UNIVERSITÀ DEGLI STUDI DI ROMA TRE

TESI DI LAUREA TRIENNALE

IN INGEGNERIA INFORMATICA

SVILUPPO DI COMPONENTI PER L'ACCESSO AI DATI IN SISTEMI NOSQL

Laureando

Barbieri Marco

Matricola

427501

Relatore

Prof. Luca Cabibbo

Anno Accademico 2011/2012

"Qual è la differenza tra un matematico, un fisico ed un informatico? Davanti ad un problema il matematico cerca di ricondurlo ad una formula matematica, il fisico cerca di risalire al fenomeno fisico che lo ha prodotto, l'informatico spegne e riaccende".

[Cit.]

Introduzione

La continua evoluzione delle applicazioni nel Web e l'esponenziale crescita del numero di utenti nella rete ha amplificato la necessità, per le imprese, di possedere tecnologie che gestissero grandi quantità di dati con la capacità di scalare in orizzontale su più sistemi distribuiti. Per questi motivi sono stati sviluppati i database NoSQL (o data-store o NRDBM) in contrasto con il tradizionale mondo relazionale. L'importanza dei NRDBMS è stata evidenziata dal fatto che molti dei giganti del Web hanno adottato queste tecnologie nell'elaborazione dei dati; basti pensare che Facebook, uno dei più grandi social network dei nostri giorni, ha sviluppato un proprio sistema NoSQL, Cassandra, utilizzato in seguito anche da Twitter e Dig.

Una delle più grandi novità portate dal movimento NoSQL nel mondo delle basi di dati è la possibilità di memorizzare i dati in documenti dinamici senza alcun bisogno di definire gli attributi a priori, tipico degli schemi statici utilizzati dai sistemi relazionali. I data-store vengono classificati in base al modello di dati utilizzato: documenti, colonne, grafi o coppie chiave-valore. La grande complessità e il difficile riutilizzo di codice nel passaggio da un sistema all'altro, ha portato molti a pensare a delle soluzioni simili agli ORM (Object-Relational Mapping); la soluzione è stata trovata nei framework ODM (Object data-store Mapping) all'interno dei quali convivono diverse tecnologie NoSQL. L'utente ha la possibilità di alternare l'utilizzo dei vari database venendo incontro alla proprie esigenze, senza effettuare drastiche modiche nel codice, sfruttando così i punti di forza della tecnologia scelta.

L'obiettivo della tesi è quello di mostrare le caratteristiche principali dei data-store, Oracle NoSQL e Cassandra, inseriti all'interno di un prortotipo ODM sviluppato in precedenza. Per integrare un sistema al framework è stato necessario implementare un nuovo modulo (uno per ogni tecnologia) composto da due classi

in particolare,il Driver che si occupa di instaurare la connessione con lo store selezionato (o tramite un file di configurazione o tramite una proprietà dell'annotazione Entity) e infine il Connector che offre i metodi per lo scambio di dati tra il lato applicativo e l'NRDBM; inoltre ogni connettore può avere più di una implentazione, ognuna delle quali rappresenta una diversa strategia di persistenza. Saranno descritti gli algoritmi utilizzati per le operazioni di lettura mostrando parallelamente gli studi di caso e l'evoluzione del codice, con le relative spiegazioni.

In particolare il lavoro è stato svolto con altri laurendi e diviso in gruppi di due persone; ogni gruppo è stato responsabile dell'implementazione delle operazioni di lettura e di scrittura nei connettori per particolari strategie di persistenza. Il frutto di tale collaborazione si è tradotto in un prodotto funzionante, capace di inserire e recuperare i dati dai due data-store, Oracle e Cassandra, attraverso il framework ODM in base alle esigenze dell'utente.

La tesi è divisa in sei capitoli:

- Nel primo capitolo viene effettuata una panoramica sul mondo NoSQL e sulle soluzioni fornite dai sistemi ODM alle problematiche relative dalle tecnologie non relazionali.
- Nel secondo capitolo vengono descritti i sistemi già presenti nel sistema ODM. del quale sarà analizzata l'architettura su cui si basa, spiegando in che modo il lato applicativo e lo strato persistente comunicano per permettere la memorizzazione dei dati; inoltre saranno elencati i vari step da seguire per la realizzazione di un nuovo modulo.
- Nel terzo capitolo vengono esposte dettagliatamente le tecnologie introdotte all'interno del framework ODM; saranno messi in risalto il modello di dati offerto e esempi relativi alle operazioni fornite dal driver di basso livello utilizzato.
- Nel quarto e quinto capitolo saranno analizzati gli algoritmi dei metodi di accesso ai dati nei connettori Oracle Structured e Cassadra KVJ tramite l'analisi di vari casi d'uso.
- Nel sesto capitolo sarà effettuato un riassunto sugli obiettivi raggiunti nel corso del lavoro di tesi.

INDICE

1 CONCETTI BASE

Di seguito verranno esposti i concetti base riguardanti le tecnologie non relazionali e gli aspetti principali dei prodotti ORM.

1.1 Concetti base sul mondo NoSQL

Il termine NoSQL è stato utilizzato per la prima volta da Carlo Strozzi nel 1998 in merito al suo Database Open Source *Light Weight*, che non disponeva di un'interfaccia SQL.Dopo circa 10 anni, Eric Evans, un impiegato Rackspace, riutilizzò il termine NoSQL per riferirsi ai database che sono non-relazionali (NRDBMS), indicando il movimento NoSQL come " *the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for* ". [Strauch]

Successivamente il termine NoSQL andò a rappresentare una vasta gamma di data-base che non seguono i popolari e consolidati principi dei data-base relazionali (RDBMS) , ovvero non memorizzano i dati in rigidi schemi tabelle-relazioni e non viene usato SQL come linguaggio di interrogazione e manipolazione dei dati. Il movimento NoSQL ha fatto notizia negli ultimi anni, poiché molti dei leader del Web hanno adottato una tecnologia NoSQL: aziende come Facebook, Twitter, Digg, Amazon, LinkedIn e Google utilizzano tecnologie non relazionali. Nel documento pubblicato da Cattell Rick nel 2010, "*scalable SQL e NoSQL datastores*"[Catell],viene esplicitato che questi nuovi sistemi sono stati sviluppati per fornire un ottima scalabilità orizzontale per semplici operazioni di scrittura/lettura distribuite su più server con un'elevata velocità nell'elaborazione delle interrogazioni, in contrasto con i tradizionali RDBMS, non capaci di scalare in orizzontale. Il termine scalabilità orizzontale è riferito all'abilità di distribuire i dati e il carico di queste semplici operazioni su molti server, senza RAM o dischi di archiviazione condivisi.

Con l'aumento esponenziale di utenti nella rete, la continua crescita dei dati da memorizzare , la necessità di elaborarli in breve tempo ma anche con la crescente disponibilità di dispositivi con accesso ad Internet (smartphone, tablet, e altri dispositivi portatili) è nato il bisogno di avere sistemi di memorizzazione molto flessibili che utilizzassero modelli di dati meno complessi , cercando di aggirare il rigido modello relazionale. Si pensi ai sistemi cloud dove i nodi possono essere molti; gestirli con un sistema relazionale risulta essere complicato sia dal punto di vista delle prestazioni che dal punto di vista della complessità.

Questi sono i motivi che hanno portato allo sviluppo e all'utilizzo dei database non relazionali caratterizzati da :

1. Capacità di scalare in orizzontale operazioni semplici su moltissimi nodi.
2. Capacità di replicare e distribuire i dati su molti nodi.
3. Flessibilità della struttura dei dati.
4. Nuove tecnologie di indicizzazione e di gestione della RAM.
5. L'uso di interfacce semplici rispetto il SQL.
6. Gestione delle transazioni meno rigorosa.
7. Possibilità di aggiungere dinamicamente nuovi attributi ai data record (strutture dati schema-free).
8. De-normalizzazione dati per renderne più efficiente l'accesso. Nei RDBMS, le tabelle sono definite da dati normalizzati che sono richiamati tramite query SQL e con un forte uso del JOIN, dispendioso in termini di performance, per assemblare i dati.

I sistemi NoSQL hanno parzialmente abbandonano le proprietà ACID ,tipiche dei database relazionali, per abbracciare il modello BASE(Basically Available, Soft State, Eventually Consistence) al fine di ottenere prestazioni elevate e maggiore scalabilità. A tal proposito viene in aiuto il teorema CAP, presentato da Eric Brewer in una conferenza "Principle of Distribuited Computing" [Bewer], nella quale afferma che non si può avere Consistency, Availability e Partition al tempo stesso ma solamente due di queste caratteristiche alla volta come mostrato in Figura 1.1. Per superare il più grosso limite dei RDBMS attuali, ovvero la scalabilità, i sistemi NoSQL generalmente rinunciano alla Consistency(coerenza), cioè la possibilità di avere , da parte dei client, la stessa visione dei dati su tutti i nodi del sistema distribuito, a favore delle altre due proprietà:

- Availability: un sistema disponibile deve avere repliche , deve essere sempre in grado di dare una risposta.
- Partition Tollerance: se le repliche sono su nodi diversi, questi possono essere isolati l'uno dall'altro, permettendo al sistema di funzionare correttamente anche se la comunicazione tra due punti del sistema viene interrotta.

Come scritto nel documento di Boraso e Guenzi, presentato alla conferenza Garr nell'ottobre 2010, se si sceglie di rinunciare alla Partition Tollerance i dati saranno presenti su di un'unica macchina con la sola possibilità di scalare in verticale, molto costosa; qualora si decidesse di venir meno all'Availability diminuiranno le prestazioni del sistema, in quanto al partizionamento bisogna attendere che esso venga risolto prima di soddisfare altre richieste.

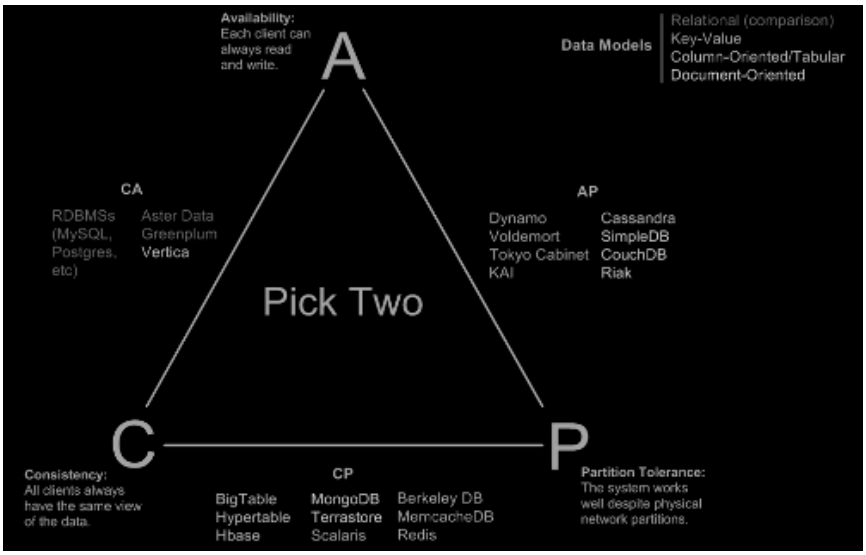


Figura 1.1

Nel tempo i maggiori esponenti del Web hanno sviluppato vari NRDBMS, tutti con gli stessi principi ma con approci implementativi leggermente diversi, in particolare sono classificati in base al tipo di modello che utilizzano per la memorizzazione dei dati:

1. **Key-Value:** sono definiti da strutture che memorizzano coppie chiave-valore come hash, array associativi, mappe, permettendo all'utente di recuperare e aggiornare il valore memorizzato data la sua chiave. Questo metodo è il più semplice da implementare, ma anche il più inefficiente se la maggior parte

delle operazioni riguardano soltanto una parte di un elemento. Tipici key-values store sono Redis, BigTable, Project Voldemort, Memcached.

2. **Column Family stores:** i dati sono organizzati in righe e colonne, ma le righe possono avere quante colonne si vogliono e non c'è bisogno di definire le colonne come prima cosa.
3. **Document store**: è l'evoluzione del metodo key/value. Rispetto ai normali database relazionali, i dati non sono memorizzati in tabelle fisse, ma inseriti in un documento (struttura schema-free) che può contenere illimitati campi di illimitata lunghezza, così se ad esempio di una persona conosciamo solo nome e cognome, ma magari di un'altra persona anche indirizzo, data di nascita e codice fiscale, si evita che per il primo nominativo ci siano campi inutilizzati che occupano inutilmente spazio.
4. **Graph Database**: rappresentano una realtà composta da una fitta rete di connessioni e la modellano sotto forma di nodi e rami di un grafo. Ai nodi come ai rami sono associate le informazioni attraverso Key-Value stores essendo 1000 volte più veloci rispetto gli altri data-base per le query che soddisfano il modello gerarchico.

Il movimento NoSQL è in continua evoluzione; la scalabilità, le performance e la prospettiva entusiasmante di pensare ai dati in maniera diversa rispetto i RDBMS hanno accompagnato la diffusione di questa nuova famiglia di data-base negli ultimi anni.

1.2 Object Data-store Mapping tool

Prima di parlare di ODM e dalla loro utilità nel mondo NoSQL accenniamo alcuni aspetti importanti dei prodotti ORM.

ORM è una tecnica di programmazione che favorisce la corrispondenza tra il dominio dell'applicazione e la base di dati risolvendo il problema dell'*impedance mismatching*[1]. Un prodotto ORM fornisce i servizi inerenti alla base di dati, mediante un'interfaccia orientata agli oggetti, nascondendo le caratteristiche implementative del RDBMS utilizzato. Nel mondo ORM c'è un forte contrasto tra chi promuove l'uso di tali sistemi e chi, invece, crede che siano un "*anti-pattern*".

[1] denota la mancata corrispondenza tra il modello a oggetti e il modello relazionale

Genn Block [Block], in "*Ten advantages of ORM"*, mostra i dieci vantaggi derivanti dall'utilizzo di prodotti ORM. Di seguito ne sono riportati solo alcuni:

- Offre funzionalità di mapping tra la logica di business e lo strato di persistenza.
- Notevole riduzione di codice. Gli strumenti ORM forniscono la possibilità agli sviluppatori di focalizzare l'attenzione principalmente sulla logica di business, mascherando le funzionalità di CRUD (che altrimenti andrebbero realizzate dal programmatore) dietro semplici operazioni.
- Rende trasparente la navigazione delle relazioni.
- Offre la possibilità di cambiare DBMS senza modificare lo strato di persistenza (solo piccole modifiche nella configurazione dell'ORM).
- Elevato supporto alla concorrenza. Più utenti possono accedere agli stessi dati contemporaneamente.

Tra i più popolari standard ORM troviamo JPA (Java Persistence API), JDO (Java Data Objects), EJB (Enterprise Java Beans), SDO (Service Data Object).

Da diversi anni l'aumento dei dati da memorizzare, l'esponenziale crescita degli utenti nella rete e la necessità di fornire scalabilità orizzontale hanno cambiato in maniera radicale la visione del mondo delle basi di dati, a favore dei database non relazionali; anche se "iniziare a lavorare con un database NoSQL risulta essere molto faticoso"[Amresh Singh]. La maggior parte delle persone ha scarse conoscenze sui NRDBMS, arrivando a conoscerne solo uno, al massimo due, e questo ha alimentato l'esigenza di sviluppare e utilizzare meccanismi simili agli ORM (Object Relational Mapping come Hibernate, Toplink, EclipseLink) per risolvere problemi d'impedance mismatching nel campo non-relazionale e fornire polyglot persistence. L'obiettivo è provvedere a uno strumento capace di sfruttare la memorizzazione di dati su più database, lasciando al framework ORM le funzionalità di CRUD e dando la possibilità agli utenti di poter utilizzare le varie strutture offerte dai sistemi NoSQL, senza averne eccellenti conoscenze.

Le sfide che i produttori di ORM affrontano sono elencate da Amresh Singh:

1. Risolvere problemi che richiedono generalizzazioni e astrazioni.
2. Ricorrere a tecniche che non appesantiscano molto le prestazioni dei NRDBMS, costruiti per ottenere scalabilità orizzontale.

3. I database NoSQL sono sistemi distribuiti con una propria architettura e topologia di rete; l'obiettivo è rispettare i presupposti sulla connettività, sulla memorizzazione e distribuzione dei dati.

L'utilizzo di un ODM per data-base NoSQL fornisce supporto alla polyglot persistence, in maniera del tutto trasparente, evitando che l'utente se ne occupi direttamente e costituendo un notevole vantaggio per la memorizzazione di dati su più database. Gli utenti possono continuare a programmare senza tener conto della complessità dei NRDBMS. Inoltre i framework ODM provvedono al mantenimento delle relazioni tra entità al momento dell'inserimento e del ritrovamento dei dati, meccanismo presente nei data-base relazionali e assente nei data-store non relazionali. Per sviluppare uno standard ODM è consigliabile seguire diversi punti che permettono di raggiungere gli obiettivi appena elencati:

- Mappare annotazioni dei framework ODM in strutture NoSQL. Ad esempio, un campo @Embedded in JPA è opportuno memorizzarlo in una Super-Column nel caso di Cassandra, o tramite collezioni in Redis.
- Se il framework fornisce un meccanismo per operare direttamente sui dati, bisogna sfruttare tale funzionalità per controllare lo scambio dati tra lo strato applicativo e persistente.
- Mantenere al minimo l'overhead dell'ODM.

2 PROGETTO ONDM

Il progetto ONDM[1] fornisce un framework che, tramite un interfaccia JPA-like e con l'ausilio di classi a livello applicativo e persistente, permette all'utente di memorizzare dati nel data-store NoSQL che meglio soddisfa le proprie esigenze.

Il lavoro di tesi è stato svolto basandosi sul prototipo di ODM con l'obiettivo di integrare due nuovi data-store, Oracle NoSQL e Cassandra (per una trattazione più approfondita vedere paragrafo), a quelli già presenti.

2.1 NRDBMS presenti

I sistemi utilizzati alla base del progetto furono due:

1. Redis.
2. MongoDB.

I motivi che hanno portato alla scelta di questi database system per realizzare un primo strato di persistenza sono da ricercare nelle loro caratteristiche: il primo scelto perché fornisce meccanismi di polyglot persistence e utilizza strutture dati già note; il secondo per l'uso di documenti, un'ottima base per rappresentare strutture del mondo Orientato agli oggetti.

2.1.1 Redis

Redis (REmote DIctionary Server) è definito key-value in-memory store, open-source, o anche data structured server, in quanto supporta diversi tipi di dati come hash, hash set, map, list, sorted list, sets, string, e fornisce diverse primitive per manipolarli. Nel marzo 2009, Redis si presentò al mondo dei data-store open-source e subito fu adottato da un elevato numero di persone. In [Russo] sono

[1] Sviluppato da un tesista nella laurea magistrale di Ingegneria informatica Uniroma 3.

evidenziate alcune delle caratteristiche chiavi che hanno permesso a Redis di affermarsi tra i NRDBMS.

1. L'utilizzo di strutture di archiviazione già note agli sviluppatori e il mantenimento dei dati in memoria, anche se persistiti su disco (utilizzato solo in caso di riavvio del sistema, per leggere i dati in memoria), hanno garantito elevate **performance**:
    - Tutti gli accessi in lettura/scrittura vengono gestiti in RAM, permettendo ad un singolo server Redis di effettuare più di 100 mila query al secondo (circa 100 mila Set al secondo e 80 mila get al secondo).
2. Tutti i comandi di una transazione sono serializzati ed eseguiti in sequenza. Una nuova richiesta da parte di un altro cliente non può essere eseguita nel mezzo di una precedente transazione e questo garantisce che i comandi sono eseguiti come una singola operazione isolata e atomica(gestione della concorrenza).
    - L'atomicità è una diretta conseguenza del fatto che Redis è single-thread; questo ne permette una gestione semplice, evitando meccanismi di sincronizzazione e loocking (spesso fonte di bug).

Come già detto in precedenza, Redis, oltre a salvare i dati in memoria, persiste anche su disco, offrendo due possibili strategie con i loro pro e contro. Se ottimizzare la durabilità dei dati non è il principale obiettivo, viene utilizzata la modalità **snapshoting** (configurabile in base alle esigenze dall'utente e dall'applicazione). A intervalli regolari o dopo un certo numero di operazioni, Redis prende uno snapshot del dataset in memoria e lo salva su di un file; a tal fine è generato un processo figlio, "*in parallelo*", che si occupa del salvataggio del dump su disco mentre il processo padre continua ad erogare il servizio. Questo permette di ripristinare facilmente l'insieme dei dati, tuttavia vi è la possibilità di perdere l'ultima versione dei dati in caso di arresto non previsto del sistema. Nella versione 1.2, è stata aggiunta una nuova modalità di persistenza, **Append On File (AOF)**, che salva, su di un file sequenziale log, tutte le operazioni di scrittura, inviategli dal server; qualora il log diventasse troppo grande, Redis lo compatterà in background senza bloccare ulteriori richieste. In caso di arresto critico del sistema, le operazioni saranno recuperate dal file e rieseguite per riportare il data-

set al suo ultimo stato. Le prestazioni dell'AOF dipendono dalla configurazione: se l'aggiornamento del log avviene ad ogni scrittura, risulta estremamente lento, ma con elevata durabilità; se configurato per eseguire l'update ogni secondo fornisce ottime performance e una ragionevole affidabilità.

La persistenza, in Redis, non porta solo aspetti positivi, ma anche alcuni problemi:

- Richiesta di memoria aggiuntiva.
- Uso pesante delle risorse I/O.
- Salvataggi asincroni possono bloccare il server per un lungo periodo.

Per i motivi appena elencati, c'è anche la possibilità di disabilitare la persistenza, affidando allo slave il salvataggio sincrono del dataset in memoria. Che cosa succede quando la memoria non è sufficiente? I dati acceduti meno frequentemente sono salvati nella *memoria virtuale* lasciando inalterate le prestazioni del database e permettendo ad una singola istanza di Redis di supportare data-set molto più grandi della memoria stessa.

Come tutti i data-store NoSQL, Redis ha la caratteristica di scalare in orizzontale attraverso meccanismi di replicazione master-slave, con lo slave che mantiene una copia esatta dei dati presenti nel master. Ogni master può avere più slave e ogni slave può accettare connessioni da parte di altri slaves.

Per avere ulteriori informazioni sulle caratteristiche di Redis e sulle strutture dati utilizzate si consiglia di consultare il sito ufficiale di Redis[1].

2.1.2 MongoDB

MongoDB è un NRDBM open-source, appartenente alla famiglia dei data-store Document oriented, sviluppato per ottenere elevate performance in lettura e scrittura e per essere facilmente scalabile con failover automatico. Il termine MongoDB deriva da "*humongous*", che significa enorme/immenso e si riferisce alla capacità di immagazzinare grandi quantità di dati. Le caratteristiche fondamentali di tale data-store NoSQL sono:

[1] www.redis.io

- Modello di dati orientato ai documenti in formato BSON[1] con uno schema dinamico.
- Auto-sharding che rende la capacità di scalare orizzontalmente molto semplice.
- Replicazione asincrona.
- MapReduce.

MongoDB organizza i dati in documenti, nei quali è possibile aggiungere attributi dinamicamente senza effettuare modifiche ai dati già inseriti; tale flessibilità e la ripetizione degli attributi in ogni documento comportano un eccessivo consumo di spazio su disco. I valori associati alle chiavi possono essere semplici dati o valori complessi come array, array di documenti. Nei data-base relazionali gli oggetti business sono rappresentati da ennuple memorizzate in tabelle con uno schema ben definito e non modificabile dinamicamente. Al contrario MongoDB salva i dati in collezioni (simile alle tabelle dei RDBMS) che a loro volta contengono uno o più documenti con grandezza massima di 4MB e senza uno schema ben preciso(schema-less); le collezioni sono organizzate in namespace con un limite massimo di 24000 (valore di default configurabile). Per ottenere alte performance con le operazioni di lettura MongoDB supporta la presenza di indici, preferibilmente sui campi acceduti frequentemente; è possibile indicizzare un solo campo o più campi. L'indice è una struttura (B-tree) dati che migliora le prestazioni delle query, riducendo l'insieme dei dati da ricercare e fornendo veloci performance di interrogazioni.

Come tutti i NDBMS, MongoDB è stato progettato per essere scalabile orizzontalmente attraverso l'utilizzo della tecnica dell'auto-sharding, che permette di dividere i dati e salvare gli shard - ovvero le partizioni ottenute dal database di partenza - su macchine differenti. L'auto-sharding divide le collezioni in partizioni più piccole e le distribuisce sui vari shard, che può risiedere su una stessa macchina o su macchine differenti. Quando tale tecnica viene attivata, bisogna scegliere una chiave, chiamata shard-key, la quale è contenuta all'interno del documento a cui fa riferimento. La scelta della shard-key dipende dalla struttura dei dati e dal modo in cui l'applicazione effettua query e operazioni di scrittura; può avere un grande impatto sulle prestazioni e sul funzionamento del

[1] BSON è basato sul termine JSON e significa "Binary JSON". Per ulteriori informazioni consultare http://www.mongodb.org/display/DOCS/BSON e en.wikipedia.org/wiki/BSON.

database e dei cluster. Sul sito ufficiale del database sono elencate alcune caratteristiche di una shard-key ideale:

- Avere un ampio numero di valori possibili.
- Non essere assegnate in maniera sequenziale ma casuale.
- Utilizzare i valori della chiave per le query di accesso ai documenti.

Spesso viene aggiunto uno o più campi appositi per definire la SK e riuscire così ad ottenere contemporaneamente queste tre caratteristiche.

Come già detto in precedenza MongoDB supporta la replicazione asincrona dei dati, realizzata in due modalità. Con il metodo di replica Master-Slave solo il nodo master ha il compito di modificare la base di dati, mentre lo slave viene sincronizzato con il master periodicamente, ed è compito dell'amministratore promuovere un nodo secondario a primario nel caso in cui il master fallisce. La modalità Replica-set invece fornisce un meccanismo di failover automatico: il nodo master sarà sostituito automaticamente con uno slave, e re-inserito nel sistema come nodo secondario non appena torna a funzionare. Per entrambe le modalità le operazioni di scrittura sono eseguite sul master. La replicazione è uno strumento importante anche per garantire la durevolezza dei dati, mantenendo le copie dei dati su più nodi.

Un'altra caratteristica importante in MongoDB è l'utilizzo di query dinamiche, tipiche dei database relazionali come ad esempio la possibilità di mostrare nel risultato di un interrogazione solo un sotto-insieme di campi(SELECT) oppure la possibilità di accedere ai dati attraverso dei criteri(WHERE). Ad esempio "*select * from posts*" è l'interrogazione SQL equivalente a "*db.posts.find()*" di MongoDB. Nel caso di query con aggregazioni complesse viene utilizzato il meccanismo di Map-Reduce che si divide in due step: nel primo passo, su ogni documento della collezione viene invocato il comando map, che produce delle coppie chiave-valore utilizzate dalla funzione reduce - secondo passo – per aggregare i dati e fornire il risultato.

Queste informazioni sono state estrapolate e riassunte dal sito ufficiale del database [Mongo] e da due articoli, [Mau] e [MokaByte]. Dopo aver fatto un quadro generale dei due data-store utilizzati all'inizio nel progetto ondm-plus, possiamo tornare a parlare dell'architettura e delle funzionalità fornite dall'ONDM.

2.3 Progetto ODMN

L'ONDM in questione ha lo scopo di permettere agli sviluppatori di avvicinarsi al mondo NoSQL senza averne elevate conoscenze. Infatti le competenze da acquisire per utilizzare uno specifico NRDBMS sono marcate e non riutilizzabili da un sistema all'atro e costituisce ancora oggi un ostacolo che spesso allontana gli utenti dall'utilizzo di tecnologie non relazioni a favore di database relazionali. È stato ridotto il rischio di lock-in, svincolando il codice dalla sintassi dei data-store cosi da consentire alle aziende di migrare da un sistema all'altro, senza eccessive modifiche nel lato della persistenze, e alle applicazioni di memorizzare dati, sfruttando le caratteristiche dei vari data-store.

2.3.1 Architettura

L'architettura del prototipo ONDM ha posto le basi per il raggiungimento degli obiettivi precedentemente elencati. Divisa in due livelli, come mostrato in Figura 2.1, permette all'utente di interagire con il sistema attraverso il livello applicativo che a sua volta si occupa di inviare i dati, pronti per la memorizzazione, al lato persistente attraverso una classe astratta DriverManager (vedere più avanti), senza tener conto del data-store utilizzato(la scelta avviene in fase di configurazione, tramite una proprietà, pesistedIn, dell'annotazione @Entity).

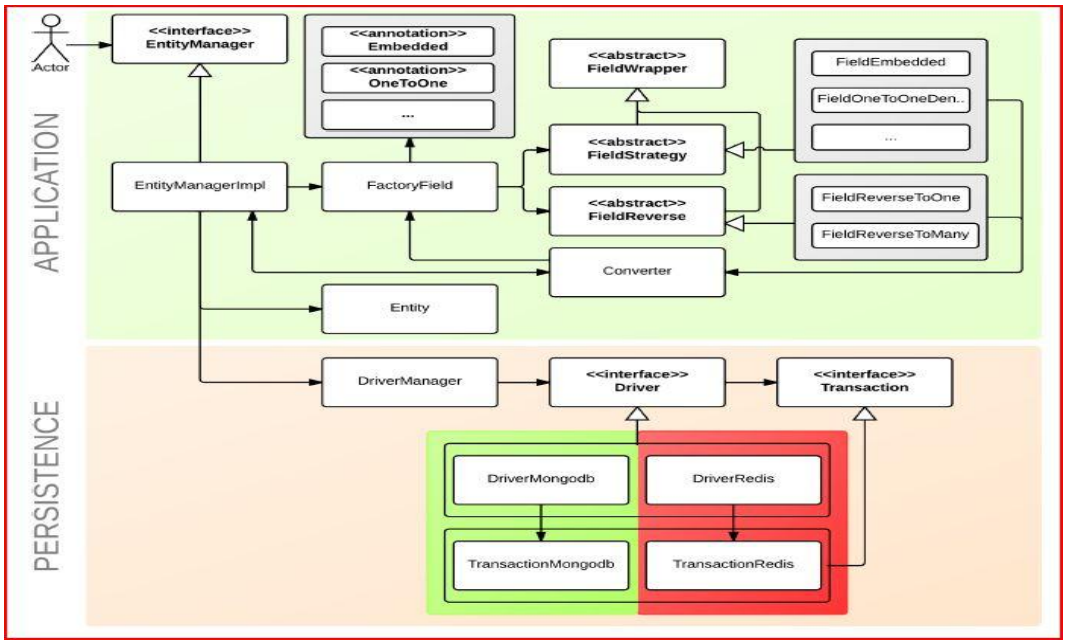


Figura 2.1 Architettura ONDM

Gestione delle funzionalità applicative

Nel lato applicativo sono presenti classi che hanno il compito di manipolare i dati inviati dall'utente attraverso un'interfaccia, **EntityManager**; dati, che incapsulati in oggetti JSON, arrivano nello strato persistente tramite il driver. L'EntityManager è la classe che offre all'utente la possibilità di effettuare transazioni e di gestire il contesto della persistenza rimuovendo, aggiornando entità già memorizzate nel database system o aggiungendone di nuove. (aggiungere intestazione metodi, concorrenza ancora non gestita). Il contesto della persistenza o *persistence context* è l'insieme delle istanze presenti nella transizione corrente; per evitare cicli nella gestione delle entità, sono state utilizzate tre strutture: due mappe che memorizzano le entità da rimuovere e da modificare, e uno stack per le entità da persistere.

Le classi Java del modello di dominio devono essere codificate in oggetti (entity beans) che possono essere persistiti. I riferimenti tra oggetti saranno mappati come relazioni unidirezionali o bidirezionali, e gli oggetti business come entità o embeddable. Lo strumento che ci permette di mappare gli oggetti della logica applicativa in oggetti dello strato persistente sono le **annotazioni**.

@Entity: Identifica una classe java come entità che può essere persistita nel data-base system scelto. L'utente può decidere su quale NRDBMS memorizzare i dati modificando la proprietà persistedIn. I valori possibili: redis, mongodb, oraclonosql, cassandra. Ogni entità avrà un identificatore all'interno del database e per il momento tale valore è estratto da un campo che obbligatoriamente si chiamerà id. L'annotazione in questione permette anche di stabilire con quale strategia rappresentare l'entità. Alle opzioni, presenti nel prototipo ONDM, sono stati aggiunti due nuovi data-store con le loro strategie:

- persistedIn= "oraclenosql", strategy= "kvj"
  Le chiavi sono rappresentate esclusivamente tramite majorKey (per la trattazione si rimanda al paragrafo sul Oracle NoSQL), composta dall'EntityName e dall'id mentre il valore è rappresentato da un oggetto JSON.
- persistedIn= "oraclenosql", strategy= "kvs"
  Ogni campo dell'entità è usato per formare una chiave(minotKey) insieme all'entityId(majorKey). I valori associati alle chiavi sono

semplici per attributi semplici, in formato JSON per attributi complessi.

- persistedIn= “oraclenosql”, strategy= “structured”

  La strategia di persistenza structured è quella che si avvicina di più alle funzionalità tipiche del data-store Oracle NoSQL. Ogni coppia campo valore formerà la coppia chiave/valore persistita nel sistema. È abbandonato il formato JSON.

- persistedIn= “cassandra”, strategy= “kvj”

  L’entityId rappresenterà la chiave di riga o row-key, univoca per ogni colonna che farà riferimento alla singola entità in formato JSON.

- persistedIn= “cassandra”, strategy= “kvs”

  L’unica differenza con la strategia kvj risiede nel fatto che ogni row-key avrà un numero di colonne pari al numero dei campi dell’entità, dove ognuna di esse conterrà valori semplici o in formato JSON(solo nel caso in cui siamo di fronte a campi complessi).

@Embeddable: rappresenta una classe che non ha una vita propria nel database ma può essere persistita solo se è contenuta in un campo di un entità.

@Embedded: l’annotazione riguarda uno o più campi di un’entità che hanno come tipo una classe segnata con l’annotazione @Embeddable. A disposizione vi sono anche due modalità di fetch: di default il fetch è EAGER, altrimenti può essere impostato a LAZY.

@ElementCollecion: indica una lista di riferimenti ad istanze di classi marcate con @Embedded. Di default il fetch è LAZY e anche qui può essere configurato a EAGER in base alle esigenze dell’utente.

In stile JPA, anche le relazioni sono rappresentate tramite annotazioni: **@OneToOne, @ManyToOne, @OneToMany, @ManyToMany**. Possono rappresentare relazioni unidirezionali o bidirezionali; in quest’ultimo caso nell’annotazione che rappresenta la parte inversa della relaziona bisogna definire la proprietà mappedBy con il nome del campo nel lato proprietario. E’ inoltre possibile definire in che modo le entità sono rappresentate tramite la proprietà *persistence*.

I valori possibili sono:

- *Normalized:* nella rappresentazione del campo viene mantenuto solo l'enityId dell'entità a cui si riferisce.
- *Denormalized:* il valore del campo viene denormalizzato, in particolare rappresenta l'intera entità a cui si riferisce con le sue relazioni.
- *Fulldenormalized:* come per il denormalized, con l'unica differenza che eventuali campi normalized di altre entità vengono rappresentati in maniera denormalizzata.

Sono due invece le modalità di fetch di un campo contrassegnato dalle annotazione di relazioni: EAGER (default per associazioni toOne) o LAZY (default per le relazioni toMany). Un campo in lazy mode viene recuperato dal database su richiesta dell'utente e non al momento della ricostruzione di una particolare istanza di entità.

Gestione della persistenza

Nello strato persistente dell'architettura sono presenti tutte le classi che si occupano di interagire direttamente con il data-store d'interesse, richiesto per la memorizzazione di una determinata entità. In particolare il **DriverManager** ha il compito di mantenere un pool con le istanze di **Driver**, create in maniera dinamica, in base al contenuto del file config.txt. Ogni volta che c'è uno scambio di dati tra lo strato applicativo e quello persistente(operazioni CRUD) l'EntityManagerImpl richiama il Driver, che tramite il metodo *getConnector(Class<?> entityClazz)* seleziona un gestore del driver(scelto tra quelli istanziati al momento della creazione dell'oggetto Driver) del particolare data-store specificato nella proprietà, persistedIn, dell'annotazione Entity. A sua volta il driver seleziona un connector che comunica direttamente con il NRDBM. Come viene selezionato il connector? Viene letto il contenuto della proprietà "strategy" dell'entità che si vuole rendere persistente, richiamando il metodo *getConnector(String strategy)* della classe Driver; il connector sarà poi restituito all'entitymanagerimpl dal drivermanager. Tramite il metodo init() il driver stabilisce una connessione con il data-base system attraverso i driver Java di basso

livello (ad esempio jedis per Redis, store per Oracle NoSQL) e una volta terminata l'interazione con il sistema la chiude con il metodo close().

L'interfaccia Connector offre i metodi per le operazioni CRUD attraverso l'utilizzo dei low-level driver messi a diposizione per il sistema NoSQL in uso. Ogni strategia di persistenza deve avere un'implementazione dell'interfaccia conncetor, in modo da venir incontro alle diverse esigenze. La creazione di nuovi moduli dà l'opportunità di implementare nuove strategia di persistenza, senza riportare ridondanza di codice: i vari metodi avranno implementazioni diverse per strategie diverse. Le strutture di input e output sono in formato JSON per perseguire il polimorfismo e l'indipendenza dai driver di basso livello; la conversione da Java a JSON e viceversa viene effettuata a livello applicativo dalla classe Converter.

Tramite le implementazioni delle interfacce Driver e Connector è possibile realizzare un nuovo modulo di persistenza dove il primo si occupa della connessione con il data-store e il secondo dello scambio dei dati con il sistema scelto dall'utente, attraverso le strutture native del driver low-level in modo del tutto trasparente al lato applicativo, che può essere messo in relazioni con diversi NRDBMS senza eseguire importanti modifiche al codice; infatti come detto più volte, la scelta è effettuata impostando le proprietà dell'annotazione @Entity.

2.3.2 Realizzazione di un nuovo modulo

Il lavoro di tesi è stato svolto con l'obiettivo di integrare due data-store NoSQL, Oracle e Cassandra, al prototipo ONDM già esistente. In particolare sono stati realizzati due nuovi moduli, ognuno dei quali devo essere costituito da due classi:

1. Implementazione della classe Driver.
2. Almeno un'implementazione della classe Connector.

Oltre ai due punti sopra elencati, vanno aggiunte le specifiche del nuovo modulo nel file config.txt e le configurazioni per le singole strategie adottate.

Implementazione del driver

Il driver ha il compito di inizializzare le connessioni con data-base system attraverso l'uso dei driver low-level messi a disposizione per il sistema, oltre ad altre funzionalità di gestione dei connector e di utilità legate ai test. L'instaurazione della connessione avviene all'interno del metodo init().

Implementazione del Connector

L'interfaccia Connector conta venti segnature di metodi semplici e rappresentanti un'unica funzionalità. È la classe che permette di scambiare dati con il database sfruttando le funzionalità del client utilizzato nella relativa classe Driver e che rappresenta una singola strategia di persistenza. L'input e l'output dei metodi è espresso in formato JSON; la conversione dei dati avviene nello strato applicativo ad opera della classe Converter.

File di configurazione

Nel file di configurazione sono presenti i nomi dei driver e dei connector con i loro path relativi, per permettere al driver di selezionare il modulo espresso dalle proprietà dell'annotazione @Entity.

3 TECNOLOGIE UTILIZZATE

In questo capitolo verranno esposte le tecnologie integrate al prototipo ONDM con le loro funzionalità, mostrando i punti di forza dei due sistemi che possono essere sfruttate contemporaneamente attraverso il meccanismo di polyglot persistence, e le differenze che intercorrono tra loro.

3.1 Oracle NoSQL

Le caratteristiche riportate sono frutto di una rielaborazione delle informazioni raccolte sul sito ufficiale della oracle[1].

Oracle NoSQL è un distribuited key-value data-store concepito per fornire un alto livello di scalabilità tramite meccanismi di replicazione basati su un insieme di sistemi configurabili, chiamati nodi d'archiviazione. I dati vengono memorizzati come coppie chiave-valore in base al valore della major-key. Una delle caratteristiche principali del database Oracle risiede proprio nell'organizzazione delle chiavi, composte dalla concatenazione di due componenti: la major Key e la minor Key. I dati che condividono la stessa parte major vengono persistiti sullo stesso nodo di archiviazione .

Le richieste al database vengono effettuate tramite un driver, che è collegato all'applicazione come una libreria .jar e acceduto usando una serie di API java. Attraverso l'utilizzo di tale driver è possibile effettuare le operazioni CRUD con accettabili garanzie di durabilità: il database offre un'alta disponibilità dei dati e un eccellente throughput.

[1] http://www.oracle.com/technetwork/products/nosqldb/overview/index.html

Le caratteristiche chiavi del database NoSQL della Oracle sono:

- ✓ **Modello dati semplice**: struttura dati di tipo chiave-valore che fornisce meccanismi di partizionamento.
- ✓ **Scalabilità**: partizione e distribuzione automatica attraverso funzioni hash sui dati. Il driver è a conoscenza della topologia della rete, consentendo uno sfruttamento ottimale dei nodi.
- ✓ **Alta disponibilità**: i nodi sono replicati e il failover è automatico in caso di un guasto imprevisto del sistema. Inoltre viene supportato il Disaster recovery attraverso la replicazione dei data center.
- ✓ **Atomicità**: offre la possibilità di eseguire le operazioni che condividono la stessa major key come una singola operazione atomica.

3.1.1 Modello dati

Il database NoSQL della Oracle memorizza i dati come coppie chiave-valore, fornendo meccanismi di ripartizione su diversi nodi che offrono un alto livello di scalabilità.

Chiavi

I dati sono distribuiti su nodi di archiviazione(descrivere più avanti), in base al valore hash della chiave primaria. Le chiavi sono formate da due componenti: MajorKey(obbligatoria) che identifica univocamente i vari record e MinorKey (opzionale); entrambe sono rappresentate da una lista di stringhe. Ad esempio, se si devono memorizzare più records rappresentanti profili utenti, l'applicazione potrà utilizzare come Major path la combinazione di nome e cognome (si può pensare alla key component come ad un percorso del file system dove ogni elemento è delimitato dallo slash "/") e identificare eventuali altri attributi usando la minor path, come mostrato in figura 3.1.[1]

```
/Smith/Bob/-/birthdate
/Smith/Bob/-/phonenumber
/Smith/Bob/-/image
/Smith/Bob/-/userID
/Smith/Patricia/-/birthdate
/Smith/Patricia/-/phonenumber
/Smith/Patricia/-/image
```

[1] Immagine ripresa dal sito ufficiale del database.

```
/Smith/Patricia/-/userID
/Wong/Bill/-/birthdate
/Wong/Bill/-/phonenumber
/Wong/Bill/-/image
/Wong/Bill/-/userID
```

Figura 3.1

Da notare che la separazione della major con la minor key avviene tramite "-".

Tutte le chiavi che condividono la stessa major key risiedono sulla stesso nodo, rendendo molto più efficienti le *performance delle query* sia in lettura sia in scrittura. Altra particolarità di tale organizzazione è la possibilità di eseguire operazioni multiple di get o delete, sugli elementi della stessa partizione, come una singola operazione *atomica*. Bisogna, però, essere consapevoli del fatto che il nodo su quale vengono persistiti i dati non può essere scelto a tempo di compilazione ma sarà deciso dal driver di basso livello a run-time.

Valori

I valori sono organizzati come array di byte e il mapping(serializzazione e deserializzazione) con la struttura data persistita è affidata interamente all'applicazione. Nel caso in cui i dati da memorizzare risultano essere troppo grandi, come immagini o video, si usa la tecnica LOS, *Large-Object-Support*, tipica del database Oracle NoSQL. Ogni valore avrà una particolare versione, utile alla gestione della concorrenza, come si vedrà più avanti(paragrafo 3.1.4).

3.1.2 Garanzie di durabilità

La durabilità è la politica che descrive quanto fortemente i dati sono persistiti, cioè la probabilità di non perderli nel caso in cui vi è un arresto critico del sistema. Le operazioni di scrittura vengono effettuate sul nodo master, che si occuperà in seguito di trasmetterle ai nodi replica prima di conseiderarle completate, con meccanismi o di acknoledgement o di sincronizzazione. Nel primo caso, le scritture sono eseguite sul master e poi inviate ai nodi replica. Quest'ultimi applicano le scritture ai propri data-base locali cosi da essere consistenti relativamente al master; viene così inviato un messaggio di acknoledgement al nodo master, il quale capisce che le scritture sono state

ricevute ed eseguite correttamente sui data-base locali e si rende disponibile per una nuova operazione. Il numero dei nodi replica può essere configurato: nessun nodo, la maggior parte(su 5 nodi almeno 3) oppure tutte le repliche. Avere un'alta durabilità significa venir meno ai requisisti di performance: un'operazione di scrittura richiede lo scambio di vari messaggi tra il master e i nodi replica e la memorizzazione dei dati su più partizioni. Nel secondo caso, le modifiche hai dati vengono effettuate prima nella cache, poi nel buffer del file system che sarà sincronizzato con il nodo d'archiviazione più stabile. È possibile configurare il tempo d'attesa per il quale il master considera un'operazione completata.

3.1.3 Replicazione

Il database Oracle NoSQL memorizza i dati su una serie di macchine, fisiche o virtuali, che vengono chiamate nodi d'archiviazione. Ogni nodo d'archiviazione ospita uno o più nodi di replicazione, che a loro volta sono divisi in partizioni; ad un alto livello d'astrazione i nodi di replicazione possono essere pensati come database che contengono le coppie chiave-valore. Questa collezione di nodi prende il nome di Kvstore.

All'interno di queste macchine, un solo nodo di replicazione funge da master, in grado di eseguire le operazioni di scrittura; i rimanenti, chiamati repliche, eseguono solo operazioni di lettura. Oracle NoSQL usa meccanismi di replicazioni per assicurare la disponibilità dei dati in caso di arresto critico. Un'architettura single-master/multiple-replica, come per il data-store oracle, richiede che le operazioni di scrittura siano applicate al nodo master per poi essere propagate alle repliche. In caso di fallimento del master sarà attivato un processo di failover automatico che permette di eleggere un nodo replica a master.

3.1.4 Gestione della concorrenza

In Oracle NoSQL la concorrenza è gestita tramite un valore, assegnato al record al momento dell'inserimento/aggiornamento nel data-store; tale valore è chiamato versione. Ogni volta che la coppia chiave-valore è acceduta, viene ritornato anche il numero della versione.

In applicazioni multi-thread è utile che le varie operazioni di aggiornamento o di cancellazione vengano eseguite solo se il numero della versione associato alla coppia persistita non è cambiato rispetto a quello ottenuto da una precedente lettura. Vi sono dei metodi della classe KVstore che permettono di raggiungere questi requisiti.

3.1.5 Esempi di operazioni

Per eseguire qualsiasi tipo di accesso al data-store, KVstore, utilizzando la classe KVstoreFactory; in particolare il metodo statico getstore() al quale va fornito un'istanza della classe KVstoreConfig. Quest'ultima ha il compito di descrivere alcune proprietà dello store in uso:

- Consistenza.
- Durabilità.
- Coppia hostname/port(identifica l'host di rete su cui risiede il nodo).
- Nome dello store(deciso al momento dell'installazione).

KVStore rappresenta il driver di basso livello attraverso il quale è possibile effettuare le operazioni sullo store una volta ottenuta la connessione al server tramite il comando run-kvlite. Kvlite è un single-process con un singolo nodo d'archiviazione e con un solo gruppo di replicazione; è ottenuto dopo l'installazione del database.

Per l'inserimento e l'aggiornamento di un record si usano diversi metodi put:

- *putIfAbsent* : inserisci l'oggetto se non presente nello store.
- *putIfPresent* : aggiorna l'oggetto.
- *putIfVersion* :consente alle applicazioni multi-thread di implementare le operazioni di lettura-modifica-scrittura mantenendo alto il livello di consistenza dei dati.
- *put* : esegue le operazioni putIfAbsent o putIfPresent in maniera trasparante all'applicazione. Se l'oggetto non è presente lo inserisce altrimenti lo aggiorna.

I metodi sopra elencati, per essere eseguiti, hanno bisogno di due parametri: la chiave con le sue due componenti major e/o minor(opzionale) e il valore da inserire/aggiornare. Di seguito viene esposto un semplice esempio.

```
/*valore da persistere*/
String valueString = "Big Data World!";

/*Si ricorda che l'uso della minor Key è opzionale.*/
Key key = Key.createKey("MajorPath","MinorPath");

/*il valore è organizzato come un array di byte*/
Value value = Value.createValue(valueString.getBytes());
/*inserisce la coppia chiave valore*/
store.put(key, value);

/*chiude la connessione allo store*/
store.close();
```

La cancellazione di un record è basata sulla chiave del record. È inoltre possibile eseguire l'operazione solo nel caso in cui il valore della versione non è cambiato, con l'uso del metodo deleteIfVersion().

```
/*Si ricorda che l'uso della minor Key è opzionale.*/
Key key = Key.createKey("MajorPath","MinorPath");

/*elimina la coppia chiave valore, in base al valore del parametro key*/
store.delete(key);

/*chiude la connessione allo store*/
store.close();
```

Come già accennato, le chiavi che condividono la stessa major key sono persistite sullo stesso nodo e su queste è possibile effettuare operazioni multiple come una singola transazione atomica.

```
/*Si definiscono le due componenti della major key. È possibile
aggiugnere eventuali componenti minor*/
ArrayList<String> majorComponents = new ArrayList<String>();

majorComponents.add("Smith");
majorComponents.add("Bob");
Key myKey = Key.createKey(majorComponents);

/*vengono eliminate tutte le chiavi che hanno come major key /Smith/Bob/
con i relativi valori*/
kvstore.multiDelete(myKey, null, null);
```

L’operazione di fetch è fornita dal metodo get che restituisce un oggetto ValueVersion che a sua volta mantiene i riferimenti agli oggetti Version e Value; si ha anche in questo caso la possibilità di effettuare accessi multipli.

```
ArrayList<String> majorComponents = new ArrayList<String>();
ArrayList<String> minorComponents = new ArrayList<String>()
majorComponents.add("Smith");
majorComponents.add("Bob");
minorComponents.add("phonenumber");
Key myKey = Key.createKey(majorComponents, minorComponents);
/*otteniamo l’oggetto ValueVersion relativo alla coppia myKey-valore */
ValueVersion vv = kvstore.get(myKey);
/*il metodo getValue restituisce il valore, mentre con il metodo getVersion otteniamo informazioni sulla versione*/
Value v = vv.getValue();
/*l’oggetto value organizzato come un array di byte va convertito in un oggetto String*/
String data = new String(v.getValue());
```

In aggiunta alle operazioni CRUD, Oracle NoSQL permette di iterare su tutti i record presenti nello store o solamente sulle coppie chiave-valore che hanno la stessa major Key, senza garantire la consistenza dei dati.

```
/* Crea l’iteratore. È possibile configurare il numero massimo di record che il metodo storeIterator deve ritornare */
Iterator<KeyValueVersion> iter =store.storeIterator(Direction.UNORDERED, 100);

/* Gli oggetti puntati dall’iteratore sono composti da tre istanze, Key-Value-Version */
while (iter.hasNext()) {
KeyValueVersion keyVV = iter.next();
Value val = keyVV.getValue();
Key key = keyVV.getKey();
}
```

Per eseguire più operazioni su di un singolo record (con la stessa major key), viene supportata l’abilità di raggruppare una lista di operazioni passata come

parametro al metodo execute(), fornendo efficienti meccanismi transazionali. L'oggetto che si occupa di creare le istanze delle operazioni è l'OperationFactory ottenuto direttamente dallo store.

```
/*oggetto che si occupa della creazione delle single operazioni*/
OperationFactory of = store.getOperationFactory();
List<Operation> opList = new ArrayList<Operation>();
List<String> majorComponents = new ArrayList<String>();
List<String> minorLength = new ArrayList<String>();
List<String> minorYear = new ArrayList<String>();
majorComponents.add("Katana");
minorLength .add("length");
minorYear.add("year");
Key key1 = Key.createKey(majorComponents, minorLength);
Key key2 = Key.createKey(majorComponents, minorYear);
String lenVal = "37";
String yearVal = "1454";

/* vengono create due operazioni di inserimento */
opList.add(of.createPut(key1, Value.createValue(lenVal.getBytes())));
opList.add(of.createPut(key2, Value.createValue(yearVal.getBytes())));
/* transazione: le operazioni vengono eseguite come un unico blocco
atomico*/
store.execute(opList);
```

3.2 Cassandra

Apache Cassandra è un NRDBMS distribuito open source, realizzato per gestire grandi quantità di dati. È una soluzione NoSQL che fu inizialmente sviluppata da uno dei più grandi social network odierni, Facebook, e che fornisce una struttura di memorizzazione orientata alle colonne, con Eventual Consistency. Alle chiavi, rappresentate da righe, corrispondono dei valori, raggruppati in famiglie di colonne, definite al momento della creazione del database. Cassandra spesso viene utilizzato in sistemi bancari o finaziari e in applicazioni per l'analisi dei dati in tempo reale, in quanto ha il vantaggio di avere le scritture molto più veloci delle letture.

Cassandra mette a disposizione dell'applicazione varie caratteristiche:

- ✓ **Schema-less**: a differenza di un database relazionale, ha uno schema molto flessibile delle struttura dati che comporta la possibilità di aggiungere colonne alle varie chiavi, senza comportare cambiamenti nei dati precedentemente inseriti. Abbandono del rigido schema tabelle-relazioni.
- ✓ **Alto livello di scalabilità**: come tutti i NRDBMS, Cassandra è stato concepito per fornire scalabilità orizzontale. Il server è installato su di una configurazione clustered nella quale più nodi cooperano per ottimizzare e distribuire le informazioni. Il throughput delle operazioni di lettura e scrittura cresce linearmente con il numero di macchine aggiunte all'interno del cluster, senza interruzione dell'applicazione.
- ✓ **Eliminazione dei colli di bottiglia**: i nodi del cluster sono tutti identici (non è presente un'architettura master-slave) senza nessun punto di fallimento. I dati vengono replicati su più nodi automaticamente.
- ✓ **Consistenza configurabile**: il livello di consistenza, sia in lettura che in scrittura, può essere configurato: ci sono diverse politiche per garantire la consistenza dei dati in scrittura ed altre per le letture. Se si aumenta la consistenza diminuiscono le performance, quindi è utile stabilire un buon compromesso per la propria applicazione.
- ✓ **Replicazione**: meccanismi di fail-over automatici. L'utente decide il numero di repliche, chiamato fattore di replicazione, e la loro distribuzione sul cluster.

3.2.1 Modello di dati

Cassandra appartiene alla famiglia dei data-store column-oriented dove i dati vengono memorizzati tramite la combinazione di righe e colonne; sembrerebbe molto simile ad una struttura dati di un qualsiasi RDBMS ma non è così. Il modello dati è semplice ma al tempo stesso complesso: una tabella risulta essere una mappa multi-dimensionale, distribuita e indicizzata da una chiave.

In un sistema relazionale, gli oggetti business sono rappresentati tramite tabelle, le relazioni attraverso l'utilizzo di chiavi esterne. I dati sono spesso normalizzati per ridurre ridondanza e sulle tabelle vengono effettuate operazioni di join, in base

alla foreign key, per soddisfare le query d'accesso, di modifica o di eliminazione. In Cassandra, il keyspace è il conteiner dell'applicazione, simile al data-base del modello relazionale, ed è usato per raggruppare al suo interno le column family, che hanno il compito di supportare le query in base al valore della chiave di riga, row-key, identificatore di un insieme univoco di colonne. L'utente può aggiungere o eliminare colonne dinamicamente, senza attenersi ad uno schema rigido, tipico del modello di dati relazionale. Vi sono altre strutture, messe a disposizione da Cassandra, come le Super Column Family e le Super Column, che saranno trattate più avanti.

Keyspace

Il data-base in Cassandra è chiamato keyspace, contente un set di column family: tipicamente un cluster utilizza il keyspace, uno per ogni applicazione, per fornire meccanismi di replicazione. Attraverso un file di configurazione è possibile scegliere la politica di replicazione del keyspace adatta alle proprie esigenze.

Column Family

La ColumnFamily è una struttura molto simile alle comuni tabelle dell'universo SQL. Identificata da un nome,unico per ogni keyspace, ha il compito di raggruppare un insieme ordinato di colonne in base al valore di una row-key determinato dall'applicazione. In Cassandra esiste anche il concetto di *Super Column Family*, che anziché contenere un insieme di colonne, contiene oggetti Column Family.

Row-key

Funge da identificatore o per un insieme di colonne all'interno di una column family o per una collezione di super column qualora si stesse definendo una super column family. Il suo utilizzo aumenta le prestazioni delle query.

Column

La struttura dati più importante del data-store Cassandra è la column. Da un punto di vista orientato agli oggetti la column rappresenta i vari campi di un oggetto, ma in pratica è un hash, formato dalle chiavi name,value e timestamp, stabilite a priori. Il nome può essere statico, stabilito al momento della creazione della column family, oppure può essere scelto dinamicamente dall'applicazione al momento della creazione della colonna. In applicazioni multi-thread, più clienti

possono richiedere all'applicazione di aggiornare contemporaneamente la stessa chiave; quindi per *gestire la concorrenza* Cassandra sfrutta il valore salvato nel timestamp.

Gruppi di colonne possono essere raggruppati nelle *Super Column*. In questo caso il campo name conterrà un array di oggetti colonna e il campo timestamp è assente. Gli esempi sulle operazioni e sull'inizializzazione di tali costrutti saranno mostrati nel paragrafo 3.2.4.

3.2.2 Garanzie di durabilità

Cassandra realizza i requisiti di durabilità aggiungendo le operazioni di scrittura alla fine del commit log, il solo ad essere sincronizzato con il file system periodicamente o tramite una finestra di batch, così da non perdere i dati in caso di fallimento del sistema. Aggiornato il commit log, i dati sono salvati all'interno della memoria, in una struttura chiamata Memtable; quando quest'ultima risulta essere piena, i valori vengono scaricati su disco in un file, SStable, e una nuova memtable viene creata. Come si può notare non vi sono meccanismi master-slave per garantire la durabilità dei dati. La SSTable permette di raggiungere alte performance nelle scritture; è una struttura rigida, che può essere cambiata solamente da un processo di compattazione e non dall'applicazione; lo spazio vuoto tra i dati viene eliminato e gli stessi vengono ordinati,indicizzati e riscritti su di un nuovo file. La durabilità è configurabile all'interno del file di configurazione. Di default il commit log viene sincronizzato ad intervalli regolari ma questo comporta il rischio di perdere dati se durante la scrittura su disco c'è un arresto critico del sistema. È possibile cambiare il valore nel file di configurazione da periodic a batch per specificare che il commit log deve essere completamente sincoronizzato con il disco prima che Cassandra riconsosca una nuova operazione di scrittura; e questo ha un impatto negativo sulle performance.

3.2.3 Replicazione

Cassandra supporta meccanismi di replicazione configurabili mediante diversi data-center, distribuendo automaticamente le copie dei dati su più nodi e

provvedendo, in caso di arresto critico del sistema, alla sostituzione dei nodi senza alcun downtime. Si basa su un'architettura peer-to-peer distribuita molto elegante e facile da mantenere, dove i nodi comunicano tra loro tramine protocolli. Quando viene creato il keyspace l'utente può decidere la strategia di replica,che indica al cluster la posizione fisica dei vari nodi; il numero totali di nodi è denominato "*fattore di replicazione"*, il quale consente di determinare il livello di consistenza a discapito delle performance.

Scegliere la giusta strategia di replica consente a Cassandra di sincronizzarsi in funzione della topologia della rete e delle esigenze dell'applicazione. La più semplice modalità, *RackUnwareStrategy*, posiziona le repliche sui nodi vicini di un unico data center, diviso in rack, ignorando la topologia fisica del cluster. La stategia *NetworkTopologyStrategy* è una strategia utilizzata per applicazioni sviluppate su più data center e realizzata attraverso l'utilizzo dello snithes, che determina la posizione dei nodi tramite IP o file di configurazione. La prima replica viene gestita come in rackUnwareStrategy, mentre le restanti sono posizionate su nodi in rack differenti.

Per la descrizione del data-store sono state utilizzate e rielaborate le informazioni presenti sul sito data-sax [1], ma soprattutto i white papers [laskshman] e [Featherston].

3.2.4 Esempi di operazioni

Tutte le operazioni elencate in seguito sono ottenute tramite l'utilizzo di uno specifico client di alto livello per Cassandra, Hector[2]. Le caratteristiche principali che hanno portato Hector ad essere uno dei client più utilizzati dagli sviluppatori Java, in procinto di avvicinarsi al mondo di Cassandra, sono elencate da Eben Hewitt in [Oreily Cassandra]:

1. *Alto livello di API orientati agli oggetti*
2. *Supporto al fail-over automatico*
3. *Gestione del pool di connessioni*

[1] http://www.datastax.com/docs/1.1/index
[2] http://hector-client.github.com/hector/build/html/documentation.html

Dopo aver installato correttamente Cassandra sul calcolatore, sarà possibile avviare il server direttamente dal file *cassandra.bat*, presente nella directory bin. A questo punto è necessario utilizzare un client, come ad esempio Hector, per interfacciarsi direttamente col database ed eseguire le operazioni opportune.

Il codice seguente viene utilizzato per instaurare una connessione con il data-store:

```
/* Viene inizializzato il cluster se non esiste già */
Cluster cluster = HFactory.getOrCreateCluster("myCluster",
"localhost:9160" );

/* Si crea il keyspace sul cluster precedentemente inizializzato  */
Keyspace keyspace = HFactory.createKeyspace("nome_keyspace", cluster);
```

Da notare: il cluster all'interno dell'applicazione viene identificato dal Hector tramite il nome; può essere diverso dal vero nome utilizzato in Cassandra. Una volta ottenuta la connessione, si possono effettuare le operazioni CRUD sulle ColumnFamily o sulle column.

Verranno proposti le intestazioni dei metodi che permettono eseguire le varie operazioni effettuate tramite l'utilizzo della classe HFactory:

- *Creazione di una Column Family*

```
static ColumnFamilyDefinition createColumnFamilyDefinition(String
keyspace, String cfName)
```

La column family creata per uno specifico keyspace va aggiunta al cluster

```
String addColumnFamily(ColumnFamilyDefinition cfdef)
```

- *Inserimento di una chiave con una colonna all'interno della column family*

A tal proposito, viene in aiuto l'oggetto Mutator<K>. Ottenuto dalla classe HFactory dal metodo *createMutator(Keyspace keyspace, Serializer<String> keySerializer)*, si occupa dell'inserimento e dell'eliminazione dei valori dal cluster.

```
insert(K key, String cf, HColumn<N,V> c)
```

Inserisce l'oggetto HColumn, che rappresenta una singola colonna, con chiave key all'interno della column family, specificata dalla stringa cf passata come parametro.

```
/* Crea l'oggetto HColumn<N,V>*/
HFactory.createStringColumn(String key, String value)
```

Come per Oracle, mutator offre la possibilità di effettuare più operazioni come una singola operazione tramite il metodo *execute()* e i metodi *addInsertion(...) e addDeletion(...)*; al contrario non è necessario che le colonne condividano la stessa chiave.

- *Fetch di una colonna*

```
/* ColumnQuery<String, String, String> columnQuery */
HFactory.createStringColumnQuery( Keyspace keyspace)
/* Modifica dei campi dell'oggetto columnQuery*/
setColumnFamily(String name_Column_Family)
setKey(String key)
setName(String name_Column)
/* Esegue la query restituendo QueryResult<HColumn<String, String>>*/
execute()
/* Restituisce l'oggetto HColumn dall'oggetto QueryResult<...>*/
get()
```

Per ottenere il valore relativo al nome, passato come parametro al metodo setName(...), si esegue il metodo *getvalue()* sull'oggetto HColumn<...>.

- *Delete di una colonna*

```
/* metodo di Mutator*/
delete(K key, String cf, N columnName, Serializer<N> nameSerializer)
```

Per eliminare tutte le colonne relative alla chiave key, basta impostare a *null* il paramtetro columnName.

- *Update di una colonna*

In Cassandra non esiste una query che richiama operazioni di aggiornamento; tuttavia, è possibile ottenere gli stessi effetti richiamando il metodo insert di mutator; se la colonna esiste già viene aggiornata, altrimenti viene aggiunta alla chiave passata come parametro al metodo sopra indicato. (vedere *Inserimento di una colonna).*

3.3 JSON

JSON, acronimo di Java Scritpt Object Notation, è divenuto con gli anni un formato semplice ed elegante per lo scambio di dati in applicazione client-server. Permette a diversi linguaggi di programmazione di inviare e ricevere dati tra loro senza eseguire alcun tipo di conversione.

JSON racchiude una lista di coppie chiave-valore, separate da virgole, tra parentesi graffe. I tipi di dati supportati

- *NULL*
- *BOOLEAN*
- *INTEGER*
- *REAL e FLOAT*
- *ARRAY DI STRINGHE*
- *ARRAY ASSOCIATIVI*
- *OGGETTI*

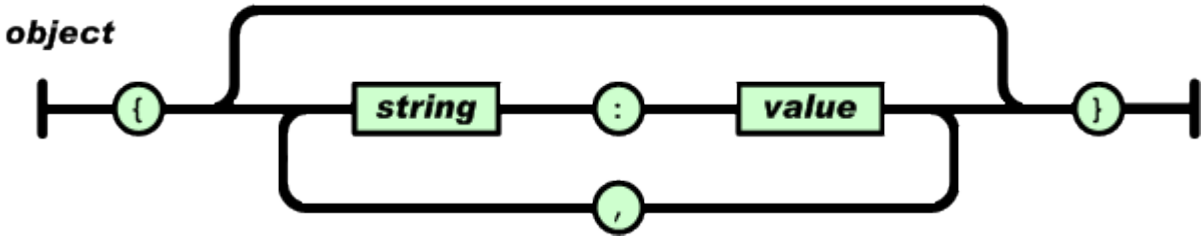
sono presenti nella maggior parte dei linguaggi come C, C++, C#, Java, JavaScript, Perl, Ruby, etc.

Rappresentazioni

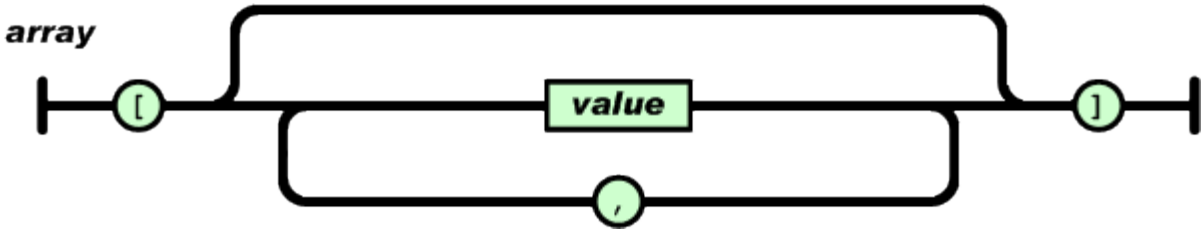
Le figure e le varie descrizioni dei formati sono state riprese da www.json.org/json-it.html.

Un *oggetto* è una serie non ordinata di nomi/valori. Un oggetto inizia con { (parentesi graffa sinistra) e finisce con } (parentesi graffa destra). Ogni

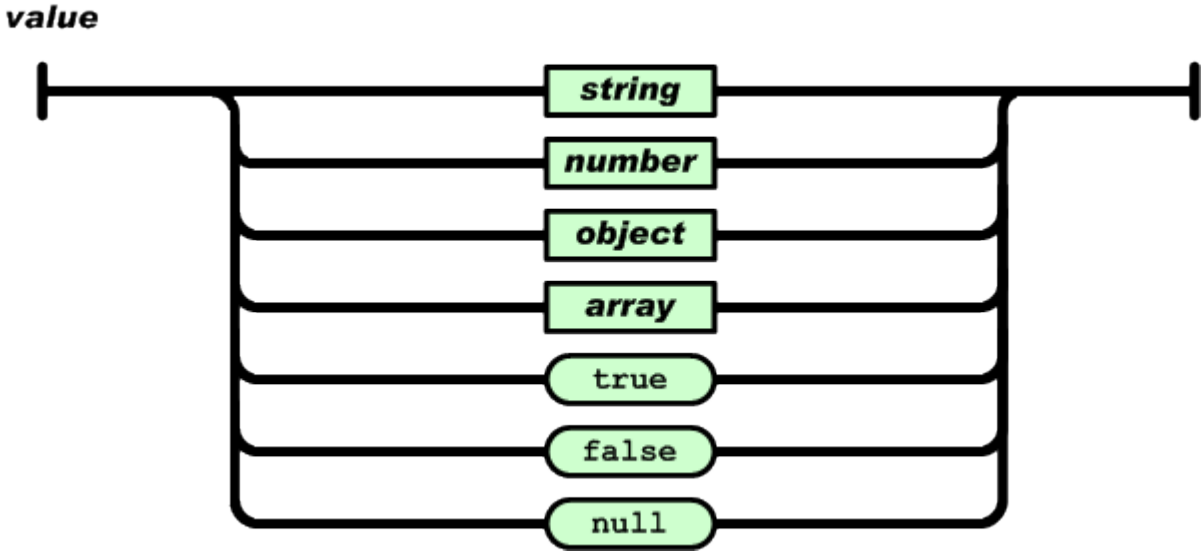
nome è seguito da : (due punti) e la coppia di nome/valore sono separata da , (virgola).



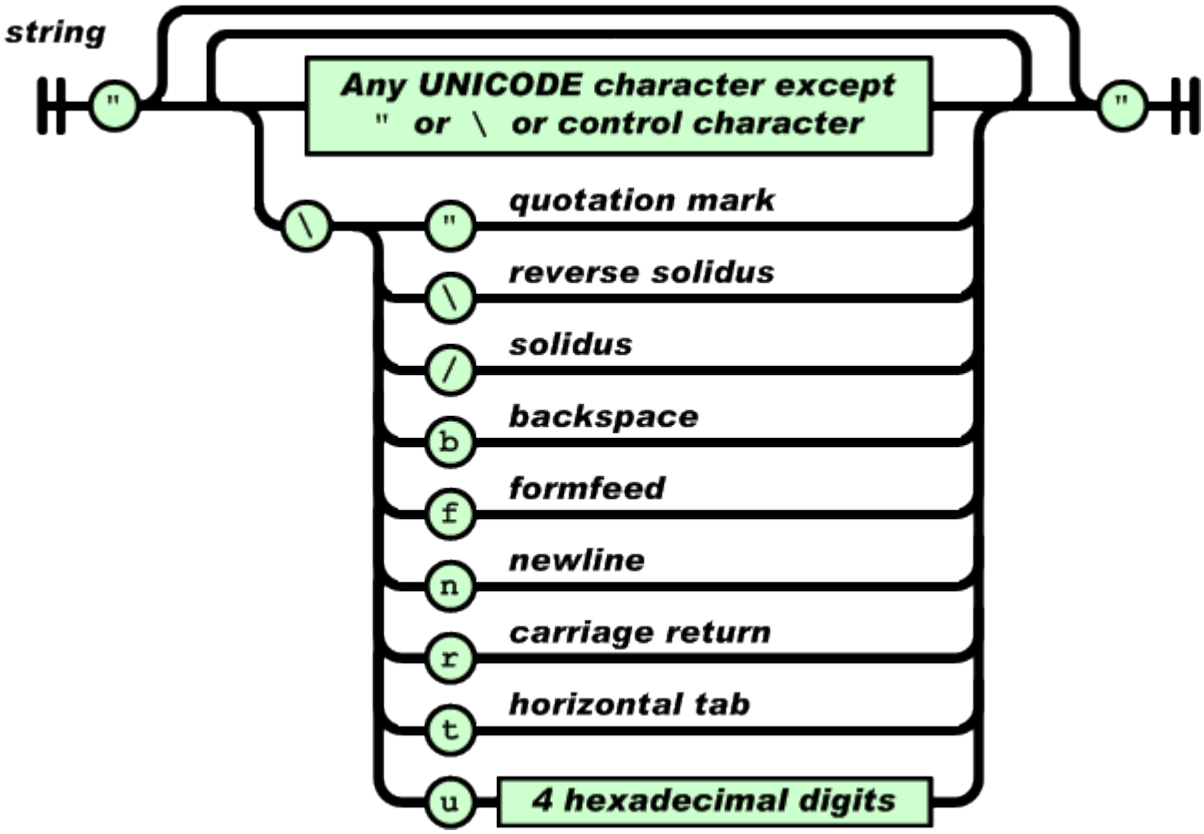
Un *array* è una raccolta ordinata di valori. Un array comincia con [(parentesi quadra sinistra) e finisce con] (parentesi quadra destra). I valori sono separati da , (virgola).



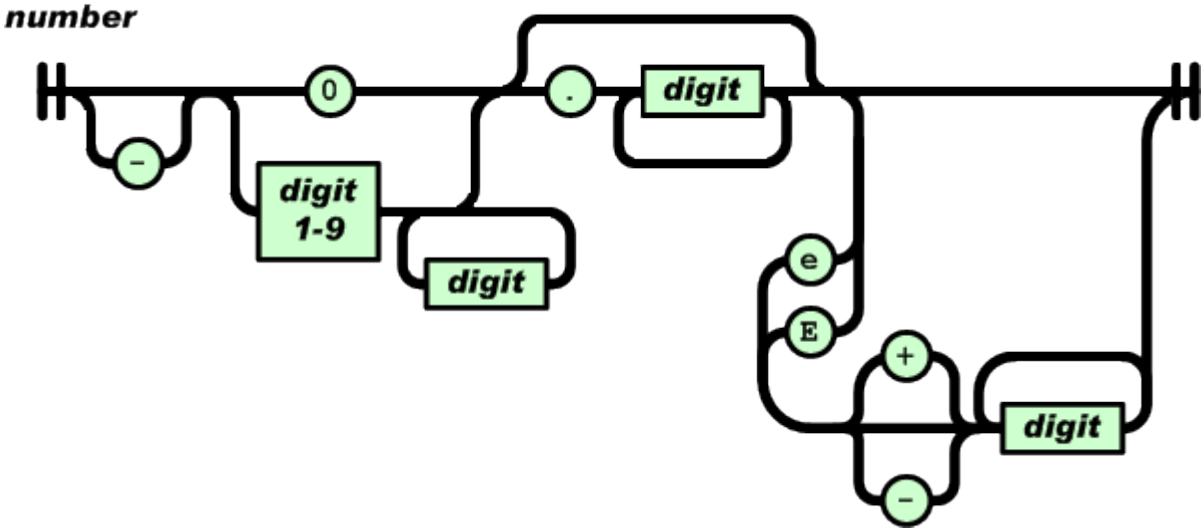
Un *valore* può essere una stringa tra virgolette, o un numero, o vero o falso o nullo, o un oggetto o un array. Queste strutture possono essere annidate.



Una *stringa* è una raccolta di zero o più caratteri Unicode, tra virgolette; per le sequenze di escape utilizza la barra rovesciata. Un singolo carattere è rappresentato come una stringa di caratteri di lunghezza uno. Una stringa è molto simile ad una stringa C o Java.



Un numero è molto simile ad un numero C o Java, a parte il fatto che i formati ottali e esadecimali non sono utilizzati.



4 CONNETTORE ORACLENOSQL STRUCTURED

In questo capitolo saranno elencate le funzionalità del connector structured relativo al data-store della Oracle, in particolare le implementazioni dei metodi che si occupano dell'accesso ai dati. Come già visto nel paragrafo 2.3.1, la classe astratta Connector offre i metodi che permettono di interagire direttamente con il sistema in uso.

Il ConnectorOracleNoSqlStructured estende la classe ConnectorOracleNoSql (che a sua volta implementa l'interfaccia Connector) e rimodella i metodi per venire incontro alle proprie esigenze, senza avere duplicazione di codice in quanto ogni strategia avrà un implementazione diversa. Per il data-store Oracle sono stati aggiunti tre nuovi connettori:

1. *ConnectorOracleNoSqlKVJ*: rappresenta la strategia più semplice, dove i dati sono memorizzati con chiavi semplici e valori complessi. La chiave è composta dalla sola MajorKey (nome più id dell'entità), mentre l'oggetto sarà persistito col formato JSON.
2. *ConnectorOracleNoSqlKVS*: chiavi complesse e valori semplici. La chiave sarà formata dalla combinazione dell'entityId (nella majorKey) e del nome dell'attributo (minorKey). Il valore sarà semplice se semplice sarà il tipo del campo a cui si riferisce, in JSON se si tratta di un riferimento ad un'altra entità o ad un oggetto @Embeddable.

3. *ConnectorOracleNoSqlStructured*: è la strategia che utilizza le funzionalità native del data-store con qualche differenza. Come sarà mostrato più avanti, all'interno della minor key un campo di tipo List o Set sarà seguito da un carattere numerico che indica la posizione all'interno della lista.

La creazione delle chiavi è affidata ad una classe specifica, OracleKeyStructured che estende la classe astratta OracleKey che offre metodi per la conversione di un oggetto String ad un oggetto Key e viceversa.

4.1 Struttura generale

Il Connector conta una ventina di metodi che sfruttano i driver di basso livello del sistema per lo scambio di dati con la tecnologia d'interesse. In Oracle viene utilizzato Kvstore come driver. ConnectorOracleNoSql ha un riferimento a tale driver, passato come parametro al suo costruttore.

```
public class ConnectorOracleNoSqlStructured extends ConnectorOracleNoSql
{
…
private OracleKey key = new OracleKeyStructured();
…
public ConnectorOracleNoSqlStructured(KVStore store) {
              super(store);
       }
...}
```

Non appena il connector è stato inizializzato dal driverOracle, sarà subito possibile effettuare le operazioni richiamate dai metodi dell'entity manager, tra cui i più utilizzati sono:

- `public void persist( Object entityObj );`
- `public <T> T find( String id, Class<T> entityClazz );`
- `public void remove( String id, Class<?> entityClazz );`

Di seguito saranno elencate le intestazioni dei metodi pubblici insieme ad ulteriori metodi di supporto. Vi sono operazioni che riguardano due tipi di strutture utilizzate all'interno del prototipo per facilitare le operazioni di aggiornamento e cancellazione in cascata in specifici casi come si vedrà più avanti.

Insert

```
public void insertEntity( String entityId, JSONObject jsonEntity );

public void addToCollectionField( String entityId, String fieldName,
JSONArray jsonAddElements );

public void addOwnerToReferredByStruct( String entityIdTarget, String
entityIdOwner, String fieldName );

public void addOwnerToIndirectReferredByStruct( String entityIdTarget,
String entityIdOwner, String fieldName );

/* metodo di supporto */
private List<Operation> recursiveInsertEntity(String recursiveKey,
Object jsonValue);
```

Retrieve

```
public JSONObject retrieveEntity( String entityId );

public Map<String, JSONObject> retrieveAllEntityInstances( String
entityName );

public Map<String, JSONObject> retriveEntitiesByField( String fieldName,
Object jsonFieldValue, String entityName );

public Object retrieveField( String entityId, String fieldName );

public JSONArray retrieveReferredByStruct( String entityIdTarget );

public JSONArray retrieveIndirectReferredByStruct( String entityIdTarget
);

public List<String> retrieveReverseField( String entityId, String
ownerFieldName );

/* Metodi di supporto */
private Object RecursiveRetrieveField(String parentKey, FieldStrategy
field,String typeSeparator)

private int lenghJSonArrayofElementCollectio( String[]
ArrayComponentOfparentKey )

public Object retrieveFieldFullDenormalized(String
parentKey,FieldStrategy field, String typeSeparator)
```

Remove

```
public void removeEntity( String entityId );

public void deleteField( String entityId, String fieldName  );

public void remFromCollectionField( String entityId, String fieldName,
JSONArray jsonRemElements );

public void remOwnerFromReferredByStruct( String entityIdTarget, String
entityIdOwner, String fieldName );
```

```
public void remTargetReferredByStruct( String entityIdTarget );

public void remOwnerFromIndirectReferredByStruct( String entityIdTarget,
String entityIdOwner, String fieldName );

public void remTargetFromIndirectReferredByStruct( String entityIdTarget
);

private JSONArray retrieveStruct( String idStruct );
```

Update

```
public void setField( String entityId, String fieldName, Object
jsonFieldValue );
```

Come si può notare dal nome di alcuni metodi privati, il connettore in analisi supporta meccanismi di ricorsione, in particolare quando si tenta di accedere ai dati; per una trattazione più dettagliata si rimanda al paragrafo successivo, dove l'analisi degli algoritmi utilizzati è divisa in vari casi d'uso.

Per capire come avviene il retrieve di un entità bisogna è importante conoscere l'organizzazione delle chiavi all'interno del datastore nei diversi casi. La minor Key è composta da tutte le chiavi presenti all'interno dell'oggetto JSON (parametro di insertEntity), rispettando l'ordine; se il campo è di tipo List o Set sarà inserito un valore numerico che rappresenta la posizione all'intero della collezione, altrimenti (caso di Embedded o OneToOne o ManyToOne) il nome dell'attributo sarà seguito dal carattere "#", utilizzato per facilitare l'accesso ricorsivo dei dati. Un breve esempio:

Palazzo e indirizzi sono rispettivamente campi Embedded (dell'entità Dipartimento) ed ElementCollection (dell'oggetto Embeddable Palazzo)

```
Oggetto JSON

Dipartimento=

{

"id":"1005",

"palazzo":{

"indirizzi":[

{"via":"Via Verdi","cap":"00127","città":"Milano","numeroCivico":"46"},

{"via":"Via Roma","cap":"00127","città":"Milano","numeroCivico":"26"}
```

```
],
"numeroPiani":"4"},
"nome":"Architettura"
}
```

In Oracle:

```
/Dipartimento/1005/-/id
1005
/Dipartimento/1005/-/nome
“Architettura”
/Dipartimento/1005/-/palazzo/#/indirizzi/0/via
“Via  Verdi”
/Dipartimento/1005/-/palazzo/#/indirizzi/0/cap
“00127”
/Dipartimento/1005/-/palazzo/#/indirizzi/0/città
“Milano”
/Dipartimento/1005/-/palazzo/#/indirizzi/0/civico
“46”
/Dipartimento/1005/-/palazzo/#/indirizzi/1/via
“Via Roma”
/Dipartimento/1005/-/palazzo/#/indirizzi/1/cap
“00127”
/Dipartimento/1005/-/palazzo/#/indirizzi/1/città
“Milano”
/Dipartimento/1005/-/palazzo/#/indirizzi/1/civico
“26”
/Dipartimento/1005/-/palazzo/#/numeroPiani
“4
```

4.2 Algoritmi per l'accesso ai dati e casi d'uso

In questo paragrafo saranno esposti gli studi di caso effettuati durante il lavoro di tesi facendo particolare attenzione agli algoritmi utilizzati per le operazioni di lettura nel data-store Oracle. L'obiettivo è quello di mostrare le soluzioni adottate per i vari problemi incontrati nel corso dell'implementazione dei metodi d'accesso ai dati. Vengono ora ricordate le intestazioni di tali metodi:

- `public JSONObject retrieveEntity( … )`
- `public Map<String, JSONObject> retrieveAllEntityInstances( … )`
- `public Map<String, JSONObject> retriveEntitiesByField( … )`
- `public Object retrieveField( … )`
- `public JSONArray retrieveReferredByStruct( … )`
- `public JSONArray retrieveIndirectReferredByStruct( … )`
- `public List<String> retrieveReverseField( … );`
- `private Object RecursiveRetrieveField( … )`
- `private int lenghJSonArrayofElementCollectio( … )`
- `public Object retrieveFieldFullDenormalized( … )`

Come visto in precedenza le chiavi sono formate dal nome dei campi dell'entità; ogni campo ha una particolare politica di persistenza che è possibile ottenere dalla classe FieldFactory che tramite il metodo getFields(Class<T> entityClass) ritorna una lista di oggetti che implementano la classe astratta FiledStrategy.

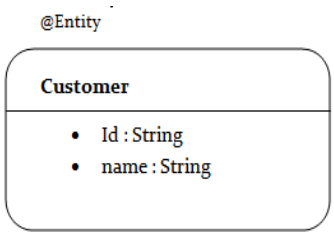
List<FieldStrategy> fields = FieldFactory.getFields(Entity.getClass(entityId));

FieldStrategy ha varie implementazioni;una per ogni strategia di persistenza: FieldSimpleType,FieldEmbedded,FieldRelationshipToOneNormalized,FieldRelationshipToOneDenormalized e altri simili per le relazioni a molti. Nel retrieve sono state sfruttate queste funzionalità per individuare il tipo dinamico dei campi delle classi coinvolte e scegliere di conseguenza la modalità d'accesso.

Le operazioni del connector, come già accennato, sono richiamate all'interno dei metodi dell'EntityManagerImpl, la cui istanza è fornita dalla classe EntityManagerFactory, ottenuta dalla classe Persistence. Al costruttore dell'oggetto factory va passata una stringa che rappresenta lo specifico costruttore da utilizzare nella persistenza dei dati, nella forma "nomeDatastore-strategia".

4.2.1 Singola Entità

```
@Entity
public class Customer{
private String id;
private String name;
public Customer(){…}
/* metodi get e setter omessi*/
}
```



Le proprietà dell'annotazione @Entity persistedIn e strategy sono rispettivamente oraclenosql e structured; saranno omesse anche per il resto degli esempi.

Si mostra anche il test che permettere di accedere all'oggetto Customer in base al valore del campo id:

```
public void testFindCustomerById() {

/* E' possibile inserire la stringa nel costruttore che indica datastore
e strategia di persistenza da utilizzare se non inserite
nell'annotazione @Entity*/
EnitityManagerFactory emf= PersistenceEntityManagerFactory();

EntityManager em = emf.createEntityManager();

em.begin();

Customer customer = em.find("1001", Customer.class);

em.close();
}
```

Il dump relativo a una singola istanza:

```
/Customer/1001/-/id
1001

/Customer/1001/-/name
Mario
```

In questo semplice caso, che non ha costituito grandi problemi, vengono utilizzati due soli metodi per ricostruire l'oggetto Customer dal data-store: retrieveEntity che sfrutta il retrieveField.

```
@Override
public JSONObject retrieveEntity(String entityId) {
JSONObject jsonEntity = new JSONObject();
List<FieldStrategy>fields=
FieldFactory.getFields(Entity.getClass(entityId));

for ( FieldStrategy field : fields ) {
      Object value=this.retrieveField(entityId, field.getName());
      jsonEntity.put(field.getName(), value);
    }
if (jsonEntity.isEmpty()) {
    return null;
    } else {
           return jsonEntity;
           }
 }
```

Come si può notare, viene presa una lista di FieldStratgy in base all'entityId e successivamente è richiamato il metodo retrieveField per ogni elemento della lista. Il valore ritornato sarà inserito nel JSONObject in corrispondenza del nome del campo a cui si riferisce. Viene ritornato un valore diverso da null solo se l'oggetto jsonEntity non è vuoto. Come si comporta retrieveField? (viene mostrata solo la parte del codice relativo al caso d'uso in analisi).

```
@Override
public Object retrieveField(String entityId, String fieldName) {
Object jsonField = null;
List<FieldStrategy> fields = new ArrayList<FieldStrategy>();
fields = FieldFactory.getFields(Entity.getClass(entityId));
for(FieldStrategy f : fields){
…
        if(f instanceof FieldSimpleType){
           Key k = key.keyFromEntityId(entityId+"_"+fieldName);
           ValueVersion vv = this.store.get(k);
           if(vv!=null){
              String value = new String(vv.getValue().getValue());
              jsonField=Converter.parseStringToJSON(value);
              }
           }
...
return jsonField;
}
```

Come avviene nel retrieveEntity si scorre una lista di FieldStrategy ed in particolare viene analizzato solo il campo il cui nome corrisponde al parametro fieldName. Poiché in questo studio di caso si ha un'entità senza nessun riferimento ad altri oggetti, i suoi campi avranno tutti valore basic type. Questa situazione rappresenta il caso base del processo di ricorsione nel quale si accede direttamente al data-store tramite il driver di basso livello. Il metodo get di

KVStore ha bisogno di un oggetto Key, come parametro, e quindi è necessaria una conversione dal formato string all'oggetto desiderato, effettuata dalla classe OracleKeyStructured nel metodo *keyFromEntityId(String entityId)*, richiamato tramite la variabile d'istanza *key* (l'oggetto key sarà sempre lo stesso per il resto degli esempi).

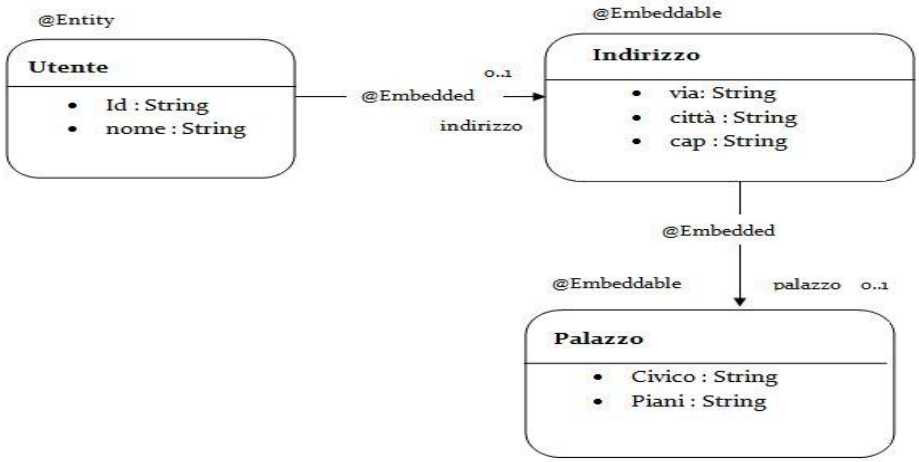
È stato pensato di impostare il codice in modo tale da poter aggiungere facilmente altri possibili casi attraverso una cascata di if else dove viene controllato il tipo di un FieldStrategy. Il caso base viene inserito all'inizio così da terminare immediatamente il controllo senza effettuare controlli superflui, come nell'esempio sopra riportato.

Vengono tralasciati esempi con singoli embedded o elementCollection e si passa subito ad esaminare casi d'uso con la presenza di molteplici classi embeddable.

4.2.2 Entità con Embedded di Embedded

```
@Entity
public class Utente{
private String id;
private String nome;
@Embedded
private Indirizzo indirizzo;
public Utente(){…}
/* get e setter omessi*/

@Embeddable
public class Palazzo{
private String civico;
private String piani;
public Palazzo(){…}
/* get e setter omessi*/
```

```
…}


@Embeddable
public class Indirizzo{
private String via;
private String città;
private String cap;
@Embedded
private Palazzo palazzo;
public Indirizzo(){…}
/* get e setter omessi*/
```



Dump

```
/Utente/1001/-/id
1001

/Utente/1001/-/nome
Mario

/Utente/1001/-/indirizzo/#/palazzo/#/civico
155

/Utente/1001/-/indirizzo/#/palazzo/#/piani
4

/Utente/1001/-/indirizzo/#/cap
00142

/Utente/1001/-/indirizzo/#/via
Via Verdi

/Utente/1001/-/indirizzo/#/città
Roma
```

I campi, indirizzo di Utente e palazzo di Indirizzo, contengono oggetti complessi, o meglio @Embeddable, e devono quindi essere gestiti in maniera diversa rispetto a come visto nel paragrafo 4.2.1. Il driver di Oracle NoSQL offre la possibilità di effettuare accessi multipli per le chiavi che condividono la stessa major key; in questo caso il metodo multiGet sulla chiave /Utente/1001/ fornirà un oggetto Map con tutte le chiavi e i rispettivi valori, ma ricostruire l'oggetto Utente dalla mappa ottenuta risulta essere problematico in quanto le operazioni multiple non possono essere effettuate su chiavi che hanno solo una parte della minor key uguale. Oltre ad un primo controllo sui campi dell'entità Utente, andranno controllati anche i campi della classe Indirizzo e della classe Palazzo tramite l'utilizzo di metodi ricorsivi.

Al metodo retrieveField è stato aggiunto il caso in cui un elemento della lista FieldStrategy è un istanza di FieldEmbedded.

```
if(f instanceof FieldEmbedded){
   Object embedded=
   Connverter.parseStringToJSON((
              recurisveRetrieveField(entityId+"_"+f.getName(), f,
              SEPARATOR).toString()));
   if(!((JSONObject)o).isEmpty())
       jsonField = o;
}
```

Come si può vedere dalla figura 4.1, sul campo Embedded viene applicato il metodo ricorsivo recursiveRetrieveField(…) ; non è stato possibile rendere retrieveField ricorsivo per due motivi:

1. La lista di FieldStrategy si ottiene tramite l'id di una entità e aggiungendo il nome del campo alla stringa enitityId, il metodo Entity.getClass(entityId) non è più utilizzabile.
2. Non era possibile cambiare la signatura del metodo.

A causa di queste considerazioni, è stato opportuno aggiungere un metodo che supportasse la ricorsione, all'interno del quale vengono analizzati ricorsivamente i campi degli oggetti Indirizzo e Palazzo ed eventuali altre classi.

Il metodo recursiveRetrieveField possiede 3 parametri:

1. String parentKey: nel caso base sarà convertita in un oggetto Key per accedere ad un particolare valore nel data-store.
2. FieldStrategy field: utilizzato per ottenere i campi della classe a cui si riferisce.
3. String separator: # se il campo rappresenta una relazione a uno oppure un Embedded; un carattere numerico nel caso di una relazione a molti o di un elementCollection.

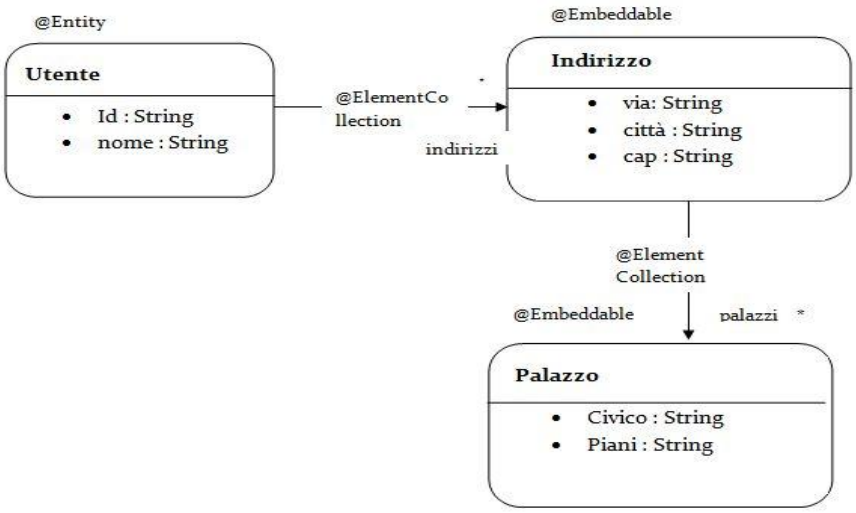
```
private Object recursiveRetrieveField(String parentKey, FieldStrategy
field,String typeSeparator) {
JSONObject result = new JSONObject();

/* Ritorna il tipo di field */
Class<?> clazzField = field.getType();

List<FieldStrategy> fields = FieldFactory.getFields(clazzField);
for(FieldStrategy f : fields){

    /* Caso base della ricorsione */
    if (f instanceof FieldSimpleType){
        Key complexKey = key.keyFromEntityId(
                          parentKey+"_"+typeSeparator+"_"+f.getName());
        ValueVersion simpleValueVersion= store.get(complexKey);
        if (simpleValueVersion!=null){
            result.put(f.getName(),
            new String(simpleValueVersion.getValue().getValue()));
            }
    }

    else if(f instanceof FieldEmbedded){
            Object value = recursiveRetrieveField(
                          parentKey+"_"+typeSeparator+"_"+f.getName()
                          , f, SEPARATOR);
            result.put(f.getName(),value);
```

```
            }

...
return (Object)result;
}
```

Tutti i campi semplici di Indirizzo vengono subito inseriti all'interno del JSONObjcet con i loro valori, mentre viene richiamata la ricorsione per il campo palazzo, sul quale vengono effettuate le stesse operazioni; poiché l'oggetto Palazzo non ha riferimenti ad altre classi si entra nel caso baso e termina la ricorsione.

4.2.3 Entità con ElementCollection di ElementCollection

```
@Entity
public class Utente{
private String id;
private String nome;
@ElementCollection
private ArrayList<Indirizzo>
indirizzi;
public Utente(){…}
/* get e setter omessi*/

@Embeddable
public class Palazzo{
private String civico;
private String piani;
public Palazzo(){…}
/* get e setter omessi*/


@Embeddable
public class Indirizzo{
private String via;
private String città;
private String cap;
@ElementCollection
private ArrayList<Palazzo>
palazzi;
public Indirizzo(){…}
/* get e setter omessi*/
```



Dump

```
/Utente/1001/-/id
1001

/Utente/1001/-/nome
Mario

/Utente/1001/-
/indirizzo/0/palazzo/0/civico
155

/Utente/1001/-
/indirizzo/0/palazzo/0/piani
4

/Utente/1001/-
/indirizzo/0/palazzo/1/civico
157

/Utente/1001/-
/indirizzo/0/palazzo/1/piani
3

/Utente/1001/-/indirizzo/0/cap
00142

/Utente/1001/-/indirizzo/0/via
Via Verdi

/Utente/1001/-/indirizzo/0/città
Roma

/Utente/1001/-
/indirizzo/1/palazzo/0/civico
15

/Utente/1001/-
/indirizzo/1/palazzo/0/piani
10

/Utente/1001/-/indirizzo/1/cap
00123

/Utente/1001/-/indirizzo/1/via
Via Milano

/Utente/1001/-/indirizzo/1/città
Milano
```

Questo studio di caso introduce le problematiche riguardanti l'accesso di collezioni, senza considerare alcun tipo di relazione; per i campi contrassegnati dall'annotazione @ElementCollection, viene inserita nella minorKey la posizione all'interno della Lista. Tale valore sarà utilizzato per la ricostruzione ordinata della lista.

Ai metodi retrieveField e recursiveRetrieveField viene aggiunto il controllo per l'annotazione ElementCollection. Grazie all'organizzazione dei due metodi non vi sono stati particolari problemi nella ricostruzione della lista di oggetti, se non quello di conoscere a priori il numero di elementi da inserire nella lista; a tale scopo è stato utilizzato un ulteriore metodo di supporto che data una chiave in formato stringa ritorna un numero negativo se non ci sono elementi relativi alla chiave stessa, altrimenti un valore numerico positivo.

RetriveField

```
if(f instanceof FieldElementCollection){
   JSONArray a = new JSONArray();
   int lenght = this.lenghJSonArrayofElementCollection(
                (entityId+"_"+f.getName()));
   if(lenght>=0){/*ci sono elementi*/
      for(int i=0;i<=lenght;i++){
           a.add(Converter.parseStringToJSON(
                 this.recursiveRetrieveField(entityId+"_"+f.getName()
                 ,f,""+i).toString()));
```

```
            }
        }
    jsonField = a; /* jsonField è di tipo Object*/
}
```

RecursiveRetrieveField

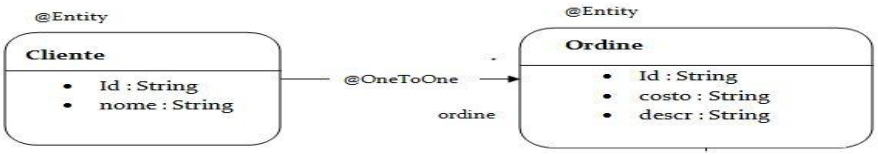
```
if(f instanceof FieldElementCollection){
   JSONArray a = new JSONArray();
   int numberOfElement = this.lenghJSonArrayofElementCollection(
                         (parentKey+"_"+typeSeparator+"_"+f.getName()));
   if(numberOfElement>=0){
      for(int i=0;i<=numberOfElement;i++){
          Object elemColl= this.RecursiveRetrieveField
                           (parentKey+"_"+typeSeparator+"_"+f.getName()
                             ,f,""+i);
          a.add(elemColl);
          }
       }

   result.put(f.getName(),a); /*result è un JSONObject */
   }
```

In entrambi i metodi sarà un JSONArray ha contenere gli oggetti relativi al campo @ElementCollection; i vari elementi contenuti all'interno dell'array saranno di tipo JSONObject. Trovato il numero di elementi, indirizzo o palazzo, persistiti, all'interno del ciclo for vengono ricostruiti gli oggetti singolarmente in base alla posizione attraverso il processo di ricorsione. In questo specifico esempio il caso base è rappresentato dall'oggetto Palazzo che ha solo campi basic type.

4.2.4 Entità con relazioni toOne

```
@Entity
public class Cliente{
private String id;
private String nome;
@OneToOne(persistence="normalized")
private Ordine ordine;
public Cliente(){}
/* setter e getter omessi*/
…}
```

```
@Entity
public class Ordine{
private String id;
private String costo;
private String descr;
public Ordine(){}
/* setter e getter omessi*/
…}
```



La proprietà persistence dell'annotazione @OneToOne può avere valore normalized, denormalized, full denormalized, a seconda di come si intende gestire la persistenza del campo. In questo caso l'attributo ordine avrà come valore l'entityId dell'oggetto. L'implementazione relativa alla classe FieldStrategy è FieldRelationshipToOneNormalized. Nell'operazione di lettura, il campo viene gestito come se fosse un tipo semplice; infatti non ci sono informazioni relative all'intera entità a cui si riferisce, ma il solo entityId, e può quindi essere considerato come caso base.

Dump *(normalized)*

```
/Cliente/1002/-/id
1002

/Cliente/1002/-/nome
Mario Rossi

/Cliente/1002/-/ordine
Ordine_5002

/Ordine/5002/-/id
5002

/Ordine/5002/-/descr
Ordine di Mario Rossi

/Ordine/5002/-/referTo
["Cliente_1002#ordine"]

/Ordine/5002/-/costo
23
```

Dal dump, si può notare come non sia necessario accedere ai campi dell'entità Ordine per ricostruire l'oggetto Cliente; la minor Key è composta dai soli attributi della classe owner side. È stato pensato di gestire FieldRelationshipToOneNormalized come se fosse un FieldSimpleType, aggiungendo un OR all'interno dell'if sia nel retrieveField che in recursiveRetrieveField.

```
if(f instanceof FieldSimpleType || f instanceof
   FieldRelationshipToOneNormalized){
   ...
   }
```

Nota: da notare come nel datastore è stata persistita la chiave /Ordine/5002/-/referTo, dove referTo non è un attributo di Ordine ma una struttura, che salva array di stringhe, utilizzata per effettuare operazioni dal lato inverso della relazione.

Dump *(denormalized)*

```
/Cliente/1002/-/id
1002

/Cliente/1002/-/nome
```

```
Mario Rossi

/Cliente/1002/-/ordine/#/id
5002

/Cliente/1002/-/ordine/#/descr
Ordine di Mario Rossi

/Cliente/1002/-/ordine/#/costo
23
```

Non è riportato il dump relativo all'ordine in quanto identico al precedente.

Al contrario del caso normalizzato, i campi dell'entità Ordine vengono utilizzati per costruire la chiave e quindi è necessario effettuare dei controlli sulle particolare istanze dei vari attributi.

Molto simile all'esempio in cui vi era un campo contrassegnato con l'annotazione @Embedded (vedi paragrafo 4.2.2), è bastato accorpare FiledEmbedded e FieldRelationshipoToOneDenormalized all'interno dello stesso if tramite la relazione logica OR.

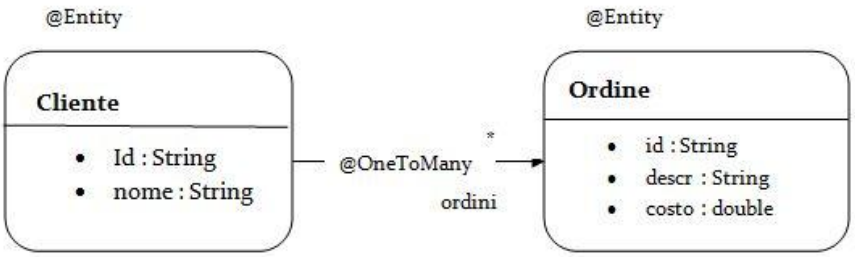
```
if(f instanceof FieldEmbedded||f instanceof FieldRelationshipToOneDenormalized)
```

Nelle relazioni bidirezionali deve essere definita la proprietà *mappedBy* unicamente nel lato inverso della relazione, con valore il nome del campo del lato proprietario. Il fetch avviene tramite il metodo *retrieveReverseRealtionship* di *EnityManagerImpl* che a sua volta richiama *retrieveReverseField* in *ConnectorOracleNoSqlStrcutured*. In questo caso vengono sfruttare le strutture dirette ed indirette per operazioni di aggiornamento, lettura ed eliminazione nel lato inverso della relazione; quindi non vi è stato il bisogno di effettuare cambiamenti nei metodi utilizzati per l'accesso ai dati.

4.2.5 Entità con relazioni toMany

```
@Entity
public class Cliente{
private String id;
private String nome;
@OneToMany(persistence="normalized")
private ArrayList<Ordine> ordini;
public Cliente(){}
/* setter e getter omessi*/
…}
```

```
@Entity
public class Ordine{
private String id;
private String costo;
private String descr;
public Ordine(){}
/* setter e getter omessi*/
…}
```



Un campo toMany normalizzato viene rappresentato tramite un array di stringhe e questo ha portato alla realizzazione di un ulteriore controllo nel metodo retrieveField.

Dump (*della sola entità cliente, owner side della relazione)*

```
/Cliente/1001/-/id
1001

/Cliente/1001/-/nome
Mario Rossi

/Cliente/1001/-/ordine/0
Ordine_5001

/Cliente/1001/-/ordine/1
Ordine_5002
```

Nella fetch, il campo ordine deve eserre ricostrutito come un oggetto JSONArray, i cui elementi sono stringhe. Nonostante l'organizzazione delle chiavi è simile al caso in cui è presente un campo @ElementCollection (nome campo+posizione

all'interno della lista), non è stato possibile aggiungere il controllo del Field all'interno di un if già implementato, come fatto in precedenza per le relazioni OneToOne; un campo ElementCollection è un JSONArray con all'interno vari JSONObject.

Per ovviare al problema è stato inserito un ulteriore controllo condizionale if, che controlla se il fieldStrategy è un istanza di FieldRelationshipToManyNormalized. Viene creato un oggetto JSONArray,vuoto all'inizio, e recuperato il numero di oggetti ordine persistiti per il cliente con id 1001(si fa riferimento al modello di dominio e al dump in figura 4.5); all'interno del ciclo for viene costrutia la chiave (cambiando di volta in volta l'indice di ricerca) che verrà usata per accedere al datastore e ottenere il valore desiderato; convertito da ValueVersion a String, sarà inseito all'interno dell'array.

```
if(f instanceof FieldRelationshipToManyNormalized){
   JSONArray a = new JSONArray();
   int l= this.lenghJSonArrayofElementCollectio(
               (entityId+"_"+f.getName()).split("_"));
   /* ciclo for per ricostruire l'array*/
   for(int i=0;i<=l;i++){
       Key k = key.keyFromEntityId(entityId+"_"+fieldName+"_"+""+i);
       ValueVersion vv = this.store.get(k);
       if(vv!=null){
          String value = new String(vv.getValue().getValue());
          a.add(value);
          }
       }
jsonField = a;
}
```

Nel caso di campo denormalizzato il retrieve si comporta come per un FieldElementCollection.

4.2.6 Relazioni tra più di due Entità

```
@Entity
public class Group{
private Long id;
private String name;
private String descr;
@ManyToMany(persistence="denormalized")
private ArrayList<Profile> profiles;
public Group(){...}}
```

```
@Entity
public class Profile{
private Long id;
private String name;
private String descr;
@ManyToMany(mappedBy="profiles")
private ArrayList<Group> groups;
@OneToMany(persistence="normalized")
private ArrayList<Feedback>
feedbacks;
public Profile(){...}
```

```
@Entity
public class Feedback{
private Long id;
private int vote;
private String comment;
@ManyToOne(mappedBy="feedbacks")
private Profile profile;
public Feedback(){…}
}
```

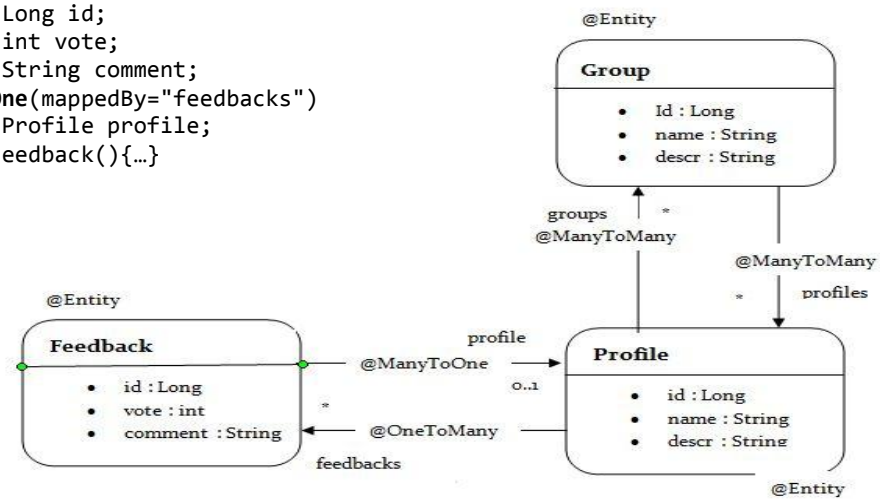


Figura 4.6

Dump

```
/Group/1/-/descr
cool!!!

/Group/1/-/id
1

/Group/1/-/name
Sport

/Group/1/-/profiles/0/descr
i'm a nice guy

/Group/1/-
/profiles/0/feedbacks/0
Feedback_1

/Group/1/-
/profiles/0/feedbacks/1
Feedback_2

/Group/1/-/profiles/0/id
/Group/1/-/profiles/0/name
Carlol
```

```
/Profile/1/-/descr
i'm a nice guy

/Profile/1/-/feedbacks/0
Feedback_1

/Profile/1/-/feedbacks/1
Feedback_2

/Profile/1/-/id
1

/Profile/1/-/name
carlol

/Profile/1/-/referTo/0
[Group_1#profiles]
```

```
/Feedback/1/-/comment
the best

/Feedback/1/-/id
1

/Feedback/1/-/indReferTo/0
[Group_1#profiles]

/Feedback/1/-/referTo/0
[Profile_1#feedbacks]

/Feedback/1/-/vote
5

/Feedback/2/-/comment
the worst

/Feedback/2/-/id
2

/Feedback/2/-/indReferTo/0
[Group_1#profiles]

/Feedback/2/-/referTo/0
[Profile_1#feedbacks]

/Feedback/2/-/vote
1
```

Il modello di dominio della figura 4.6 rappresenta lo studio di caso più complesso preso in considerazione: tre entità collegate con relazioni bidirezionali in cui:

1. Group è owner-side nella relazione toMany con Profiles.
2. Profiles è owner-side nella relazione toMany con Feedbacks.
3. Feedback è indirettamente collegato a Group tramite una particolare struttura, IndRefer.

Nell'esempio il campo profiles di Group è denormalizzato, quindi i vari sotto-campi sono considerati in base al valore della loro proprietà *persistence*.

Per le operazioni di accesso al data-store non è stato effettuato nessun cambiamento a livello di codice, in quanto i metodi retrieveField e recursiveRetrieveField erano già stati implementati aggiungendo al crescere della complessità dei casi d'uso gli opportuni controlli sui campi. A questo punto, nel lavoro di tesi, i metodi citati sono in grado di effettuare il fetch dell'entità con le relative relazioni sia se normalizzate sia se denormalizzate. Il tutto in maniera ricorsiva.

Vi è però la possibilità di persistere un campo relativo ad una relazione in maniera *fulldenormalized*(ultimo caso da prendere in considerazione). Mentre la denormalizzazione di un'entità tiene in considerazione la proprietà persistence dei sotto-campi, la modalità fulldenormalized gestisce campi normalizzati come se fossero denormalizzati. Si vede ora il dump relativo ad una singola istanza di Group, dove il valore persistence del campo profiles è cambiato in fulldenormalized.

Dump-Group

```
/Group/1/-/descr
cool!!!

/Group/1/-/id
1

/Group/1/-/name
group_1

/Group/1/-/profiles/0/descr
i'm a nice guy

/Group/1/-/profiles/0/feedbacks/0/comment
the best

/Group/1/-/profiles/0/feedbacks/0/id
1

/Group/1/-/profiles/0/feedbacks/0/vote
5

/Group/1/-/profiles/0/feedbacks/1/comment
the worst

/Group/1/-/profiles/0/feedbacks/1/id
2

/Group/1/-/profiles/0/feedbacks/1/vote
1

/Group/1/-/profiles/0/id
1

/Group/1/-/profiles/0/name
carlol
```

Considerando che il campo feedbacks di Profile è normalizzato, dovrebbero esserci solo gli enityId di Feedback referenziate dall'oggetto Profile; invece sono presenti anche i campi di Feedback. Questo ha creato problemi nel momento in cui si è tentato di aggiungere tale possibilità ai due metodi di retrieve. Non appena si incontra un campo full-denormalizzato è necessario considerare i campi normalizzati (si ricorda che sono i casi base della ricorsione) come se fossero denormalizzati, non accedere subito al data-store ma continuare nella costruzione della chiave. A tal proposito è stato opportuno implementare un nuovo metodo ricorsivo che ricostruisse i soli campi full-denormalizzati, retrieveFieldFullDenormalized che viene richiamato o da retrieveField o da recurisveRetrieveField.

```
private Object retrieveFieldFullDenormalized(String parentKey,
FieldStrategy field, String typeSeparator){

JSONObject result = new JSONObject();
Class<?> clazzField = field.getType();
if(clazzField.getSimpleName().contains("List") ||
   clazzField.getSimpleName().contains("Set"))
   clazzField = field.getGenerics()[0];

List<FieldStrategy> fields = FieldFactory.getFields(clazzField);
for(FieldStrategy f : fields){

    /*CASO BASE DELLA RICORSIONE*/
    if (f instanceof FieldSimpleType){
        Key complexKey = key.keyFromEntityId(
                          parentKey+"_"+typeSeparator+"_"+f.getName());
        ValueVersion simpleValueVersion= store.get(complexKey);
        if (simpleValueVersion!=null){
            result.put(f.getName(),
            new String(simpleValueVersion.getValue().getValue()));
            }
         }

    /*I DUE CASI RICORSIVI*/
    else if(f instanceof FieldEmbedded
            || f instanceof FieldRelationshipToOneDenormalized
            || f instanceof FieldRelationshipToOneFullDenormalized
            || f instanceof FieldRelationshipToOneNormalized){
            Object embedded=retrieveFiedFullDenormalized (
                            parentKey+"_"+typeSeparator+"_"+f.getName()
                            , f, SEPARATOR);
            result.put(f.getName(),embedded);

          }
```

```
    else if(f instanceof FieldElementCollection
            || f instanceof FieldRelationshipToManyDenormalized
            || f instanceof FieldRelationshipToManyFullDenormalized
            || f instanceof FieldRelationshipToManyNormalized){

            JSONArray a = new JSONArray();
            int numberOfElement = this.lenghJSonArrayofElementCollectio(
              (parentKey+"_"+typeSeparator+"_"+f.getName()).split("_"));
            if(numberOfElement>=0){
               for(int i=0;i<=numberOfElement;i++){
                   Object elemColl = retrieveFiedFullDenormalized
                            (parentKey+"_"+typeSeparator+"_"+f.getName()
                              ,f,""+i);
                     a.add(elemColl);
                     }
                }

            result.put(f.getName(),a);
            }
          }
            return (Object)result; }
```

Figura 4.7

Come si può vedere dalla figura 4.7, è stato considerato come caso base della ricorsione solo FieldSimpleType; mentre tutti i campi con riferimento ad una singola istanza, Embedded e relazioni ToOne, sono stati inseriti nello stesso if, così come per i campi toMany ed ElementCollection.

Si è spesso parlato di strutture particolari che offrono supporto per le operazioni sul lato inverso delle relazioni rimandando sempre la descrizione delle loro caratteristiche, verranno ora elencate in breve.

Strutture dirette

Ha il compito di mantenere una lista di riferimenti diretti, relazioni unidirezionali o bidirezionali entranti; è utilizzata per effettuare operazioni in cascata nel lato inverso della relazione. I valori persistiti all'interno di tale struttura hanno la seguente forma: *entityIdOwner+#+nome del campo nel lato owner.*

Strutture indirette

Mantiene i riferimenti indiretti ad una particola istanza di entità, come nell'esempio 4.2.6: group era owner per Profile che a sua volta rappresentava il lato proprietario della relazione Profile-Feedback. In particolare viene creata quando un entità (Feedback ad esempio) è contenuta all'interno di un campo denormalizzato o full-denormalizzato(profiles di Group) di un'altra entità senza

partecipare direttamente alla relazione(non vi è una relazione diretta tra Group e Feedback).

5 CONNETTORE CASSANDRA KVJ

Qui saranno esposti i metodi d'accesso del connector Kvj del data-store Cassandra, in particolare il retrieveEntity, che data una stringa entityId restituisce un oggetto JSON relativo all'entità persistita. I valori possibili per la proprietà strategy dell'annotazione @Entity sono due:

1. *Kvj(key-value JSON):* all'interno della column family, le chiavi di riga sono composte dall'entityId della particolare istanza di entità da inserire nello store. Ad ogni chiave corrisponde una singola colonna il cui nome è una stringa, "jsonValue", al quale corrsisponde un oggetto JSON.
2. *Kvs(key-value simple):* ogni row-key contiene una serie di colonne, ognuna delle quali corrisponde al singolo campo dell'entità nella forma nome_campo-valore; solo i campi complessi sono rappresentati tramite oggetti JSON.

5.1 Struttura generale

La classe ConnectorCassandraKVJ utilizza il client Hector per effettuare le operazioni di CRUD sul database system. Il suo costruttore ha due parametri: un oggetto cluster e un riferimento al keyspace creato in fase di configurazione. Entrambi vengono inizializati dalla classe DriverCassandra: il primo è utilizzato per avviare una connessione al server cassandra, il secondo per poter gestire le query sulle colonne contenute nelle column family.

```
public void init( ) {
CassandraHostConfigurator chc = new
                            CassandraHostConfigurator("localhost:9160");
cluster = HFactory.getOrCreateCluster(CLUSTER_NAME, chc);
if (cluster.describeKeyspace(KEYSPACE_NAME)==null)
     firstInit();
```

```
this.dumper = null;
connectorMap = new HashMap<String, ConnectorCassandra>();
}
```

Ogni volta che viene instaurata un connessione, si controlla se il keyspace è già presente all'interno del cluster; se la condizione all'interno dell'if ha valore false allora viene creata una nuova istanza del keyspace e successivamente aggiunta al cluster , tramite il metodo firstInit().

Si andrà ora ad analizzare i vari studi di caso attraverso l'analisi del metodo retrieveEntity come è stato già fatto per il connettore di oracle.

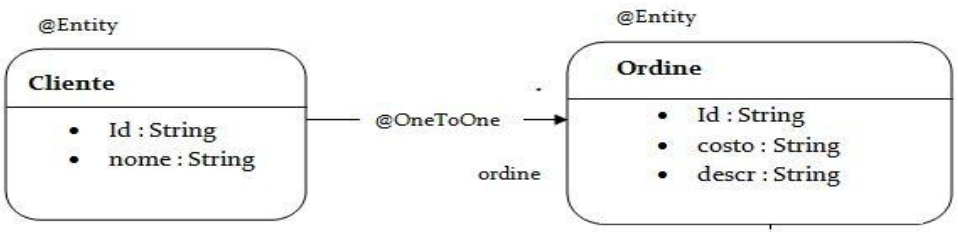
5.2 Algoritmi d'accesso per l'accesso ai dati e casi d'uso

Per comprendere al meglio il funzionamento dell'algoritmo utilizzato per l'accesso ai dati, non sarà necessario considerare tutti i casi d'uso effettuati durante l'analisi del connettore Oracle, in quanto ad ogni chiave corrisponde un singolo oggetto JSON e non vi sono problemi di ricorsione. Si partirà da subito con un caso d'uso di complessità media tralasciando la persistenza di entità singole e con oggetti Emebeddable.

5.2.1 Entità con relazioni toOne

```
@Entity
public class Cliente{
private String id;
private String nome;
@OneToOne(persistence="normalized")
private Ordine ordine;
public Cliente(){}
/* setter e getter omessi*/
…}
```

```
@Entity
public class Ordine{
private String id;
private String costo;
private String descr;
public Ordine(){}
/* setter e getter omessi*/
…}
```



Dump

```
Column Family Name: Cliente

Row_Key: Cliente_1003

HColumn(jsonValue={"id":"1003",
"ordine":{"id":"5003","costo":"0.0","descr":"Ordine di Mario Rossi"},
"name":"Mario Rossi"})
```

Come si può notare la row-key contiene una singola colonna che al suo interno mantiene l'intera istanza di una particolare entità in formato JSON. Per effettuare il fetc è necessario sapere in quale column family è salvata la chiave che identifica l'oggetto desiderato.

```
public JSONObject retrieveEntity(String entityId) {

String cfName = getCfName(entityId);
ColumnQuery<String, String, String> cQuery =
           HFactory.createColumnQuery(keyspace, StringSerializer.get(),
                        StringSerializer.get(), StringSerializer.get());
cQuery = cQuery.setColumnFamily(cfName).
        setKey(entityId).setName("jsonValue");
QueryResult<HColumn<String,String>> queryResult = cQuery.execute();
HColumn<String, String> resultColumn = queryResult.get();
if ( resultColumn != null ){
     JSONObject result = null;
     String resultValue = resultColumn.getValue();

     if ( resultValue != null ) {
          Object fieldValue = Converter.parseStringToJSON(resultValue);
          result = (JSONObject) fieldValue;
          }
return result;
}
 else {
      logger.info("CASSANDRA: Entity not found");
      return null;
      }
}
```

Il metodo getCfName(String enitityId) prende il nome dell'entità con il quale sarà identificata la column family all'interno della quale si deve accedere. Viene definita una column Query relativa al keyspace, inizializzato e aggiunto al cluster nel DriverCassandra. L'oggetto cQuery ha tre parametri che devono essere settati per effettuare efficentemente l'interrogazione:

1. Nome della column family
2. La row-key

3. Nome della colonna

Impostati i parametri alla columnQuery, è possibile eseguire l'interrogazione tramite il metodo execute(), che resituisce una collezione di oggetti Colonna (nel caso in analisi una singola istanza). A questo punto non rimane che estrarre il valore memorizzato nella colonna utilizzando getValue();infine la stringa andrà convertita in un oggetto JSON (tramite la classe Converter).

Come si può notare, l'algoritmo utilizzato esegue una query sulla column family in base al valore della chiave e al nome della colonna, sfruttando la semplicità del modello di dati di cassandra. Anche nel caso di campo denormalizzato o full denormalizzato, il metodo retrieveEntity non ha nessun problema a ricostruire l'oggetto JSONObect, in quanto memorizzato all'interno di una singola colonna.

Qualora il fetch della relazione è impostato a LAZY, il campo viene caricato su richiesta dell'utente, il quale invia all'entityManagerImpl il comando retrieveLazyField(…), che a sua volta richiama il metodo retrieveField della classe Connector.

```
public Object retrieveField(String entityId, String fieldName) {
JSONObject jsonEntity = this.retrieveEntity(entityId);
if ( jsonEntity == null )
     return null;
return jsonEntity.get(fieldName);
}
```

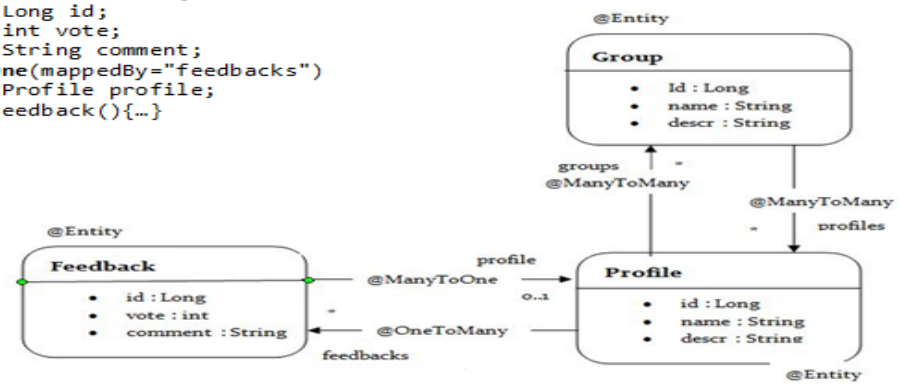
Tale metodo non fa altro che recuperare l'entità dallo store in formato JSON e ottenere il valore presente all'interno dell'oggetto JSONObject in base alla chiave fieldName.

5.2.2 Relazioni tra più di due Entità

```
@Entity
public class Group{
private Long id;
private String name;
private String descr;
@ManyToMany(persistence="denormalize
d")
private ArrayList<Profile> profiles;
public Group(){...}}
```

```
@Entity
public class Profile{
private Long id;
private String name;
private String descr;
@ManyToMany(mappedBy="profiles")
private ArrayList<Group> groups;
@OneToMany(persistence="normalized")
private ArrayList<Feedback>
feedbacks;
public Profile(){...}
```

```
@Entity
public class Feedback{
private Long id;
private int vote;
private String comment;
@ManyToOne(mappedBy="feedbacks")
private Profile profile;
public Feedback(){…}
}
```



Dump

Column Family Name: Profile

Row_Key: Profile_1

HColumn(referTo=[" Group_1#profiles",
" Group_1#profiles"])]))

HColumn(jsonValue={"id":"1","name":"carlol",
"feedbacks":[" Feedback_1",
" Feedback_2"],
"descr":"i'm a nice guy"})

Column Family Name: Feedback

Row_Key: Feedback_1{

HColumn(indReferTo=["Group_1#profiles"])

HColumn(jsonValue={"id":"1","vote":"5","comment":"the best"})

HColumn(referTo=["Profile_1#feedbacks"])

},

Row_Key: Feedback_2{

HColumn(indReferTo=["Group_1#profiles",
" Group_1#profiles"])

```
HColumn(jsonValue={"id":"2","vote":"1","comment":"the worst"})

HColumn(referTo=["Profile_1#feedbacks",
"Profile_1#feedbacks"])

}
```

Column Family Name: Group

```
Row_Key: Group_1{

HColumn(jsonValue={"id":"1","name":"group_1",
"profiles":[{"id":"1","name":"carlol",
            "feedbacks":[{"id":"1","vote":"5","comment":"thebest"},
                         {"id":"2","vote":"1","comment":"the worst"}
           ],
"descr":"i'm a nice guy"}],"descr":"cool!!!"})
```

Il retrieve di una qualsiasi entità avviene semplicemente attraverso una query dalla quale si ottiene l'oggetto column contente l'oggetto JSON da inviare al lato applicativo. L'unica differenza con l'esempio precedente risiede nel fatto che una chiave può contenere al massimo tre colonne, due delle quali memorizzano le strutture dirette e indirette accennate nel capitolo 4.

6 CONCLUSIONI

Il lavoro di tesi è stato incentrato sull'intergrazione di due nuovi sistemi nel prototipo ONDM già esistente, sviluppato sulla base degli store Redis e Mongo. Durante questi tre mesi mi sono avvicinato al mondo NoSQL, di cui fanno parte varie tecnologie sviluppate ed utilizzate per ovviare ai problemi dei data-base relazionali.

È stato portato avanti uno studio approfondito di due data-store in particolare, Oracle e Cassandra, con l'obiettivo di permetterne un utilizzo all'interno dell'ONDM, lasciando all'utente la possibilità di scegliere con quale NRDBMS memorizzare i dati. Per la realizzazione dei due nuovi moduli sono state implementate le interfacce delle classi astratte Driver e Connector; la prima si occupa di instaurare una connessione con lo store, la seconda è responsabile dello scambio dati tra il lato applicativo del framework e il sistema in uso.

Il prodotto finale è frutto di una collaborazione con altri tre laureandi, che ha portato allo sviluppo dei moduli Oracle e Cassandra per diverse modalità di memorizzazione. Ogni gruppo si è occupato di implementatre una particolare strategia, ad esempio operazioni insert/retrieve per il connettore OracleNoSqlKVJ oppure per la classe CassandraKVS. In particolare mi sono occupato degli algoritmi relativi alle operazioni di accesso ai dati per i connettori Oracle con strategia structured e Cassandra nella modalità key-value json.

Si è tentato di sfruttare al massimo le feautures e le funzionalità offerte dalle tecnologie studiate per riuscire ad ottenere ottime performance sia in scrittura che in lettura; il raggiungimento di questi obiettivi è stato portato avanti da un lato attraverso lo studio di casi d'uso di complessità via via crescente, dall'altro mediante l'implementazione dei vari metodi, cercando di lasciare spazio alla possibilità di aggiungere codice, in base alle esigenze, senza effettuare drastiche

modifiche. Tenendo conto della complessità dei vari sistemi NoSQL, l'integrazione di una tecnologia ad un prodotto ODM può portare ad un abbassamento delle performance. Infatti vi sono ancora margini di miglioramento soprattutto per le operazioni di lettura sfruttando ancor più le caratteristiche messe a disposizione dagli store: è possibile migliorare i metodi ricorsivi della classe ConnectorOracleNoSqlStructured, tentando di minimizzare il controllo delle istanze di FieldStrategy e cercando di sfruttare l'organizzazione delle chiavi persistite e le operazioni atomiche fornite dal driver low-level dello store Oracle NoSQL.

BIBLIOGRAFIA

1) Amresh Singh, “SQLifying NoSQL – Are ORM tools relevant to NoSQL?”.

2) Avinash Lakshman, “Cassandra - A Decentralized Structured Storage System”.

3) Carlo Luchessa, “Gestione della persistenza di oggetti mediante basi di dati non relazionali”. (Tesi)

4) Christof Strauch, “NoSQL Databases”.

5) Dietrich Featherston, “Cassandra: Principles and Application”.

6) Eric Brewer, “Towards robust distribuited system”.

7) Glenn Block, “Ten advantages of an ORM (Object Relational Mapper)”.

8) Mathias Meyer, “Notes on MongoDB”.

9) Michael J. Russo, “Redis, from the Ground Up”.

10) Onofrio Panzarino, “MongoDB, un database scalabile e orientato ai documenti”.

11) Oracle, “Oracle NoSQL Database”.

12) Rick Cattel, “Scalable SQL and NoSQL Data Stores”.

13) Rodolfo Borazo, “Not Only-SQL”.

14) Scott Fulton, “Oracle Formally Embraces NoSQL, Implies It Ivented NoSQL”.

Ringraziamenti

Anche se è l'utlima pagina che scrivo, è la prima a cui ho pensato.

Se ora sono qui, a scrivere questa tesi e questi ringraziamenti lo devo alla mia famiglia: i miei fratelli, Andrea e Simone, mia madre e mio padre, che mi hanno sempre sostenuto ed incoraggiato nelle mie scelte e che mi hanno permesso di giungere a questo primo traguardo della mia vita, nonostante i molti sacrifici.

Nel mio percorso universitario ho conosciuto persone fantastiche, sempre pronte a darmi una mano in caso ne avessi avuto bisogno. In particolare vorrei ringraziare Davide, Luca e Antonello che hanno condiviso con me questa grande esperienza fino all'ultimo e su cui ho potuto sempre contare, cosi come Giampaolo, Mauro e Riccardo, conosciuti al di fuori dell'Università ma a cui devo molto per l'essere stati sempre disponibili.

Un ringraziamento va anche ai componenti della squadra 'SuperSantos' con cui ho giocato insieme nel torneo interfacoltà.

Come non parlare poi dei 'torricellesi, ragazzi fantastici, direi i migliori di tutti. Non sto qui ad elencarli, ma a tutti loro dico grazie per aver reso indimenticabile ogni momento passato insieme. Veramente grazie di cuore a tutti voi.

Infine ci sono Mauro e Mattia, i miei coinquilini nonché due delle persone più importanti della mia vita che non hanno mai dubitato di me e che mi sono stati sempre vicino, incoraggiandomi ad andare avanti nei momenti di difficoltà, sia nel contesto 'Univeristà' che nella vita di tutti i giorni; a loro un GRAZIE SPECIALE.