUSH listaNumeri 34
QUEUED
127.0.0.1:6379(TX)> LPUSH listaNumeri 45
QUEUED
127.0.0.1:6379(TX)> EXEC
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) (integer) 4


127.0.0.1:6379> 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`.

Quindi, passando all'illustrazione delle proprietá ACID:

- **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 vari comandi `MULTI`, `EXEC` ...;
- **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.

Inoltre, vi é un meccanismo che riesce a fornire delle garanzie aggiuntive, 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.

```
1       num = GET sampleKey
2       num = num + 1
3       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 a livello di transazioni.

- **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.

### 3.1.3 Ambiti di utilizzo

### 3.1.4 sistema distribuito (cenni)

# Capitolo 4

# Interrogazioni SQL $\rightarrow$ *Key-Value*

parlare del tipo di interrogazioni che vengono fatte, c'é un vero e proprio linguaggio strutturato come SQL?

# 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