



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

Questa relazione presenta una panoramica delle basi di dati non relazionali NoSQL, compiendo un passo verso una progettazione concettuale/logica generalizzata ed indipendente da dbms concreti. Dal capitolo 3 si vuole entrare piú nel dettaglio con lo studio di un database key-value molto particolare: Redis. In questo capitolo e in quello successivo si vogliono mostrare le principali peculiarit , somiglianze e differenze che si hanno rispetto ai database tradizionali. Nell'ultimo capitolo sono stati messi in pratica gli studi fatti nei capitoli precedenti, sono stati creati diversi applicativi che sfruttano le potenzialit  di Redis in un ambito che si sta espandendo sempre piú negli ultimi anni: l'Industrial of Thing.

Indice

1	DataBase NoSQL	1
1.1	Perche é nato?	1
1.2	Perché utilizzare un database <i>NoSQL</i> ?	2
1.2.1	Vantaggi	2
1.2.2	Limiti	2
1.3	Tipologie	3
2	Progettazione Concettuale/Logica	7
2.1	Modellazione NoAM → NoSQL abstract model	7
2.1.1	Modellazione Concettuale e design degli Aggregati	8
2.1.2	Partizionamento degli Aggregati e modellazione NoSQL	9
2.1.3	Implementazione	12
3	Redis → Remote Dictionary Server	15
3.1	Caratteristiche	16
3.1.1	Strutture Dati	16
3.1.2	Confronto con <i>proprietá ACID</i>	17
3.1.3	Ambiti di utilizzo	21
3.1.4	sistema distribuito	23
4	Interrogazioni Redis	29
4.1	Data Manipulation Language	30
4.1.1	Comandi di Modifica	30
4.1.2	Comandi di Query	32
4.2	Store procedure	33
5	Caso Concreto - IoT sensore Temperatura	35
5.1	Scelte di progetto	35
5.2	Arduino e DHT11	37

5.2.1	Connessione con Redis	38
5.2.2	NTP	39
5.2.3	Invio dataSet	39
5.3	Java	41
5.3.1	Jedis e RedisTimeSeries	41
5.3.2	DataSet RealTime	43
5.3.3	JFreeChart	45
Bibliografia		47

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 scalabilità 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 più flessibile alla memorizzazione dei dati rispetto a un database relazionale, un controllo migliore e una maggiore semplicità nelle applicazioni.

1.1 Perché è nato?

A partire dagli anni 2000 si è 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 così 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 più veloce. Quindi è una soluzione ideale per i dati semi-strutturati e non strutturati. È possibile arricchire le applicazioni di nuovi dati e informazioni senza dover sottostare ad una rigida struttura;
- **Scalabilità:** grazie alla semplicità vi è la possibilità 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 è la possibilità 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 velocità computazionale;
- **Riduzione dei tempi di sviluppo:** grazie alla definizione di logiche di lettura dati molto più semplici rispetto a quelle da scrivere con database relazionali.

1.2.2 Limiti

La semplicità di questi database comporta anche degli svantaggi:

- **Integrità:** mancando i controlli fondamentali sull'integrità dei dati, il compito ricade totalmente sull'applicativo che dialoga con il database;
- **Scrittura:** problema strettamente collegato all'integrità, poiché ogni volta che si deve aggiornare un dato ridondato in più entità diventa necessario aggiornare il dato su tutte le entità in cui è 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;
- **Espressività:** problema legato ai linguaggi di interrogazione, infatti risulta essere meno espressivo rispetto a SQL nella grande maggioranza dei casi, portando ad una maggiore difficoltà di correzione massiva (data fixing), report ed export dei dati.

1.3 Tipologie

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 proprietà 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 flessibilità 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.

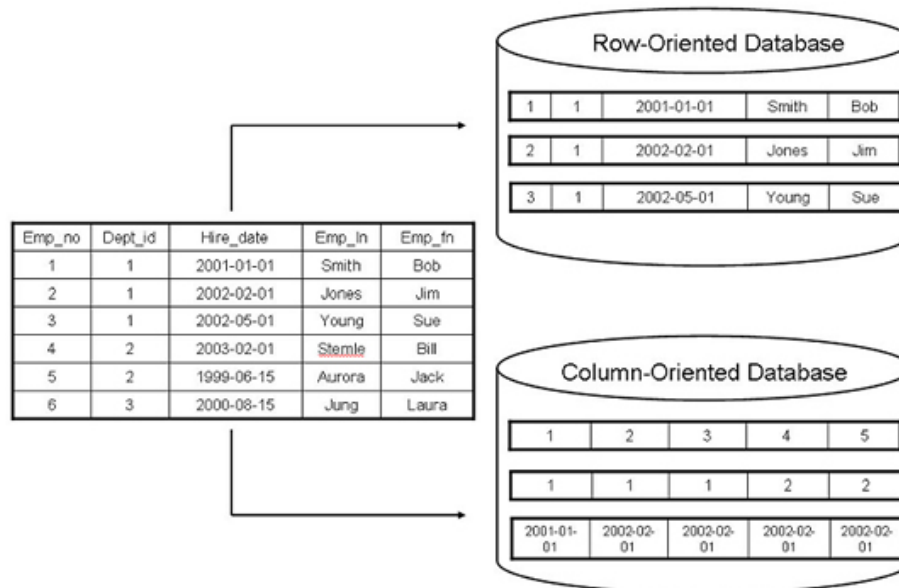
<pre>{ Nome:"Mario", Indirizzo:"Via Veneto 10", Hobby:"Calcio" }</pre> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin-top: 10px;"> Documento1 : person1 </div>	<pre>{ Nome:"Simone", Indirizzo:"Via del Popolo 20", Figli:[{Nome:"Annamaria", Eta:3}, {Nome:"Luigi", Eta:2}] }</pre> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin-top: 10px; float: right;"> Documento2 : person2 </div>
---	--

- **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 verrà 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.

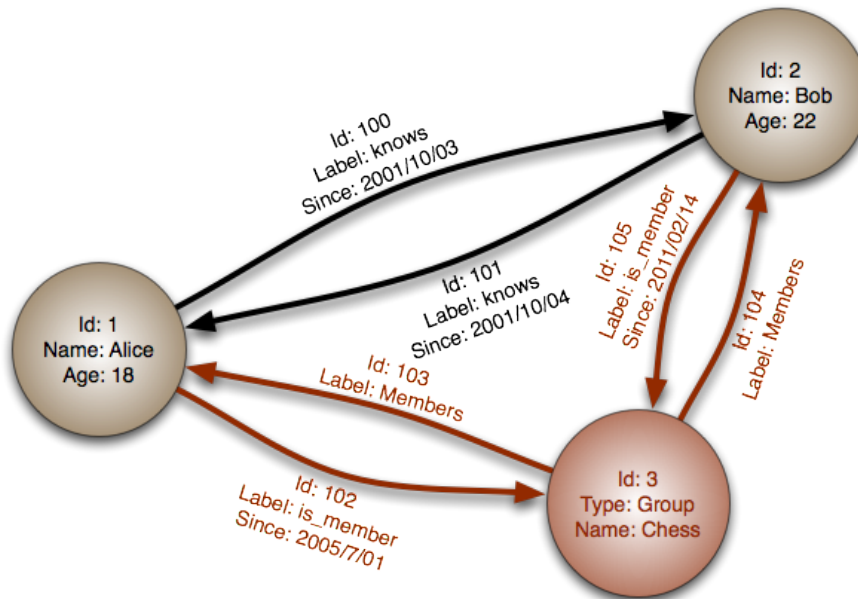
Tra gli esempi di maggiore interesse vi sono: HBase, Cassandra, Vertica.



- **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 *nod*i per archiviare le entità di dati e gli *archi* per archiviare le relazioni tra entità. Le

relazioni che un nodo può avere sono illimitate. In questo tipo di database attraversare collegamenti o relazioni è molto veloce perché le relazioni tra i nodi non vengono elaborate al momento della query, ma sono già presenti nel database. I casi d'uso più tipici sono i Social Network, motori di raccomandazioni e rilevamento di frodi, ovvero in tutti quegli ambiti dove è 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 qualità che devono essere soddisfatti in un'applicazione vera e propria (scalabilità, prestazioni, coerenza). Il mondo NoSQL è altamente eterogeneo, quindi questa attività di progettazione di solito si basa su pratiche e linee guida da seguire in base al sistema adoperato. Però, sono stati studiati diversi approcci che vogliono generalizzare il problema di progettazione che è alla base di ogni sistema di persistenza.

NoAM è uno dei principali strumenti di modellazione astratto per database NoSQL. Grazie ad esso riusciamo a definire una progettazione che è 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 \rightarrow NoSQL abstract model

La metodologia *NoAM* è 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'è un aggregato? Un aggregato è una porzione di dati correlati, con una struttura più o meno complessa ed ha un identificatore univoco. Gli aggregati regolano anche la distribuzione dei dati, infatti per supportare la scalabilità 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, è guidato dai casi d'uso, ovvero dai requisiti funzionali, e da esigenze di scalabilità e coerenza all'interno dell'aggregato. Si procede nel modo seguente:

- I dati persistenti di un'applicazione sono modellati in termini di entità, oggetti di valore e relazioni. Un'entità è un oggetto persistente che ha un'esistenza indipendente ed è caratterizzata da un identificatore univoco, mentre un oggetto di valore è caratterizzato appunto da un suo valore senza un proprio identificatore
- Entità e oggetti di valore vengono raggruppati in *aggregati*. Un aggregato ha un'entità come radice e può contenere molti oggetti di valore.

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

- Gli aggregati dovrebbero essere abbastanza grandi per poter includere tutti i dati coinvolti da certi vincoli di integrità.
- Gli aggregati dovrebbero essere i più piccoli possibile, in quanto dimensioni ridotte consentono di soddisfare requisiti di prestazioni e scalabilità.

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 è un'entità, altrimenti è 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'entità principale. Le frecce uscenti indicano la composizione dell'entità e, ricorsivamente, degli oggetti di valore. Se consideriamo l'entità **mary:Player**, essa sarà composta, oltre che dai suoi attributi, da:

due oggetti di valore, rispettivamente **games[0]** e **games[1]**, a sua volta **games[0]** è formato dall'oggetto di valore **:GameInfo** così composto:

il valore di **game** sarà **2345:Game**, il valore di **opponent** sarà composto da **rick:Player**. Stesso procedimento verrà fatto per **games[1]**.

Bisognerà ripetere il procedimento per ogni entità 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 \rightarrow 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 granularità:

- **blocco**, unità di dimensione maggiore, ha massima consistenza
- **entry**, unità 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** è un insieme di **collections**;

Ogni **collection** ha un nome distinto; Una **collection** è un insieme di **blocchi**;

Ogni **blocco** all'interno della **collection** è identificato da una chiave di blocco, che deve essere univoca; Un **blocco** è un insieme non vuoto di **entries**;

Ogni **entry** è composta da una coppia chiave-valore, la quale è univoca all'interno del blocco, il valore può 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 può essere fatto con la programmazione ad oggetti, in cui una **collection** è una classe, ed un **blocco** è l'istanza di quella classe.

Vi sono due modalità principali per rappresentare gli aggregati:

- *Entry per Aggregate Object (EAO)*: Rappresenta ogni aggregato utilizzando una singola **entry**, la chiave della **entry** è vuota, il valore contiene l'intero aggregato. Metodologia dedicata maggiormente ai database documentali.
- *Entry per Atomic Value (EAV)*: Rappresenta ogni aggregato per mezzo di più **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 più semplice la ricerca; i valori di ogni **entry** sono atomici.
Questa metodologia è proprio quella che rispecchia in modo più appropriato i database key-value.



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 più entry in modo da avere valori atomici al suo interno

Ovviamente queste due sono le rappresentazioni più estreme che si possono ottenere.

È 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.

Player		
mary	€	\langle username:"mary", firstName:"Mary", lastName:"Wilson", games : { \langle game : Game:2345 , opponent : Player:rick \rangle , \langle game : Game:2611 , opponent : Player:ann \rangle } \rangle
rick	€	\langle username:"rick", firstName:"Ricky", lastName:"Doe", score:42, games : { \langle game : Game:2345 , opponent : Player:mary \rangle , \langle game : Game:7425 , opponent : Player:ann \rangle , \langle game : Game:1241 , opponent : Player:johnny \rangle } \rangle

Game		
2345	€	\langle id : "2345", firstPlayer : Player:mary , secondPlayer : Player:rick , rounds : { \langle moves : ..., comments : ... \rangle , \langle moves : ..., actions : ..., spell : ... \rangle } \rangle

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/EA-V/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, però, comporterá una maggiore complessitá delle chiavi. Mentre, se vogliamo chiavi molto semplici dovremo utiliz-

zare *EAO*, però 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 complessità, cosa non sempre presente nei database key-value.

<i>key (/major/key/-/minor/key)</i>	<i>value</i>	EAO
<i>/Player/mary/-</i>	<i>{ username: "mary", firstName: "Mary", ... }</i>	

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

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

La figura *EAV* é quella che rispecchia maggiormente un database key-value classico con valori atomici. Si può 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*, è 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 è abbinato ad una chiave univoca che ne permette il recupero. È 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 varietà di tipi di dati associabili alle chiavi, infatti il valore archiviato in corrispondenza di una certa chiave può essere molto differente da un tipo semplice come la stringa ed il valore stesso può addirittura rappresentare una struttura dati. Inoltre vi é una grandissima possibilità di manipolazione grazie all'elevato numero di funzioni presenti.

I principali tipi di dato disponibili sono:

- **Stringhe**: é il tipo più 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 possibilità di trattare una lista come una coda (First In First Out) tramite il comando di inserimento LPUSH e il comando di prelievo RPOP oppure può 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 possibilità 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 più, includono un valore detto **score** associato ad ogni elemento, in base al quale é possibile ordinare in senso ascendente 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 può 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

```
1 > MULTI
2 OK
3 (TX)> LPUSH listaNumeri 3
4 QUEUED
5 (TX)> LPUSH listaNumeri 10
6 QUEUED
7 (TX)> LPUSH listaNumeri 34
8 QUEUED
```

```
9 (TX)> LPUSH listaNumeri 45
10     QUEUED
11 (TX)> EXEC
12     1) (integer) 1
13     2) (integer) 2
14     3) (integer) 3
15     4) (integer) 4
16
17 > LRANGE listaNumeri 0 -1
18 1) "45"
19 2) "34"
20 3) "10"
21 4) "3"
```

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

Quali proprietà ACID implementa Redis?

- **Atomicità:** Redis può avere due livelli di atomicità:
 - 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 sarà mai soddisfatta nel bel mezzo dell'esecuzione di una transazione. Ciò garantisce che i comandi vengano eseguiti come un'unica operazione isolata.
Se vogliamo avere un isolamento multi-transazionale, vi è un meccanismo che riesce a fornire delle garanzie, in cui viene fatta una sorta di operazione di check-and-set. Questo meccanismo 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 verrà 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 più 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 verrà interrotta. Avremo bisogno di ripetere la transazione quando la race condition non sarà più presente.

Quindi questo è un modo efficace per ottenere un buon livello di isolamento nel caso di transazioni multiple.

- **consistenza:** I vincoli di integrità sono dei concetti relazionali, quindi è difficile fare un collegamento con un database di questo tipo. L'unica chiave che esiste è quella primaria e deve essere univoca; l'integrità referenziale non è mantenuta da Redis stesso e deve essere gestita dalle applicazioni client.
- **persistenza(durability):** l'efficienza di Redis è dovuta in buona parte al suo modo di gestire questa proprietà. È un database in memoria ma con possibilità di essere persistente su disco, quindi rappresenta un compromesso in cui si ottengono velocità di scrittura e lettura molto elevate con la limitazione di avere un set di dati non più grande della memoria. Questo database mette a disposizione la possibilità di scegliere tra diversi meccanismi offrendo l'opportunità 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 continuit , 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 pi  comodo.

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

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

Database In Memory   possibile rinunciare ad entrambi i meccanismi definiti precedentemente per dare vita ad un database in memory, risultando molto pi  efficiente, poich  non deve pi  occuparsi dei salvataggi su disco, e pu  essere utilizzato per immagazzinare dati ad uso temporaneo la cui perdita non risulterebbe irreparabile per il sistema.

Redis pu  anche essere utilizzato come memoria cache, in cui viene fissata la quantit  massima di memoria utilizzabile. Quando questa sar  colma, i dati pi  vecchi verranno eliminato con una politica LRU o LFU.

Come configurare i diversi livelli di persistenza?

Per fare ci  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 possibilit  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 già impostati i seguenti parametri di default:

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

Ciò 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 necessità di avere intervalli aggiuntivi o diversi é molto semplice andare a modificarli aggiungendo o togliendo questi parametri predefiniti.

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

```
1 10 changes in 300 seconds. Saving...
2 Background saving started by pid ...
3 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:

```
1 save ""
2 appendonly no
```

3.1.3 Ambiti di utilizzo

Redis é estremamente flessibile e grazie alla sua efficienza può 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 può essere utilizzato in modo estremamente efficace in ambito **IoT**. Si può 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 verrà mostrato un modo in cui Redis può 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) può essere utilizzato per la messaggistica. Infatti vengono offerte prestazioni elevate per chat e flussi di commenti in tempo reale;
- **Classifiche di videogiochi:** è un servizio molto utilizzato per la creazione di classifiche in tempo reale. Infatti, è 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, è 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 rapidità. Praticamente è 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 basterà fare nel modo seguente:

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

Per aggiornare la posizione dell'auto andrà eseguito un nuovo **GEOADD** sulla stessa auto.

È possibile determinare la distanza in metri tra diverse auto con il comando **GEODIST**:

```
1 > GEODIST auto auto-p1 auto-p2
2 "347.0365"
```

Quindi, si può 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 quantità 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 più macchine con la possibilità di collaborazione tra di loro. Oltre alle soluzioni proprietarie fatte su misura in base al caso d'uso specifico, Redis offre la possibilità di implementare lato server un sistema distribuito in modo piuttosto semplice ed efficiente.

vi sono due tipologie di sistemi distribuiti che è possibile implementare:

- **Redis Master-Slave**
- **Redis Cluster**

Per sistemi estremamente avanzati, vi è la possibilità di fondere queste due modalità.

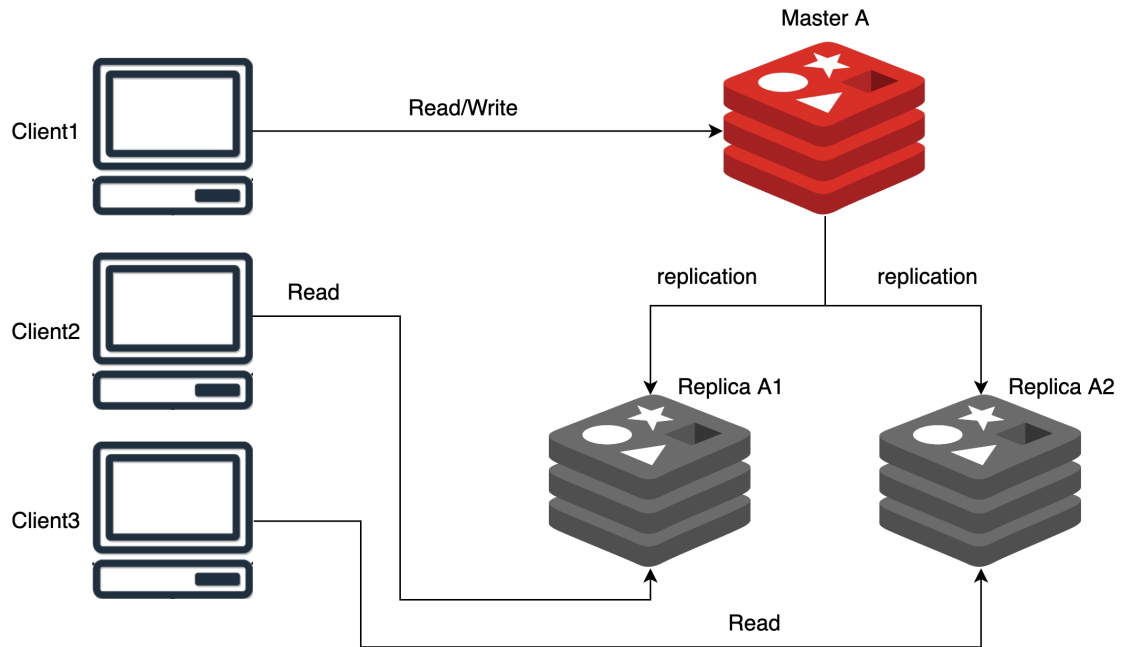
Redis Master-Slave

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

il server Redis può essere eseguito in due modalità:

- Modalità Master (Redis Master);
- Modalità Slave (Redis Slave o Redis Replica).

Redis Master funge da interfaccia con il mondo esterno, gestendo tutte le richieste scrittura esterne e può 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 può avvenire in due modi:

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

La replica master-slave è in gran parte non bloccante, il che significa che il database master può 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** è che nel momento in cui aggiungiamo una nuova macchina che fa da Master e questa sincronizzerà tutti i dati sui vari Slaves perderemo

tutti i dati.

L'approccio **2** é quello migliore perché lo slave esistente avrà già tutti i dati e una volta che lo avremo impostato in modalità 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 proprietà 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 proprietà ACID, infatti questa fa riferimento al valore più aggiornato di un certo dato, mentre nelle proprietà 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 più 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 può avere, in quanto tutto il set di dati archiviato non deve mai superare la dimensione massima.

Vi é la possibilità di dividere i dati in diverse istanze eseguite su piú server, in modo che ogni istanza contenga solo un sottoinsieme delle chiavi; 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 può 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 dovrà essere reindirizzato all'istanza corretta.

In realtà, 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 più chiavi distribuite su più 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 capacità più 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 è presente il linguaggio SQL (Structured Query Language), il quale è comune a tutti i dbms di questo tipo, quindi è diventato negli anni un vero e proprio standard.

nei NoSQL vi è 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, è compito del produttore del dbms di dover mettere a disposizione un insieme di comandi che permettano di sfruttare pienamente le potenzialità del NoSQL da lui fornito. Ne deriva che il linguaggio SQL non è supportato; sono disponibili delle interfacce 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 è 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 possibilità di definire domini, tabelle, vincoli sulle strutture e così 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 sarà 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 così 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 sarà 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 può 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 più record tramite SQL:

```
1 UPDATE Utenti
2 SET password = 'newpsw'
3 WHERE username = 'matteo00'
```

In Redis non vi è 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:

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

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

4.1.2 Comandi di Query

In SQL le interrogazioni hanno una struttura **select-from-where**, che può essere estesa in diversi modi. Sono comandi che possono anche avere una certa complessità semantica, questo è 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 complessità paragonabile a quella SQL.

Per fare una selezione di un record in SQL:

```
1 SELECT *
2 FROM Utenti
3 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.

```
1 > HGETALL utente:matteo00
2 1) "nome"
3 2) "luca"
4 3) "cognome"
5 4) "rizzo"
6 5) "password"
7 6) "myspw"
8 7) "dataCreazione"
9 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.


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

Vi é anche la possibilità 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.

```
1 > KEYS utente:*
2 1) "utente:user1"
3 2) "utente:user2"
4 3) "utente:user3"
5 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 già stati definiti nel capitolo precedente nella sezione del confronto con le proprietà ACID.

4.2 Store procedure

In SQL vi é la possibilità di definire procedure che vengono memorizzate nella base di dati come parte dello schema ed eseguite dal dbms, ovvero con una modalità opposta rispetto all'esecuzione dei client. Queste procedure svolgono una specifica attività 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 poiché gli script memorizzati sono eseguiti completamente server-side. É garantita l'esecuzione atomica dello script e durante la sua esecuzione tutte le attività 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 modalità 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 può 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.

- È 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 è sempre volatile, non è considerato come parte del database e non è persistente, quindi gli script memorizzati sono temporanei e il contenuto della cache può essere perso in qualsiasi momento.

Capitolo 5

Caso Concreto - IoT sensore Temperatura

Uno degli ambiti in cui viene maggiormente sfruttato Redis é quello IoT.

Cos'è l'IoT?

L'Internet of Things é una tecnologia che permette di massimizzare le capacità di raccolta e di utilizzo dei dati da una moltitudine di sorgenti, le quali possono essere prodotti industriali, sistemi di fabbrica, veicoli di trasporto e così via. L'IoT consente di rendere disponibili i dati che servono a comprendere meglio il mondo reale. Permette di estrarre informazioni utili ai processi decisionali.

Queste informazioni utili devono essere immagazzinate in maniera più o meno persistente e devono essere elaborate con dei tempi di latenza piuttosto brevi, ed é proprio qua che interviene Redis.

In questo capitolo verrà analizzata una possibile applicazione in ambito industriale, ma non solo: verrà utilizzato un sensore di temperatura e umidità che preleva delle informazioni ad intervalli regolari e sfrutteremo Redis per la persistenza e l'organizzazione dei dati e l'elaborazione di essi.

5.1 Scelte di progetto

Per questo tipo di applicazione non si necessita di una fase di progettazione, in quanto nell'ambito che stiamo discutendo abbiamo una mole di dati elevati ma semplici. Però, Prima di sviluppare lo sketch vero e proprio dobbiamo decidere che strutture dati messe a disposizione si adattano maggiormente al nostro caso d'uso. Possono esserci diversi tipi di implementazione; approfondiamo due modalità potenzialmente

utilizzabili: una per casi più semplificati ed una per casi più avanzati (in cui si potrebbe sfruttare anche una rete di sensori).

Queste modalità dipendono direttamente dalla struttura dati che intendiamo utilizzare in Redis per salvare i valori; i due approcci in questione sono:

1. metodo semplificato, dove ogni valore prelevato dal sensore lo memorizziamo in Redis con il tipo stringa, quindi avremo una tupla che viene generata per ogni prelevamento ad intervalli regolari, e sarà così composta:

key -> **nameSensor.time**

value -> **sensor.value**

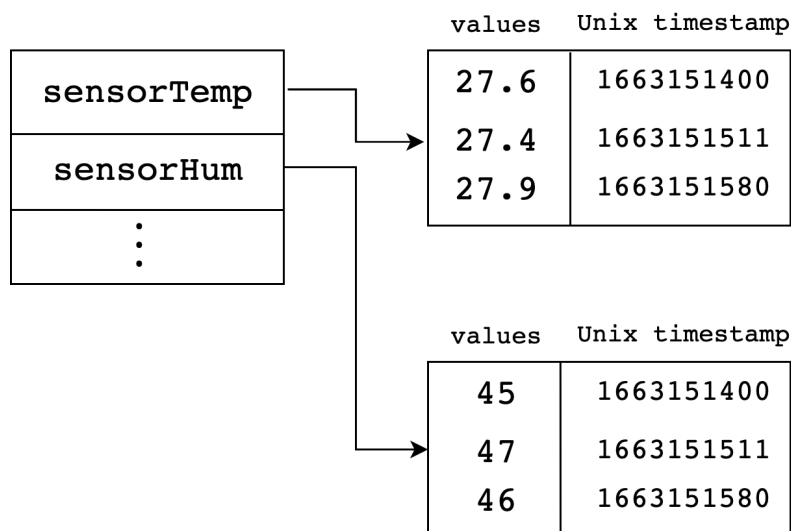
Un esempio possibile quindi:

sensorTemp.11:23:49 → 27.8

2. metodo avanzato, dove viene utilizza una struttura più complicata, non presente nelle strutture dati di base di Redis, chiamata **RedisTimeSeries**, è un modulo aggiuntivo che è stato fatto su misura per questo tipo di applicazioni. Permette un alto volume di inserimenti di valori con letture a bassa latenza. Può essere vista come una SortedSet che associa ad ogni valore inserito un timestamp come score.

Infatti sarà fatto in questo modo:

si crea una serie temporale il cui nome sarà quello del sensore e all'interno della serie temporale verranno aggiunti i valori prelevati a intervalli temporali, e questi valori verranno associati al timestamp che corrisponde al momento in cui vengono inseriti.



Prima dello sviluppo vero e proprio dello sketch necessitiamo di un ultimo passaggio, ovvero apportare dei cambiamenti al file di configurazione del server Redis. Per fare questo vi è una libreria, chiamata `Redis.h` la quale permette di comunicare con il database Redis. Eseguiamo una connessione sulla rete Locale sfruttando il modulo WiFi di ESP32.

in quanto con la configurazione standard il server comunica solo su `localhost`(127.0.0.1). Dobbiamo aggiungere un parametro per fare in modo che il server sia raggiungibile sulla rete locale aggiungendo indirizzo IP privato della macchina sulla quale viene eseguito il processo `redis-server`. Di conseguenza, il server sarà raggiungibile tramite indirizzo IP e Port.

Nel nostro caso é: **192.168.179.25:6379**. Dobbiamo anche aggiungere una password di sicurezza per l'autenticazione con il server, anche questo viene fatto aggiungendo un parametro nel file di configurazione.

La configurazione viene modificata così:

```
1 bind 127.0.0.1 192.168.178.25
2
3 requirepass "admin"
```

Per quanto riguarda la persistenza nei file di configurazione, possiamo lasciare quella base, che é una RDB. Per queste applicazioni non é necessario avere un massimo livello di persistenza.

5.2 Arduino e DHT11

Dopo aver fatto le considerazioni precedenti possiamo passare allo sviluppo dello sketch vero e proprio, che avrà il compito di inviare dati/informazioni al nostro database. Il materiale necessario per questo progetto é:

- Un microcontrollore compatibile con l'ambiente di sviluppo arduino e, possibilmente, con WiFi integrato, nel nostro caso utilizziamo un ESP32.
- un sensore digitale di temperatura e umidità dell'aria, utilizziamo un DHT11: costituito da una parte resistiva che si occupa della rilevazione dell'umidità e da un NTC che rileva la temperatura, queste due parti sono gestite da un microcontrollore che é parte integrante del sensore.

Dopo che abbiamo collegato il tutto e verificato che il sensore funzioni correttamente, ovvero che mandi dei dati coerenti all'ESP32 possiamo cominciare a sviluppare il progetto.

5.2.1 Connessione con Redis

Per comunicare ed inviare dati a Redis vi é una libreria apposita, sviluppata proprio per l'ambiente Arduino, chiamata `Redis.h`, la quale permette di comunicare con il `redis-server`.

Collegiamoci come client alla rete Locale, grazie al modulo WiFi di ESP32, sfruttando la libreria `WiFi.h`.

Nel metodo di `setup()`, che é il metodo che viene eseguito solo alla prima esecuzione dello sketch, andiamo a creare una connessione alla rete Locale tramite WiFi e, soprattutto, con il `server-redis`.

```

1 #define WIFI_SSID "wifi-name"
2 #define WIFI_PSW "wifi-psw"
3 #define REDIS_ADDR "192.168.178.25"
4 #define REDIS_PORT 6379
5 #define REDIS_PASSWORD "admin"
6
7 WiFiClient RedisConn;
8 Redis redis(redisConn);
9
10 void setup(){
11     WiFi.mode(WIFI_STA); //ESP32 associato come client alla rete WiFi
12     WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
13     Serial.print("Connecting to the WiFi");
14     while (WiFi.status() != WL_CONNECTED)
15         delay(250);
16
17     if (!redisConn.connect(REDIS_ADDR, REDIS_PORT))
18     {
19         Serial.println("Failed to connect to the Redis server!");
20         return;
21     }
22
23     auto connRet = redis.authenticate(REDIS_PASSWORD);
24     if (connRet == RedisSuccess)
25     {
26         Serial.println("Connected to the Redis server!");
27     }
28     else
29     {
30         Serial.printf("Failed to authenticate to the Redis server! Errno: %d\n", (
31             int)connRet);
32         return;
33     }
34 }
```

Per quanto riguarda la connessione al Server Redis, con `redisConn.connect(...)` creiamo una connessione con il server passando indirizzo Ip e Porta che individuano il processo `redis-server`, successivamente proviamo ad autenticarci con `redis.authenticate(...)`. Se tutto viene eseguito correttamente abbiamo il nostro `redis-client` che può comunicare.

5.2.2 NTP

Come specificato nella sezione dedicata alle scelte progettuali, qualunque modalità venga utilizzata dobbiamo ricavare l'orario, indipendentemente che sia uno Unix timestamp o una data in formato hh:mm:ss.

Come facciamo a ricavare l'ora attuale?

Arduino essendo sprovvisto di RTC (Real Time Clock), ovvero un componente elettronico che misura il tempo, ha bisogno di ottenere l'informazione sull'orario in qualche modo. Per fare questo sfruttiamo **NTP** (Network Time Protocol), che è un protocollo client-server utilizzato appunto per allineare orari di applicazioni e prodotti elettronici. È disponibile una libreria arduino chiamata **NTPClient** che permette di collegarsi a un server NTP specificato ed ottenere l'ora da questo server.

```

1 const char* ntpServer = "pool.ntp.org";
2 const long  gmtoffset_sec = 0;
3 const int   daylightOffset_sec = 7200;
4
5 void setup(){
6   configTime(gmtoffset_sec, daylightOffset_sec, ntpServer); //configurazione per
   ottenere orario in formato hh:mm:ss con fuso orario Europeo
7 }
```

Questo è il setup necessario per ottenere l'ora nel formato hh:mm:ss.

Invece, se vogliamo ottenere l'orario in Unix Timestamp, la configurazione è la seguente:

```

1 const char* ntpServer = "pool.ntp.org";
2
3 void setup(){
4   configTime(0, 0, ntpServer); //configurazione per ottenere orario in Unix Timestamp
5 }
```

5.2.3 Invio dataSet

Adesso possiamo creare la parte di codice che permette di inviare il dataSet a Redis. Analizziamo i due casi specificati nelle scelte progettuali:

1. Il primo tratta la casistica in cui viene adoperata la struttura dati più semplice, ovvero la stringa:

```

1 tm getTime(){
2   struct tm timeinfo;
3   if(!getLocalTime(&timeinfo)){
4     Serial.println("Failed to obtain time");
5   }
}
```

```

6   return timeinfo;
7 }
8
9 void loop(){
10  struct tm timeinfo;
11  timeinfo = getTime();
12  char timeHour[3];
13  strftime(timeHour, 3, "%H", &timeinfo);
14  char timeMinutes[3];
15  strftime(timeMinutes, 3, "%M", &timeinfo);
16  char timeSeconds[3];
17  strftime(timeSeconds, 3, "%S", &timeinfo);
18  sprintf(currentTime, "%s:%s:%s", timeHour, timeMinutes, timeSeconds );
19
20  char valueTemp[10];
21  sprintf(valueTemp, "%.1f", float(dht.readTemperature()));
22  char keyTemp[10];
23  sprintf(keyTemp, "sensorTemp.%s", currentTime);
24  if(redis.set(keyTemp, valueTemp)){
25      redis.expire(keyTemp, 600);
26  }
27
28  char valueHum[10];
29  sprintf(valueHum, "%d", int(dht.readHumidity()));
30  char keyHum[10];
31  sprintf(keyHum, "sensorHum.%s", currentTime);
32  if(redis.set(keyHum, valueHum)){
33      redis.expire(keyHum, 600);
34  }
35
36  delay(5000);
37 }

```

con il metodo `getTime()` si ottiene l'informazione sull'orario grazie al protocollo NTP.

Per quanto riguarda Redis utilizziamo il metodo `redis.set()` a cui passiamo due parametri: il primo é la chiave ed il secondo il valore. Ripetiamo il procedimento per la temperatura e l'umidità. Con `redis.expire()` diamo una scadenza alla chiave, la durata della chiave viene espressa in secondi e viene passata come parametro del metodo. Alla fine di `loop()` mettiamo un `delay()` che indica la pausa che deve intercorrere tra l'esecuzione di un loop e quello successivo, ovvero dovrà ripetersi ogni 5000 ms (5 secondi).

2. Di seguito analizziamo la casistica in cui viene adoperata la struttura più avanzata, `RedisTimeSeries`:

```

1 long getTime() {
2     time_t now;
3     struct tm timeinfo;
4     if (!getLocalTime(&timeinfo)) {
5         return 0;
6     }
7     time(&now);

```



```

8   return now;
9 }
10
11 void loop()
12 {
13     redis.tsadd("sensorTemp", getTime(), dht.readTemperature()*
        fattoreMoltiplicativoTemp))
14     redis.tsadd("sensorHum", getTime(), dht.readHumidity())){
15         delay(5000);
16 }

```

Con `getTime()` otteniamo l'orario in Unix Timestamp, quindi rispetto allo sketch precedente sarà molto più semplice l'implementazione, in quanto non dovremo fare conversioni particolari per ottenere un formato come quello che dovevamo ottenere nel caso precedente. Con il metodo `redis.tsadd` andiamo ad aggiungere ad una serie temporale un nuovo valore associandolo con il proprio timestamp, se la serie temporale non esiste viene creata direttamente con questa istruzione. Questo metodo ha un difetto, i valori inseriti possono essere solo interi, quindi, quando andiamo ad inserire una nuova temperatura dovremo utilizzare un fattore moltiplicativo pari ai decimali presenti nella sua rilevazione. Bisogna ricordarsi di tenerne conto nel software applicativo.

5.3 Java

Per sviluppare il software applicativo viene adoperato Java, lo scopo é quello di generare due grafici differenti, uno per la temperatura e uno per l'umidità, sfruttando il dataset presente in Redis ed aggiornandolo in tempo reale, con una frequenza di aggiornamento pari alla frequenza con cui i dati vengono inviati al server.

5.3.1 Jedis e RedisTimeSeries

Redis mette a disposizione un numero elevato di librerie, per i due approcci vengono utilizzate due librerie differenti, in quanto la libreria base adoperata nel primo approccio non fornisce metodi necessari per sfruttare la struttura dati più avanzata utilizzata nel secondo approccio: É stato utilizzato:

1. il client Jedis, uno dei più conosciuti. In particolare é stata sfruttata la classe `JedisPool` per stabilire la connessione con `redis-server`.

```

1 private static JedisPool jedisPool;
2
3 public RedisClient(String ip, int port, int timeout, String password) {
4     try {
5         if (jedisPool == null) {

```

```

6     JedisPoolConfig poolConfig = new JedisPoolConfig();
7     jedisPool = new JedisPool(poolConfig, ip, port, timeout, password);
8 }
9 } catch (Exception e) {
10     System.out.println("Malformed server address");
11 }
12 }
13
14 public Set<String> getKeysByString(String interrogation){
15     try (Jedis jedis = jedisPool.getResource()) {
16         return jedis.keys(interrogation);
17     } catch (Exception ex) {
18         System.out.println("Exception caught in KEYS");
19     }
20     return null;
21 }
22
23 public List<String> getValuesByKeys(Set<String> keys){
24     String[] arrayKeys = Utility.convertSetToArray(keys);
25     try (Jedis jedis = jedisPool.getResource()) {
26         List<String> values = jedis.mget(arrayKeys);
27         return values;
28     } catch (Exception ex) {
29         System.out.println("Exception caught in mget");
30     }
31     return null;
32 }

```

RedisClient.java

Nel costruttore `RedisClient(...)` creiamo la connessione con il server. Successivamente, sono stati creati due metodi fondamentali:

- `getKeysByString(...)`: riusciamo ad ottenere l'insieme chiavi presenti nel database data una certa stringa, viene sfruttato il metodo `jedis.keys(interrogation)` che ritorna un set di stringhe dato un certo pattern. L'idea é quella di ottenere un set composto da tutte le chiavi che cominciano per `sensorTemp` ed un altro set che comincia per `sensorHum`.
- `getValuesByKeys(...)`: riusciamo ad ottenere l'insieme di valori dato un set di chiavi, questo viene ottenuto grazie al metodo `jedis.mget(arrayKeys)`, che ritorna una lista di valori. Questo metodo vuole come parametro in ingresso un array di chiavi, quindi bisogna fare una conversione da set ad array.

2. il client `RedisTimeSeries`, creato appositamente per sfruttare la omonima struttura dati.

```

1 private static final long RANGE_OF_TIME = 600000; //in milliseconds
2 private RedisTimeSeries client;
3

```

```

4 public RedisClient(String host, int port, int timeout, int poolSize, String
    password){
5     try {
6         client = new RedisTimeSeries(host, port, timeout, poolSize, password);
7     } catch (Exception e){
8         System.out.println("connection failed!");
9     }
10 }
11
12 public Value[] getValues(String key){
13     Value[] arrayTemp = client.range(key, System.currentTimeMillis()-
        RANGE_OF_TIME, System.currentTimeMillis());
14     return arrayTemp;
15 }
16
17 public void alterRange(String key){
18     client.alter(key, RANGE_OF_TIME, null);
19 }

```

RedisClient.java

Nel costruttore `RedisClient(...)` viene sempre creata la connessione con `redis-server`.

Analizziamo i due metodi creati:

- `getValues(...)`: otteniamo un array di `Value`, che é una classe messa a disposizione dalla libreria **RedisTimeSeries**, la quale possiede come attributi valore e timestamp. Viene sfruttato il metodo `client.range(...)`, il quale ha il compito di ottenere un insieme di valori associati a timestamp in un certo range temporale. Come parametro in ingresso a `getValues()` dobbiamo fornire una stringa che corrisponde alla chiave della serie temporale.
- `alterRange(...)`: serve per far scadere valori all'interno di una serie temporale. Viene sfruttato il metodo `client.alter(...)`. Viene fatta la differenza con il timestamp del valore piú recente all'interno della serie; i valori con una differenza maggiore del range specificato vengono eliminati. Metodo utilizzato per non far saturare la serie temporale con valori vecchi e poco significativi.

5.3.2 DataSet RealTime

Dopo che abbiamo ottenuto l'intero dataset dal server bisogna modificarlo in modo da renderlo adatto per l'elaborazione grafica. Per rendere semplice la sua elaborazione in entrambi gli approcci é stata sfruttata una **TreeMap** che ha le chiavi di tipo `LocalTime` ed i valori di tipo `String`. Come abbiamo fatto nei due approcci :

1. abbiamo dovuto estrarre da ogni elemento del set di chiavi la data, quindi da ogni chiave abbiamo dovuto estrarre la data e convertirla in un `LocalTime`. Ad esempio, se abbiamo una chiave fatta in questo modo `sensorTemp.12:24:25` estraiamo `12:24:25` e lo convertiamo in tipo `LocalTime` con l'apposito metodo. Dopo di che, associamo ad ogni data il proprio valore e inseriamo tutto in una `TreeMap` in modo da mantenere l'ordine temporale.
2. qui invece siamo partiti da un array di tipo `Value`, e abbiamo dovuto convertire lo Unix timestamp in un `LocalTime` e successivamente associare date e valori nella `TreeMap` come sopra.

Dopo questo passaggio i due approcci si fondono insieme, in quanto da qui in poi si lavora sulle `TreeMap` generate che sono esattamente identiche.

A questo punto rimane un ultimo problema, ovvero aggiornare il dataset in tempo reale. Ogni 5 secondi bisogna comunicare con Redis per ottenere i dati aggiornati.

Per fare questo sfruttiamo i `Thread`.

```

1 private TimeSeries ts = new TimeSeries("f(t)","Time","No:of files");
2
3 public void run() {
4     while(true) {
5         DataSet dataSet = new DataSet(...);
6         for(Map.Entry<LocalTime, String> entry : dataSet.getKeyValueMap().entrySet()) {
7             LocalTime localTime = entry.getKey();
8             double val = Utility.convertStringToDouble(entry.getValue(), div);
9             Second istant = new Second(localTime.getSecond(), localTime.getMinute(),
              localTime.getHour(), LocalDate.now().getDayOfMonth(), LocalDate.now().getMonth()
              .getValue(), LocalDate.now().getYear());
              ts.addOrUpdate(istant, val);
10        }
11    }
12    try {
13        Thread.sleep(5000);
14    } catch (InterruptedException ex) {
15        System.out.println(ex);
16    }
17 }
18 }

```

`DataSetRealTime.java`

In questa classe abbiamo implementato l'interfaccia `Runnable` e abbiamo fatto un override sul metodo `run()`. l'attributo di tipo `TimeSeries` `ts` fa parte della libreria **JFreeChart**, che é stata sfruttata per il disegno dei grafici. Questa classe funziona in modo analogo ad una `Map`, però contiene dei metodi aggiuntivi dedicati alle serie temporali.

Con l'istruzione `new DataSet(...)` andiamo a creare la `TreeMap` spiegata in precedenza, successivamente andiamo ad estrarre orario e valore di ogni entry con un ciclo

for ed andiamo ad aggiungerlo alla `TimeSeries` con il metodo `ts.addOrUpdate(...)`. Questo metodo aggiunge nuove entry all'interno della sua struttura se vede nuovi istanti, le lascia inalterate se sono composte da stesso istante e stesso valore associato, mentre le modifica se vede stesso istante e valore associato modificato.

Tutte queste istruzioni vengono ripetute all'infinito grazie a `while(true)` ed eseguite ad intervalli temporali regolari, grazie a `Thread.sleep(5000)`, che ha il compito di disattivare il thread per 5000 ms e dopo riavviarlo.

5.3.3 JFreeChart

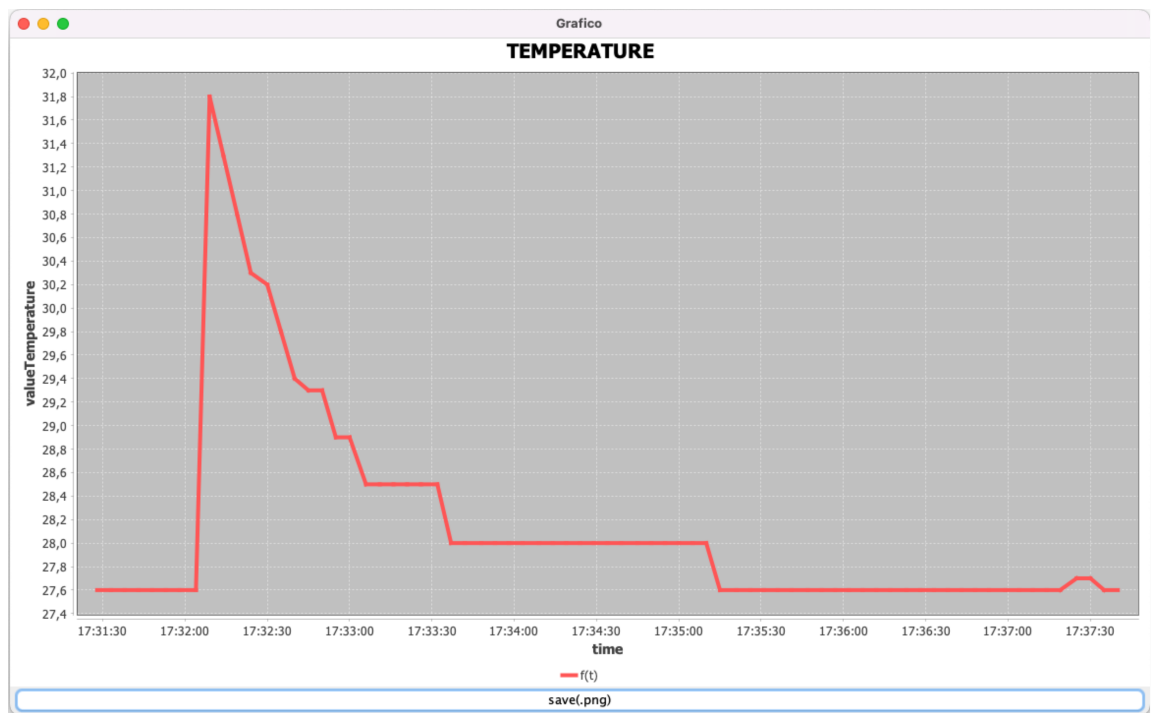
Per generare il grafico finale vero e proprio abbiamo sfruttato una libreria adatta alla creazione di diagrammi temporali, che supporta una frequenza di aggiornamento di questo tipo. Nel caso in cui volessimo sfruttare frequenze di aggiornamento molto più elevate questa libreria non è adatta, bisognerebbe utilizzare soluzioni proprietarie, o fatte ad hoc per il caso d'uso in questione.

```
1 DataSetRealTime dataRealTime = new DataSetRealTime(...);
2 new Thread(dataRealTime).start();
3
4 TimeSeriesCollection dataset = new TimeSeriesCollection(dataRealTime.getTimeSeries()
5 );
6 JFreeChart chart = ChartFactory.createTimeSeriesChart(..., dataset, ...);
... //istruzioni grafiche
```

DataSetRealTime.java

Con l'istruzione `new Thread(dataRealTime).start()` eseguiamo il metodo `run()` illustrato in precedenza su un thread specifico.

Il grafico vero e proprio viene creato con `ChartFactory.createTimeSeriesChart()`, a cui va passato come parametro d'ingresso il dataset e altri parametri per la grafica.



Nell'immagine sopra riportata possiamo apprezzare il risultato finale del progetto. gysdfsdffdsfsdffdffsdffdsfds dashas ciao coes ajfankdsfnjadfasdfasndhjjsadf fdanasdfjhsnjadsjnanjasdjndasknjka fdas dasddasdasdsadas

Bibliografia

Bibliografia