



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

Relazione Finale

Titolo 1

Titolo 2

Titolo 3

Relatore: Chiar.mo Prof. Melchiori Michele

Laureando:
Rizzo Matteo
Matricola n. 727499

Anno Accademico 2021/2022

Sommario

Ringraziamenti

Indice

1	DataBase NoSQL	3
1.1	Perche é nato <i>NoSQL</i> ?	3
1.2	Perché dovrei utilizzare un database <i>NoSQL</i> ?	4
1.3	Tipologie di <i>NoSQL</i>	4
2	Progettazione Concettuale/Logica	9
2.1	Modellazione NoAM → NoSQL abstract model	9
2.1.1	Modellazione Concettuale e design degli Aggregati	10
2.1.2	Partizionamento degli Aggregati e modellazione NoSQL	11
2.1.3	Implementazione	14
3	Approfondimento Redis	15
4	Interrogazioni SQL → <i>Key-Value</i>	17
5	<i>proprietá ACID</i>	19
6	Caso Concreto - IoT sensore Temperatura	21
	Bibliografia	23

Introduzione

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

Capitolo 1

DataBase NoSQL

I database *NoSQL*, che sta per *not only SQL*, sono database non tabellari che archiviano i dati in maniera completamente differente dai classici relazionali. Le caratteristiche principali sono la progettazione specifica per carichi elevati e il supporto nativo per la 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 *NoSQL*?

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 dovrei utilizzare un database *NoSQL*?

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

- **Flessibilità:** 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 dei dati;
- **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.
- **Altamente funzionali:** non vi è più un linguaggio generale (SQL) come nei database relazionali, ma vi sono API in base al database specifico che si va ad utilizzare.

1.3 Tipologie di *NoSQL*

Tipologie principali di database *NoSQL*:

- **documentali:** la rappresentazione dei dati 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 appunto 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, ad esempio nei dati polimorfici. Inoltre, diventa più 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.

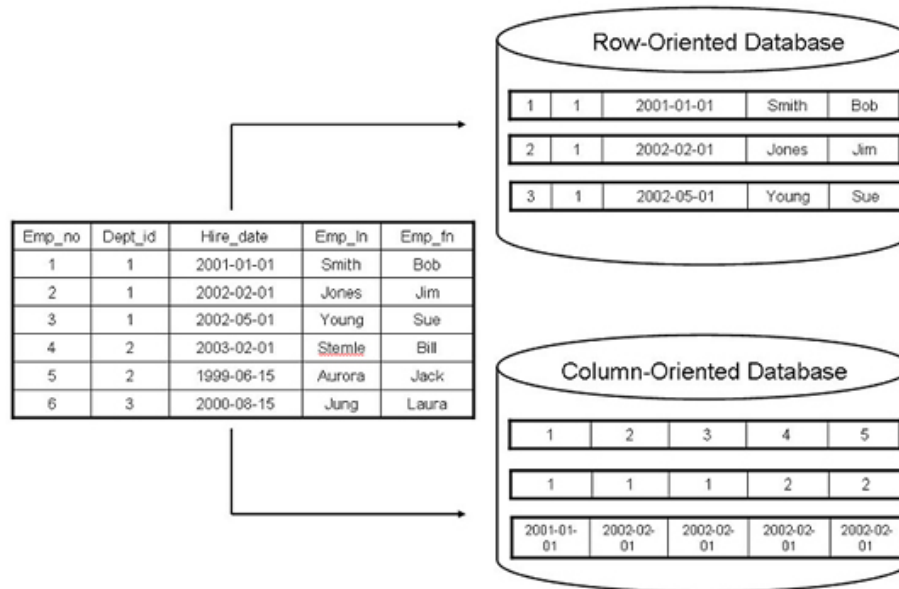
<pre>{ Nome:"Mario", Indirizzo:"Via Veneto 10", Hobby:"Calcio" }</pre>	<pre>{ Nome:"Simone", Indirizzo:"Via del Popolo 20", Figli:[{Nome:"Annamaria", Eta:3}, {Nome:"Luigi", Eta:2}] }</pre>
<div style="border: 1px solid black; padding: 5px; display: inline-block;">Document1 : person1</div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;">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.

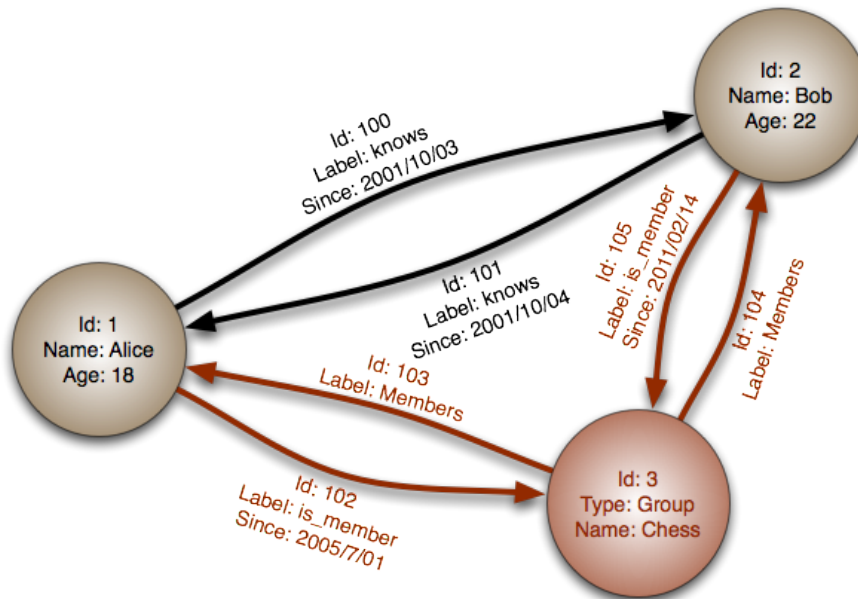
- **colonnari**: i dati vengono archiviati per colonne, anziché per righe come avviene nei database relazionali classici. Queste colonne vengono raccolte per formare dei sottogruppi. Le chiavi e i nomi delle colonne di questo tipo di database non sono fissi. Ogni colonna é memorizzata separatamente.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



Capitolo 2

Progettazione Concettuale/Logica

Sebbene i database *NoSQL* vengono definiti *schemaless*, la progettazione dell'organizzazione dei dati richiede di prendere decisioni significative. Infatti, i dati persistenti delle applicazioni hanno un impatto sui principali requisiti di 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 selezionato. 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

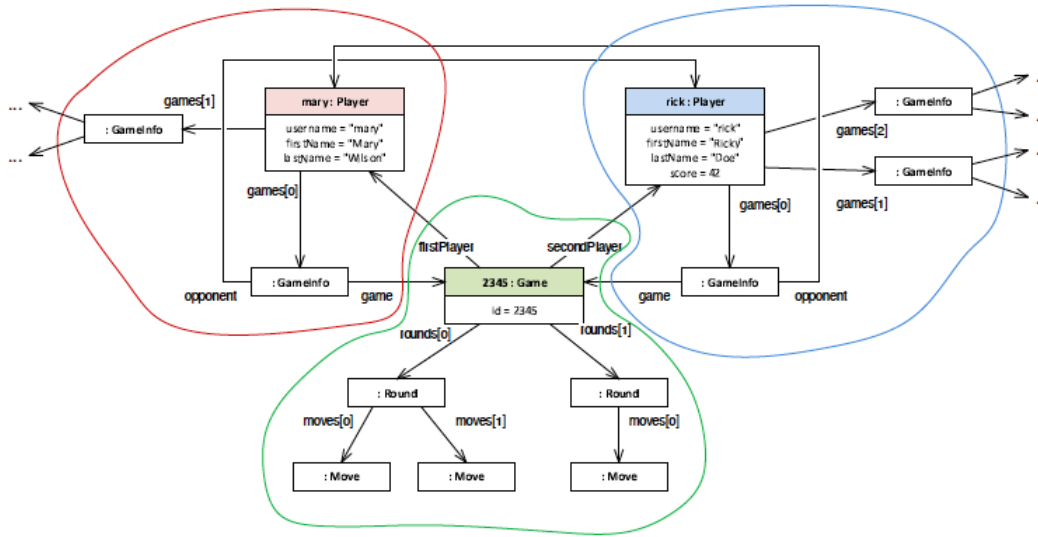
Riguarda la vera e propria progettazione del modello di dominio, e comporta l'identificazione delle diverse classi di aggregati necessari in un'applicazione. Sono possibili diversi approcci per identificare classi di aggregati per una particolare applicazione. L'approccio Domain-Driven Design(*DDD*), 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.

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

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

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

Quindi, si ha che: Un **database** 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** é un insieme non vuoto di **entries**, ogni **entry** é composta da una coppia chiave-valore, la quale é univoca all'interno del blocco, il valore 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**.

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

Player		
mary	€	{username:"mary", firstName:"Mary", lastName:"Wilson", games : { { game : Game:2345, opponent : Player:rick }, { game : Game:2611, opponent : Player:ann } }}
rick	€	{username:"rick", firstName:"Ricky", lastName:"Doe", score:42, games : { { game : Game:2345, opponent : Player:mary }, { game : Game:7425, opponent : Player:ann }, { game : Game:1241, opponent : Player:johnny } }}
Game		
2345	€	{id : "2345", firstPlayer : Player:mary, secondPlayer : Player:rick, rounds : { { moves : ..., comments : ... }, { moves : ..., actions : ..., spell : ... } }}

EAO

Player	
mary	username "mary"
	firstName "Mary"
	lastName "Wilson"
	games[1].game Game:2345
	games[1].opponent Player:rick
	games[2].game Game:2611
	games[2].opponent Player:ann
rick	username "rick"
	firstName "Ricky"
	lastName "Doe"
	score 42
	games[1].game Game:2345
	games[1].opponent Player:mary
	games[2].game Game:7425
	games[2].opponent Player:ann
	games[3].game Game:1241
	games[3].opponent Player:johnny
Game	
2345	id 2345
	firstPlayer Player:mary
	secondPlayer Player:rick
	rounds[1].moves. ...
	rounds[1].comments. ...
	rounds[2].moves. ...
	rounds[2].actions. ...
	rounds[2].spell ...

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

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	username "mary"
	firstName "Mary"
	lastName "Wilson"
	games { { game: Game:2345, opponent: Player:rick }, { game: Game:2611, opponent: Player:ann } }
rick	username "rick"
	firstName "Ricky"
	lastName "Doe"
	score 42
	games { { game: Game:2345, opponent: Player:mary }, { game: Game:7425, opponent: Player:ann }, { game: Game:1241, opponent: Player:johnny } }
Game	
2345	id 2345
	firstPlayer Player:mary
	secondPlayer Player:rick
	rounds { { moves: ..., comments: ..., }, { moves: ..., actions: ..., spell: ... } }

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 corrette per il database 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), praticamente bisogna decidere che livello di complessità vogliamo avere sulle chiavi e sui valori.

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

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

Capitolo 3

Approfondimento Redis

Parlare di Redis in generale vantaggi/svantaggi perché utilizzarlo strutture dati
String Map Set Json ...

Capitolo 4

Interrogazioni SQL \rightarrow *Key-Value*

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

Capitolo 5

proprietá ACID

Parlare delle proprietà *ACID* per quanto riguarda questo tipo di database Redis quali di proprietà dispone? È possibile decidere se avere una certa proprietà o meno ...

Capitolo 6

Caso Concreto - IoT sensore Temperatura

implementazione librerie Redis in linguaggi di programmazione Parlare di Jedis,
libreria Java. Riportare esempi di codice che implementa DataBase

Bibliografia

Bibliografia