Contents

Chapter 1

Introduction

The world of the information technology in the 21^{th} century, mainly due to the success of the internet, in particular of the world wide web, has seen the raise of many problems that until now have been ignored, first of all the handle of big data sets.

The classic way to manage data is using an SQL solution, like MySQL, or PostgreSQL, but the truth is that this is not, and it will never can be, *the* solution. Its force, but also its weakness, is the strict theory on which relational databases are based on. I say that this is a pro because a good phase of design can produce a well-formed database, data-consistent and error less. On the other hand this is con, because this theory is not so recent: relational model has been introduced in the 70s, so it's half century old.

This work is not only about databases: it is part of a bigger project whose hit is building a FUSE driver for OrientDB, a multi-model non-relational database. This in not the whole project, and what will be seen here is how the library to manage informations stored in OrientDB as if they were real files directories or links has been written.

Chapter 2

As we will see in this chapter, RDBMS is not the best solution for big complex data storing, mainly due to their expansive join operations and to the lien of using non-structured attributes, while it is still a good solution in non efficiency-critic scenarios.

As each type of task requires a different language (I would never write an

operative system in Java or Python, as I would never write a simple script in C or C++), each complex data persisting problem could be well solved using an appropriate solution. This approach is known as *polyglot persistence*.

After this short *excursus* on the noSQL world, will be presented the used database, OrientDB, introducing its main characteristics, such as the possibility of use it as a graph, object or documental database.

Chapter 3

Because this is not an isolated work, the purpose of the chapter 3 is to present FUSE to the reader. It is a default kernel module since version 2.6.14 and allows users to create filesystems in their userspace without recompiling kernel, so without administrative privileges. Moreover, FUSE allows to work with its filesystems transparently, using standard commands and programs.

Here will be introduced also the virtual file system concept, that is the kernel layer that allows to not distinguish the different filesystems, and how FUSE interact with it.

Chapter 4

This is the most important part, and it's here that the library architecture will be examined in its main parts: how filesystem concepts have been translated into a non-filesystem resource, how data browsing is actuated, how permissions hare handled and how functions are implemented.

A relevant importance in this chapter is given to methods implementation, and, after a short description of what they do is presented their pseudocode.

Chapter 5

In this last chapter the focus is on performances of adopted solutions and on the analysis of how OrientDB can help the developer, and in the end will be presented new possible improvements.

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 (di Google), e di Dynamo (di Amazon), e che avevano ispirato il lavoro di altri informatici (portando alla nascita di progetti 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 congresso sulle novità che il mondo dei database stava vivendo.

[1] Framework, rilasciato con licenza 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 lo sono, ed offrono assistenza a pagamento o versioni definite *Enterprise*, ovvero 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:

- *Document store.* Il concetto centrale è il documento, che incapsula i dati

[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.

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 ne permette il recupero attraverso una chiave, come accade 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].

2.1.3 NoSQL ed il teorema CAP

Quando si deve risolvere un problema di persistenza dei dati, come previsto dal *teorema CAP*, bisogna sempre sapere, prima di fare una scelta, cosa si vuole ottenere.

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)

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.

[3]Resource Description Framework, standard definito dal W3C

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, per la maggior parte, 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. OrientDB, ad esempio, permette di scegliere quale approccio utilizzare, introducendo il concetto di *transazione atomica*.

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 ACID, o di non usarle, effettuando i salvataggi ad ogni modifica, rendendo l'ambiente BASIC.
- *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.

Oltre a ciò non va tralasciata l'elevato *throughput* 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, 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'apertura 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 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

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

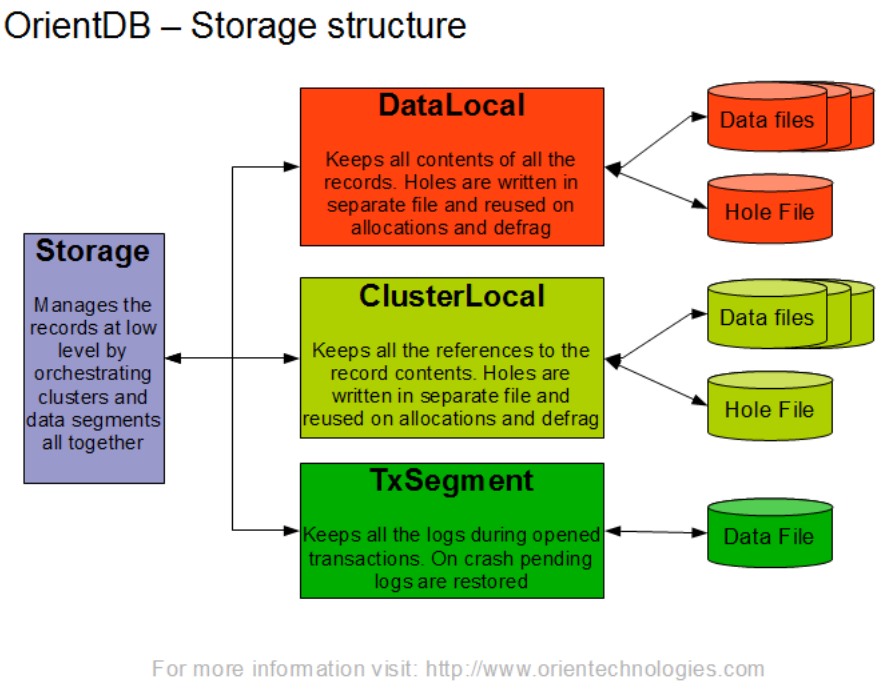

Figure 2.1: Schematizzazione dell'architettura del data storage di OrientDB

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, come si può vedere in figura 2.2, quando ad OrientDB riceve una richiesta per un record si attiva il seguente flusso: 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.

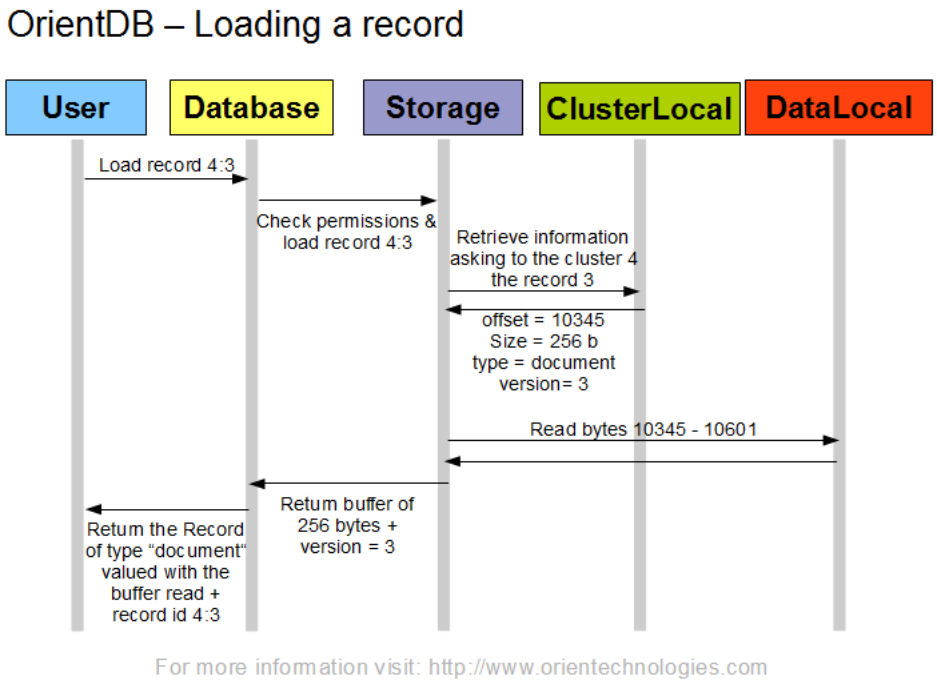

Figure 2.2: Flusso generato da OrientDB per rispondere alla richiesta di un record

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 nuova classe equivale a creare un nuovo 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 classi astratte ed evitare così 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 è il fatto che non si ha più il vincolo delle

forme normali, e quindi neanche dell'atomicità dei valori. Questo permette allo sviluppatore di implementare soluzioni performanti, come vedermo più avanti, e 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 di quello 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, ed è accessibile solo dal record contenitore. Se si elimina il record esterno vengono eliminati ricorsivamente anche tutti i record embedded.

Chapter 3

Filesystem in userspace

3.1 Cos'è FUSE

Sebbene questo lavoro riguardi l'implementazione di un filesystem virtuale sulla base di OrientDB, poichè l'obbiettivo finale è quello di avere un filesystem effettivamente funzionante e accessibile usando le classiche API Unix, è bene dedicare qualche pagina a FUSE.

Come riportato sia in *en.wikipedia.org/wiki/Filesystem_in_userspace* che nella documentazione ufficiale reperibile alla pagina *fuse.sourceforge.net* FUSE è un modulo presente di default nel kernel di sistemi Unix-like[1] a partire dalla versione 2.6.14. Rilasciato sotto licenza GPL, come già il suo nome, Filesystem in USErspace, lascia intuire, il suo scopo è quello di permettere anche ad utenti non privileggiati di implementare dei filesystem nel proprio *spazio utente*, senza necessità di ricompilare il kernel. Oltre al modulo, viene fornita ai programmatori la libreria *libfuse* (rilasciata sotto liscenza LGPL), che permette il *binding* tra il modulo ed il filesystem implementato.

Il progetto FUSE nasce come *fork* di AVFS, acronimo di A Virtual FileSystem, un sistema che permette a tutti i programmi di accedere ad archivi, file compressi o remoti senza necessità di essere ricompilati o di modificare il kernel. Entrambi hanno lo stesso obbiettivo: rendere disponibile all'utente delle risorse che, attraverso altre vie non sarebbero disponibile; quello che li differenzia è però la volontà da parte di FUSE di essere il più possibile generico: qualsiasi supporto può diventare un filesystem.

[1] Attualmente Fuse è disponibile per Linux, FreeBSD, OpenSolaris, Minix 3, Android e OS X

3.1.1 Il concetto del virtual filesystem

FUSE si basa sul concetto di *virtual filesystem*, conosciuto anche con l'acronimo VFS; il suo scopo è quello di permettere ad un utente di accedere a diversi tipi di filesystem in maniera uniforme (ad esempio un utente può accedere allo stesso modo ad un filesystem locale o remoto senza rendersi conto delle differenze).

Fondamentalmente un VFS offre un un'interfaccia da utilizzare per comunicare tra il kernel ed il filesystem concreto; un primo esempio è stato introdotto nel 1985 dalla Sun Microsystems e permetteva di comunicare in maniera trasparente all'utente sia con un filesystem locale (UFS) che con uno remoto (NFS).

Durante il boot del sistema, quando i filesystem vengono inizializzati, ciascuno di essi si registra presso il VFS; essi possono essere presenti direttamente nel kernel o caricati successivamente come moduli (un esempio è costituito dai filesystem VFAT) solo quando necessario.

Seguendo questa politica, il VFS sa sempre redirigere le richieste ricevute al filesystem di competenza, per poi restituire le risposte in maniera appropriata. Inoltre, proprio perchè nasconde la vera implementazione, non si ha una differenza tra i vari filesystem, lo stesso gestore desktop sarà in grado di visualizzare tutto in maniera omogenea.

3.2 Come funziona FUSE

Per scrivere un filesystem FUSE è necessario, come abbiamo visto, far uso della libreria libfuse; esistono vari linguaggi di binding, e quindi varie versioni di questa libreria da utilizzare, ma, storicamente, viene utilizzato il C, in quanto è l'unico per il quale è fornito supporto ufficiale. Tutte le altre implementazioni di libfuse sono implementazioni non native, e, ad oggi, soffrono di mancanze e bug rilevanti.

Quello che si ottiene al termine del lavoro è un eseguibile che, al momento necessario, va semplicemente eseguito specificando il path in cui eseguire il *mount* del filesystem, ossia il punto in cui visualizzare il suo contenuto. Va sottolineato che, qualunque sia il path specificato, esso deve essere una directory, non fa differenza se vuota o meno: dal momento del mounting tutto il suo contenuto non sarà più accessibile per lasciare spazio alle nuove risorse, mentre tornerà disponibile una volta eseguito l'*umount*.

Come abbiamo visto, FUSE permette di scrivere filesystem non convenzionali; a differenza di questi, non si occupa di scrittura e lettura su disco, ma si

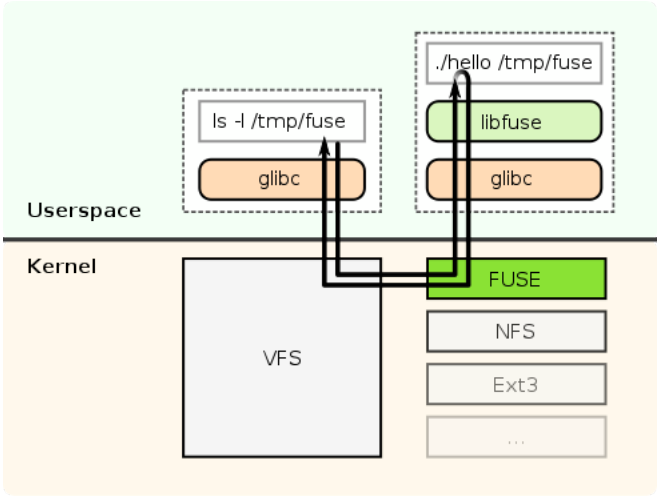

Figure 3.1: Il flusso generato da un filesystem FUSE

comporta da traduttore di informazioni già presente su disco o strumenti di storage. Esempi concreti di come questo venga sfruttato saranno presentati nei prossimi paragrafi.

Il meccanismo con cui FUSE agisce è rappresentato nella figura 3.1. Con riferimento ad essa, abbiamo rappresentato un filesystem denominato *hello*, il cui contenuto è montato nella directory */tmp/fuse*. Al momento in cui un utente richieda di eseguire il comando *ls -l /temp/fuse*, viene eseguito il codice ad esso corrispondente, che contiene chiamate alla libreria *glibc*, la quale compie delle chiamate all'interfaccia kernel VFS. Il sistema, al momento del mount del filesystem (ossia al momento dell'esecuzione di *hello*), ha registrato presso il VFS il modulo FUSE, collegando ad esso la directory in cui è stato montato; sa quindi a quale modulo inoltrare la richiesta, modulo che si occuperà di contattare nuovamente glibc per arrivare a libfuse. Da qua la libreria si occuperà di interagire con il filesystem hello: se chi lo ha realizzato ha implementato la funzione richiesta, essa gli sarà rigirata e la risposta percorrerà il percorso inverso a quello iniziale, altrimenti verrà comunicato all'utente che la funzione in questione non è disponibile.

Non tutte le funzioni, infatti, sono richieste per avere un filesystem funzionante: alcune di essere sono dedicate a dispositivi speciali, altre, invece, fanno parte di un set che offre funzionalità aggiuntive. Nel nostro progetto è stato deciso di implementare un set che permetta di eseguire tutti i comandi essenziali

in Unix (ma non è questo il lavoro che sè ne occupa).

Da aggiungere, infine, che, a partire dalla versione 2.4 di FUSE, è possibile aggiungere un qualsiasi filesystem realizzato al file *fstab* per permettere il mount automatico già in fase di avvio del sistema operativo.

3.3 Perchè FUSE?

Come abbiamo visto, sembra evidente che FUSE aggiunga un sufficiente *overhead* rispetto all'uso di un filesystem come potrebbe essere Ext4 o Ext3, quindi, perchè scegliere FUSE?

La verità è che *il* perchè usare FUSE non esiste. Ciascuna implementazione di un filesystem ha delle proprie pecularità, e quindi ciascun utente potrebbe avere delle necessità per cui usare questo sistema.

Un esempio è costituito da WebDrive, un filesystem con cui è possibile utilizzare in maniera trasparente il servizio Amazon S3, o GmailFS, con cui è possibile memorizzare dati utilizzando lo spazio messo a disposizione da Gmail. Non tutti possono essere interessati a queste possibilità (si pensi a chi non ha un account per tali servizi), ma esiste sicuramente una, seppur piccola, fetta di utenti che ne tra giovamento.

Il lavoro che abbiamo deciso di realizzare offre un servizio complesso e, si spera, utile. Utilizzando un database non relazionale potente come OrientDB in accoppiata alle API Blueprints, si intende fornire un servizio di memorizzazione che sia effettivamente fruibile a tutto tondo.

Innanzi tutto il progetto prevede la distinzione di un processo client e di uno server: il filesystem è implementato nel primo. Quando questo riceve una richiesta, verifica la possibilità di contattare il server; se questo è disponibile, gli viene chiesto di attuarla, utilizzando le API che verranno discusse nel capitolo successivo. Se tutto va a buon fine, il server notifica al client il risultato dell'operazione, che a sua volta restituirà i dati al processo che ha invocato la chiamata iniziale.

Una tale architettura permette, oltre che la semplice esecuzione del filesystem in un ambiente locale, la condivisione contemporanea degli stessi dati in remoto tra più utenti, mantenendo la politica dei permessi tipica di Unix.

Utilizzato in locale, invece, permette di condividere il filesystem tra più utenti proprio come si trattasse di una pendrive, con il vantaggio aggiuntivo che può essere spedito per mail, ad esempio, o caricato su un servizio di hosting.

Infine un altro motivo che ha fatto nascere questo progetto è stata la volontà di partecipare ad un progetto open source rispondendo alle richieste del gruppo di OrientDB.

Chapter 4

L'implementazione del filesystem virtuale

4.1 Una vista d'insieme

4.1.1 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, libfuse 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.

4.1.2 Le API Blueprints

L'analisi inizia dal 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 *throughput*, 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 modelli 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 sia 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].

4.2 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.

4.2.1 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 che fa uso di questa libreria 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.

4.2.2 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 corretta

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 questo filesystem lavora *nativamente* con una struttura a grafo simile a quella simulata da un comune dispositivo di memorizzazione.

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 creato, 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, semplificandone 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).

4.2.3 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 rispettivamente 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, è sufficiente che Alice imposti come permessi sul 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 abbiano quindi intenzioni malevole[7].

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 esso autoesplicativo ci si limiterà a questo.

[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 grandi responsabilità*

Algorithm 1 La funzione fondamentale per il controllo dei permessi

```
    function getPermission(resource, user, group)
        uOwn ← resource.get(owner)
        uGroup ← resource.get(group)
        mode ← resource.get(mode)
 5:     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
10:         return mode.substr(3, 4)
        end if
    end function
```

Algorithm 2 La funzione che verifica se si hanno permessi in scrittura

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

Algorithm 3 La funzione che verifica se si hanno permessi in lettura

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

Algorithm 4 La funzione che verifica se si hanno permessi in esecuzione

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

4.2.4 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 permetta 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.

Per finire è presente il metodo utilizzato 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 La funzione fondamentale del data browsing

```
function getParentResource(path, retValue, user, group)
    if path.length = 1 then
        retValue.value ← 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.value ← EACCESS
            return null
        end if
        parent ← parent.follow(path[index])
        if parent = null then
            retValue.value ← ENOENT
        else if parent.isNotDirectory then
            retValue.value ← ENOENT
            parent ← null
        end if
        index + +
    end while
end function
```

Algorithm 6 La funzione che ritorna il padre di una risorsa

```
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.value ← EEXIST
    return null
end function
```

Algorithm 7 La funzione che ritorna una risorsa

```
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.value ← EACCESS
        return null
    end if
    resourceName ← canonicalPath.lastElement
    resource ← parent.follow(resourceName)
    if resource = null then
        retValue.value ← ENOENT
        return null
    end if
    return resource
end function
```

4.2.5 Funzioni sul filesystem

Verranno ora analizzate le funzioni, facenti parte della classe *Functions*, che vanno ad operare materialmente sul database, seguite dalla loro implementazione in pseudocodice. Tutte loro operano sul database simulando il comportamento abituale di un filesystem, e sono state ispirate dal set di funzioni implementate da FUSE. Un'ulteriore caratteristica da evidenziare è l'atomicità delle operazioni, che vengono commissionate al database solo alla fine, e mai passo passo, permettendo il *rollback* ad uno stato sicuro in caso di errore.

Nelle funzioni che richiedono un valore di ritorno diverso da quello indicante il risultato dell'operazione (ad esempio nel caso della read che ritorna un array di byte), poichè in Java non è previsto il passaggio di parametri per riferimento, si è reso necessario l'uso di istanze della classe *IntWrapper*, che contengono al loro interno un valore intero che, se settato, indica un errore[8].

Creazione di risorse

La funzione principale di cui un filesystem deve disporre è quella che permette la creazione di nuove risorse, siano esse file o directory (i link verranno trattati più avanti).

Senza di essa un filesystem sarebbe statico, e nessun utente avrebbe interesse nell'usarlo. Per implementare questa funzionalità basilare è stata implementato il metodo *create_resource*, che richiede come parametri il path completo della risorsa da creare, i suoi permessi, le sue credenziali (nome utente e gruppo) ed il tipo di risorsa da creare.

Il metodo non permette di creare una risorsa che si trovi gerarchicamente alla pari della root, per evitare di avere un filesystem con più di una radice, e parimenti non permette ad una risorsa file di avere un nome terminante con uno slash, dal momento che tale terminazione è associata alle directory.

Al suo interno, come abbiamo visto nel capitolo relativo al data browser, fa uso della funzione *getResourcePath*, che restituisce l'elemento che accoglierà la nuova risorsa nel caso in cui l'utente abbia i permessi di esecuzione e scrittura su di esso. Il passo successivo è creare il nuovo vertice, inizializzarlo con i dati specificati dall'utente e collegarlo alla risorsa padre, di cui vengono aggiornati i

[8] Infatti in Java seppur è vero che l'unico passaggio di parametri previsto è quello per valore, non bisogna dimenticare che, nel caso di oggetti, il valore passato non è l'oggetto in sè, quanto piuttosto il riferimento all'oggetto; quindi in un metodo non è possibile, ad esempio scambiare due o più oggetti passati come parametri, perchè significherebbe modificare i loro riferimenti, ma risulta possibile modificare i loro attributi.

tempi di modifica e cambiamento (mtime e ctime).

Poichè la funzione di creazione viene richiamata non solo per una semplice creazione di risorsa, ma anche nell'ambito di un più complesso flusso di esecuzione (come la write su una risorsa non esistente), è stato impostato un flag binario che specifica se, al termine dell'esecuzione, il database deve commissionare la transazione oppure attendere perchè la sua conclusione sarà compito del metodo invocante.

Algorithm 8 La funziona che crea directory e file

```
    function CREATE_RESOURCE(path, mode, user, group, type, commit)
        if path = "/" then
            return EPERM
        end if
 5:     if type.isNotLink and type.isNotDirectory then
            return EINVAL
        end if
        retVal ← newIntWrapper()
        parent ← getResourcePath(path, retValue, user, group)
10:     if parent = null then
            return retValue.value
        end if
        if canExecute(parent, user, group) and canWrite(parent, user, group)
    then
            parent.ctime ← now
15:         parent.mtime ← now
            resName ← path.getResName
            resource ← initializeResource(mode, user, group, resName)
            parent.addOutEdge(resource, resName)
            if commit = true then
20:             database.commit
            end if
        else
            return EPERM
        end if
25:     return 0
    end function
```

Link a risorse

Un'ulteriore risorsa standard dei filesystem Unix è il link, ovvero un collegamento ad una risorsa che può essere di due tipi: logico, e si ha un *soft link*, noto anche come *symbolic link* o fisico, noto come *hard link*. Un link simbolico, come

riportato dall'Università dell'Indiana alla pagina internet *http://kb.iu.edu/data/abbe.html* è un tipo speciale di risorsa che punta ad un'altra risorsa, proprio come gli *shortcut* in Windows, e a differenza degli hard link non contiene dati, ma semplicemente punta un'altra risorsa del filesystem; questo è particolarmente evidente al momento dell'eliminazione della risorsa linkata, infatti in questo caso il link simbolico diventa inusabile, mentre l'hard link continua ad avere i dati memorizzati. Inoltre, poichè non linka dei blocchi di dati, ma un inode, è possibile collegargli non solo file, ma anche directory e risorse su computer remoti (ma questa opportunità non è stata affrontata).

Ciò detto, la realizzazione dei link in questo filesystem è stata compiuta effettundo una scelta, ossia l'implementazione dei soli soft link, dal momento che gli hard link avrebbero comportato l'aumento della complessità del sistema globale senza fornire funzionalità aggiuntive significative.

L'algoritmo di creazione dei link richiede come parametri di ingresso il path del link, quello della risorsa linkata e le credenziali dell'utente, e restituisce il valore corrispondente allo stato dell'operazione. Inizialmente, come per la creazione di un file o di una directory, viene richiesto il vertice sotto cui creare il link, se si hanno su di esso permessi di esecuzione e scrittura allora viene richiesto anche il vertice corrispondente alla risorsa da linkare; viene inizializzata una nuova risorsa con label *Link* ed i parametri stabiliti dall'utente. Infine viene creato un arco tra la risorsa padre (a cui vengono aggiornati i tempi mtime e ctime) ed il link, oltre che tra il link stesso e la risorsa linkata (a cui viene aggiornato il tempo ctime).

Al termine di questa operazione l'albero iniziale si trasforma in un grafo.

Algorithm 9 La funzione che crea un link ad una risorsa

```
    function LINK(linkPath, linkedResource, user, group)
        if linkPath = "/" then
            return EPERM
        end if
 5:     retValue ← newIntWrapper()
        linkParent ← getResourcePath(linkPath, retValue, user, group)
        if linkParent = null then
            return retValue.value
        end if
10:     if canExecute(linkParent, user, group) = false or
    canWrite(linkParent, user, group) = false then
            return EPERM
        end if
        linked ← getResource(linkedResource, retValue, user, group)
        if linked = null then
15:         return retValue.value
        end if
        linkName ← linkPath.getResName
        link ← initializeNewLink(linkParent, linked, linkName, linkedResource)
        linkParent.addOutEdge(link, linkName)
20:     link.getOutEdge(linked, "link")
        linked.ctime ← now
        linkParent.ctime ← now
        linkParent.mtime ← now
        return 0
25: end function
```

Il follow dei link

Come detto sopra, creare un link significa creare una risorsa che punta ad un altra risorsa; si rende quindi indispensabile avere una metodo che, dato un link, sia in grado di dire qual è la risorsa ad esso collegata (può ancora trattarsi di un link, o non avere risorse linkate nel caso esse siano state eliminate).

Per fare ciò è presente il metodo *readlink*, che richiede il path del link e le credenziali dell'utente, e restituisce la stringa contenente il path della risorsa collegata, ed in caso di errore setta un errore. Fondamentalmente la funzione chiede al data browser il vertice corrispondente alla risorsa richiesta, verifica che sia un link, e restituisce il path per del vertice ad esso collegato.

Algorithm 10 La funzione che effettua il follow dei link

```
    function READLINK(path, retValue, user, group)
        link ← getResource(path, retValue, user, group)
        if link = null then
            return null
5:      end if
        if link.isNotLink then
            retValue.value ← EINVAL
            return null
        end ifreturn link.linkedPath
10: end function
```

Le informazioni sulle risorse

La funzione *gettatr* permette di ottenere un set di informazioni su uno specifico nodo. Un tipico esempio di quando questa funzione viene richiamata è quello in cui l'utente richiede da terminale di eseguire il comando *stat* su una risorsa. Le informazioni restituite dalla funzione sono i permessi, lo username ed il gruppo del possessore della risorsa, la sua dimensione, l'ultimo tempo di accesso, di modifica e di cambiamento ed il tipo di risorsa.

Poichè Java non permette la restituzione di valori multipli come la funzione richiederebbe, è stata usata la classe *Stat* avente come attributi privati tutti i campi di interesse che divengono accessibili tramite dei *getter* pubblici (così da garantire l'integrità dei dati).

La procedura eseguita da getattr è abbastanza semplice: richiede al data browser la risorsa specificata nel path, se è presente controlla che non sia un link, in caso negativo segue la risorsa da essa linkata in maniera ricorsiva fino

ad arrivare ad un file o ad una directory. Poichè un link può anche essere interrotto, questa eventualità viene gestita restituendo *null*, poichè non è stato possibile determinare la risorsa su cui eseguire lo stat. Infine viene creato un oggetto Stat con i parametri corretti e viene restituito.

Algorithm 11 La funzione che restituisce i metadati di una risorsa

```
function GETATTR(path, retValue, user, group)
    resource ← getResource(path, retValue, user, group)
    if resource = null then
        return null
    end if
    if resource.isLink then
        resource ← resource.followLinks
    end if
    if resource.isLink then
        retValue.value ← ENOENT
        return null
    end if
    return new Stat(resource.mode, resource.user, resource.group, resource.size, resource.atime, resource.ctime, resource.mtime, resource.type)
end function
```

La rimozione di risorse

Operazione importante tanto quanto la creazione di nuove risorse è la loro rimozione, senza la quale un filesystem verrebbe rapidamente saturato, e non potrebbe mai essere ordinato. In Unix sono previsti due tipi di rimozione di risorse, accessibili tramite system call diverse, la *unlink*, che permette la rimozione di file e link, e la *rmdir*, che permette la rimozione di directory vuote.

La rimozione di un file o di un link prevede che l'utente abbia i permessi di scrittura sulla risorsa in questione, e quelli in scrittura ed esecuzione sulla cartella al livello superiore; il metodo inizia richiedendo il vertice corrispondente all risorsa richiesta, da questa, se non è una directory, viene presa la risorsa padre, e se ne verificano i permessi. Se la risorsa da eliminare è un file vengono eliminati tutti i dati a lui relativi, infine si procede alla rimozione di tutti gli archi uscenti o entranti e, quindi, del vertice stesso. Vengono poi aggiornati i tempi di cambiamento e modifica della risorsa padre.

Analogo meccanismo è seguito dal metodo che gestisce la rimozione delle cartelle, con l'unica differenza che il controllo deve confermare che si sta tentando di eliminare una directory e che essa non sia vuota.

Algorithm 12 La funzione di rimozione di file e link

```
    function removeResource(path, user, group)
        retValue ← newIntWrapper()
        resource ← getResource(path, retValue, user, group)
        if resource = null then
5:          return retvalue.value
        end if
        if resource.isDirectory then
            return EISDIR
        end if
10:     parent ← resource.getParent
        if canWrite(parent, user, group) = false or
    canWrite(resource, user, group) = false then
            return EPERM
        end if
        if resource.isFile then
15:         resource.removeData
        end if
        resource.removeEdge
        resource.delete
        parent.ctime ← now
20:     parent.mtime ← now
        return 0
    end function
```

Algorithm 13 la funzione di rimozione delle directory

```
    function REMOVEDIRECTORY(path, user, group)
        retValue ← newIntWrapper()
        resource ← getResource(path, retValue, user, group)
        if resource = null then
 5:         return retvalue.value
        end if
        if resource.isNotDirectory then
            return ENOTDIR
        end if
10:     parent ← resource.getParent
        if canWrite(parent, user, group) = false or
    canWrite(resource, user, group) = false then
            return EPERM
        end if
        if resource.isNotEmpty then
15:         return ENOTEMPTY
        end if
        resource.removeEdge
        resource.delete
        parent.ctime ← now
20:     parent.mtime ← now
        return 0
    end function
```

Il rename delle risorse

Nel sistemi Unix è possibile rinominare una risorsa attraverso la system call *rename*; un'analoga funzione è stata implementata nel filesystem virtuale.

In realtà la rename effettua un'operazione ben più complessa di quello che ci si aspetterebbe; se infatti il nuovo nome che si vuole assegnare alla risorsa esiste già allora il filesystem tenterà di sovrascrivere la seconda con la prima, seguendo la seguente politica:

- se la risorsa da rinominare è una directory allora quella che deve essere sovrascritta deve essere a sua volta una directory;
- se la risorsa da rinominare non è una directory allora non può essere sovrascritta una risorsa che sia una directory;
- infine se la risorsa da rinominare è una directory e quella che deve essere sovrascritta è non vuota allora non è possibile portare a termine l'operazione.

Oltre a ciò, la ridenominazione offre anche il supporto allo spostamento del file, proprio come il comando *mv*. L'operazione di *rename* comporta inoltre l'aggiornamento dei tempi di modifica e cambiamento delle due risorse padre, e del solo tempo di cambiamento per quella rinominata.

Algorithm 14 La funzione di rename

```
function RENAME(path, newPath, user, group)
    retValue ← newIntWrapper()
    resource ← getResource(path, retValue, user, group)
    if resource = null then
        return retValue.value
    end if
    parent ← resource.getParent
    if canWrite(parent, user, group) = false then
        return EPERM
    end if
    newParent ← getResourcePath(path, retValue, user, group)
    if newParent = null then
        return retValue.value
    end if
    if newParent.isNotDirectory then
        return ENOTDIR
    end if
    if canWrite(newParent, user, group) = false or
canEcecute(newParent, user, group) = false then
        return EPERM
    end if
    oldName ← path.getResName
    newName ← newPath.getResName
    if newParent.hasNoOutVertex(newName) then
        resource.name ← newName
        resource.removeInEdges
        newParent.addOutEdge(resource, newName)
    else
        otherRes ← newParent.getOutEdge(newName)
        if otherRes.isNotDirectory and resource.isDirectory then
            return ENOTDIR
        end if
        if otherRes.isDirectory and resource.isNotDirectory then
            return EISDIR
        end if
        if otherRes.isDirectory and otherRes.isEmpty then
            return ENOTEMPTY
        end if
        resource.removeInEdges
        edgeSet ← otherRes.getInEdges
        while edgeSet.isNotEmpty do
            edge ← edgeSet.extract
            edge.setVertices(newParent, resource)
        end while
        resource.name ← newName
        otherRes.delete
    end if
    parent.ctime ← now
    parent.mtime ← now
    resource.ctime ← now
    newParent.ctime ← now
    newParent.mtime ← now
    return 0
end function
```

La gestione della policy delle risorse

Come è stato analizzato nei paragrafi precedenti, la gestione della policy di una risorsa è determinata e dal proprietario (inteso sia come utente che come gruppo) e dai permessi che il proprietario ha stabilito per essa.

Questi tre elementi fanno parte di quella parte di dati che non concorrono a creare l'informazione vera su cui un utente lavora, ma fanno parte dei *metadati* della risorsa, ossia quei dati *sui dati* che permottono di identificarla univocamente e di gestirla in maniera opportuna.

Tralasciando la presenza del superutente, che può tutto, in un sistema Unix la suddivisione utente/gruppo e permessi genera questo tipo di gestione: quando un utente desidera effettuare un'operazione viene verificato se i permessi da lui posseduti per quella risorsa sono sufficienti per portarla a termine; come si effettua il check? Semplicemente, in base al nome utente ed al gruppo vengono estratti dai permessi quei bit necessari alla verifica. Chi, però, stabilisce i permessi su un file, però, è solo ed esclusivamente l'utente proprietario della risorsa, che può cambiarli con il comando *chmod* (che fa riferimento all'omonima system call) da terminale. Ecco perchè è importante questo dualismo proprietario-permessi: è il primo che concede o revoca i secondi, anche a se stesso se necessario.

Il possesso di una risorsa può anche essere delegato ad un altro utente da terminale tramite il comando *chown* (che richiama l'omonima system call).

Entrambi i metodi implementati fanno uso di un metodo privato, che si occupa di prendere la risorsa specificata nel path e restituirla nel caso in cui l'utente sia effettivamente il suo proprietario (e quindi abbia i diritti di eseguire una chmod o una chown) o l'utente *root*, ne modificano il ctime ed aggiornano i dati con quelli richiesti.

Algorithm 15 La funzione che verifica i diritti per eseguire chmod o chown

```
function getResourceForCh(path, user, group, retValue)
    resource ← getResource(path, retValue, user, group)
    if resource = null then
        return null
    end if
    if resource.isLink then
        resource ← resource.followLinks
        if resource = null then
            return null
        end if
    end if
    if user.isRoot then
        return resource
    end if
    if user = resource.user and group = resource.group then
        return resource
    end if
    return null
end function
```

Algorithm 16 La funzione chown

```
function chown(path, user, group, newUser, newGroup)
    retValue ← newIntWrapper()
    resource ← getResourceForCh(path, retValue, user, group)
    if resource = null then
        return retValue.value
    end if
    resource.ctime ← now
    resource.user ← newUser
    resource.group ← newGroup
    return 0
end function
```

Algorithm 17 La funzione chmod

```
function chmod(path, user, group)
    retValue ← newIntWrapper()
    resource ← getResourceForCh(path, retValue, user, group)
    if resource = null then
        return retvalue.value
    end if
    resource.ctime ← now
    resource.mode ← mode
    return 0
end function
```

Le funzione write e read

Sebbene le funzioni che sono state fin'ora analizzate siano tutte importanti, in un filesystem sono presenti altre due funzioni che sono indispensabili (oltre alle già discusse funzioni di creazione di risorse): la *write* e la *read*.

Basandosi sulla rappresentazione delle risorse già introdotta, abbiamo che un file può (e deve) avere un'informazione da mettere a disposizione dell'utente per l'utente. In un comune filesystem i dati vengono divisi in blocchi di dimensione fissa e sono accedibili tramite gli inode; si ha così un'allocazione dinamica della memoria che segue le necessità dell'utente (una dimensione del file maggiore comporta un maggior numero blocchi allocati), che limita gli sprechi di spazio e che permette di riallocare i blocchi disallocati.

Una simile implementazione appare evidente essere impossibile da realizzare in un dispositivo che di sua natura non è a blocchi, seppur appaia come la scelta più saggia da intraprendere. La simulazione di un tale comportamento è però offerta dal database su cui si basa questa implementazione: OrientDB, a differenza di un database relazionale, in cui la *prima forma normale* richiede che i valori associati ad una tupla siano tutti di tipo elementare, permette di memorizzare degli oggetti, ed in questo caso l'oggetto necessario è una lista, in particolare è stato utilizzato un *ArrayList*.

Questo oggetto, va sottolineato, viene utilizzato in coppia con un'altra caratteristica propria di OrientDB, la capacità di memorizzare informazioni binarie; fin'ora abbiamo visto che tutto ciò che è stato memorizzato nel database è stato salvato sotto forma di ODocument[9], ma si ha a disposizione anche un'altra risorsa concreta, l'OrecordBytes, che come riportato nella documentazione ufficiale, è un tipo di record capace di memorizzare dati binari senza necessità di effettuare alcuna conversione. Questo tipo di record può prendere o cedere informazioni in stream o direttamente in un array di byte (il vantaggio di usare questa loro implementazione piuttosto che un array di byte Java verrà discusso nel capitolo seguente).

OrientDB permette di salvare record binari senza necessità di stabilirne una dimensione fissa, possiamo quindi avere record che memorizzano 27byte ed altri record che ne memorizzano un milione, e in questa imnplementazione essa viene imposta al momento della creazione del database specificandola e quindi settando la variabile CHUNK_SIZE. Inoltre un vantaggio significativo dell'uso dei

[9] OVertex, OEdge e qualsiasi altra risorsa fin'ora usata altro non era che un'estensione della classe ODocument, ma concretamente si è operato con dei documenti

record binary è quello di non avere necessità di caricare in memoria tutto il file richiesto in una volta, ma caricare solo i *riferimenti* ai record. Abbiamo quindi una struttura in cui, da una parte, abbiamo l'ArrayList che tiene dinamicamente i puntatori ai record, e dall'altra parte i record.

Si supponga che venga richiesta la lettura di un file la cui dimensione è di 500MB, e di creare un filesystem i cui blocchi hanno una dimensione di 256kB, questo comporta l'allocazione di 2000 blocchi; nel momento in cui al record vengono richiesti i dati non si renderà necessario caricare in memoria l'intero file, con il rischio di terminare spazione nello heap, ma sarà possibile caricare blocco per blocco, con un'occupazione in memoria costante di soli 256kB, permettendo così anche un accesso di tipo ibrido tra il sequenziale (in cui la lettura inizia per forza dalla posizione 0) ed il random (in cui è possibile effettuare una lettura in qualsiasi punto arbitrario). Se poi, durante un'operazione di scrittura si renderà necessario allocare nuovi blocchi, o, al contrario, rimuoverli, sarà possibile farlo in maniera efficiente.

Il metodo che permette la scrittura di un file richiede che vengano specificati il path della risorsa da scrivere, un array di byte rappresentanti i dati, un offset da cui iniziare la scrittura, un numero che indica quanti byte scrivere, oltre che le credenziali dell'utente.

Inizialmente viene verificato se la risorsa specificata nel path esiste, se non dovesse esistere e fosse possibile crearla, viene richiesto al metodo che si occupa della creazione di risorse, di inizializzare un nuovo file. Successivamente, se la risorsa specificata è un Link, allora si segue fino ad arrivare alla risorsa linkata, se invece viene richiesta la scrittura su una directory viene ritornato un errore; controllati i permessi in scrittura sul file, si passa ad allocare eventuali nuovi blocchi necessari, per poi effettuare la scrittura vera e propria, che termina con un aggiornamento dei tempi di modifica e cambiamento, oltre che della dimensione del file e dei puntatori di record.

Analizzata la funzione *write*, non resta molto da aggiungere per poter introdurre la *read*, se non che questa richiede un offset da cui iniziare la lettura ed il numero di quanti byte (al più) è necessario leggere.

Algorithm 18 La funzione write

```
function WRITE(path, data, offset, size, user, group)
    retValue ← newIntWrapper()
    resource ← getResource(path, retValue, user, group)
    if resource = null and offset = 0 then
        err ← create_resource(path, 0755, user, group, "File", noCommit)
        if err = 0 then
            return write(path, data, offset, size, user, group)
        else
            return err
        end if
    end if
    if resource = null then
        return ENOENT
    end if
    if resource.isLink then
        resource ← resource.folowLinks
    end if
    if resource.isLink then
        return ENOENT
    else if resource.isDirectory then
        return EISDIR
    end if
    if canWrite(resource, user, group) = false then
        return EPERM
    end if
    fileSize ← resource.size
    if fileSize < offset − 1 then
        return EOF
    end if
    resource.allocRequiredBlocks
    remaining ← size
    index ← ⌊offset/CHUNK_SIZE⌋
    dataIndex ← 0
    while remaining > 0 do
        buffer ← resource.loadBlock(index)
        if remaining = size and offset%CHUNK_SIZE > 0 then
            offPos ← offset%CHUNK_SIZE
            copiedBytes ← min(size, CHUNK_SIZE − offPos)
            arraycopy(data, 0, buffer, offPos, copiedBytes)
        else
            copiedBytes ← min(remaining, CHUNK_SIZE)
            arraycopy(data, dataIndex, buffer, 0, copiedBytes)
        end if
        buffer.saveBlock
        remaining ← remaining − copiedbytes
        dataIndex ← dataIndex + copiedBytes
        index + +
    end while
    resource.size ← max(offset + size, fileSize)
    resource.ctime ← now
    resource.mtime ← now
    return size
end function
```

Algorithm 19 La funzione read

```
function READ(path, offset, size, user, group, retVal)
    resource ← getResource(path, retValue, user, group)
    if resource = null then
        return null
    end if
    if resource.isLink then
        resource ← resource.folowLinks
    end if
    if resource.isLink then
        retValue.value ← ENOENT
        return null
    else if resource.isDirectory then
        retValue.value ← EISDIR
        return null
    end if
    if canRead(resource, user, group) = false then
        retValue.value ← EPERM
        return null
    end if
    fileSize ← resource.size
    if fileSize < offset − 1 then
        retValue.value ← EOF
        return null
    end if
    if offset + size > fileSize then
        size ← fileSize − offset
    end if
    data ← byte[size]
    startRecord ← ⌊offset/CHUNK_SIZE⌋
    endRecord ← ⌊(offset + size)/CHUNK_SIZE⌋
    if (offset + size)%CHUNK_SIZE = 0 then
        endRecord − −
    end if
    index ← startRecord
    dataIndex ← 0
    while dataindex < size do
        buffer ← resource.loadBlock(index)
        if index = startRecord then
            copiedBytes ← min(size − dataIndex, CHUNK_SIZE −
offset%CHUNK_SIZE)
            data.append(buffer, offset%CHUNK_SIZE, copiedBytes)
        else if index = endRecord then
            copiedBytes ← size − dataIndex
            data.append(buffer, 0, copiedBytes)
        else
            copiedBytes ← CHUNK_SIZE
            data.append(buffer, 0, CHUNK_SIZE)
        end if
        dataIndex ← dataIndex + copiedBytes
        index + +
    end while
    resource.atime ← now
    return data
end function
```

La funzione truncate

Insieme alla *read* ed alla *write* è presente una terza funzione di gestione dei file, la *truncate*; come riportanto alla pagina internet *http://unixhelp.ed.ac.uk/CGI/man-cgi?truncate+2*, questo metodo comporta il ridimensionamento del file su cui è invocato, e prevede la specifica di una dimensione. Se questa dimensione è minore di quella attuale del file, allora l'informazione eccedente è persa, mentre se è maggiore il file viene esteso, e la parte nuova viene colmata con dei byte 0.

Poichè ridimensionare un file comporta la necessità di allocare o disallocare blocchi di memoria, anche questa funzione opera con gli ORecordBytes e la lista di puntatori ad essi.

Proprio in questa situazione si può apprezzare una delle differenze tra un dispositivo a blocchi virtuali ed uno a blocchi fisici: mentre l'operazione di disallocamento comporta classicamente la sola cancellazione degli inode interessati, con conseguente frammentazione del disco, in OrientDB questo non accade, poichè non sono solo i puntatori degli ORecordBytes ad essere eliminati, ma i record stessi. Questa strategia, pur se non ottimale (dal momento che la creazione di un record non è un'operazione *costless*), è stata obbligata per permettere di sfruttare tutto lo spazio necessario e non di più: mantenere infatti dei record nel database per permettere il loro *riciclo* avrebbe comportato la necessità di occupare spazio su disco inutilmente, con conseguente rischio di saturarlo.

Poichè il comportamento standard richiede che, nel caso di un'estensione del file, i nuovi byte siano posti a 0, si è reso necessario compiere una scelta: o i byte eccedenti del blocco vengono messi a 0 (ovvero viene eseguita una sovrascrittura di tutti 0) quando il file è ridimensionato in positivo, o quando viene ridimensionato in negativo. La scelta è completamente arbitraria, e nessuna soluzione comporta perdite prestazionali rispetto all'altra. In questa implementazione è stato preferito eseguire l'annullamento in fase di resize negativo. Per il resto il comportamento è analogo a quello tenuto dalla *write*, verrà presentato lo pseudocodice senza dare ulteriori spiegazioni.

Algorithm 20 La funzione truncate

```
    function TRUNCATE(path, size, user, group)
        resource ← getResource(path, retValue, user, group)
        if resource = null then
            return ENOENT
 5:     end if
        if resource.isLink then
            resource ← resource.folowLinks
        end if
        if resource.isLink then
10:         return ENOENT
        else if resource.isDirectory then
            return EISDIR
        end if
        if canWrite(resource, user, group) = false then
15:         return EPERMS
        end if
        if size = 0 then
            resource.deallocAll
            return 0
20:     end if
        fileSize ← resource.size
        if fileSize = size then
            return 0
        end if
25:     resource.allocRequiredBlocks
        if fileSize > size then
            resource.zeroExceedBytes
        end if
        resource.size ← size
30:     resource.mtime ← now
        resource.ctime ← now
        return 0
    end function
```

La funzione readdir

Una caratteristica dei filesystem che li rende navigabili è la possibilità di visualizzare la struttura in cui ci troviamo, ossia le risorse che da una determinata directory sono accessibili. La funzione che rende possibile tutto ciò è un'analoga logica della *read*, la *readdir*; come espresso anche dal nome, questa esegue una lettura del contenuto di una directory, e nell'implementazione del driver fuse essa restituisce un puntatore ad una sequenza di strutture contenenti le informazioni richieste. Poichè in Java il concetto di *struct* non è presente, l'attuale implementazione restituisce una lista di oggetti, istanze della classe DirList, che ha due attributi privati, reName e resType, accessibili tramite i corrispondenti *getter*.

Questo metodo richiede in input il path della risorsa di cui leggere il contenuto, oltre alle credenziali dell'utente e una varibile dove inserire un eventuale codice di errore. Se la risorsa richiesta non esiste, è un file, o è un link la cui risorsa linkata è stata eliminata allora l'esecuzione termina riportando il codice appropiato. Successivamente viene preso l'insieme di vertici uscenti dalla directory richiesta e, elemento per elemento, vengono aggiunte le informazioni alla lista che poi sarà restituita.

Algorithm 21 La funzione readdir

```
function READDIR(path, user, group, retValue)
    retValue ← newIntWrapper()
    resource ← getResource(path, retValue, user, group)
    if resource = null then
        retValue.value ← ENOENT
        return null
    end if
    if resource.isLink then
        resource ← resource.folowLinks
    end if
    if resource.isLink then
        retValue.value ← ENOENT
        return null
    else if resource.isFile then
        retValue.value ← ENOTDIR
        return null
    end if
    if canRead(resource, user, group) = false then
        retValue.value ← EPERM
        return null
    end if
    vertices ← resource.getOutVertices
    while vertices.isNotEmpty do
        vertex ← vertices.extract
        list.add(newDirList(vertex.name, vertex.type))
    end while
    resource.atime ← now
    return list
end function
```

La funzione utime

Un'ultima funzionalità messa a disposizione dai filesystem è quella di poter modificare di tempi di accesso e/o di modifica di una risorsa

La policy attuata per poter usare questo metodo nei filesystem comuni, e riproposta in questa implementazione, è la seguente:

- se chi invoca il metodo è il proprietario della risorsa o è l'utente amministratore, allora atime e mtime sono modificati coerentemente con i dati passati, e sostituiti con il timestamp attuale se nulli;
- se chi invoca il metodo non è il proprietario della risorsa, ma ha comunque permessi di scrittura su essa allora i valori atime e ctime sono settati con il timestamp attuale;
- ogni altro tentativo viene bloccato.

Algorithm 22 La funzione utime

```
    function UTIME(path, user, group, retValue)
        retValue ← newIntWrapper()
        resource ← getResource(path, retValue, user, group)
        if resource = null then
 5:         retValue.value ← ENOENT
            return null
        end if
        if resource.isLink then
            resource ← resource.folowLinks
10:     end if
        if resource.isLink then
            retValue.value ← ENOENT
            return null
        end if
15:     if user.isRoot or user.isOwner then
            if atime = null then
                atime ← now
            end if
            if mtime = null then
20:             mtime ← now
            end if
        else if atime = null and ctime = null and
    canWrite(resource, user, group) then
            atime ← now
            mtime ← now
25:     else
            return EPERM
        end if
        resource.atime ← atime
        resource.mtime ← mtime
30:     return 0
    end function
```

Chapter 5

I test sulle prestazioni

In questo capitolo verranno analizzati i test affrontati per valutare la reale bontà delle scelte fin qui discusse per quanto riguardo le due funzioni critiche di un filesystem: la *read* e la *write*.

Il motivo per chui si è scelto di analizzare solamente queste due funzioni è il seguente: sebbene l'intero filesystem dovrebbe essere ottimizzato, per la maggior parte del codice non è stato possibile implementare strategie alternative a quelle realizzate dal momento che il vero *bootlneck* è il database, ed anche diminuendo il numero di istruzioni scritte non sarebbe diminuita la complessità generale del sistema.

Come abbiamo visto invece nel capitolo precedente, nel realizzare la persistenza di dati binari si è reso necessario compiere delle scelte che, come vedremo, comportano significative differenze.

5.1 L'analisi della persistenza dei dati

In questo paragrafo verranno presentate le possibili scelte realizzative che si sono prese in considerazione nell'implenetazione della persistenza dei dati binari.

In tutto il progetto il requisito fondamentale che è stato sempre rispettato è stata la flessibilità, intesa a tutto tondo: possibilità di sostituire il backend, possibilità di scalare orizzontalmente su un cluster di macchine e non porre limiti a quello che un utente, o un gruppo di essi, può fare con il suo filesystem, coerentemente con una gestione dei permessi di tipo Unix-like.

Proprio quest'ultimo aspetto della modularità è stato affrontato nella real-

izzazione delle due funzioni che trattano dati binari; non si è voluto stabilire un limite massimo per la dimensione dei file memorizzati nel database (se non quello imposto dal dispositivo fisico su cui si lavora).

Prima di giungere a come il problema è stato risolto è opportuno ricordare come un filesystem comune funziona: almeno per quanto ci possa interessare, è possibile distinguere inode e blocchi di dati; nei primi sono memorizzati i metadati ed i puntatori ai secondi[1].

La prima soluzione ideata prevedeva che un solo array di byte si occupasse di memorizzare l'intero file binario; si può ben capire che questa tecnica può essere sufficiente fin quando un file è di piccole dimensioni, ma non è più sufficiente per la gestione di grandi quantità di dati, in quanto la lettura, fosse anche di un solo byte, avrebbe richiesto il caricamento in memoria dell'intero array, e poichè lo spazio nello *heap* della Virtual Machine è limitato questa soluzione è apparsa da subito improponibile. Per di più, nel caso di un ridimensionamento del file si sarebbe reso necessario copiare in un nuovo array tutti i dati da salvare, operazione banale da un punto di vista algoritmico, ma decisamente complessa meno da un punto di vista computazionale e di memoria (lo spazio necessario sarebbe diventato la somma di quello necessario per contenere entrambi).

Una seconda soluzione, nata dalla precedente, prevedeva, invece di un solo array di byte, un array di array di byte, che sarebbero stati tutti dimensionati a 0, ed allocati al momento del bisogno. Questa soluzione è stata immediatamente abbandonata perchè sofferente di un problema ancor maggiore della precedente: gli array di byte avrebbero rappresentato i dati salvati, ma poichè l'array di array è inizializzato ad una dimensione fissa, sarebbe stato necessario imporre all'utente dei vincoli sulla dimensione massima dei file memorizzati (se ad esempio abbiamo blocchi da 512kB e fosse stato allocato in memoria un array di array di dimensione 50, la dimensione massima sarebbe stata 25600kB).

A questo punto è stato deciso di sacrificare, parzialmente, le performance globali sostituendo l'array di array con un ArrayList, che garantisce le stesse funzionalità della soluzione precedente, senza però imporre un vincolo sulla dimensione massima.

Infine, analizzando la documentazione di OrientDB, è stato possibile trovare una quarta possibilità, propria del database, utilizzando il record ORecordBytes, appositamente studiato per dati binari, in sostutuzione dell'array di byte e mantenendo l'ArrayList.

[1] Questa è un'approssimazione, in quanto un inode contiene un blocco di puntatori a dati, e due puntatori a blocchi di altri puntatori a dati

Nel corso del prossimo capitolo verranno analizzati i risultati dei test che hanno confrontato la terza e la quarta soluzione, spiegando poi perchè la scelta sia ricaduta su quest'ultima.

5.1.1 Come sono stati condotti i test

Prima di presentare i risultati dei test è necessario dedicare qualche parola per descrivere come i test sono stati condotti.

Innanzi tutto per confrontare le due versioni è stato necessario creare due versioni della classe che implementa la read e la write, ciascuna delle quali realizza una soluzione diversa, e quindi sono stati ottenuti due pacchetti *jar* differenti. Poichè il progetto presentato è una libreria, è stato realizzato un metodo *main*, in modo da rendere il pacchetto eseguibile, che esegue azioni diverse in base ai parametri con cui viene lanciato.

Oltre a ciò è stato scritto un semplice script bash in modo da rendere automatica l'esecuzione dei test, ciascuno dei quali è stato ripetuto con una dimensione dei blocchi di allocazione man mano crescente per dieci volte, così da ridurre al minimo la possibilità che il risultato sia legato al carico complessivo del sistema operativo.

Il test è stato così condotto: da un file binario la cui dimensione è 26.1MB viene estratta la sua sequenza di byte che è poi memorizzata in un array di byte. Poi, a seconda che sia richiesta una lettura, una scrittura o entrambe, il sistema si comporta di conseguenza. L'istruzione precedente, così come la successiva, all'invocazione del metodo richiesto è una chiamata della funzione *System.currentTimeMillis()*, ed il tempo di esecuzione viene calcolato effettuando una sottrazione tra i due tempi: il risultato così ottenuto è preciso al millesimo di secondo.

5.2 Il test di scrittura

In questo paragrafo saranno analizzati i risultati dei test condotti sull'operazione di scrittura di dati. Come abbiamo detto il numero di byte memorizzati è $26{\cdot}056{\cdot}704$, e non sono stati generati in maniera casuale dal sistema, ma sono byte estratti da un file concreto.

I risultati, va sottolineato, forniscono un *upper bound* sul tempo effettivamente richiesto, dal momento che, al termine di ogni esecuzione, il database è stato cancellato per costringere il sistema a creare nuovi record senza permet-

	1kB	**2kB**	**4kB**	**16kB**	**64kB**	**256kB**	**1MB**
ORecordBytes	3266.8	2356.4	1506.8	691.3	322.2	170.7	122.8
Array di byte	1139.7	1058.9	1053.5	1019.9	1060.6	1049.7	1066.8
Differenza	2127.1	1297.5	453.3	-328.6	-738.4	-879.0	-944

Figure 5.1: Tabella che elenca i tempi medi di scrittura (espressi in ms) nelle due soluzioni in base alle dimensioni dei blocchi allocati e loro differenza

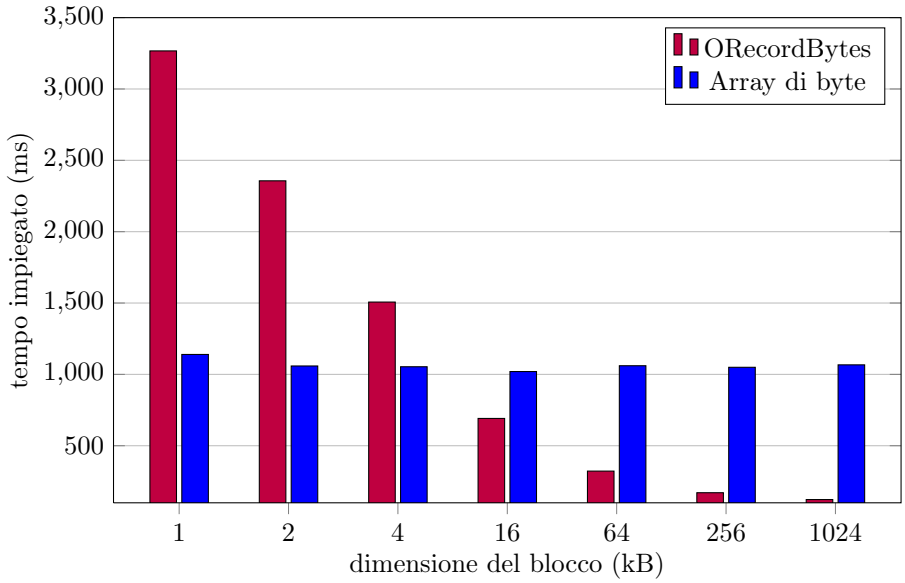

Figure 5.2: Grafico del tempo impiegato per la memorizzazione di un file da 26,1MB

tere il riciclo di quelli vecchi; come abbiamo visto, infatti, nuovi blocchi vengono allocati dinamicamente solo se necessario in quanto questa è un'operazione computazionalmente pesante e che richiede molto tempo[2]. Si garantisce quindi che, a regime, il tempo necessario per la scrittura effettivamente necessario sarà *mediamente* minore.

Nelle figure 5.1 e 5.2 sono riportati i risultati dei test, che sono abbastanza significativi.

Come è possibile apprezzare, l'implementazione degli ORecordBytes permette una sensibile variazione delle performance al variare della dimensione dei blocchi allocati (principalmente a causa del fatto che la creazione di oggetti di

[2] Ad esempio, l'operazione di sovrascrittura non prevede alcuna allocazione di nuovi record

	1kB	**2kB**	**4kB**	**16kB**	**64kB**	**256kB**	**1MB**
ORecordBytes	3764.5	1782.4	1062.2	462.4	237.4	127.3	134.9
Array di byte	2873.0	2816.9	2853.1	2778.4	2842.5	2866.3	2867.7
Differenza	891.5	-1034.5	-1790.9	-2316.0	-2605.1	-2739.0	-2732.8

Figure 5.3: Tabella che elenca i tempi medi di lettura (espressi in ms) nelle due soluzioni in base alle dimensioni dei blocchi allocati e loro differenza

tipo ORecordBytes è molto più costosa della semplice allocazione in memoria di un array di byte). Questa differenza, però, svanisce non appena diminuisce il numero di record allocati: infatti, mentre la soluzione che prevede gli array di byte mantiene (quasi) costante il suo tempo di esecuzione, l'altra soluzione, nel passare da un blocco di dimensione 1kB ad uno di dimensione 1MB, lo abbatte di un fattore 26, passando da 3266.8ms a 122.8ms.

Naturalmente aumentare la dimensione dei blocchi è un'operazione non priva di costi aggiuntivi visto che allocare spazio extra significa generare sprechi. Statisticamente, lo spazio eccedente occupato ma non sfruttato è la metà della dimensione del blocco. Va quindi ben ponderata la scelta di quanto spazio dedicare a cisacun record, considerando anche che, ad oggi, non è previsto di poterlo modificare.

Una possibile spiegazione al motivo per cui la soluzione che prevede l'array di byte non è sensibile alla variazione della dimensione dei blocchi (i tempi di esecuzione differiscono al massimo di 120ms, e raggiungono il loro minimo con dimensione 16kB) è che il sistema non prevede tuning specifici; inoltre questa soluzione prevede sempre e comunque che un file venga caricato interamente nello heap per poterlo manipolare, con conseguente appesantimento del sistema.

5.3 Il test di lettura

Successivamente sono stati eseguiti i test di lettura del file appena scritto. Va evidenziato che per non sfruttare potenzialità proprie di OrientDB alla scrittura iniziale e ad ogni lettura è stata fatta seguire una chiusura del database, così da permettere un confronto equo tra le due soluzioni. La lettura richiesta è stata di tipo completo, ossia a partire dalla posizione 0 per tutto il file. Questo non comporta falsificazioni del risultato finale visto che, in entrambe le implementazioni, posizionarsi in una posizione qualsiasi del file introduce un overhead nullo.

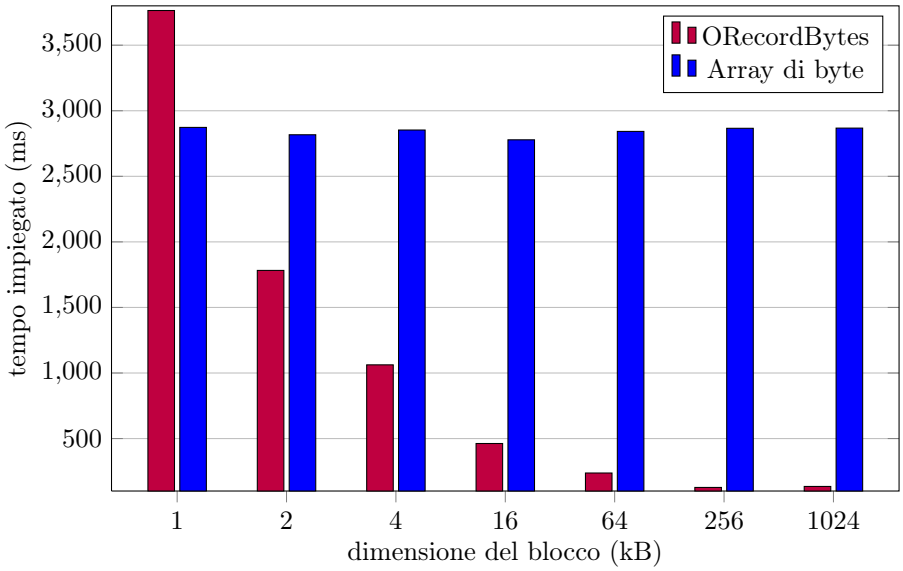

Figure 5.4: Grafico del tempo impiegato per la lettura di un file da 26,1MB

Anche in questo caso, come riportato nelle figure 5.3 e 5.4, si confermano i dati ottenuti nel test precedente, ossia l'uso di ORecordBytes permette una diminuzione massima dei tempi di esecuzione pari a 2732.8ms rispetto alla soluzione che fa uso degli array di byte.

Va notato che però, a differenza dei risultati ottenuti in lettura, la diminuzione dei tempi di esecuzione non è costante, ma, nel passaggio della dimensione dei blocchi da 256kB ad 1MB risulta essere di segno opposto.

Un'ultima annotazione riguarda un confronto tra i tempi di lettura e quelli di scrittura è la seguente: generalmente i tempi di lettura di un dispositivo, a parità di dimensione richiesta, sono minori, ma in questo caso accade il contrario. Questo è indice di cosa in realtà accade in ORecordBytes: quando si chiede a quest'oggetto di memorizzare, in realtà viene semplicemente copiato un array già presente in memoria, mentre quando viene richiesto di trasformare il suo contenuto in un array di byte il sistema deve allocare dello spazio aggiuntivo non previsto nello heap.

5.3.1 OrientDB e la cache

Nel test precedente per evitare di sfruttare tutte le peculiarità di ORecordBytes non disponibili per le altre soluzioni si è agito evitando di utilizzare la

	1kB	**2kB**	**4kB**	**16kB**	**64kB**	**256kB**	**1MB**
Con cache	2334.1	1432.7	406.7	144.2	43.6	27.0	22.7
Senza cache	3764.5	1782.4	1062.2	462.4	237.4	127.3	134.9
Differenza	-1430.4	-349.7	-655.5	-318.2	-139.8	-100.3	-112.2

Figure 5.5: Tabella che elenca i tempi medi di lettura (espressi in ms) nelle due soluzioni in base alle dimensioni dei blocchi allocati e loro differenza

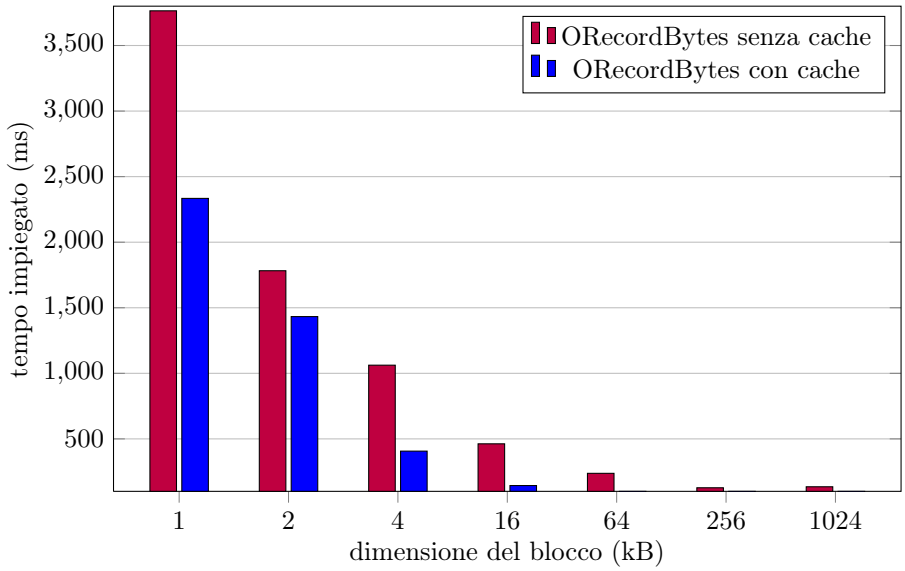

Figure 5.6: Grafico del tempo impiegato per la lettura di un file da 26,1MB senza e con cache

cache. Anche in questo caso, quindi, si è trovato un upper bound (affermazione falsa per quanto riguarda la soluzione che prevede gli array di byte).

Volendo vedere qual è la prestazione massima effettivamente fruibile in lettura è stato fatto un test così strutturato: un file viene scritto, poi, senza chiudere il database, ne viene richiesta la lettura.

Il risultato è riportato nelle figure 5.5 e 5.6, ed è sintetizzabile facendo notare come usare la cache significa ottenere un risparmio medio del 61,68%, con punte minime di 19,62% e punte massime di 83,17%. Questo perchè i record non vengono effettivamente caricariti e riscaricati sul disco, ma rimangono in memoria e vengono automaticamente recuperati dall'engine del database quando sono richiesti. In quest'ottica è anche spiegabile come mai il risparmio di tempo è crescente: il numero di record da leggere è inversamente proporzionale alla di-

mensione del blocco usato, quindi, minori sono i record utilizzati e minore sarà il numero di letture effettivamente da disco da effettuare (infatti il numero totale di record mantenibili in cache è pressochè costante).

List of Algorithms

List of Figures