



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

Introduzione

Per decenni l'approccio alle basi di dati è sempre stato quello relazionale: la gestione dei dati e il relativo linguaggio di riferimento per le interrogazioni sono stati punti inamovibili dell'IT. Il modello relazionale è collaudato, conosciuto e adatto alle applicazioni più esigenti. La sua affidabilità è un elemento essenziale per il mondo aziendale e le applicazioni business in cui viene adottato.

In altri ambiti, però, specialmente nello sviluppo di applicazioni aperte al web, la grande affidabilità è stata messa in secondo piano dal prezzo che si paga per averla. Quando si pensa ad applicazioni che possono essere usate da milioni di utenti via Internet e con profili di carico a volte imprevedibili, scalabilità e performance diventano più importanti della correttezza delle informazioni. Proprio da questa esigenza nasce il mondo oggi conosciuto come NoSQL.

A tal proposito, questa relazione vuole mostrare una panoramica delle basi di dati non relazionali NoSQL, mostrando i vantaggi e i limiti di questo approccio. Inoltre, verrà riportata una tassonomia in base al modello che viene utilizzato per la memorizzazione dei dati. Ogni categoria porterà come conseguenza un diverso interfacciamento con la base di dati stessa.

Nel secondo capitolo verrà presentato un modello di progettazione concettuale-logica astratto ed adattabile alle differenti famiglie di NoSQL, completamente indipendente dalle implementazioni concrete.

Dal terzo capitolo verrà approfondito un particolare dbms chiave-valore: **Redis**.

A riguardo, verranno mostrate le principali peculiarità che lo contraddistinguono ed i propri punti di forza. Verranno messe in evidenza le somiglianze e differenze che si hanno con le basi di dati tradizionali.

Nell'ultimo capitolo verranno messi in pratica gli studi fatti nei capitoli precedenti, si sfrutteranno le caratteristiche di Redis per creare un applicativo utile in un ambito che si sta espandendo sempre di più, ed in maniera sempre più rapida negli ultimi anni: l'Industrial of Thing.

Indice

1 DataBase NoSQL	1
1.1 Origine	1
1.2 Perché utilizzare NoSQL	2
1.2.1 Vantaggi	2
1.2.2 Limiti	2
1.3 Tassonomia	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	13
3 Redis → Remote Dictionary Server	15
3.1 Caratteristiche	16
3.1.1 Strutture Dati	16
3.1.2 Ambiti di utilizzo	17
3.2 Confronto con <i>proprietá ACID</i>	18
3.3 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 Stored procedure	33
5 Caso reale nell'IoT	35
5.1 Scelte progettuali	35
5.2 Sketch Arduino	37

5.2.1	Connessione con Redis	38
5.2.2	NTP	39
5.2.3	Invio dataSet	40
5.3	Java	42
5.3.1	Jedis e RedisTimeSeries	42
5.3.2	DataSet RealTime	44
5.3.3	JFreeChart	46
	Conclusioni	49
	Bibliografia	51

Elenco delle figure

1.1 esempio Document Stores database	4
1.2 esempio Column Family Stores database	5
1.3 esempio Graph Model database	6
2.1 Esempio Modello di Dominio	9
2.2 Esempio Entry per Aggregate Object (EAO) ed Entry per Atomic Value (EAV)	11
2.3 Esempio Entry per Top level Field (ETF)	12
2.4 Esempio fase di Implementazione	13
2.5 Confronto modellazione Relazione e modellazione NoSQL	14
3.1 Modello client-server Redis	15
3.2 Redis distribuito master-slave	24
3.3 Redis distribuito Cluster	27
3.4 Redis distribuito avanzato con Cluster e Master-Slave	28
5.1 struttura RedisTimeSeries	36
5.2 ESP32 e DHT11 su breadboard	38
5.3 Grafico Temperatura JFreeChart	47
5.4 Grafico Umidità JFreeChart	47
5.5 Sistema di Acquisizione ed Elaborazione completo	48

Capitolo 1

DataBase NoSQL

I database *NoSQL*, che sta per *not only SQL*, sono database non tabellari che archiviano le informazioni 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 ad un database relazionale, un controllo migliore ed una maggiore semplicità nelle applicazioni.

1.1 Origine

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 quelli non strutturati (di differenti dimensioni, semi-strutturati, polimorfici...), che non permettono di definire un modello relazionale organico. I database NoSQL, di conseguenza, sono diventati estremamente popolari perché permettono di lavorare principalmente con dati variabili, anche di enormi dimensioni.

1.2 Perché utilizzare 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 alle 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

L'elevata dinamicità 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 aggiornarlo 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 è una mancanza di uno standard universale come SQL. Quindi, il passaggio da un database ad un altro può richiedere alcuni cambi più 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 Tassonomia

I database NoSQL possono essere implementati seguendo diversi approcci a seconda del modello con cui vengono rappresentati i dati. A tal proposito, le principali categorie in cui vengono suddivisi sono le seguenti:

- **Key-Value Stores:** 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. Sono altamente partizionabili, in quanto consentono una distribuzione orizzontale su scale molto maggiori rispetto ad altri database. (Questa tipologia verrà sviluppato maggiormente nel corso dei prossimi capitoli);

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

- **Document Stores:** la rappresentazione dei dati è 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 può essere formattato in vari modi (XML, JSON, YAML ...), si possono trovare le informazioni. La nozione di schema è dinamica, ogni documento può contenere dei campi diversi. Questa flessibilità può essere particolarmente utile per la modellazione dei dati in cui le strutture possono cambiare da un record all'altro, come nei dati polimorfici. Inoltre, diventa più semplice l'evoluzione di un'applicazione durante il suo ciclo di vita, permettendo di aggiungere nuovi campi a nuovi record, senza dover modificare i precedenti.

Si differenziano dai key-value stores principalmente perché i dati devono venire archiviati in un formato comprensibile dal database (documenti), infatti possono essere visti come un'estensione della categoria precedente.

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

Key	Document
1001	{ Nome:"Mario", Indirizzo:"Via Veneto 10", Hobby:"Calcio" }
1002	{ Nome:"Simone", Indirizzo:"Via del Popolo 20", Figli:[{"Nome:"Annamaria", Eta:3}, {"Nome:"Luigi", Eta:2}] }
:	:

Figura 1.1: esempio Document Stores database

- **Column Family Stores:** i dati vengono archiviati per colonne, anziché per righe come avviene nei database relazionali classici. Ogni colonna può essere memorizzata separatamente, oppure possono essere raccolte per formare dei sottogruppi. Infatti, 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 è 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.

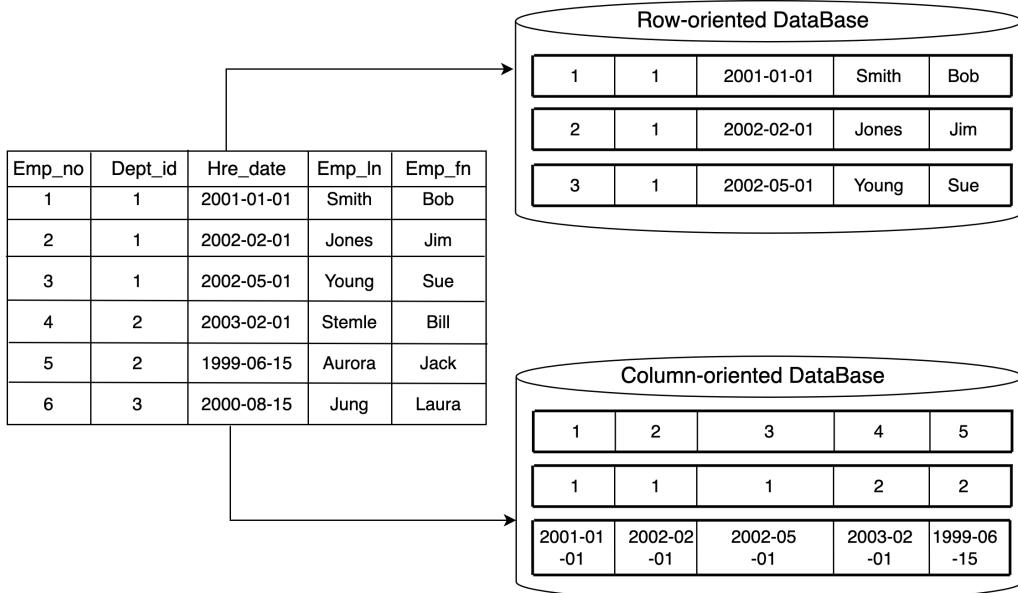


Figura 1.2: esempio Column Family Stores database

- **Graph Model:** 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 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.

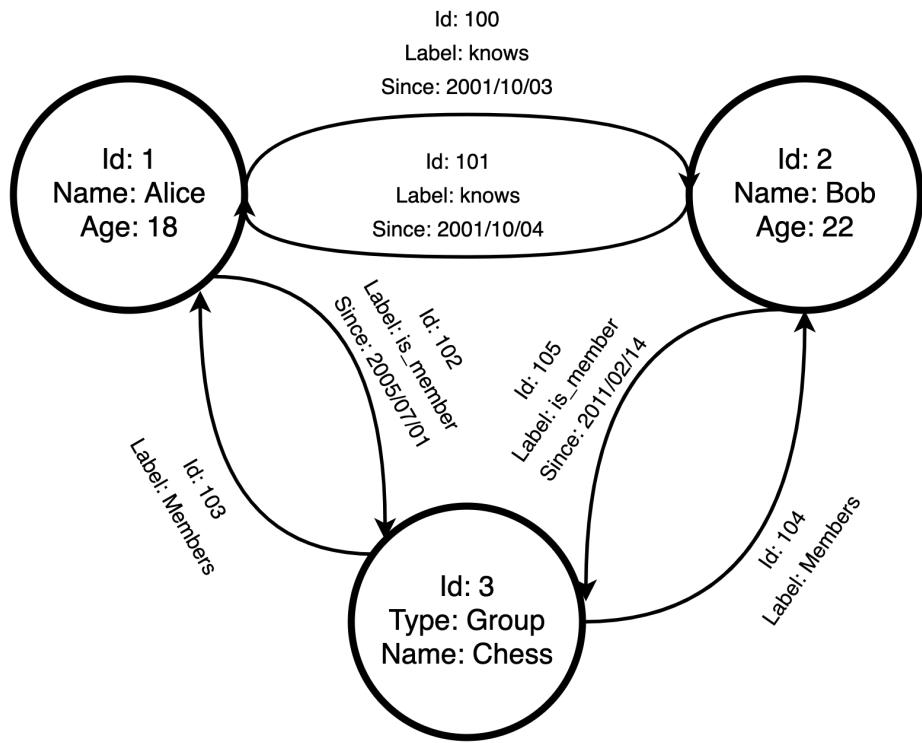


Figura 1.3: esempio Graph Model database

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 → 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.

Le classi di aggregati sono un'astrazione di un insieme di aggregati correlati tra loro.

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

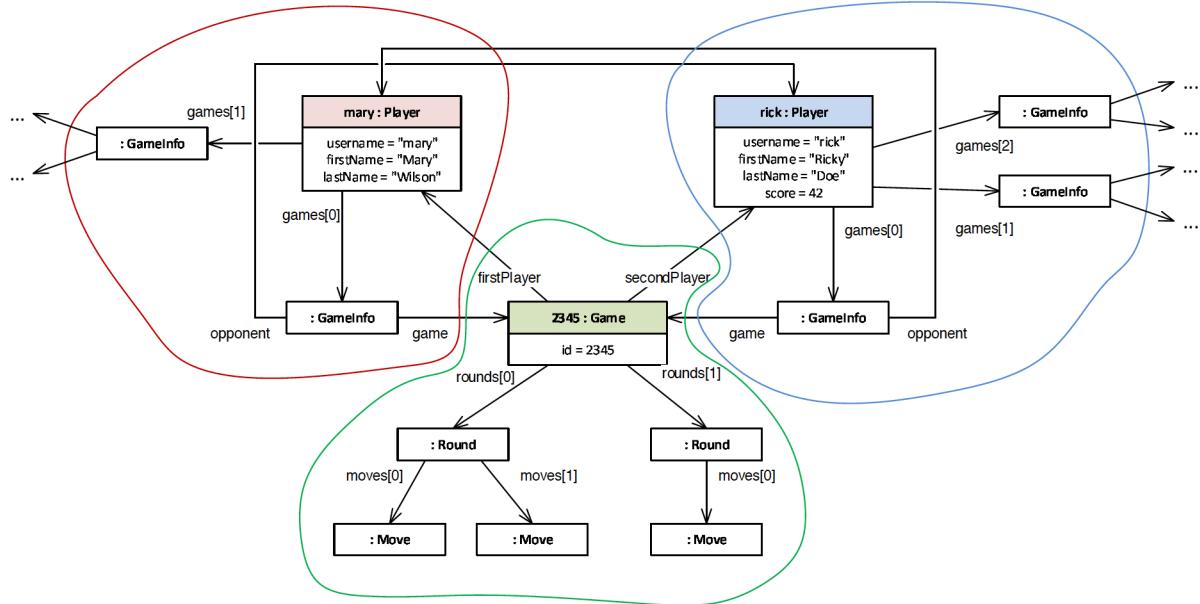


Figura 2.1: Esempio Modello di Dominio

Nella Figura 2.1 l'oggetto con lo scomparto superiore colorato è un'entità, altrimenti è un oggetto valore. La linea chiusa denota il confine di un aggregato. Pertanto possiamo definire due classi di aggregati principali: **Player** e **Game**.

Per costruire un aggregato si parte sempre da un'entità. Le frecce uscenti indicano la composizione dell'entità e, ricorsivamente, degli oggetti valore. Se consideriamo l'entità **mary:Player**, essa sarà composta, oltre che dai suoi attributi, da:

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

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

Bisognerà ripetere tutto 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 logica* fatta nei database relazionali.

Il modello *NoAM* → modello di dati astratti, ha il compito di sfruttare i punti in

comune dei vari modelli di dati, ma introduce anche astrazioni per bilanciare le variazioni che vi sono tra diversi modelli di NoSQL.

Da questa trasformazione si ottengono due strutture con diverse granularità:

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

Con riferimento in particolare alle basi di dati 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 un insieme d'istanze di una certa classe, ed un **blocco** è un'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.

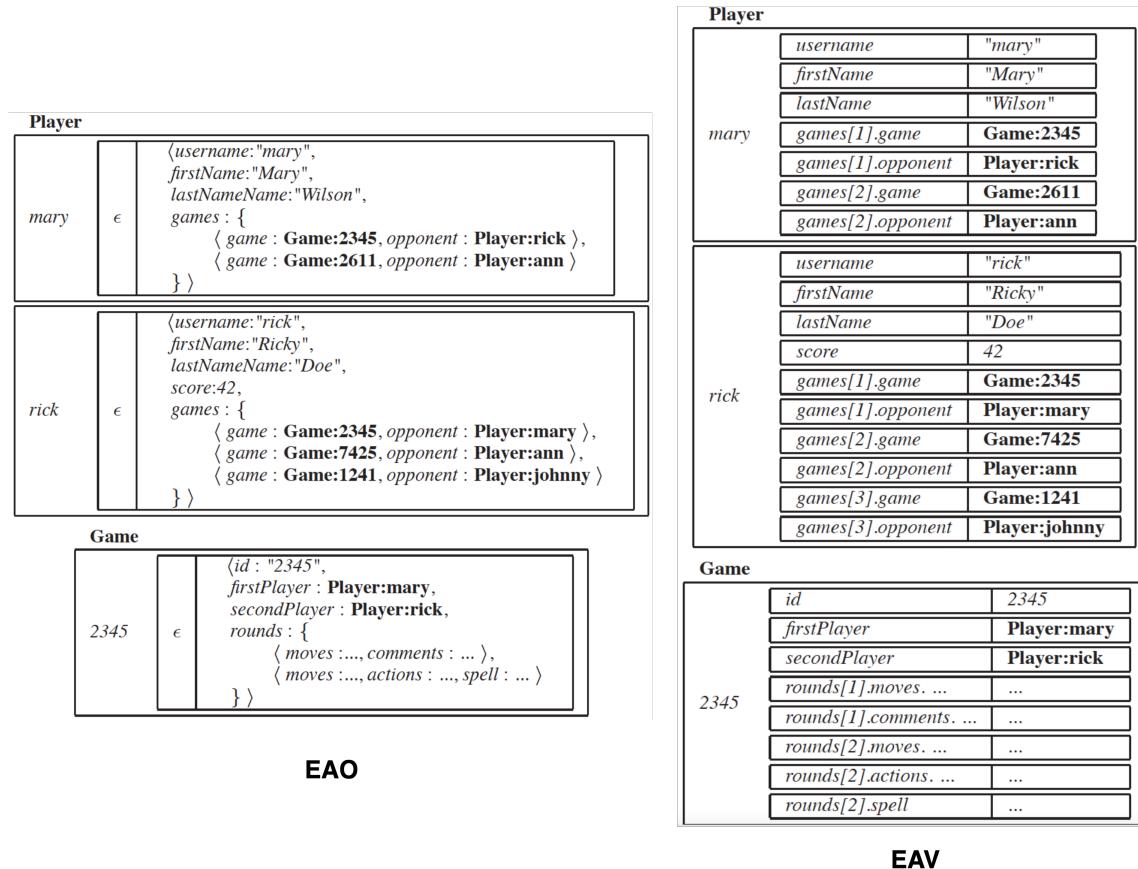


Figura 2.2: Esempio Entry per Aggregate Object (EAO) ed Entry per Atomic Value (EAV)

Nella figura *EAO* si ha per ogni blocco *Player* o *Game* una singola **entry** con chiave vuota, valore strutturato in modo complesso.

Nella figura *EAV* si ha per ogni blocco più **entry**, in modo da avere valori atomici al suo interno; ad esempio, nel primo blocco le chiavi delle entry sono **username**, **firstName**, **lastName** e così via.

Le chiavi di blocco di entrambe le figure sono `mary`, `rick` e `2345`.

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											
<code>mary</code>	<table border="1"> <tr> <td><code>username</code></td><td>"mary"</td></tr> <tr> <td><code>firstName</code></td><td>"Mary"</td></tr> <tr> <td><code>lastName</code></td><td>"Wilson"</td></tr> <tr> <td><code>games</code></td><td>{⟨ game: Game:2345, opponent: Player:rick ⟩, ⟨ game: Game:2611, opponent: Player:ann ⟩ }</td></tr> </table>	<code>username</code>	"mary"	<code>firstName</code>	"Mary"	<code>lastName</code>	"Wilson"	<code>games</code>	{⟨ game: Game: 2345 , opponent: Player: rick ⟩, ⟨ game: Game: 2611 , opponent: Player: ann ⟩ }		
<code>username</code>	"mary"										
<code>firstName</code>	"Mary"										
<code>lastName</code>	"Wilson"										
<code>games</code>	{⟨ game: Game: 2345 , opponent: Player: rick ⟩, ⟨ game: Game: 2611 , opponent: Player: ann ⟩ }										
<code>rick</code>	<table border="1"> <tr> <td><code>username</code></td><td>"rick"</td></tr> <tr> <td><code>firstName</code></td><td>"Ricky"</td></tr> <tr> <td><code>lastName</code></td><td>"Doe"</td></tr> <tr> <td><code>score</code></td><td>42</td></tr> <tr> <td><code>games</code></td><td>{⟨ game: Game:2345, opponent: Player:mary ⟩, ⟨ game: Game:7425, opponent: Player:ann ⟩, ⟨ game: Game:1241, opponent: Player:johnny ⟩ }</td></tr> </table>	<code>username</code>	"rick"	<code>firstName</code>	"Ricky"	<code>lastName</code>	"Doe"	<code>score</code>	42	<code>games</code>	{⟨ game: Game: 2345 , opponent: Player: mary ⟩, ⟨ game: Game: 7425 , opponent: Player: ann ⟩, ⟨ game: Game: 1241 , opponent: Player: johnny ⟩ }
<code>username</code>	"rick"										
<code>firstName</code>	"Ricky"										
<code>lastName</code>	"Doe"										
<code>score</code>	42										
<code>games</code>	{⟨ game: Game: 2345 , opponent: Player: mary ⟩, ⟨ game: Game: 7425 , opponent: Player: ann ⟩, ⟨ game: Game: 1241 , opponent: Player: johnny ⟩ }										
Game											
<code>2345</code>	<table border="1"> <tr> <td><code>id</code></td><td>2345</td></tr> <tr> <td><code>firstPlayer</code></td><td>Player:mary</td></tr> <tr> <td><code>secondPlayer</code></td><td>Player:rick</td></tr> <tr> <td><code>rounds</code></td><td>{⟨ moves: ..., comments: ... ⟩, ⟨ moves: ..., actions: ..., spell: ... ⟩ }</td></tr> </table>	<code>id</code>	2345	<code>firstPlayer</code>	Player: mary	<code>secondPlayer</code>	Player: rick	<code>rounds</code>	{⟨ moves: ..., comments: ... ⟩, ⟨ moves: ..., actions: ..., spell: ... ⟩ }		
<code>id</code>	2345										
<code>firstPlayer</code>	Player: mary										
<code>secondPlayer</code>	Player: rick										
<code>rounds</code>	{⟨ moves: ..., comments: ... ⟩, ⟨ moves: ..., actions: ..., spell: ... ⟩ }										

ETF

Figura 2.3: Esempio Entry per Top level Field (ETF)

Se dovessimo aver bisogno di una rappresentazione degli aggregati ancora più flessibile è 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 adatte al dbms specifico che stiamo utilizzando.

Per quanto riguarda i database chiave-valore si utilizzerà una coppia chiave-valore per ogni **entry** ottenuta nella struttura precedente.

In base alle scelte di progetto si decide quale modello NoAM utilizzare (EAO/EAV/ETF); bisogna decidere che livello di complessità vogliamo avere su chiavi e valori. Se vogliamo ottenere dei valori semplici, rappresentabili semplicemente con dei tipi atomici, utilizzeremo il metodo *EAV*, questo, però, comporterà una maggiore complessità delle chiavi. Mentre, se vogliamo chiavi molto semplici dovremo utilizzare *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</i> (/major/key/-/minor/key)	<i>value</i>	
/Player/mary/-	{ username: "mary", firstName: "Mary", ... }	EAO
<i>key</i> (/major/key/-/minor/key)	<i>value</i>	
/Player/mary/-/username	mary	
/Player/mary/-/firstName	Mary	
/Player/mary/-/lastName	Wilson	
/Player/mary/-/games/1/game	"Game:2345"	EAV
/Player/mary/-/games/1/oppont	"Player:rick"	
/Player/mary/-/games/2/game	"Game:2611"	
/Player/mary/-/games/2/oppont	"Player:ann"	
<i>key</i> (/major/key/-/minor/key)	<i>value</i>	
/Player/mary/-/username	mary	
/Player/mary/-/firstName	Mary	
/Player/mary/-/lastName	Wilson	
/Player/mary/-/games	[{ ... }, { ... }]	ETF

Figura 2.4: Esempio fase di Implementazione

La figura *EAV* è quella che rispecchia maggiormente un database key-value clas-

sico 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.

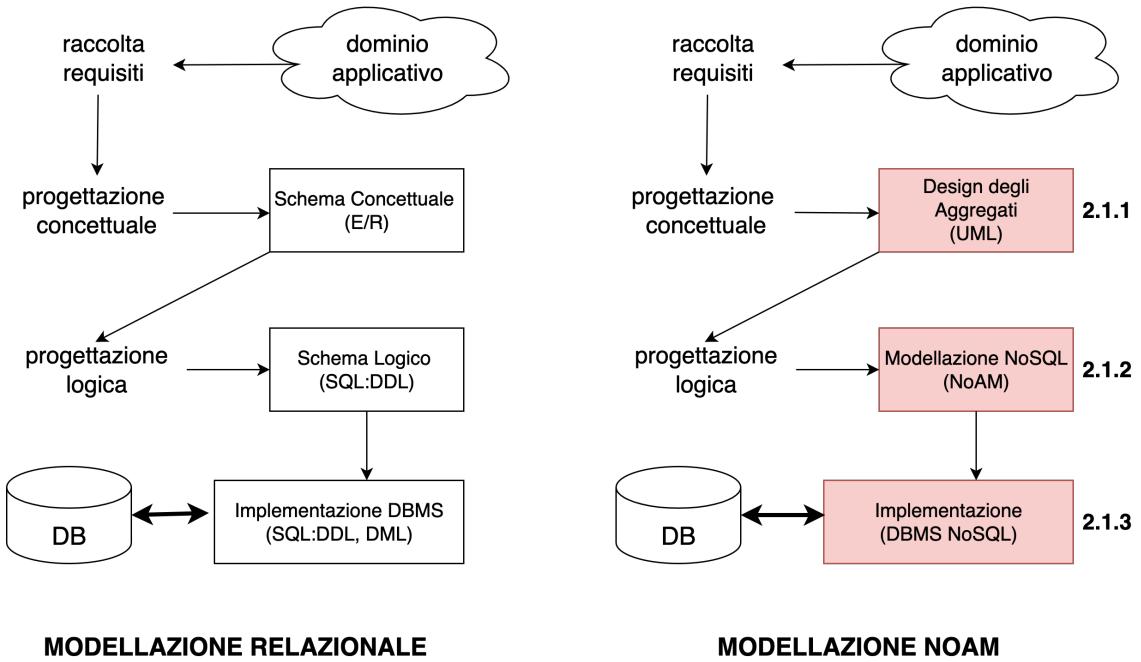


Figura 2.5: Confronto modellazione Relazione e modellazione NoSQL

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. È stato sviluppato nel linguaggio di programmazione C, e funziona principalmente con sistemi unix based, non esiste un supporto ufficiale per Windows. 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.

```
redis-cli
> set key value
OK
> get key
"value"
```

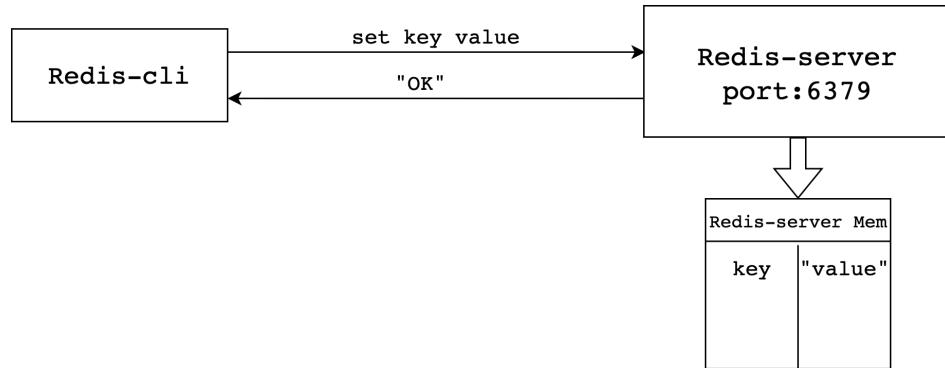


Figura 3.1: Modello client-server Redis

3.1 Caratteristiche

3.1.1 Strutture Dati

Una peculiarità di Redis è 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 è una grandissima possibilità di manipolazione grazie all'elevato numero di funzioni presenti.

I principali tipi di dato disponibili sono:

- **Stringhe:** è 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 è 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:** è una raccolta non ordinata di stringhe univoche (sono chiamate *membri* del set); vi è 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 è 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 consultare <https://redis.io/docs/data-types>.

3.1.2 Ambiti di utilizzo

Redis è 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; può essere utilizzato in modo estremamente efficace in ambito **IoT**. Infatti, si può immaginare un sensore, o una rete di sensori, che invia dati in maniera continua; questa mole di dati andrà memorizzata ed elaborata con una bassa latenza.
Verrà analizzato un caso reale nel quinto capitolo, in cui si mostrerà lo sviluppo di un applicativo nell'ambito IoT basandosi su Redis;
- **Caching:** può essere utilizzato per implementare caching in memoria e ridurre la latenza di accesso ai dati. È in grado di servire elementi richiesti con maggiore frequenza e con tempi di risposta molto brevi. Un caso tipico è la memorizzazione di risultati di query complesse di database relazionali; in questo modo se l'applicazione dovrà eseguire una query svolta già precedentemente, non dovrà più comunicare con il database relazionale, ma potrà comunicare direttamente con Redis che manterrà in memoria le ultime operazioni svolte.
- **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;
- **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. 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"
```

- **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.

3.2 Confronto con *proprietà ACID*

Nelle basi di dati relazionali ogni transazione gode delle proprietà ACID. Nei database NoSQL non è vera questa affermazione.

In questa sezione si vogliono mettere in evidenza quali proprietà 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 viene riportato 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.

- **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 è molto efficiente. Il salvataggio dei dati viene eseguito su file ad intervalli di tempo e non con continuità, 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 più comodo.

AOF → Append Only File: È un meccanismo di persistenza che consente al server Redis di tenere traccia e registrare ogni comando eseguito dal server. Vi è un file di log dove vengono aggiunti i comandi ogni volta che vengono eseguiti. Questo registro di comandi può essere riprodotto all'avvio del server, ricreando il database al suo stato originale. Il vantaggio è che basandosi su un log scritto continuamente, non vi è il rischio di incorrere in perdite in caso di crash. Inoltre, il formato dei file che vengono prodotti da questa modalità permette un recupero più semplice in caso di corruzione. Però, i file AOF risultano meno compatti e più voluminosi rispetto a quelli in formato RDB e da ciò

consegue un ripristino del database meno rapido all'avvio. Questo meccanismo viene utilizzato quando la principale preoccupazione è la perdita di dati.

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

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

Questa modalità solitamente è quella utilizzata per fare caching, nella quale viene fissata la quantità massima di memoria utilizzabile. Quando questa sarà colma, i dati più vecchi verranno eliminato con una politica LRU o LFU.

Come configurare i diversi livelli di persistenza?

Per fare ciò 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 possibilità di avere un effetto immediato sulle modifiche apportate alla configurazione del server senza doverlo riavviare.

Di seguito viene illustrato un esempio di modifica del file di configurazione, denominato `redis.conf`.

RDB è 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 è 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 è 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 ottenere un *database in memory* senza persistenza è sufficiente includere questi comandi:

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

3.3 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 avere 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 di 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.

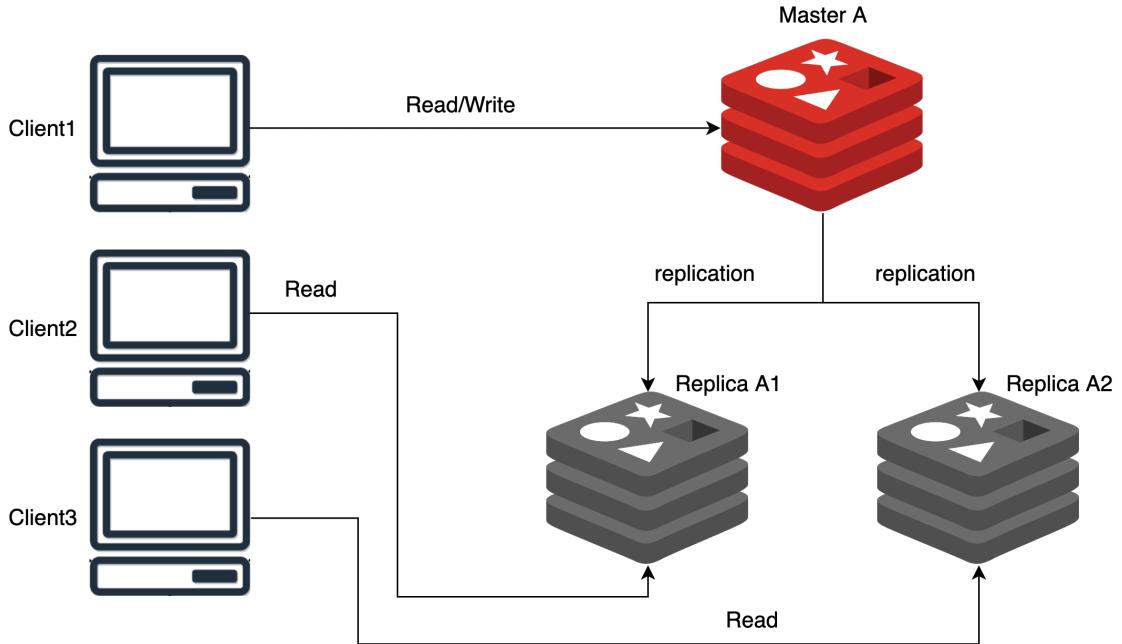


Figura 3.2: Redis distribuito master-slave

La propagazione della replica puó avvenire in due modi:

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

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

Come avviene il failover nel caso di guasto di Redis Master?

vi sono due scelte:

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

Il problema con l'approccio **1** é che nel momento in cui aggiungiamo una nuova macchina che fa da Master e questa 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 replicherà/sincronizzerà 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 vengono memorizzati in un sistema 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 piú aggiornato di un certo dato, mentre nelle proprietà ACID si fa riferimento al rispetto dei vincoli d'integrità.
- **Availability:** l'accesso ai dati é sempre garantito, non si ricevono mai messaggi d'errore, ma i dati restituiti non sono necessariamente consistenti, ovvero potrebbero non coincidere con quelli utilizzati nell'ultima operazione di scrittura degli stessi.
- **Partition Tolerance:** il sistema continua a funzionare nonostante alcuni messaggi vengano persi o subiscano rallentamenti nelle comunicazioni tra i nodi della rete.

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

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 più 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 lo scenario in cui uno Slave non ha ricevuto la scrittura e viene promosso a Master.

Redis Cluster

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

Vi é la possibilità di suddividere 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 invierà le risposte ai client. Il limite principale di questa implementazione é che il proxy puó diventare un "collo di bottiglia", poiché tutte le richieste e risposte passano per esso.

- **instradamento della query:** i client inviano la query a un’istanza Redis casuale e l’istanza si assicurerà 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.

A che nodo assegnare una chiave?

Si introduce l’idea degli **Hash Slot**, in particolare un cluster dispone di 16384 slot. Essi vengono assegnati in maniera più o meno equa tra i vari nodi. Ad ogni chiave viene associato un numero, ottenuto applicando alla chiave una Hash Function chiamata CRC16 e facendo il modulo di 16384. Il numero ottenuto indicherà lo slot di appartenenza, e di conseguenza, il nodo di appartenenza. Quindi, al momento dell’inserimento/richiesta di informazioni di una chiave viene applicata questa funzione alla chiave stessa, ed in base al numero ottenuto il client verrà reindirizzato al nodo proprietario del numero di slot ottenuto, ed eseguirà la richiesta.

Dopo un certo numero di interrogazioni i client riescono ad ottenere una mappatura completa di quali nodi sono proprietari di quali slot, potendo contattare direttamente i nodi corretti.

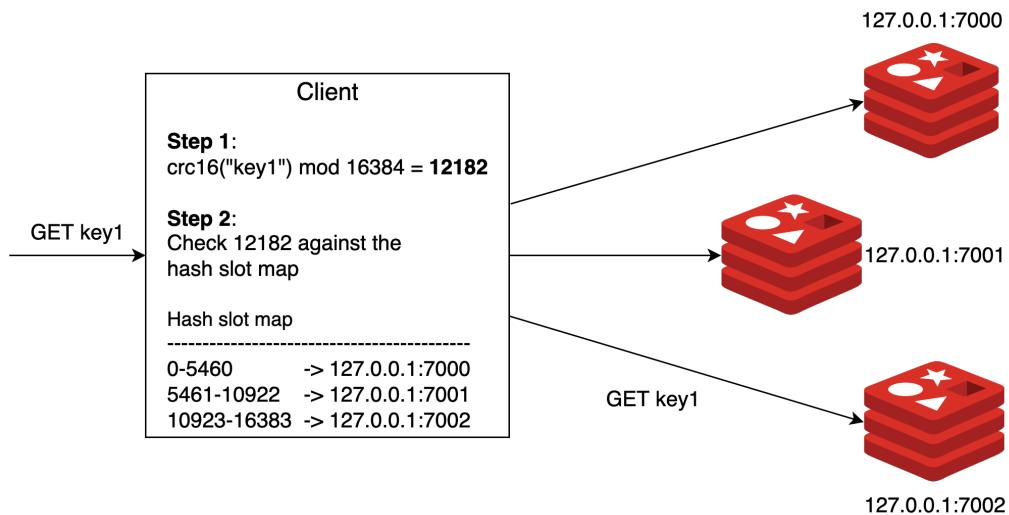


Figura 3.3: Redis distribuito Cluster

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 evitare, o quanto meno diminuire, 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.

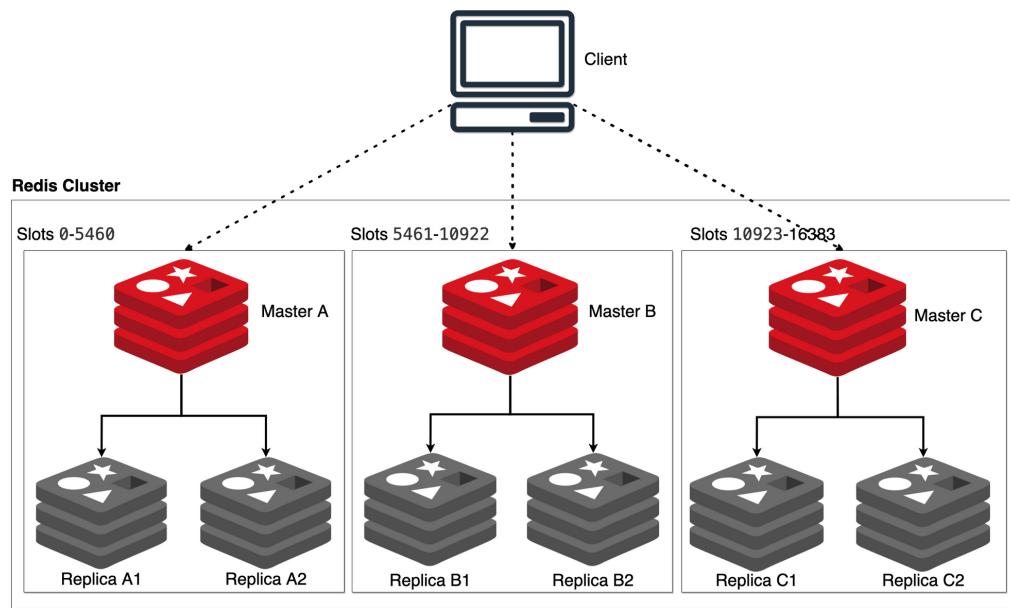


Figura 3.4: Redis distribuito avanzato con Cluster e Master-Slave

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

Capitolo 4

Interrogazioni Redis

Le capacità piú essenziali per il lavoro su database sono saperli popolare di informazioni in modo coerente e interrogare in modo intelligente, per ricavare l'informazione che si sta cercando in mezzo a volumi di dati che possono essere immensi. Quindi, per raggiungere questo obiettivo viene adoperato un linguaggio per l'interrogazione dei database, il quale serve per formulare delle query, in modo adeguato alla struttura delle informazioni. Nei database di tipo relazionale é presente il linguaggio SQL (Structured Query Language), uno standard che si è sviluppato durante il corso degli anni.

nei NoSQL vi é un linguaggio standard da adoperare?

i NoSQL, di conseguenza anche Redis, si caratterizzano da schemi non fissi, che possono variare in modo dinamico e, soprattutto, non vi sono vincoli referenziali che possono associare diverse tabelle all'interno della base di dati (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. È compito del produttore del dbms di mettere a disposizione un insieme di comandi che permettano di sfruttare pienamente le potenzialità del NoSQL da lui fornito. Ne deriva che non vi é un linguaggio standard comune ai NoSQL, o comune ad una specifica categoria.

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 é presente un elenco completo di tutte le API disponibili per i principali linguaggi: <https://redis.io/docs/clients>

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, verrà adoperata questa libreria.

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'inserimento.

4.1 Data Manipulation Language

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

4.1.1 Comandi di Modifica

I comandi di modifica sono quelle istruzioni che permettono:

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

Come esempio di confronto verrà adoperato un dominio che rappresenta i profili di utenti. Un utente sarà composto da: nome utente, nome, cognome, password, data di creazione.

In Redis ogni tupla sarà composta nel seguente modo:

- **chiave → utente:nomeutente**

- **valore** → tipo **hash**, in cui saranno contenute le informazioni riguardo a nome, cognome, password e data di creazione.

I comandi che verranno analizzati sono associati al tipo Hash, in quanto Redis dedica dei comandi specifici per ogni struttura dati.

Nel database relazionale si avrà 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', 'mypass', '12092020')
```

Un inserimento di una tupla in Redis viene fatta nel modo seguente:
se utilizziamo una Hash come valore, il comando di cui abbiamo bisogno è HSET, il quale viene utilizzato per aggiungere una nuova tupla.

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

Si può notare che 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 sarà possibile avere utenti con una Hash non coerente alle altre.

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

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

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

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

La cancellazione di una tupla è poco utilizzata, 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 vengono passati come parametri la chiave ed il numero di secondi che definiscono la scadenza.

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

Il valore 60 indica i secondi di tempo dopo i quali la chiave utente:matteo00 deve essere cancellata.

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

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

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

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

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

4.1.2 Comandi di Query

In SQL le interrogazioni hanno una struttura **select-from-where**, che puó essere estesa in diversi modi. Sono comandi che possono anche avere una certa complessità semantica, questo é dovuto principalmente dal fatto che si possono eseguire dei join tra diverse tabelle presenti nel database.

Questo non avviene in Redis, infatti le interrogazioni di una base di dati chiave-valore non possono avere una complessitá paragonabile a quella SQL.

Per fare una **SELEZIONE** di un record in SQL:

```
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) "matteo"
4 3) "cognome"
5 4) "rizzo"
6 5) "password"
7 6) "mypass"
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 Stored 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. È garantita l'esecuzione atomica dello script e durante la sua esecuzione tutte le attività del server vengono bloccate. È 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 puó essere il seguente:

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

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

- 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 "c664a3bf70bd1d45c4284ffeb65a6f2299bfc9f" \\SHA1 digest
3 >EVALSHA "c664a3bf70bd1d45c4284ffeb65a6f2299bfc9f" 0
4 "example of script load!"
```

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

Capitolo 5

Caso reale nell'IoT

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 molte altre.

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 piú o meno persistente e devono essere elaborate con dei tempi di latenza piuttosto brevi, ed é in questa circostanza che interviene Redis.

In questo capitolo verrà analizzata una possibile applicazione in ambito industriale: si sfrutterà un sensore di temperatura e umidità che preleverà delle informazioni ad intervalli regolari e verrà sfruttato Redis per la persistenza (se sarà necessaria), l'organizzazione dei dati e l'elaborazione di essi.

5.1 Scelte progettuali

Per questo tipo di applicazione non si necessita di una fase di progettazione concettuale/logica, in quanto, nell'ambito che stiamo discutendo abbiamo una mole di dati elevati ma, semplici. 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; verranno approfondite due modalità potenzialmente utilizzabili: una per casi piú semplificati ed una per casi piú 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 viene memorizzato in Redis con il tipo **String**, quindi si avrà 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 utilizzata una struttura più complessa, non presente nelle strutture dati di base di Redis, chiamata **RedisTimeSeries**, la quale 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 con membri non univoci, che associa ad ogni valore inserito un timestamp come score.

Sarà fatto in questo modo:

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

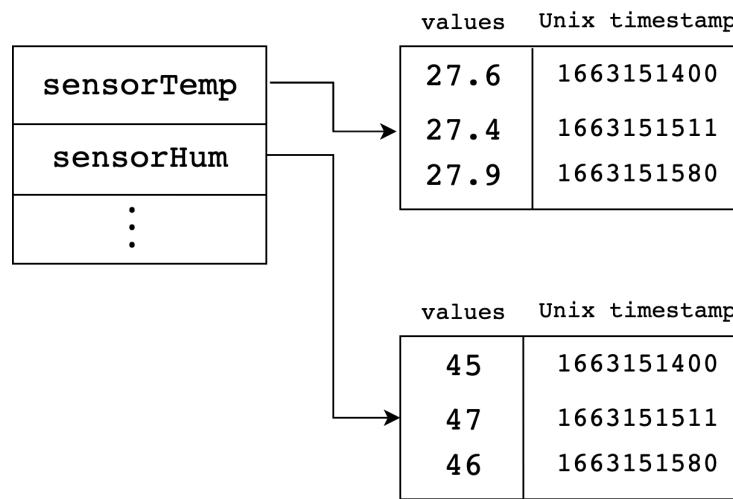


Figura 5.1: struttura RedisTimeSeries

Prima dello sviluppo vero e proprio dello sketch, che avrà il compito di far comunicare il sensore con Redis, si necessita di un ultimo passaggio, ossia apportare dei cambiamenti al file di configurazione del server Redis.

Con la configurazione standard il server comunica solo su `localhost`(127.0.0.1). Bisogna 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 Porta.

Nel nostro caso é: **192.168.179.25:6379**.

Bisogna 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 aggiungendo le seguenti righe in `redis.conf`:

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

Per quanto riguarda la persistenza, può essere lasciata quella base, che é una RDB. Per questa applicazione non é necessario avere un massimo livello di persistenza, anzi si potrebbe pensare di avere addirittura un database in memory senza salvataggi su disco.

5.2 Sketch Arduino

Si può passare allo sviluppo dello sketch vero e proprio, che avrà il compito di inviare dati/informazioni al database.

Il materiale necessario per questo progetto é:

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

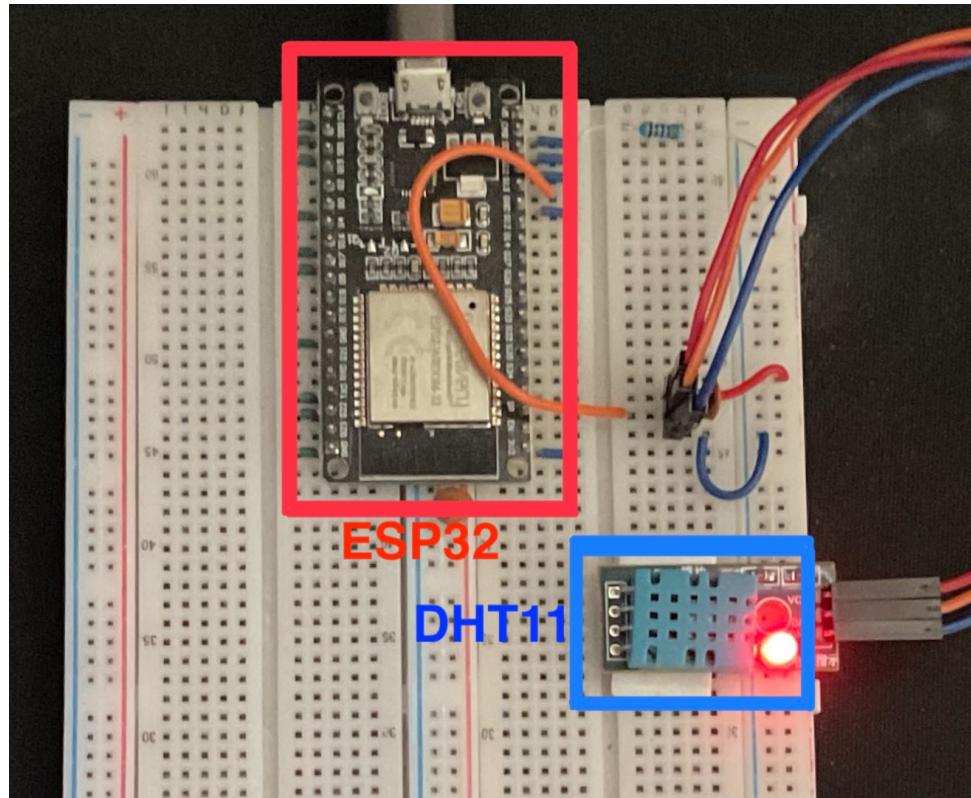


Figura 5.2: ESP32 e DHT11 su breadboard

Dopo aver collegato il tutto e verificato che il sensore funzioni correttamente, ovvero che mandi dei dati coerenti al microcontrollore, si può iniziare a sviluppare lo sketch da caricare su ESP32.

5.2.1 Connessione con Redis

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

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

Nel metodo di `setup()`, che viene eseguito solo alla prima esecuzione dello sketch, viene creata 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_PSW);
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         connRet);
32         return;
33 }

```

Per quanto riguarda la connessione al redis-server, con `redisConn.connect(...)` viene creata una connessione con esso passando indirizzo Ip e Porta, le quali individuano il processo `redis-server`, successivamente si prova l'operazione di autenticazione con `redis.authenticate(...)`. Se tutto viene eseguito correttamente si dispone di `redis-client` su ESP32 che comunica in modo corretto.

5.2.2 NTP

Come specificato nella sezione dedicata alle scelte progettuali, qualunque modalità venga utilizzata bisogna 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 da

qualche servizio esterno. Per fare questo sfruttiamo **NTP** (Network Time Protocol), che é 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 é il setup necessario per ottenere l'ora nel formato **hh:mm:ss**.

Invece, se vogliamo ottenere l'orario in Unix Timestamp, la configurazione é 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

In questa sezione viene creare la parte di codice che permette di inviare il dataSet a Redis. Analizziamo i due casi specificati nelle scelte progettuali:

- Il primo tratta la casistica in cui viene adoperata la struttura dati piú 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, "%2.1f", 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 è la chiave ed il secondo il valore. Il procedimento viene fatto per temperatura e umidità. Con `redis.expire()` viene impostata 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: dovrà ripetersi ogni 5000 ms.

2. Di seguito analizziamo l'utilizzo della 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()*
14         fattoreMoltiplicativoTemp));
15     redis.tsadd("sensorHum", getTime(), dht.readHumidity());
16     delay(5000);
17 }

```

Con `getTime()` viene ricavato l'orario in Unix Timestamp, quindi rispetto allo sketch precedente sarà molto più semplice l'implementazione nel codice, in

quanto non andranno fatte conversioni particolari per ottenere un formato come quello che bisognava ottenere nel caso precedente.

Con il metodo `redis.tsadd` viene aggiunto 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 verrà inviata una nuova temperatura bisognerà utilizzare un fattore moltiplicativo proporzionale ai decimali presenti nella sua rilevazione, per non perdere informazioni sulla risoluzione dei valori rilevati. Bisognerà ricordarsi di tenerne conto nell'interpretazione dei valori presenti nel database, soprattutto quando verranno elaborati nell'applicativo Java.

5.3 Java

Per sviluppare il software applicativo viene adoperato Java, lo scopo è 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 database.

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: È stato utilizzato:

1. il client Jedis, uno dei più conosciuti. In particolare è 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(...)` viene creata la connessione con il server. Successivamente, sono stati creati due metodi fondamentali:

- `getKeysByString(...)`: si ottiene l'insieme di 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 è quella di ottenere un set composto da tutte le chiavi che cominciano per `sensorTemp` ed un altro set che comincia per `sensorHum`.
- `getValuesByKeys(...)`: si ottiene 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 millisecondi
2 private RedisTimeSeries client;
3
4 public RedisClient(String host, int port, int timeout, int poolSize, String
5     password){
6     try {
7         client = new RedisTimeSeries(host, port, timeout, poolSize, password);
8     }catch (Exception e){
9         System.out.println("connection failed!");
10    }
11
12 public Value[] getValues(String key){
```

```

13     Value[] arrayTemp = client.range(key, System.currentTimeMillis() -
14         RANGE_OF_TIME, System.currentTimeMillis());
15     return arrayTemp;
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 è 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 con timestamp associati, in un certo range temporale. Come parametro in ingresso a `getValues()` bisogna fornire una stringa che corrisponde al nome (chiave) della serie temporale.
- `alterRange(...)`: serve per eliminare valori all'interno di una serie temporale. Viene sfruttato il metodo `client.alter(...)`. Vengono confrontati i timestamp di tutti i valori con quello più recente, il confronto viene fatto facendo la differenza con il timestamp del valore più recente all'interno della serie; i valori che hanno un timestamp associato che ha una differenza maggiore del range specificato come parametro in ingresso vengono eliminati. Metodo utilizzato per non far saturare la serie temporale con valori vecchi e poco significativi.

5.3.2 DataSet RealTime

Dopo che è stato 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 è stata sfruttata una `TreeMap`, che ha le chiavi di tipo `LocalTime` ed i valori di tipo `String`. La conversione viene fatta nel modo seguente nei due approcci :

1. da ogni elemento di tipo stringa appartenente al set di chiavi si estrae la data e la si converte 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, viene associata ad ogni data il proprio valore inserendo tutto in una `TreeMap` in modo da mantenere l'ordine temporale.

2. Si parte da un array di tipo `Value`, si estraе il timestamp associato ad ogni valore e lo si converte in un `LocalTime`; successivamente vengono associate 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 vengono sfruttati 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 instant = new Second(localTime.getSecond(), localTime.getMinute(),
10                 localTime.getHour(), LocalDate.now().getDayOfMonth(), LocalDate.now().getMonth()
11                 .getValue(), LocalDate.now().getYear());
12             ts.addOrUpdate(instant, val);
13         }
14         try {
15             Thread.sleep(5000);
16         } catch (InterruptedException ex) {
17             System.out.println(ex);
18         }
19     }
20 }
```

DataSetRealTime.java

Si implementa l'interfaccia `Runnable` e viene fatto un override sul metodo `run()`. Il tipo dell'attributo `TimeSeries ts` fa parte della libreria **JFreeChart**, che è stata sfruttata per il disegno dei grafici. Questo oggetto funziona in modo analogo ad una `Map`, però contiene dei metodi aggiuntivi dedicati alle serie temporali.

Con l'istruzione `new DataSet(...)` viene generata la `TreeMap` spiegata in precedenza, successivamente vengono estratti orario e valore di ogni entry con un ciclo `for` e si aggiungono alla `TimeSeries` con il metodo `ts.addOrUpdate(...)`. Questo metodo aggiunge nuove entry all'interno della sua struttura se trova nuovi istanti nei cicli successivi, 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 durante l'intera esecuzione dell'applicazione, 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 è stata sfruttata 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, bisognerà 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 JFreeChart chart = ChartFactory.createTimeSeriesChart(..., dataset, ...);
6 ... //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.

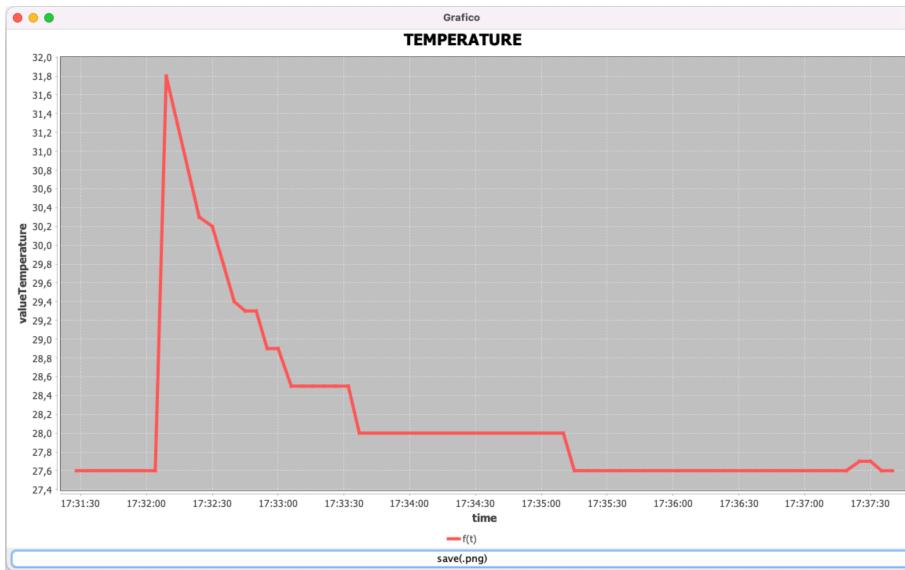


Figura 5.3: Grafico Temperatura JFreeChart

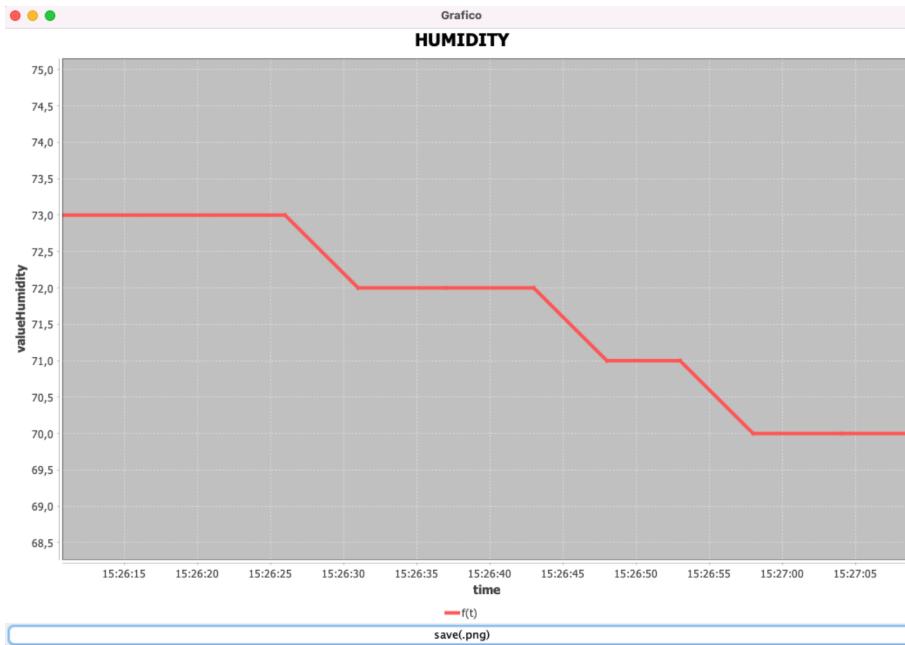


Figura 5.4: Grafico Umidità JFreeChart

Nei grafici sopra riportati può essere apprezzato che sull'asse delle ascisse abbiamo il tempo, quindi la parte di chiavi del primo approccio o i timestamp convertiti del secondo approccio; mentre sull'asse delle ordinate abbiamo i valori legati alle chiavi oppure i valori inseriti nella serie temporale.

Prima dell'avvio del software bisogna avviare il **redis-server** con il file di configurazione specifico, successivamente il grafico verrà disegnato coerentemente con i valori memorizzati in Redis. Si aggiornerà automaticamente in tempo reale con una frequenza di 5 secondi.

Questo software può essere sfruttato anche per elaborare dati completamente diversi da temperatura e umidità, quindi può essere adattato in modo molto semplice per dataset di diversa natura.

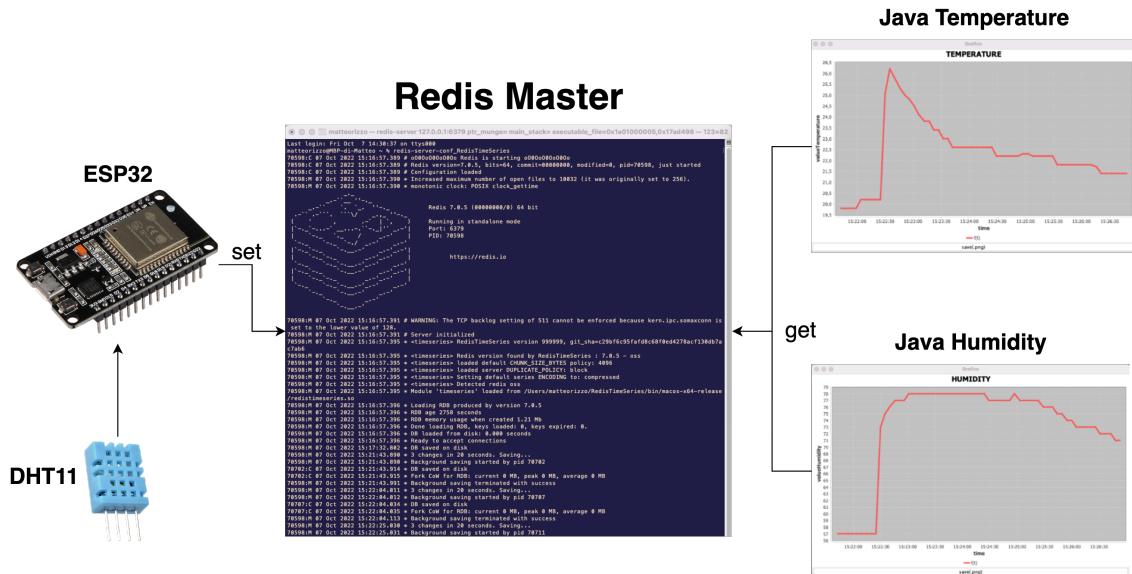


Figura 5.5: Sistema di Acquisizione ed Elaborazione completo

Conclusioni

Dopo aver fornito una panoramica delle basi di dati NoSQL, è stata definita una modellazione astratta ed indipendente dal dbms specifico, grazie alla quale si ha una libertà di progettazione dal sistema utilizzato, ad eccezione dell'ultima fase.

Dal terzo capitolo si è preso in considerazione come caso di studio **Redis**, sono state fatte notare le proprie caratteristiche ed ambiti di utilizzo.

Si è visto che non vengono implementate tutte le proprietà ACID; inoltre, certe proprietà possono essere modificate a livello di file di configurazione del server, oppure devono essere gestite a livello applicativo.

Nel corso del quarto capitolo è stato fatto un confronto tra il Data Manipulation Language di SQL e quello di Redis; in Redis è stata sfruttata la struttura dati Hash per questo paragone.

Nell'ultimo capitolo, l'applicativo software è stato progettato sfruttando due strutture dati differenti: String e RedisTimeSeries.

Per il disegno del grafico è stata utilizzata la libreria JFreeChart.

Un possibile miglioramento potrebbe essere l'aumento della frequenza di campionamento da parte dei sensori, con un conseguente aumento di velocità di aggiornamento del grafico. Per farlo con risultati estetici apprezzabili, si avrebbe la necessità di implementare una libreria diversa da quella utilizzata, adatta ad intervalli di refresh più brevi.

Inoltre, potremmo volere un grafico che abbia al suo interno maggiori funzionalità, ad esempio per fare un confronto tra diversi sensori appartenente ad una stessa rete, oppure ripercorrere indietro nel tempo i prelevamenti. A questo punto, converrebbe utilizzare librerie che mettono a disposizione costrutti per ottenere grafici che dispongono di funzionalità più avanzate.

Si potrebbe pensare di memorizzare i dati meno recenti in database non in memory, in modo da avere uno storico molto più prolungato di dati, con garanzie di persistenza. Redis verrebbe sfruttato solo per dati istantanei, o comunque per dati generati in un intervallo temporale abbastanza vicino a quello attuale.

Bibliografia

- [1] <https://www.geeksforgeeks.org/types-of-nosql-databases>
- [2] <https://www.ionos.it/digitalguide/hosting/tecniche-hosting/nosql>
- [3] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, Riccardo Torlone - A logical approach to NoSQL databases
- [4] <https://redis.io/docs/data-types>
- [5] <https://aws.amazon.com/it/redis>
- [6] <https://linuxhint.com/redis-watch-command>
- [7] <https://redis.io/docs/clients>
- [8] <https://www.javacodegeeks.com/2015/09/redis-clustering.html>
- [9] <https://redis.io/docs/reference/cluster-spec>
- [10] <https://www.quora.com/What-is-Redis-in-the-context-of-the-CAP-Theorem>
- [11] <https://severalnines.com/blog/installing-redis-master-slave-manual-failover>
- [12] <https://www.arduino.cc/reference/en/libraries/redis-for-arduino>
- [13] <https://www.jfree.org/jfreechart>
- [14] <https://www.arduino.cc/reference/en/libraries/ntpclient>
- [15] <https://redis.io/docs/modules>

Ringraziamenti

Un sentito grazie a tutte le persone che mi hanno permesso di arrivare fin qui.
Grazie al mio relatore M. Melchiori, per avermi concesso la sua disponibilità ed avermi guidato durante la stesura di questo elaborato.
Alla mia famiglia ed alla mia fidanzata per essermi stati sempre affianco durante l'intero percorso ed essere stati sempre i primi a sostenermi.
Ai miei amici e compagni di università per essere stati sempre presenti in tutti questi anni, regalandomi momenti di gioia e spensieratezza.