Contents

Chapter 1

Introduzione

Chapter 2

NoSQL e OrientDB

2.1 Il movimento NoSQL

Prima di parlare di OrientDB è necessario introdurre il concetto di *database non relazionale* e spiegare il come ed il perchè del movimento NoSQL.

2.1.1 Un pò di storia

Possiamo rilevare che la prima comparsa del termine NoSQL, sia riconducibile ad un italiano, Carlo Strozzi, che, sul finire degli anni '90, ideò un database, lo *Strozzi NoSQL*, che non si avvaleva della sinstassi SQL per accedere ai dati memorizzati. Nonostante ciò non è possibile rilevare in questo lavoro alcuna caratteristica che sia precorritrice dell'attuale *movimento* NoSQL.

La nascita formale del movimento è datata 11 giugno 2009. All'epoca già esistevano dei database che non si basavano sul modello relazionale dei dati, principalmente stiamo parlando di BigTable (da Google), e di Dynamo (da Amazon), e che avevano ispirato il lavoro di altri informatici (portando alla nascita di colossi quali MongoDB, Cassandra e Voldemort), ma non avevano ancora una loro collocazione precisa. In tale data Johan Oskarsson si trovava a San Francisco per un metting su Hadoop[1] ed in tale occasione decise di organizzare lui stesso un convegno sulle novità che il mondo dei database stava vivendo.

[1]Framework, rilasciato con licensa open source e sviluppato in Java, che permette il calcolo distribuito scalando da un server fino a migliaia di macchine. Il pacchetto include, oltre ai pacchetti base: un file system distribuito (HDFS), un framework per il job scheduling e la gestione delle risorse ed un sistema per l'elaborazione parallela.

Il nome di questo talk doveva essere breve, significativo e con pochi risultati su Google, così che una successiva ricerca permettesse di trovare senza difficoltà i riferimenti a questo storico metting. Così, dopo una breve consultazione online, si optò per NoSQL, nome proposto da Eric Evans sul canale IRC di Cassandra. La ricerca degli interventi che si sarebbero succeduti durante il meeting richiedeva che i database presentati fossero *open-source*, *distribuiti* e *non relazionali*. Durante questo *Woodstock* si discusse delle nuove tecnologie emergenti: Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB e MongoDB.

2.1.2 Perchè NoSQL

Il termine NoSQL sebbene sia un termine forte, che colpisce, come venne riconosciuto già all'epoca del meeting, non determina in positivo i database che a questa categoria appartengono. Piuttosto descrive tutto ciò che un database NoSQL *non* è.

A tutt'oggi non esiste una definizione standardizzata di NoSQL; non si è neppure concordi su quale sia il significato da attribuire a questo acronimo: c'è chi sostiene che NoSQL sia l'acronimo di Non SQL, e c'è chi sostiene, e sono sempre di più, che sia l'acronimo di Not only SQL. Secondo l'autore di questo testo, l'interpretazione corretta da attribuire a NoSQL è la prima, ossia Non SQL: a tal proposito si noti che, se così non fosse, la sigla sarebbe stata NOSQL.

Tuttavia è possibile individuare nel mondo del NoSQL delle caratteristiche condivise, seppur con delle eccezioni, da tutti i facenti parte di questa categoria:

- *Rinuncia all'SQL.* Evidentemente i database NoSQL non fanno uso di SQL, sebbene alcuni di essi offrano all'utente un'interfaccia SQL-like per rendere più dolce il passaggio dai database relazionali a quelli non relazionali (Cassandra, con CQL, ne è un esempio).

- *Progetti open-source.* Come richiesto da Johan, i database NoSQL dovrebbero essere open-source. Attualmente la maggior parte di essi sono open-source, ed offrono assistenza a pagamento, o versioni definite *Enterprise*, ovvero versioni con dei tool di gestione avanzati non disponibili se non pagando una licenza.

- *Scalabilità.* Quasi la totalità dei database NoSQL offre la capacità di scalare orizzontalmente, piuttosto che verticalmente. Torneremo su questo

argomento nei paragrafi successivi.

- *Tempistiche.* Questo è l'unico punto che viene condiviso da tutti: solo i database nati nel XXI secolo sono definiti NoSQL, in quanto nati con lo scopo di soddisfare le necessità che in questo periodo si sono manifestate. Tutti i database non relazionali nati nel XX secolo vengono chiamati *BC*, acronimo di *Before Codd*[2]

- *Schema-less.* La quasi totalità dei database non relazionali permette di lavorare in modalità *schema-less*, ovvero senza uno schema fisso da rispettare. Questo permette di aggiungere campi ai record senza necessità di modificare lo schema. Questa caratteristica è particolarmente utile nelle situazioni in cui si ha una non uniformità dei dati, situazione che in database relazionale costringe ad un inutile spreco di spazio. Non si deve pensare, però, che non sia possibile definire, *volutamente*, uno schema da rispettare: ci sono database non relazionali che permettono di operare in modalità *schema-full* - orientDB è uno di questi.

Torniamo ora sul termine NoSQL: i sostenitori del "Not only SQL", seppur errando nell'interpretazione dell'acronimo, hanno perfettamente capito qual'è lo spirito di questo movimento: non si ha una tecnologia specifica, un pattern con cui risolvere dei problemi. Piuttosto abbiamo una classe di soluzioni, un *set* di possibilità, l'una diversa dalle altre. A tal proposito, va sottolineato che NoSQL non significa rinnegare l'SQL, ne abbandonarlo. Tutt'altro: il 90% dei problemi di persistenza potrebbe essere risolto usando dei database relazionali.

Va capito che partecipare al movimento NoSQL significa ampliare le proprie possibilità, avere delle soluzioni innovative e *specifiche* per il problema in questione: *i database relazionali sono una possibilità per il data storage, e non la soluzione.* Questo approccio alla persistenza viene detto *polyglot persistence.*

Bisogna sottolineare che, proprio per il periodo che li ha visti nascere, i database NoSQL rappresentano, come abbiamo visto sopra, la soluzione ideale per lo storage di grandi dimensioni di dati o per la divisione su più macchine (sia con lo *sharding*, che con la replicazione, sia essa *multi-master* o *master-slave*), rendendoli la soluzione ideale per applicazioni web.

Come abbiamo visto, i database NoSQL offrono caratteristiche non omogenee, ed in base a come operano è possibile riconoscere delle classi:

[2]Edgar Frank "Ted" Codd, informatico inglese che, durante la sua esperienza lavorativa presso la IBM, ideò il modello relazionale per la gestione delle basi dati relazionali.

- *Document store.* Il concetto centrale è il documento, che incapsula i dati e li codifica in uno standard che può essere, tra gli altri, l'XML, il JSON o il BSON.
- *Grafi.* Questo tipo di database è ottimale per soluzioni in cui tra i dati intercorrono un numero non predeterminato di relazioni, come nel caso dei social network, o delle mappe stradali.
- *Key-Value store.* Il database memorizza i dati senza schema, e e permette il loro recupero attraverso una chiave, come nelle hashmap.
- *Object database.* Il database memorizza dei veri e propri oggetti.
- *RDF database.* Il database memorizza le informazioni e le cataloga in base ai loro metadati, così come specificato dal RDF[3].
- *Tabulare.*dtgh???
- *A tuple*dfg???

2.1.3 NoSQL ed il teorema CAP

Quando si ha a che fare con un database, a causa di un teorema, detto *teorema CAP*, bisogna sempre sapere, prima di fare una scelta, sapere cosa si vuole ottenere.

Prima di tutto, però, analiziamo questo teorema, diventato tale nel 2002, quando S. Gilbert e N. Lynch dimostrarono la congettura di E. Brewer, risalente al 2000. Esso afferma che in un sistema distribuito è impossibile soddisfare contemporaneamente tutte e tre le seguenti garanzie:

- *Consistenza* (tutti i nodi vedono gli stessi dati nello stesso momento).
- *Disponibilità* (ogni richiesta deve ricevere una risposta su ciò che è andato a buon fine e su ciò che non lo è)
- *Tolleranza alla partizione* (il sistema continua a funzionare nonostante un arbitrario numero di messaggi venga perso o ci sia un problema in una parte del sistema)

[3]Resource Description Framework, standard definito dal W3C

In base a quale garanzia si decide di abbandonare, si avrà un sistema definito ACID o un sistema definito BASE. Nel primo caso, ACID è l'acronimo di Atomico, Consistente, Isolato, Duraturo, e, come sembra evidente, questo tipo di database garantisce una sicurezza senza pari, compromettendo, però, le prestazioni e la disponibilità.

Al contrario, un sistema BASE, coerentemente con suo significato (consistenza effettiva, soft-state, fondamentelmente disponibile), risulta essere orientato alle performance ed alla disponibilità parallela, a scapito della coerenza.

Attualmente i database relazionali incarnano il principio ACID, e risulta possibile una scalabilità verticale, ovvero aggiungere risorse (ad esempio CPU) ad un singolo server, su cui è mantenuto il database, mentre i database non relazionali incarnano il principio BASE, rendendo possibile la scalabilità orizzontale, ovvero aggiungere risorse in parallelo, quindi distribuire il database su più macchine ed incrementare le prestazioni parallelizzando l'elaborazione.

2.2 Introduzione ad OrientDB

OrientDB nasce nel 2010 per opera di Luca Garulli, già autore, presso Asset Data, di Roma Framework. Come database NoSQL si caratterizza per le sue prestazioni e per la portabilità, derivante dal fatto di essere completamente scritto in Java e di non dipendere da librerie esterne, oltre che per le sue dimensioni estremamente ridotte: il pacchetto, completo di API, si aggira intorno ai 6MB. Il suo punto di forza è la versatilità, potendo gestire diverse modalità:

- *Grafo* o *Documentale*. L'utente decide che tipologia di database creare al momento dell'inizializzazione, se documentale o a grafo.
- *Transazionale* o *non transazionale*. L'utente decide al momento della stesura del codice che si interfaccia con il database se usare le transazioni, rendendo l'ambiente performante nel caso di un ambiente client-server che richiede continue modifiche sui dati, o di non usarle, effettuando i salvataggi ad ogni modifica.
- *Schema-less* o *schema-full* o *mixed*. L'utente decide se impostare uno schema predefinito, se lavorare senza vincoli, o se impostare per alcuni elementi un vincoli, e lasciare liberi altri. Tutto ciò anche a database popolato.

- *Local* o *embedded* o *in memory.* L'utente, all'atto della creazione dell'istanza, decide se creare un database remoto, con cui sarà possibile comunicare solo tramite server, locale, con cui sarà possibile comunicare usando direttamente le API fornite, senza necessità di un server attivo (come SQLite) o in modalità in memory, salvando, cioè, i dati in RAM, ed accedendovi solo tramite API.

Oltre a ciò non va tralasciata la potenza di OrientDB, che si basa su un innovativo algoritmo di indicizzazione, definito MVRB-tree (multi value red black tree), derivato dai red-black tree e dai B+tree con i benefici di entrambi, che permette di raggiungere i 150000 record memorizzati al secondo[4], e la sua predisposizione ad un ambiente web, con il supporto nativo ad HTTP e Rest.

Infine, da segnalare la presenza di una sintassi SQL-like che permette la migrazione facilitata da un RDBMS ad OrientDB.

2.3 L'architettura di OrientDB

2.3.1 Lo storage fisico

Come abbiamo precedentemente visto, OrientDB può funzionare con tre tipi di storage: in maniera locale, ed in questo caso l'accesso al database avviene dallo stesso processo che lo ha richiesto, rendendo impossibile l'aprtura dello stesso da parte di un altro processo; in maniera remota, ed in questo caso l'accesso avviene attraverso il server, che risiede su un processo distinto da quello del richiedente, mediante il protocollo REST, permettendo di gestire anche l'accesso concorrente; in memory, ovvero tutti i dati rimangono in RAM, e nessun dato viene scritto su file system.

Come avviene realmente lo storage dei dati su disco? OrientDB ha tre strutture di storage separate, che si riflettono poi sull'indirizzazione dei record, come vedremo successivamente. La prima, presente per garantire che si usino le transazioni sicure, è detta *TxSegment*, ed è quella parte che si occupa di fare il logging dei cambiamenti avvenuti e che, in caso di fallimento, si occupa di fare il *rollback* dei dati.

La seconda separa i dati in *Cluster*, ovvero un gruppo molto generico di dati raggruppati o in base al loro tipo (ad esempio il cluster 'Seas' potrebbe contenere tutti i record dei vari mari) o in base a dei valori (ad esempio il

[4]testato su un HP Pavillon dv6 con Intel Core i7 720q, 4GB RAM e HDD E-SATA 7200rpm

cluster 'hotelDiLusso' potrebbe contenere tutti gli hotel che hanno 4 o più stelle). OrientDB usa due o più file per gestire i cluster: uno o più file con estensione .ocl che contengono i puntatori ai record effettivi, ed uno ed un solo file con estensione.och che contiene gli *hole*, ovvero puntatori a record eliminati e che possono essere dunque rimpiazzati o deframmentati.

Infine abbiamo i *Data segment*, ossia i file che poi, concretamente, rappresentano il contenuto di ciascun record. Come per i cluster, i data segment necessitano di due o più file, almeno uno con estensione .oda, che contiene i record, ed uno ed un solo file con estensione .odh che contiene gli hole.

Ciascun record in OrientDB ha un identificatore univoco, chiamato *RecordID*, noto anche con l'acronimo RID. Esso è nella forma #<cluster>:<position>, dove cluster è l'ID del cluster considerato e position è la posizione assoluta del record nel cluster. Entrambi i numeri sono positivi, tranne nel caso si stiano trattando record temporanei (come nelle transazioni): il tal caso avremo numeri negativi. I RID in OrientDB sono univoci, e quindi non sarà necessario fornire i documenti di chiavi primarie come in un database relazionale.

Ciò detto, quando ad OrientDB viene richiesto un record si attiva il seguente processo: l'utente chiede il record #x:y, il database verifica che effettivamente l'utente abbia i permessi sufficienti per caricare il record richiesto, se si, lo storage chiede al Cluster x dove si trovi il file y, e questo gli risponderà specificando *offset*, *dimensione*, *tipo* e *versione*; a questo punto lo storage interrogherà il file corretto, leggendo i byte interessati e restituendoli al database che, a sua volta, li restituirà all'utente.

2.3.2 Lo storage logico

L'unità fondamentale dello storage logico di OrientDB è la classe, intesa come ci si aspetterebbe, ovvero come concetto preso dall'Object Oriented paradigm. A livello fisico, una classe corrisponde ad un tipo di record: quindi creare una classe equivale a creare un cluster. In OrientDB è prevista l'eredità delle classi, ovvero è possibile creare una sottoclasse che estende una classe già esistente e ne eridita tutti gli attributi. Poichè potrebbe rendersi necessario creare classi che non hanno effettivamente dei record, ma che fungono da superclasse, è stato previsto dagli sviluppatori un meccanismo per definire astratta una classe, ed evitare di creare cluster inutili.

Come abbiamo anticipato, OrientDB fornisce il supporto alla sintassi SQL, ed in questo caso le classi devono essere utilizzate come fossero tabelle, o, in

alternativa, al posto del nome di classe è possibile usare l'id del cluster cui la classe fa riferimento.

La cosa interessante dei database non relazionali che lo scrittore ha dato per scontato nella trattazione introduttiva al mondo NoSQL, è il fatto che non si ha più il vincolo dell'atomicità dei valori. Questo permette ad OrientDB di ottimizzare le relazioni tra record, introducendo il concetto di *relazione*:

- *Referenziata.* In questo caso vengono salvati nel documento link diretti agli elementi con cui il documento stesso è in relazione, permettendo il caricamento senza costose JOIN tra tabelle. Questo tipo di relazione è bidirezionale in quanto ciascun record conserva il RID del record con cui è in relazione.

- *Embedded.* In questo caso i record embedded sono contenuti nel record che li incorpora. E' una relazione forte, rappresentabile come una relazione di composizione UML. In questo caso il record embedded non ha un RID, dal momento che non può essere referenziabile da nessun altro record, ed è accessibile solo dal record contenitore. Se si elimina il record esterno automaticamente vengono eliminati ricorsivamente anche tutti i record embedded.

Chapter 3

L'implementazione del filesystem virtuale

3.1 Una vista d'insieme

3.1.1 La struttura del progetto

Il lavoro che verrà presentato di seguito costituisce non l'intera implementazione del progetto, ma una parte di essa. Per ragioni di complessità, infatti, è stato deciso di dividere il lavoro in due parti distinte e relativamente indipendenti. Realizzare un driver fuse comporterebbe effettivamente la scrittura del driver vero e proprio, l'implementazione della parte relativa al data storage e la progettazione della comunicazione tra i due strati software.

Come è noto, fuse è una libreria *poliglotta*, nel senso che esistono vari linguaggi che permettono di implementare un driver di questo tipo (C, C++, Java, Python sono tra i più noti), ma, storicamente, quello più usato, e quindi testato, è stato il C; gli altri linguaggi, oltre a non avere una documentazione comparabile a quella disponibile per C, non sono ufficiali, nel senso che sono implementazioni *custom*, con i difetti e le instabilità che questo comporta.

In base a quanto sopra, la scelta del linguaggio con cui realizzare il driver fuse è stata il C. Dall'altra parte, però, si sono presentati problemi opposti: OrientDB è un database *pure Java* e mette a disposizione API native per questo linguaggio. Anche in questo caso, la comunità ha sviluppato dei driver che permettono l'uso di OrientDB anche in altri linguaggi, quali C#, C++, C,

PHP.ettivamente Il problema che si è presentato è stato sempre lo stesso: questi driver non permettono di sfruttare in pieno le potenzialità di orientDB, ed inoltre sono troppo recenti per essere considerati una buona base su cui costruire un lavoro importante come questo.

La scelta finale, per entrambi gli *end-point* del lavoro, è stata quella di prediligere i linguaggi nativi. Questo ha comportato la necessità di realizzare un'infrastruttura di comunicazione tra i due linguaggi, e le possibilità analizzate sono state:

- *JNI* La prima idea è stata quella di usare JNI, acronimo di Java Native Interface. Come viene definita nella documentazione Oracle, e rimarcato anche in quella IBM, la JNI è un'interfaccia di programmazione utilizzata per scrivere metodi nativi ed integrare la Java virtual machine in applicazioni C/C++. Di conseguenza, quello che JNI rende possibile, spiegando è possibile invocare funzioni scritte in C e C++ da Java, e viceversa. La maggiore difficoltà che avrebbe comportato questa scelta sarebbe stata la compatibilità cone le varie JVM. L'interfaccia JNI, infatti, andando a lavorare a basso livello, non è uno standard, ma è legata alla singola implementazione, e l'uso di diverse JVM avrebbe comportato la potenziale non compatibilità, rendendo vano l'impegno. Oltre a ciò, la documentazione ufficiale è aggiornata alla versione 1.2 di Java.

- *Socket* In un secondo momento è stato ipotizzato l'uso delle socket e del framework *zeromq*, così da semplificare lo svilupppo; il motivo per cui questa idea è stata abbandonata consiste nelle difficoltà intrinseche dell'uso delle socket per applicazioni così varie: si sarebbe reso necessario creare un protocollo di comunicazione client-server, e fare ciò in maniera efficiente avrebbbe richiesto un imponente sforzo, oltre che la fase di debug dell'applicazione si sarebbe complicata e non sarebbe stato possibile ampliare il lavoro in futuro senza grossi sforzi.

- *Web service Rest* Infine si è arrivati alla scelta di creare un web service Rest per la comunicazione client-server. Ciò permette di non costringere i futuri manutentori del software a modifiche sostanziali, ma a limitarsi allo strato necessario. Infatti utilizzando questa tecnologia è possibile riconoscere nell'architettura complessiva del progetto tre *layer* distinti: il client-driver, lo strato comunicazione e lo strato storage. Inoltre è possibile, con un

minimo sforzo, ampliare i servizi offerti da fuse[1].

Quello che viene trattato in questo lavoro è la parte relativa al data storage, ovvero al terzo ed ultimo strato software. Nei paragrafi che seguono verranno analizati nel dettaglio i componenti ed il funzionamento di questo layer, a partire dal linguaggio utilizzato per interrogare il database.

3.1.2 Le API Blueprints

L'autore del testo ha deciso di iniziare l'analisi del data-storage dalla tecnologia utilizzata concretamente per interfacciarsi con il database. Come abbiamo visto nel capitolo precedente, OrientDB può essere utilizzato in due modalità: a grafo o documentale; la struttura che più si presta all'implementazione di un filesystem virtuale, naturalmente, è la prima.

Una differenza sostanziale tra un database relazionale ed un database non relazionale consiste nel modo in cui questo può essere interrogato: nel primo caso abbiamo a disposizione l'SQL, ovvero un linguaggio dichiarativo multidatabase, utilizzabile sia da riga di comando, sia da applicazione (tramite l'uso di specifici driver). Nel secondo caso, invece, non è possibile trovare un linguaggio multidatabase standardizzato, ma ciascun produttore offre delle proprie API, e queste sono strettamente legate al prodotto in questione. Nel caso di OrientDB, come abbiamo accennato, le API sono disponibili in Java; sono inoltre disponibili porting non ufficiali anche per altri linguaggi.

Lavorare a livello database è però inefficiente dal punto di vista del *throuhput*, in quanto è necessario gestire molti fattori, come la cache, l'uso di transazioni o le indicizzazioni.

Quali sono allora i vantaggi di usare OrientDB? Il vantaggio consiste dal poter usufruire della combinazione di OrientDB e dello stack Tinkerpop. Quest'ultimo, come viene definito dal sito del produttore, è uno stack open-source, Java oriented, per il panorama emergente dei grafi; esso fornisce le basi per costruire applicazioni ad alte performance di qualsiasi dimensione che operino su grafi, e sono in grado di scalare da una semplice modellazione dei dati, fino a gestire grafi da un trilione di nodi distribuiti su un cluster di computer. In particolare Tinkerpop è costituito da:

- *Blueprints* offre un'interfaccia verso il modello a grafi per i database con delle implementazioni già realizzate.

[1] Si pensi, ad esempio, alla semplicità con cui sarebbe possibile creare un'interfaccia web per la gestione del filesystem in un ambiente su cui non sia disponibile fuse.

- *Pipes* è un framework che permette il controllo del flusso dei dati, dall'input all'output.
- *Gremlin* è un linguaggio specifico per effettuare il *traversing* dei grafi.
- *Frames* permette di lavorare con gli oggetti Blueprints come fossero degli ogggetti Java, nascondendo il fatto che in realtà si sta operando con nodi e vertici.
- *Furnace* fornisce degli algoritmi specifici per lavorare con i grafi.
- *Rexster* è un server multi protocollo, con particolare attenzione a Rest, che espone i database che implementano Blueprints.

La scelta di Blueprints, e, quindi, di Tinkerpop, è stata effettuata principalmente basandosi sul criterio fondamentale della *modularità* dell'applicazione. L'implementazione dell'interfaccia Blueprints da parte di OrientDB rimane totalmente isolata al programmatore, che potrà quindi concentrarsi più sul proprio codice e sfruttare al meglio le potenzialità offerte dallo storage sottostante, con la sicurezza che eventuali bug o problemi di performance verranno risolti in maniera trasparente.

Un esempio che aiuterà a comprendere come l'uso di Blueprints aiuti il programmatore a pensare al cosa piuttosto che al come deriva dalla nuova[2] implementazione fornita da OrientDB; quando si desidera creare un nuovo *edge* tra due vertici senza che però esso stesso debba memorizzare informazioni, utilizzando le *raw* API si hanno due possibilità: creare un vero edge, comportando la creazione di un ODocument[3] vuoto, che comunque occupa spazio in memoria e richiede, nella fase di traversing, di caricare il record in questione, o memorizzare come campo del vertice *parent* un collegamento diretto ad esso. Naturalmente la prima soluzione appare più semplice da realizzare e manutenere nel tempo, seppur si riveli la più inefficiente a causa dell'inutile caricamento da disco di un record non necessario, mentre la seconda, seppur ottimale, risulta essere la più complessa da implementare, per via della macchinosità della soluzione trovata. OrientDB nell'implementazione dell'interfaccia Blueprints risolve il problema introducendo il concetto di *lightweight edge*, ovvero archi leggeri, senza campi di informazioni proprie; se la creazione richiesta è per un lightway edge allora automaticamente viene usata la seconda strategia, mentre se l'arco ha informazioni

[2] Ci si riferisce alla release del 7/6/2013
[3] L'elemento utilizzato da orientDB per memorizzare i dati

viene creato un documento. Tutto ciò viene fatto in automatico dal sistema, senza richiedere l'intervento del programmatore, che così si concentra sul cosa (creazione dell'arco), e non sul come (adottare la prima o la seconda strategia).

Oltre a ciò non va trascurato il fatto che Blueprints, così come tutto lo stack Tinkerpop, è multipiattaforma: lo stesso codice utilizzato per costruire e modellare il virtual filesystem su base OrientDB, con piccole modifiche, può essere utilizzato su altri database che implementano questa interfaccia (come ad esempio Neo4j o MongoDB[4]).

La flessibilità della scelta si dimostra anche nel momento in cui si dovesse rendere necessario scendere ad un maggior livello di dettaglio nell'uso del database: Blueprints offre la possibilità di ottenere il database a basso livello, come nel caso si necessiti di *tuning* del sistema (gestione della cache, accesso ad eventuali indici) o, più semplicemente, nel caso in cui si debbano usare delle funzionalità che Blueprints non offre[5].

3.2 Il filesystem virtuale

In questa sezione sarà analizzata l'implementazione del filesystem virtuale, analizzando le scelte e le strategie adottate, mentre i test saranno discussi nel capitolo che segue. Tutto il codice qui discusso è disponibile alla pagina internet http://github.com/litiales/fuse-driver-for-orientDB/tree/master/OVirtualFileSystem Gli argomento trattati saranno discussi seguento questo ordine:

- *Creazione ed apertura del filesystem*
- *La rappresentazione dei dati*
- *Gestione dei permessi*
- *Data browsing*
- *Funzioni sul filesystem*

[4]Anche se MongoDB non è un database a grafi gli sviluppatori hanno comunque dato un'implementazione dell'interfaccia Blueprints tale da simularne un comportamento

[5]Ad esempio Blueprints non prevede la presenza di una root. Questo comportamento deriva dal fatto che esistono due distinte correnti riguardo la gestione dei database a grafo: i produttori che considerano la root un elemento essenziale del grafo, e quelli che eliminano questo concetto. Essendo Blueprints un framework trasversale, necessita di essere quanto più possibile generico, e quindi non può offrire funzionalità che poi non tutti i produttori sono in grado di offrire. Attualmente OrientDB utilizza un approccio libero, ovvero se si vuole è possibile creare una o più root ed assegnare loro un nome, così da facilitarne il recupero.

3.2.1 Creazione ed apertura del filesystem

Fondamentale per poter operare con ogni filesystem è poterne eseguire un *mount*, operazione che, come si può apprendere eseguendo il comando *man mount*, consiste nel rendere disponibile in una directory specificata dall'utente i file contenuti in esso. Questo significa che, al momento in cui un utente richiederà di accedere al suo filesystem il sistema deve essere in grado di convertire i file di storage che abbiamo visto al Capitolo 2 in un qualcosa che abbia senso e che sia concretamente disponibile.

Per soddisfare questa richiesta, è presente la classe *OVFSManager*: la sua implementazione è piuttosto semplice, in quanto è costituita da funzioni di gestione primitiva del database, quali l'apertura e/o la creazione, la chiusura e l'eliminazione. Accanto ad esse troviamo degli atributi che contengono tutto il necessario per operare poi concretamente con il filesystem.

Il costruttore della classe è un accessibile solo attraverso metodi statici in quanto orientDB prevede la possibilità di specificare un path specifico per il salvataggio del database, oltre che la possibilità di specificare un utente ed una password specifici, ed a seconda delle necessità dell'utente verrà richiamata la funzione adatta. In effetti tutte e quattro le funzioni fanno poi una chiamata allo stesso costruttore, passando parametri standard o personalizzati, restituendo una istanza della classe. Va sottolineato come il sistema sottostante debba mantenere in memoria una sola istanza della classe per ciascun database aperto, in quanto l'uso embedded di orientDB non prevede la sincronizzazione tra processi concorrenti, la richiesta di riapertura di uno stesso filesystem comporterebbe degli errori di consistenza. Questo è il caso in cui due o più utenti richiedano il mount per uno stesso filesystem: il sistema dovrà occuparsi di gestire le richieste accedende sempre e solo alla stessa istanza di OVFSManager, mentre se la richiesta è per filesystem diversi non ci sono problemi, sarà possibile gestire più istanze simultaneamente.

Un'ulteriore caratteristica di questa classe è la capacità di gestire una prima apertura del database come se già esistesse, ovvero il sistema non ha il compito di controllare l'esistenza di uno specifico filesystem: se il filesystem esiste verrà aperto, altrimenti verrà creato usando i parametri specificati e poi aperto.

3.2.2 La rappresentazione dei dati

Sicuramente la parte più significativa da un punto di vista implementativo è il modello della rappresentazione dei dati: scegliere una rappresentazione cor-

retta significa semplicità nella scrittura delle funzioni e minor tasso di errori, scegliere una rappresentazione complessa significa rendere complessa, se non impossibile, l'aggiunta di funzionalità. Per definire il modello dei dati adottato in questa implementazione è stata seguita la rappresentazione ad alto livello di un qualunque filesystem. Questo significa che, mentre normalmente vengono usati gli *inode* a cui sono collegati i dati veri e propri, questo filesystem lavora *nativamente* con una struttura a grafo.

Capire il perchè e quali vantaggi/svantaggi questo modelli comporti si pensi che quando si richiede una risorsa ad un filesystem si chiede in realtà all'inode corrispondente di caricare i dati effettivi: abbiamo così un doppio livello di memorizzazione, il livello degli inode ed il livello fisico dei bit memorizzati, proprio come in una qualsiasi implementazione di filesystem in un database relazionale in cui non è possibile avere dati strutturati. Il problema in orientDB non sussiste, essendo possibile definire un grafo. Effettivamente questa scelta potrebbe comportare problemi di performance, in quanto per poter attraversare un nodo sarebbe necessario caricare l'intero record, magari contenente centinaia di byte; l'implementazione di orientDB, invece, non comporta questo genere di problemi, in quanto implementa un meccanismo di lazy loading, ovvero un record viene effettivamente caricato solo quando necessario, e non al suo accesso. Abbiamo quindi un meccanismo analogo a quello degli inode e dei blocchi di dati, senza però che si implementino effettivamente due strutture separate.

La rappresentazione scelta è stata dunque quella di far equivalere una qualsiasi risorsa, sia essa un link, una directory od un file, ad un vertice del grafo. Per permettere poi la consistenza dei dati, viene creata, al momento dell'inizializzazione del database, un nodo *root* che non può essere eliminato, e che consente l'accesso al filesystem da un unico punto, semplificando la navigazione. Oltre ciò, sfruttando la possibilità di dare a ciascun nodo una label, è possibile conoscere in tempo O(1) il tipo di risorsa che si sta trattando, sia essa una Directory, un File od un Link.

Definite le risorse, è possibile stabilire una gerarchia su di esse, ovvero dei collegamenti che definiscono come il filesystem è articolato. Per ottenere ciò è stato utilizzata la creazione di archi leggeri, che non occupano spazio in memoria, e che richiedono un tempo di attraversamento costante. La strategia adottata è la seguente: si supponga di avere una directory *Dir*, e di voler creare in essa il file *file*; una volta creati i due nodi si crea un arco leggero uscente da *Dir* ed entrante in *file*, con label il nome del file. Analogamente si supponga di avere una risorsa *Resource* e di volerla linkare al link *link*; una volta create le due risorse

è sufficiente creare un arco uscente da *linkedRes* ed entrante in *resource* con label *link*. Con questa rappresentazione, dato un path, sarà possibile navigare il filesystem in tempo costante dal momento che non è possibile avere due risorse in una stessa directory con lo stesso nome e quindi l'attraversamento di ciascun arco avrà costo O(1).

3.2.3 La gestione dei permessi

Tutti i filesystem più recenti dispongono della possibilità di gestire i permessi di una risorsa, ovvero cosa un utente può o non può fare. Nell'implementazione classica di Unix a ciascun file o cartella è associato un permesso, costituito da un numero ottale a tre cifre: da sinistra verso destra le cifre codificano rispettivamente i permessi sulla risorsa assegnati all'utente proprietario, agli utenti appartenenti allo stesso gruppo del proprietario ed a tutti gli altri utenti. Essendo la rappresentazione numerica di tipo ottale, ciascuna cifra viene a sua volta codificata da tre bit, che, da sinistra verso destra, se impostati ad uno, esprimono la possibilità di scrittura, lettura o esecuzione.

Una tale gestione delle risorse è stata ritenuta fondamentale per permettere la scalabilità del filesystem: nel momento in cui si debba condividere le risorse tra più utenti, si rende necessario stabilire una policy su cosa e come si possa fare. Supponiamo che Alice voglia condividere con gli utenti un file importante, ma che voglia esser certa che tale file non venga modificato; se non fosse possibile stabilire delle limitazioni, Bob potrebbe prendere il file ed aggiungere o eliminare dati importanti. Così facendo, invece, basta che Alice imposti come permessi su file, ad esempio, 0744, così chiunque può leggere il contenuto del file, ma nessuno, se non Alice stessa, può modificarlo.

Purtroppo però la gestione dei permessi in un filesystem non condiviso è lasciata al fatto che, almeno teoricamente, i permessi da superutente possa averli solo l'amministratore del sistema, che può fare tutto con i file, senza limitazione alcuna. Questo meccanismo viene meno nel caso in cui si decida di condividere lo stesso filesystem tra più utenti senza che esista una password da amministratore: potenzialmente si lascia a chiunque la possibilità di agire da superutente, proprio come se tutti fossero presenti nel file *sudoers*[6]. Questa scelta implementativa è stata fatta in quanto si suppone che gli utenti che accedono al filesystem siano utenti fidati, e che non abbino quindi intenzioni malevole[7].

[6]Il file in Unix contiene la lista degli utenti che possono usare la loro password per accedere temporaneamente come amministratori attraverso il comando *sudo*

[7]Come viene riportato al primo comando sudo in ArchLinux *"Da grandi poteri derivano*

Nell'implementazione del filesystem virtuale i permessi sono rappresentati come una stringa di quattro caratteri che codifica il numero ottale (il primo carattere è uno 0), ed è salvata nel campo *mode* dei nodi (nel caso dei Link, così come in Unix, i permessi su di essi sono gli stessi di quelli delle risorse linkate). La gestione dei permessi è effettuata dalla classe ModeManager, una classe con un unico costruttore privato (così da rendere impossibile crearne un'istanza) in cui tutti i metodi sono statici; così facendo si garantisce che la stessa classe viene condivisa tra più istanze di filesystem, risparmiando spazio in memoria.

Sarà ora presentato lo pseudocodice delle funzioni, ma essendo il codice autoesplicativo ci si limiterà a questo.

Algorithm 1 Ritorna la cifra corretta su cui controllare i permessi in base al proprio nome utente e gruppo

```
function getPermission(resource, user, group)
    uOwn ← resource.get(owner)
    uGroup ← resource.get(group)
    mode ← resource.get(mode)
    if uOwn = user and uGroup = group then
        return mode.substr(1, 2)
    else if uOwn <> user and uGroup = group then
        return mode.substr(2, 3)
    else
        return mode.substr(3, 4)
    end if
end function
```

Algorithm 2 Ritorna true se si hanno i permessi per scrivere

```
function canWrite(resource, user, group)
    if user = root and group = root then
        return true
    end if
    perm ← getPermission(resource, user, group)
    if perm = 2 or perm = 3 or perm = 6 or perm = 7 then
        return true
    end if
    return false
end function
```

granbdi responsabilità

Algorithm 3 Ritorna true se si hanno i permessi per leggere

```
function canRead(resource, user, group)
    if user = root and group = root then
        return true
    end if
    perm ← getPermission(resource, user, group)
    if perm > 3 then
        return true
    end if
    return false
end function
```

Algorithm 4 Ritorna true se si hanno i permessi per l'esecuzione

```
function canExecute(resource, user, group)
    if user = root and group = root then
        return true
    end if
    perm ← getPermission(resource, user, group)
    if perm%2 = 1 then
        return true
    end if
    return false
end function
```

3.2.4 Data browsing

Come già sottolineato, montare un filesystem significa costruire un albero, ovvero stabilire una gerarchia tra risorse, ed operare su un filesystem si riduce a due operazione fondamentali: navigare il filesystem fino alla risorsa desiderata per poi operare effettivamente.

Risulta quindi necessario avere un meccanismo sicuro ed efficiente che permette di effettuare il *browsing* del tree, altrimenti qualsiasi implementazione ottima al momento dell'operazione viene vanificata. La struttura che è stata data filesystem è già stata analizzata due paragrafi fa e non verrà ritrattata in questo luogo. Quello che sarà invece approfondito è il come questa struttura viene sfruttata dalla classe ODatabaseBrowser.

Questa classe ha un attributo in cui è memorizzata la root del filesystem, e tre metodi, di cui uno privato e due visibili solo all'interno del package, oltre ad un quarto metodo pubblico che elenca tutte le risorse del filesystem, utile in fase di testing.

La funzioni privata *getParentResource* viene richiamata dalle altre due funzioni, ed è quella che effettua il browsing, restituendo la risorsa che precede quella specificata nel path; ad esempio, si supponga di richiedere una certa operazione da effettuare sulla risorsa */home/user/resource*, verrà restituita la risorsa padre, ovvero */home/user/*. Man mano che dalla root scende verso le foglie questa funzione verifica di avere i permessi per poter continuare, permessi che nella comune implementazione Unix sono d'esecuzione, oltre che la risorsa corrente esista e sia effettivamente una Directory. Come parametri d'ingresso richiede un array dove è presente la lista ordinata delle risorse da attraversare per restituire il padre, una variabile dove inserire un eventuale codice di errore, e le credenziali dell'utente.

Si passa ora alle funzione che vengono usate all'interno del codice, e che fanno riferimento alla funzione precedente. La prima è la *getResourcePath*, che viene invocata nel caso si voglia creare una nuova risorsa: in questo caso si prende la risorsa che viene restituita da *getResource*, e si controlla che essa non sia nulla; se poi il nodo non contiene altri vertici uscenti con il nome della risorsa che si vuole creare allora viene ritornato, altrimeni la funziona ha un return nullo. Di seguito è riportato lo pseudocodice:

Per finire è presente la funzione utilizzata dalla maggior parte delle funzioni, la *getResource*, che restituisce il vertice specificato nel path, o null se l'elaborazione ha incontrato problemi. Come funzione è abbastanza elementare,

semplicemente chiama la *getParentresource*, verifica se l'utente ha i permessi d'esecuzione sulla directory padre, se si tenta di restituire la risorsa richiesta se questa è presente. In caso di errori restituisce null.

Algorithm 5 Ritorna il padre della risorsa richiesta, null se non è stato possibile determinarlo, in tal caso retValue è settato. L'elemento path[0] è la risorsa root.

```
function getParentResource(path, retValue, user, group)
    if path.length = 1 then
        retValue ← EACCESS
        return null
    end if
    index ← 1
    parent ← root
    while index + 1 < path.leght and parent <> null do
        if canExecute(parent, user, group) then
            retValue ← EACCESS
            return null
        end if
        parent ← parent.follow(path[index])
        if parent = null then
            retValue ← ENOENT
        else if parent.isNotDirectory then
            retValue ← ENOENT
            parent ← null
        end if
        index + +
    end while
end function
```

Algorithm 6 Ritorna il padre della risorsa richiesta, null se non è stato possibile determinarlo, in tal caso retValue è settato.

```
function getResourcePath(path, retValue, user, group)
    canonicalPath ← path.split("/")
    parent ← getParentResource(canonicalPath, retValue, user, group)
    if parent = null then
        return null
    end if
    resourceName ← canonicalPath.lastElement
    if parent.hasOutResource(resourceName) = false then
        return parent
    end if
    retvalue ← EEXIST
    return null
end function
```

Algorithm 7 Ritorna la risorsa richiesta, null se non è stato possibile determinarla, in tal caso retValue è settato.

```
function getResource(path, retValue, user, group)
    canonicalPath ← path.split("/")
    parent ← getParentResource(canonicalPath, retValue, user, group)
    if parent = null then
        return null
    end if
    if canExecute(parent, user, group) = false then
        retValue ← EACCESS
        return null
    end if
    resourceName ← canonicalPath.lastElement
    resource ← parent.follow(resourceName)
    if resource = null then
        retvalue ← ENOENT
        return null
    end if
    return resource
end function
```

3.2.5 Funzioni sul filesystem

Verranno ora analizzate le funzioni che vanno ad operare materialmente sul database, seguite dalla loro implementazione in pseudocodice.

getattr