Alma Mater Studiorum · Università di Bologna · Campus di Cesena

SCUOLA DI SCIENZE

Corso di Laurea in Scienze e Tecnologie Informatiche

Analisi e Sperimentazione del DBMS VoltDB

Tesi di Laurea in Laboratorio di Basi di Dati

Relatore:
Matteo Golfarelli

Presentata da:
Valentina Intrusi

Sessione II
Anno Accademico 2013/2014

Indice

1 Classificazione DBMS NewSQL **1**

1.1 Perchè NewSQL . . . 1

1.2 NoSQL vs NewSQL . . . 7

1.3 In-Memory DBMS . . . 9

2 VoltDB **11**

2.1 Le origini: il progetto H-Store . . . 11

2.2 Cos'è VoltDB . . . 14

2.2.1 Come Funziona VoltDB . . . 15

2.3 Le proprietà ACID . . . 21

2.3.1 Atomicity, Consistency ed Isolation . . . 21

2.3.2 Durability . . . 21

2.3.3 Availability . . . 23

3 Caso di studio **27**

3.1 Schema Logico del Database . . . 27

3.2 Descrizione del Cluster . . . 31

3.3 Risultati . . . 32

3.3.1 Confronti tra MySQL e VoltDB . . . 36

3.4 Punti di Debolezza di VoltDB . . . 38

3.5 Sviluppi futuri . . . 41

A Benchmark **43**

A.1 Schema Logico . . . 43

A.2 Query svolte . . . 46

B Precisazioni Pratiche **53**

Introduzione

25 anni di sviluppo di DBMS commerciali possono essere riassunti con 3 termini: standardizzazione, affidabilità e flessibilità. Questa affermazione si riferisce al fatto che la medesima architettura dei DBMS tradizionali è stata usata per supportare applicazioni aventi le più diverse caratteristiche e necessità. Tuttavia l'enorme crescita dei dati ha portato i ricercatori a sviluppare nuovi sistemi di gestione degli stessi. Questo ha visto fiorire i sistemi NoSQL: imbattibili in velocità e performance, ma discutibili per quanto riguarda la garanzia sulla corretta esecuzione delle transazioni.
C'è da chiedersi quindi: le imprese possono permettersi di rinunciare al supporto SQL e, soprattutto, alle proprietà ACID? La risposta, ovviamente, è no.

In questo contesto è emersa quindi la necessità di avere un nuovo paradigma avente il supporto (New)SQL e contemporaneamente le ottime performance per quanto riguarda scalabilità e disponibilità.
Per verificare che quanto promesso venisse mantenuto, si è scelto di testare il DBMS VoltDB, primo rappresentate commerciale dei sistemi NewSQL, che si propone come la soluzione adatta alle necessità di alte prestazioni e affidabilità dei dati, per mezzo dell'utilizzo della memoria centrale come memoria principale oltre ad un'architettura distribuita di tipo *shared-nothing* per assicurare scalabilità orizzontale.

In questa tesi verrà studiata la struttura del sistema NewSQL, ponendo particolare attenzione al software VoltDB; esso verrà poi installato su un *cluster* di macchine e testato sul *benchmark* proposto dal *Transaction Processing Performance Council*(TPC).

Nel **Capitolo 1** verrà fornita una panoramica sul sistema NewSQL, le sue caratteristiche principali e il suo funzionamento. Nel seguito sarà effettuato un confronto con la tecnologia NoSQL evidenziando le differenze tra i due. Infine sarà illustrato il modello *In-Memory* e i motivi che hanno spinto a sceglierlo rispetto a quello tradizionale.

Nel **Capitolo 2** sarà descritta l'applicazione VoltDB, dalle sue origini, con il progetto H-Store, fino al suo effettivo sviluppo commerciale; sarà rivolta particolare attenzione ai suoi punti chiave, come vengono garantite le proprietà ACID e, in aggiunta ad esse, la gestione della proprietà di Availability.

Nel **Capitolo 3**, infine, sarà proposta un'analisi prestazionale del software in esame, avendolo installato su un cluster e avendo utilizzato il benchmark TPC-H. Si proporrà successivamente un confronto con l'RDBMS MySQL sotto due aspetti fondamentali: tempo di esecuzione delle *query* e numero di transazioni al secondo. Saranno, infine, mostrati i punti di debolezza di VoltDB.

Capitolo 1

Classificazione DBMS NewSQL

In questo capitolo verrà trattata la descrizione del sistema NewSQL, quali sono le sue caratteristiche fondamentali, gli ambiti principali in cui viene utilizzato e le varie catogorizzazioni dello stesso.
Verrà poi fornito un confronto con il suo predecessore NoSQL, quali sono i motivi che hanno spinto i ricercatori a trovare un'alternativa sia a NoSQL che al SQL tradizionale. Infine sarà spiegato in che cosa consiste il modello In-Memory, di che cosa si tratta e, anche in questo caso, le motivazioni che hanno spinto verso questa soluzione rispetto alla tradizionale.

1.1 Perchè NewSQL

L'enormità dei dati in circolazione oggi ha cambiato radicalmente lo stile di vita di molti utenti; non è sorprendente constatare l'impatto ancora maggiore che ha avuto sulle imprese. L'impresa moderna, infatti, è diventata sempre più preoccupata di possedere servizi affidabili e in grado di rispondere tempestivamente alle richieste. Al contempo l'aspirazione è quella di utilizzare queste informazioni in maniera veloce ed efficace e, conseguentemente, migliorare i propri risultati, che in ambito aziendale si traducono in un aumento del profitto. Secondo una stima [1], i dati creati nel 2010 sono circa 1.200 *exabyte*, e cresceranno a 8.000 *exabyte* entro il 2015, con internet, ed in modo particolare il web, destinato a diventare il primo consumatore e conduttore di dati.

Tale crescita sta superando quella della capacità di immagazzinamento degli stessi; in letteratura si trova un metodo di risoluzione di questo problema nella nascita di sistemi di gestione delle informazioni in cui i dati sono salvati in maniera distribuita, ma che, per semplicità di utilizzo per gli utenti finali, sono acceduti ed analizzati come se risiedessero su una singola macchina.

NewSQL è una nuova classe di sistemi di gestione di database relazionali. Non è ne' un'estesione, ne' un sottoinsieme del linguaggio SQL; non è neanche un linguaggio per basi di dati orientate agli oggetti.
NewSQL è un termine coniato da *The 451 Group* - un importante gruppo di analisti nel campo della *Information Technology* - apparso per la prima volta nel loro famoso *report NoSQL, NewSQL and Beyond [2]* nel 2011. Questo termine evidenzia il suo contrapporsi con il linguaggio SQL tradizionale, descritto come ormai non più adeguato [3].
Questo linguaggio si presenta come un particolare insieme di nuovi database SQL caratterizzati da grande scalabilità e alte prestazioni: queste sono le parole con cui Matthew Aslett, *senior analyst* del *The 451 Group*, utilizza sul suo *blog* per chiarirne la natura. In passato ci si riferiva a tali prodotti con il termine *Scalable-SQL* per differenziarli dai prodotti database relazionali classici. Dal momento che questo termine implica esplicitamente scalabilità orizzontale, non necessariamente una caratteristica di tutti i prodotti che afferiscono a questa categoria, si è adottato il maggiormente descrittivo NewSQL.
I *vendor* che lo adottano hanno creato soluzioni con il fine di portare i benefici del modello relazionale in un'architettura distribuita, o di migliorarne le performance a tal punto da rendere la scalabilità orizzontale non più necessaria.
Si definisce scalabilità la particolare caratteristica di un sistema di incrementare o decrementare le proprie prestazioni in base all'incremento delle proprie risorse; in particolare la scalabilità orizzontale si riferisce all'incremento delle prestazioni di un sistema aggiungendo macchine ad un *cluster* esistente.
Con *On-Line Transactional Processing*, abbreviato in OLTP, si intende l'insieme di tutte le interrogazioni che eseguono transazioni le quali leggono e scrivono *record* da diverse tabelle presenti nel database. *Query* di questo tipo vengono fatte direttamente sui dati provenienti dai database sorgente, mantenuti sempre aggiornati.

D'altra parte però essendo operazioni molto costose dal punto di vista I/O, non sono particolarmente adatte in situazioni in cui la quantità di dati da interrogare è molto elevata.

Differenze tra architettura Shared Everything e Shared Nothing

Al paragrafo precedente si è parlato di salvataggio dei dati in maniera distribuita, ovvero di memorizzazione per mezzo di un *clustered database.* In letteratura, con questo termine, si indica un gruppo di macchine collegate in rete, che cooperano tra loro suddividendosi il carico di lavoro invece di farlo eseguire su una sola macchina *standalone*; il fine di questa architettura è garantire alta disponibilità di servizio, alta affidabilità grazie all'assenza di un singolo punto di fallimento e scalabilità, potendo aggiungere nodi alla struttura.
I *clustered database* si possono suddividere in due categorie: *shared everything* e *shared nothing* in base alle loro caratteristiche architetturali.

Un DBMS ha un'architettura *shared everything* se la memoria è centralizzata, cioè condivisa tra tutti processori; una rappresentazione schematica è fornita in figura 1.1. DBMS che lavorano in questo modo sono detti SMP - *Symmetric Multi-Processing* - ossia sistemi multiprocessori dove più CPU condividono risorse comuni come memoria e periferiche I/O, ma effettuano le proprie operazioni in maniera indipendente restando connesse tra loro tramite *bus* di sistema o *crossbar.* La ridondanza dei componenti hardware aumenta la *fault tolerance* del *cluster*; in caso di guasto di uno dei nodi, gli altri continuano a funzionare correttamente garantendo agli utenti di poter continuare il proprio lavoro [4].

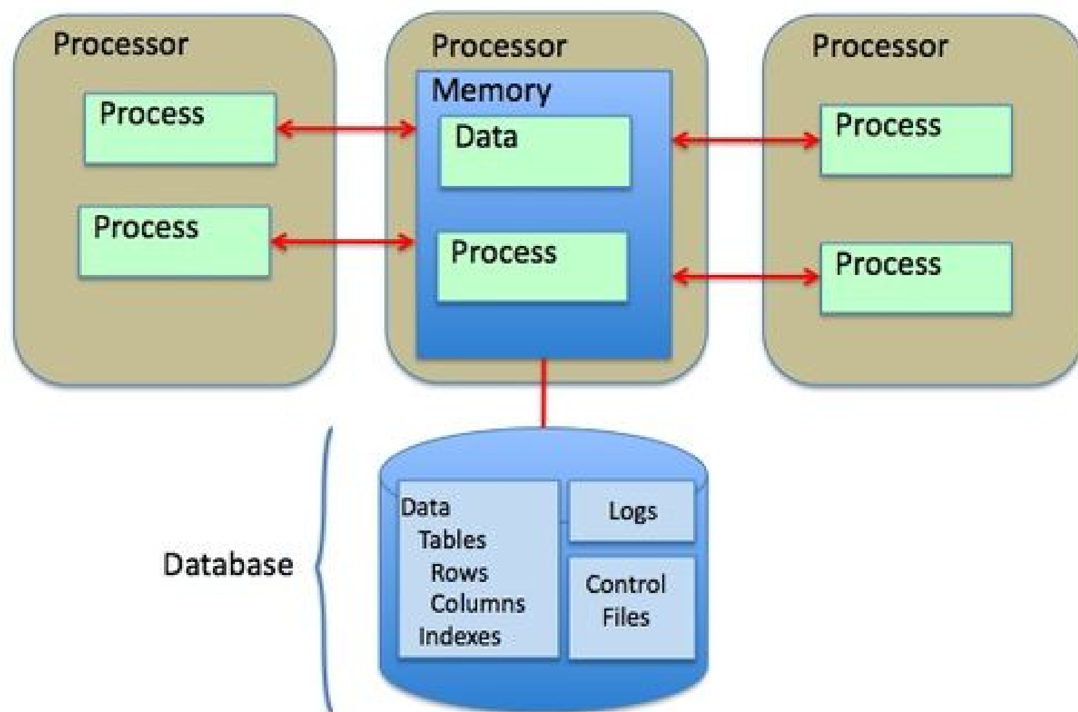

Figura 1.1: Architettura shared everything

All'opposto, un'architettura è *shared nothing*, illustrata in figura 1.2, quando ogni nodo è indipendente dagli altri e autosufficiente, più nello specifico, ogni nodo non condivide memoria RAM o spazio su hard disk con gli altri. Questa tipologia di architettura prevede nella propria struttura un coordinatore; questi si occupa di dirigere le richieste degli utenti all'istanza appropriata e di coordinare la ripartizione dei dati su tutti i nodi. In breve gestisce tutti i nodi in modo che vengano percepiti dall'utente finale come un unico complesso.
Questa caratteristica determina l'aggregazione di *cluster* con un gran numero di nodi senza però doversi preoccupare dei colli di bottiglia.

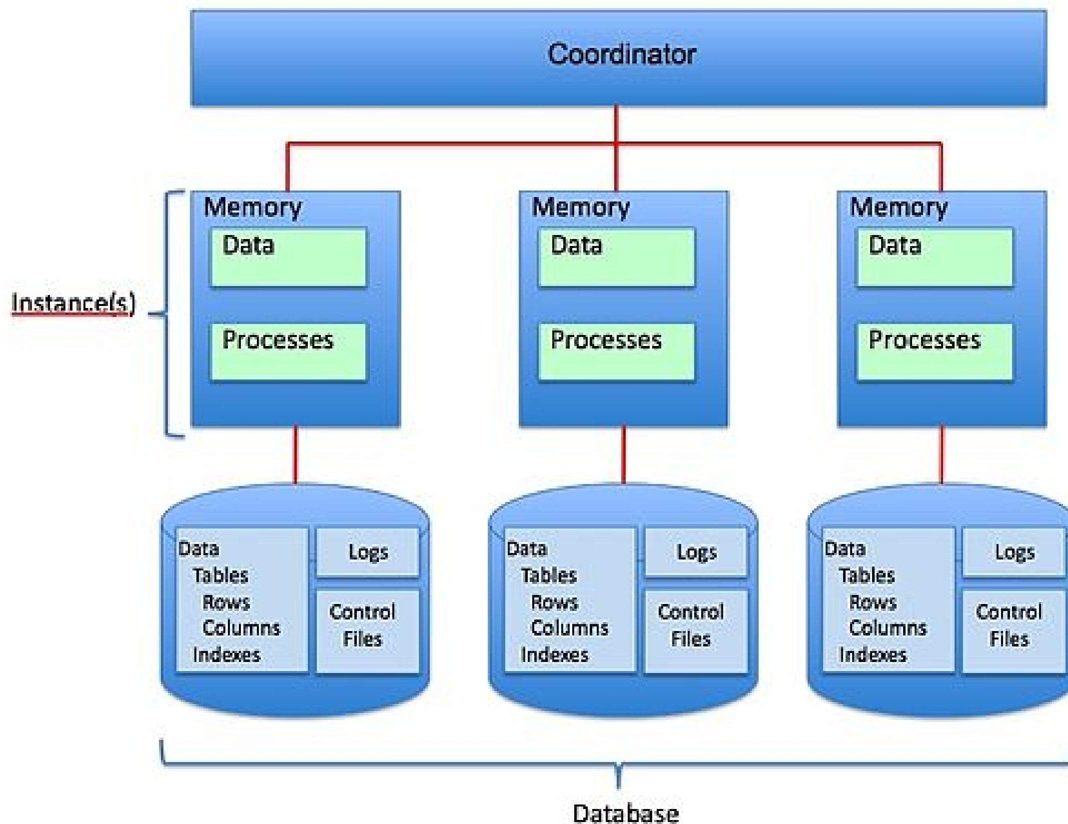

Figura 1.2: Architettura shared nothing

Le soluzioni NewSQL adottano il linguaggio SQL per le interrogazioni e garantiscono tutte le proprietà ACID - *Atomicity, Consistency, Isolation, Durability* - alle transazioni.
Un meccanismo di controllo della concorrenza non bloccante permette alle letture che avvengono contemporaneamente di non andare in conflitto con le scritture, evitando lo stallo di queste ultime. L'architettura NewSQL è *shared nothing* e consente, in questo modo, di ottenere prestazioni migliori in ogni macchina del *cluster* rispetto alle soluzioni tradizionali RDBMS.
Le caratteristiche sopraelencate dei sistemi NewSQL fanno sì che essi possano arrivare ad essere fino a 50 volte più veloci dei RDBMS tradizionali [1].

Categorizzazione approcci NewSQL

I differenti approcci adottati dai fornitori per mantenere l'interfaccia SQL, così come i problemi di prestazioni e scalabilità derivanti dalle tradizionali soluzioni OLTP, determinano una possibile classificazione degli approcci NewSQL. In [1] gli autori ne propongono una che prevede le 3 classi seguenti:

1. **Nuovi DBMS**: La considerazione chiave per il miglioramento delle performance di questi DBMS è quella di rendere la RAM o i nuovi tipi di hard disk, *flash* o SSD, la memoria primaria per il salvataggio dei dati: in questo modo si riduce il numero degli accessi a disco o il costo di tali operazioni; ricordando quanto queste siano onerose, è ovvio constatare il miglioramento. Questi sistemi sono stati recentemente creati da zero con l'obiettivo di raggiungere scalabilità e alte prestazioni. Possono essere *software-only*, fra questi si ricordano VoltDB, NuoDB e Drizzle, oppure supportate da hardware *ad hoc*, Clustrix e Translattice.

2. **New MySQL storage engines**: Lo *storage engine* è il componente software del DBMS che implementa la gestione fisica dei dati: scrittura, lettura, aggiornamento e eliminazione dei record. Una nuova gestione dei dati sostituisce quella dei motori MySQL originali mantenendone l'interfaccia: ciò viene fatto al fine di superare problemi di performance e scalabilità. Un problema non secondario, però, è l'assenza di supporto per la migrazione dei dati da molti DBMS.

3. **Clustering trasparente**: Queste soluzioni, al contrario delle altre non ridefiniscono le strutture hardware o software, ma si concentrano su un diverso uso dei DBMS OLTP allo stato dell'arte. I *cluster* possiedono catatteristiche modulari, ossia all'occorrenza possono essere collegati loro componenti aggiuntivi garandendo scalabilità in maniera del tutto trasparente: l'introduzione di un nuovo nodo viene gestita automaticamente dal DBMS.

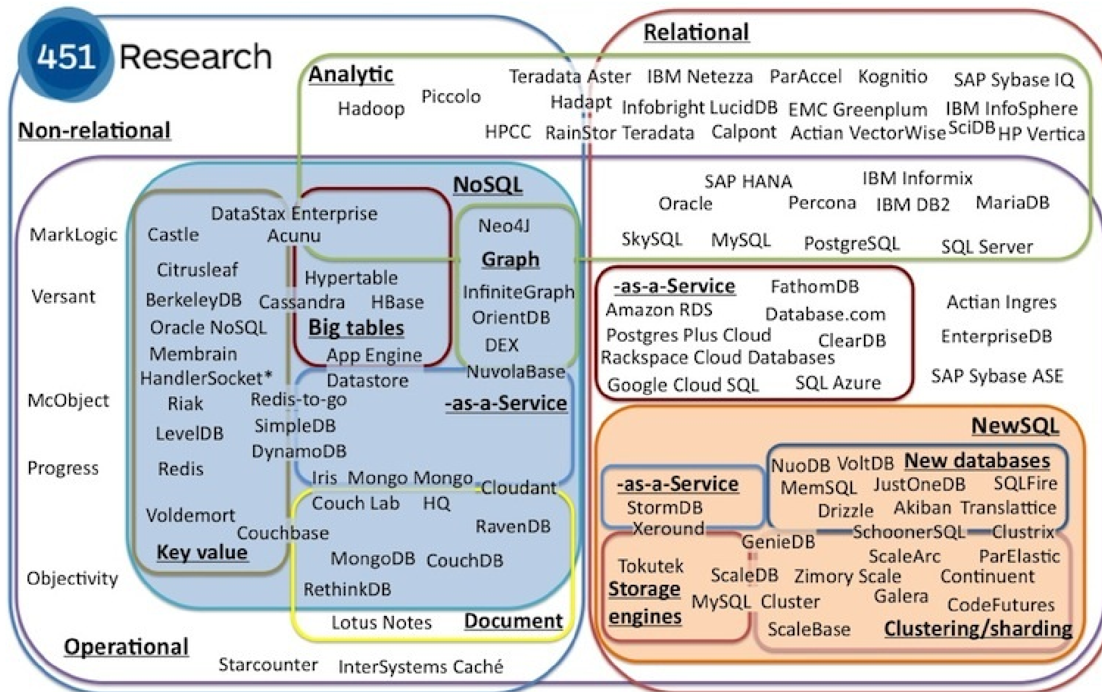

Figura 1.3: Evoluzione dello scenario dei database

1.2 NoSQL vs NewSQL

Il movimento NoSQL nasce nel 2009 conseguentemente alle necessità di gestire grandi moli di dati e scalare orizzontalmente.
NoSQL è l'acronimo di *Not only SQL* ed identifica la classe di database che non adotta un modello dati di tipo relazionale; concetti come tabella, indici, chiavi primarie o esterne non sono per forza presenti; operazioni costose come *join* tra tabelle sono spesso evitate.
Le implementazioni di NoSQL possono essere categorizzate sulla base del tipo di modello dati adottato. Quello orientato al documento, per esempio, conserva i dati in documenti, come suggerisce il nome, non in tabelle: in questo caso la differenza sostanziale rispetto ai tradizionali RDBMS sta nel fatto che le informazioni sono memorizzate aggregate per oggetto in documenti e non distribuite in diverse strutture logiche. Ogni documento aggrega tutti i dati associati ad un'entità in modo tale da poterla considerare come oggetto e valutare contemporaneamente tutte le informazioni ad esso collegate; essendo tutti i dati necessari e corrispondenti ad

un medesimo oggetto disponibili in un unico documento, si evitano come voluto le estremamente costose operazioni di *join*. Un altro esempio di implementazione NoSQL è quella a grafo. In questo caso la base di dati è rappresentata da nodi e archi per archiviare le informazioni.
L'assenza di tabelle permette ai database non relazionali di essere *schemaless*, ossia privi di un qualsiasi schema definito a priori.
Il rovescio della medaglia sta nel fatto che, per privilegiare velocità e scalabilità, occorre rinunciare in primo luogo a SQL e in secondo luogo alle proprietà ACID. Questo che cosa comporta?
Prima di tutto l'incapacità di formulare *query* altamente strutturate, con il rischio di analizzare dati in maniera incostitente, cosa che invece SQL, fondandosi su basi algebriche e di calcolo relazionale, evita. Secondariamente NoSQL, non eseguendo operazioni di tipo ACID, non garantisce che una transazione legata ad un database venga portata a termine anche in caso di interruzione del sistema.
In tutti quei casi in cui si ha a che fare con vincoli di integrità e, più in generale, ogni volta che l'accuratezza dei dati deve essere una garanzia e non una richiesta, ecco che i sistemi NoSQL non sono adatti. Volendo legare dati provenienti da archivi diversi, solo soluzioni non automatiche sono disponibili; ovvero se si memorizzano documenti in due archivi differenti l'equivalente del *join* non è gestito dal DBMS. Se si memorizza tutto in un unico documento, invece, diventa difficile correggere i dati errati o effettuare statistiche complesse. Inoltre non esistono standard consolidati: ogni database NoSQL ha il proprio linguaggio di *query*.

NewSQL, adottando il modello relazionale, supporta SQL e garantisce in questo modo le proprietà ACID, contemporaneamente cercando di mantenere la scalabilità propria dei sistemi NoSQL.

	Traditional SQL	**NoSQL**	**NewSQL**
Relational	Yes	No	Yes
SQL	Yes	No	Yes
ACID transactions	Yes	No	Yes
Horizontal scalability	No	Yes	Yes
Performance/big volume	No	Yes	Yes
Schemaless	No	Yes	No

Tabella 1.1: *Confronto tra le tipologie Traditional SQL, NoSQL, NewSQL*

1.3 In-Memory DBMS

Con il termine *in-memory database* (IMBD), che può essere tradotto in sistema di basi dati in memoria centrale (MMDB) si intende un DBMS che sfrutta la memoria principale, invece che utilizzare quelle di massa per lo *storage* dei dati.
I primi - database in memoria principale - sono molto più veloci dei secondi - database in memoria di massa - ma possono gestire quantità di dati molto inferiori; ovviamente ciò a patto che ci sia un modo per recuperarli in caso di guasti. Un IMDB può essere implementato sia con strutture utilizzanti l'approccio relazionale sia con strutture non relazionali.
In tutte quelle applicazioni che necessitano tempestività nell'accesso ai dati, un sistema *in-memory* garantisce ottime prestazioni; è il caso, per esempio, delle applicazioni *real-time* dove sono richieste performance molto elevate per quanto riguarda *throughput* e latenza.

Confronto tra MMDB e DBMS tradizionali

Con i MMDB si perde il concetto di I/O nei confronti dei dischi, poichè tutta la base di dati è caricata in memoria centrale [5].
I meccanismi di *caching* supportati dai DBMS tradizionali al fine di mantenere i *record* più utilizzati in RAM e ridurre conseguentemente gli accessi a disco, ora potranno non essere più considerati: si evita in questo modo di doversi preoccupare in primo luogo della sincronizzazione, usata per garantire la consistenza dei dati in memoria centrale con quelli in memoria secondaria, e secondariamente la

consultazione della *cache* per determinare la presenza o l'assenza del *record* richiesto. Viene in questo modo anche ridotto il carico del lavoro della CPU.
Con le loro elevate prestazioni gli MMDB rappresentano una buona soluzione per le *time-critical applications*; tuttavia non risultano più essere soluzioni adeguate in tutti quei casi in cui è importante la persistenza dei dati, dal momento che risiedono in una memoria volatile.

Capitolo 2

VoltDB

In questo capitolo verrà fornita una panoramica del RDBMS VoltDB, in un primo momento in maniera più generale per poi soffermarsi sui suoi aspetti peculiari. Verrà mostrato il progetto H-Store da cui è nato, da cui ha preso i principi fondamentali, sottolineando poi le differenze tra i due. Si illustreranno inoltre le motivazioni che hanno portato all'effettivo sviluppo di VoltDB come soluzione commerciale, così come il suo funzionamento, delinaendone i punti chiave: partizionamento, serializzazione e scalabilità. Infine si dedicherà particolare attenzione alle proprietà ACID e di come esse vengano garantite, in particolare al concetto di *Durability*. Si affiancherà a queste quattro proprietà quella di *Availability*, spiegandone l'importanza per VoltDB.

2.1 Le origini: il progetto H-Store

H-Store è la prima implementazione del sistema NewSQL teorizzato da Michael Stonebraker; si tratta di un DBMS sperimentale creato per applicazioni di tipo OLTP, capace di eseguire transazioni favorendo l'aumento del *throughput* allontanandosi dalla vecchia tecnologia legata a *System-R*, ovvero a quei primi sistemi di tipo relazionale su cui si sono basati i moderni RDBMS. H-Store è stato progettato per essere un DBMS relazionale, distribuito e orientato alle righe, eseguito su un *cluster shared-nothing*. I dati vengono partizionati in un sottoinsieme disgiunto di nodi, costituiti da calcolatori, ai cui *core* è assegnato uno e un solo motore di

esecuzione *single-threaded*. Questi ultimi hanno accesso ai soli dati delle proprie partizioni. Poichè sono implementati per mezzo di un unico *thread*, solo una transazione alla volta ha accesso ai dati memorizzati nel proprio nodo. In figura 2.1 viene mostrato uno schema di H-Store che ne descrive l'architettura.
Il *team* sviluppatore proviene dalla *Brown University*, *Carnegie Mellow University*, *Massachusetts Institute of Technology* e *Yale University*: i ricercatori che ne hanno fatto parte sono Michael Stonebraker, Sam Madden e Daniel Abadi. Alcune delle loro proposte più radicali comprendono: innanzitutto propongono l'assenza di dischi o altri dispositivi di memoria di massa da interrogare per il recupero dei dati durante le varie transazioni. In secondo luogo vogliono fare in modo che queste ultime siano svolte in maniera sequenziale e non parallela: conseguentemente non c'è nessuna necessità di impostare sistema di controllo della concorrenza [7].
Prima di lavorare su H-Store, il gruppo di lavoro specializzato nell'ambito delle basi di dati del MIT e quello della Brown collaborarono per creare un sistema orientato alle colonne [6]. C-Store era l'originale prototipo accademico per questo primo progetto di ricerca, mentre Vertica era il sistema commerciale basato su di esso; nonostante la similarità, questi due sistemi erano completamente separati e non condividevano alcun codice.
Similmente H-Store fu concepito come prototipo accademico: la sua versione originale era costituita da un sistema *single-node*, *proof-of-concept* usato nel *paper* presentato nel corso della conferenza VLDB - *Very Large Data Bases* - del 2007 [8].
Agli inizi del 2008, i ricercatori del MIT, della Brown e di Yale cominciarono a lavorare a tutti gli effetti sui sistemi H-Store. Contemporaneamente Horizontica venne lanciato come progetto segreto interno a Vertica: ancora una volta a stessi obbiettivi corrispondevano implementazioni indipendenti.
Nella primavera del 2008, venne deciso di fondere insieme i motori di esecuzione del *back-end* di H-Store con il coordinatore delle transazioni *front-end* e con il pianificatore di *query* di Horizontica. Dopo la dimostrazione avvenuta durante il VLDB nell'autunno del 2008, il progetto H-Store gettò le basi per VoltDB. I miglioramenti al codice di base originale H-Store, aggiunti dal *team* sviluppatore di VoltDB, sono stati poi importati nel repository di H-Store nell'estate del 2010: nel corso del tempo, vari componenti di VoltDB sono stati rimossi e riscritti in

H-Store al fine di soddisfare le esigenze della ricerca originale.

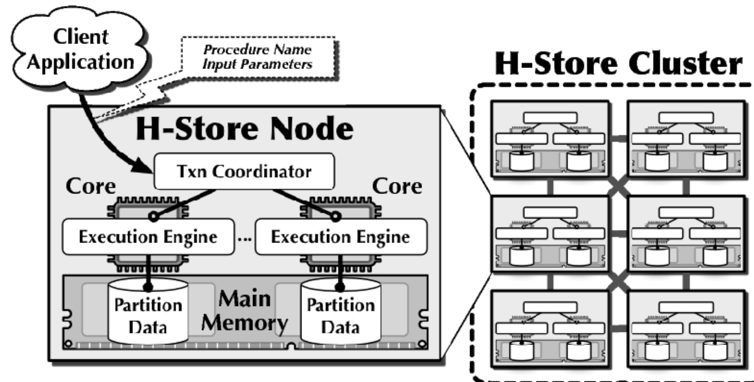

Figura 2.1: *Panoramica dell'architettura H-Store*

H-Store è distribuito sotto le licenze *BSD license* e *Gnu Public License* (GPL).

Differenze tra H-Store e VoltDB

Voltdb è stato sviluppato per aziende produttive, quindi si focalizza sulle alte performance proprio in termini di attività produttiva, ovvero di *throughput*, per transazioni su partizioni singole, fornendo anche una robusta gestione dei guasti.
H-Store non ha gli strumenti di VoltDB per mantenere e gestire un *cluster*: dal momento che H-Store è un progetto di ricerca sui database, non deve provvedere alle garanzie di sicurezza a cui deve invece provvedere VoltDB, soluzione per aziende reali; permette tuttavia il supporto per ricevere ottimizzazioni aggiuntive.
Per esempio, in VoltDB ogni transazione è o *single-partition*, cioè relativa ad una partizione singola, ossia una transazione che contiene query che hanno accesso solo ad una specifica partizione [11], o *multiple-partition*, richiedenti dati che risiedono su più partizioni. È da ricordare che qualsiasi transazione che che necessita di dati provenienti da più partizioni causerà il blocco di tutte quelle nel *cluster* ad opera del *transaction manager*, anche se la transazione avesse richiesto dati risiedenti in sole due partizioni.
Nell'ultima versione di H-Store è stato rimosso il sopracitato coordinatore interno di VoltDB; al suo posto è stato introdotto un *framework* più generico che supporta un numero arbitrario di transazioni. Per ogni *query* di tipo *batch* effettuata

da una transazione, l'*H-Store batch planner*, fortemente ottimizzato, determina quali partizioni servano a tali interrogazioni, inviando solo a quelle individuate la richiesta.

2.2 Cos'è VoltDB

VoltDB è un rivoluzionario prodotto DBMS, creato per essere la miglior soluzione per raggiungere alte performance nelle applicazioni più critiche per il business; l'architettura di VoltDB è tale per cui riesce ad archiviare dati fino a 45 volte più velocemente dei prodotti DBMS odierni. Questa architettura permette, inoltre, di scalare facilmente aggiungendo processori al *cluster* [10].
Lo scopo di VoltDB è quello di computare molto più velocemente di quanto non facciano i DBMS tradizionali concentrandosi su certe classi di applicazioni: queste includono il tradizionale *transaction processing* (TP) che esiste da anni; alcune recenti applicazioni TP includono una varietà di casi non-tradizionali, come ad esempio mantenere lo stato di un gioco su internet, oppure il posizionamento degli annunci in tempo reale nelle pagine web. Ancora, si possono trovare applicazioni che controllano e gestiscono il commercio elettronico e varie applicazioni di telecomunicazioni [12]: cioè tutte quelle soluzioni che necessitano di grande *throughput*.
Sebbene l'architettura di base dei DBMS non sia cambiata significativamente negli ultimi 30 anni, la computazione lo è, nella misura in cui sono cambiate la domanda e le aspettative riguardo ai software di business e le aziende che dipendono da essi.
VoltDB è creato per soddisfare le necessità che la computazione moderna richiede: utilizza come memoria principale quella centrale aumentando il *throughput* ed evitando i dispendiosi accessi a disco; serializza gli accessi ai dati evitando tutte le funzioni *time-consuming* tipiche dei DBMS tradizionali come il bloccaggio a livello di tabella; raggiunge scalabilità, affidabilità ed alta disponibilità grazie a *cluster* di computer e alla replicazione dei dati su più *server* o *server-farm*.
VoltDB è un DBMS transazionale che garantisce le proprietà ACID, sollevando gli sviluppatori dall'implementare propri frammenti di codice che eseguano transazioni e gestiscano *rollback*. Utilizzando un sottoinsieme del linguaggio ANSI standard SQL, VoltDB riduce la curva di apprendimento degli utenti che lo adottano come soluzione commerciale per i propri bisogni.

2.2.1 Come Funziona VoltDB

VoltDB non crea un generico unico database, ma un database distribuito dove ogni componente è ottimizzata per una specifica applicazione; la specifica istanza è ottenuta compilando uno schema, avendo scritto *stored procedure* e partizionando le informazioni in quello che viene chiamato *application catalog*. Quest'ultimo viene poi caricato su più macchine per creare il database distribuito finale; è da ricordare che, all'estremo, si potrebbe installare un *cluster* mononodo, costituito cioè da un unica macchina.

Partizionamento

Ogni *stored procedure* è definita come una transazione. Essa può andare a buon fine oppure fallire nel complesso, il tutto per mantenere il database in uno stato consistente.
Analizzando e precompilando la logica di accesso ai dati nelle *stored procedure*, VoltDB può distribuire non solo i dati, ma anche i processi associati ad essi ai singoli nodi nel *cluster*. In questo modo ogni nodo contiene una parte di dati e processi del database. Il partizionamento è fatto automaticamente da VoltDB sulla base della colonna indicata dal progettista. Occorre fare questa scelta con cautela poichè il miglior partizionamento non è sempre il più ovvio: per ottenere le migliori performance è necessario ricordare sempre che VoltDB partiziona oltre ai dati anche il lavoro su di essi facendo in modo che le operazioni più comuni possano essere eseguite in parallelo. Le caratteristiche hardware delle macchine obbligano a parallelizare il carico di lavoro per ottenere alte prestazioni. Ci sono due modi per realizzare tale parallelismo: un primo, un *multi-threading* con *lock* e memoria condivisa, che favorisce una tipologia di lavoro principalmente in lettura, in quanto le operazioni che verrebbero svolte concorrentemente, ma richiedono gli stessi dati in scrittura obbligano ad una serializzazione; un secondo dove si ha una divisione dei dati in partizioni logiche e determinate sulla base di quali transazioni li accedono. Se due operazioni richiedono lo stesso dato logico, devono essere serializzate, ma molte strutture come gli indici possono essere divise tra le partizioni per migliorare il *throughtput* complessivo e aumentare le *performance* in scrittura.

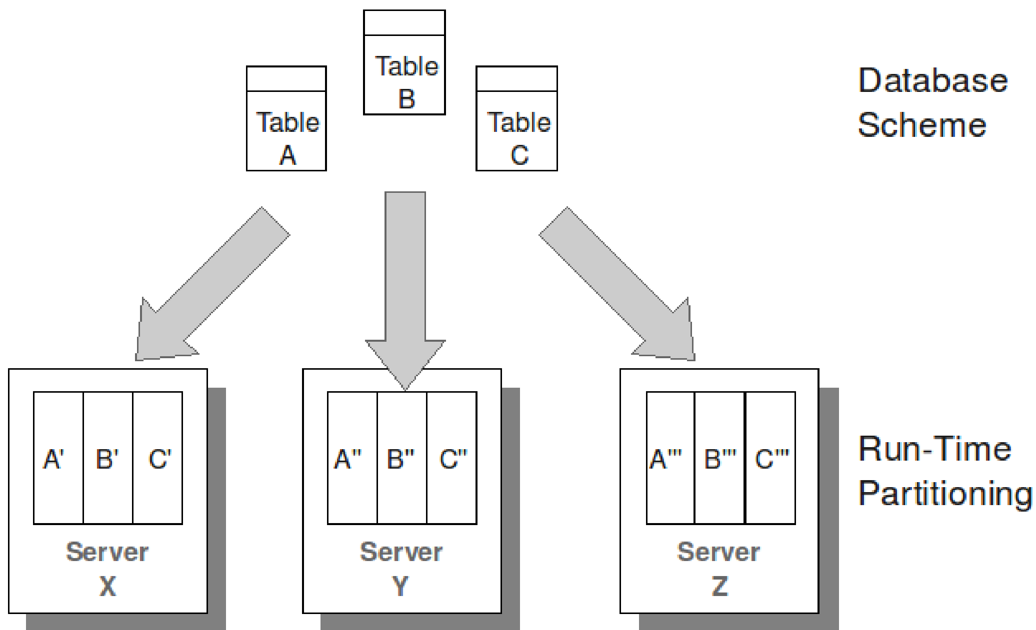

Figura 2.2: Partizionamento

Il sistema considera membri singoli del *cluster* i *core* di una CPU, assegnando ad ognuno di essi una partizione logica. La comunicazione tra le partizioni è astratta in modo tale da funzionare allo stesso modo sia in tutta la rete che attraverso il *bus* locale. Rimuovendo la concorrenza del *multi-threading* e creando code ordinate di lavoro anzichè strozzature, VoltDB riesce migliorare le prestazioni di uno o due ordini di grandezza rispetto ad altri DBMS con la stessa architettura [14].
Per raggiungere un'ottimizzazione ulteriore delle performance, VoltDB permette la replica di alcune tabelle su tutte le partizioni del *cluster*. Tali tabelle devono essere piccole e accedute prevalentemente in modalità *read-only*; questo permette alle *stored procedure* di effettuare *join* tra queste e altre di grandi dimensioni rimanendo comunque transazioni *single-partition*.

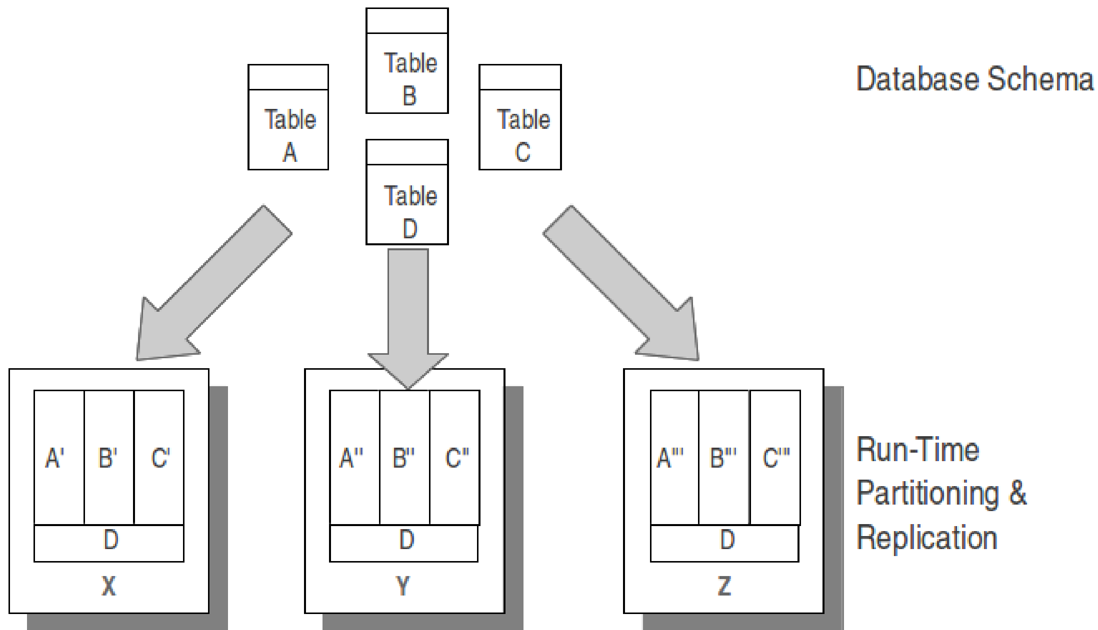

Figura 2.3: Partizionamento e Replicazione

Quando si pianifica il partizionamento dello schema, ci sono alcune linee guida che è bene seguire: prima di tutto occorre chiedersi quali *query* saranno critiche e quali saranno le più frequenti e, successivamente, per ognuna di queste, quali tabelle verranno accedute e, più precisamente, quali colonne; infine quali di queste tabelle potrebbero essere buone candidate per la replicazione.

Serializzazione

A *runtime*, le chiamate alle *stored procedure* vengono inoltrate al nodo appropriato del *cluster* dal coordinatore. Per le transazioni *single-threaded* il nodo singolo esegue la transazione indipendentemente, permettendo al resto del *cluster* di rispondere ad altre richieste in parallelo. Utilizzando un processo serializzato, VoltDB assicura la consistenza delle transazioni senza essere afflitto dall'*overhead* relativo al bloccaggio e alla gestione del *transaction log*, mentre il partizionamento del *cluster*, come osservato sopra, permette il soddisfacimento di più richieste in parallelo. Più semplicemente si può definire la seguente regola generale: più processori, quindi più partizioni sono nel *cluster*, maggiore è il numero di tran-

sazioni che VoltDB può completare al secondo. Quando una procedura richiede dati da più partizioni, un nodo funge da coordinatore e distribuisce il lavoro agli altri. I risultati vengono poi raccolti completando così il compito. Il coordinatore eletto esegue transazioni *multiple-partitioned* generalmente più lentamente di quelle *single-partitioned*. In ogni caso viene comunque mantenuta l'integrità della transazione (o fallisce o ha successo nel suo complesso) e l'architettura a partizioni multiple assicura che il *throughput* venga mantenuto al massimo. È importante no-

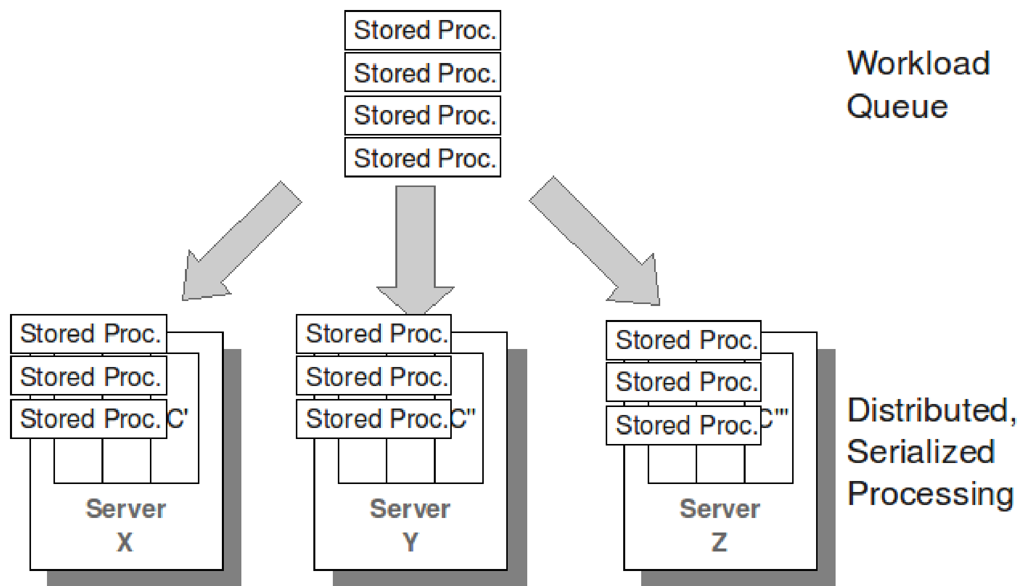

Figura 2.4: Processo Serializzato

tare che l'architettura di VoltDB è ottimizzata per quanto riguarda il *throughput*: il numero di transazioni completate al secondo è maggiore in ordine di grandezza a quello dei prodotti DBMS tradizionali, poichè VoltDB riduce il tempo in cui le transazioni restano in attesa di essere eseguite. La latenza di ogni transazione ossia il tempo che intercorre tra quando questa comincia a quando termina, invece, non si discosta da quello di altri DBMS.

Scalabilità

L'intuizione avuta per rendere efficiente il processo di scalabilità di un sistema è stata quella di spostare il database in memoria centrale, poichè nei sistemi RDBMS tradizionale l'accesso al disco è la maggior fonte di *overhead* [13]. In generale, in tali sistemi, si possono distinguere 5 fonti principali di *overhead*:

1. **Gestione degli indici**: *B-tree*, *hash table* e altri schemi di indicizzazione sono alquanto dispendiosi in termini di CPU e I/O.

2. **Write-Ahead Logging (WAL)**: i DBMS tradizionali scrivono tutto due volte: una volta nel DBMS e una volta nel *log file*. La scrittura sul *log* è forzata dal disco per garantire la *durability* e questa risulta essere un'operazione molto onerosa.

3. **Locking**: prima di prendere dati da un *record*, la transazione deve assicurarsi di aver bloccato la tabella in cui si trova il dato richiesto. L'operazione di bloccaggio della tabella è un'operazione particolarmente *overhead-intensive*.

4. **Latching**: gli aggiornamenti delle strutture condivise devono essere fatti con cautela, tipicamente con bloccaggi di breve durata.

5. **Gestione del buffer**: tradizionalmente i dati sono salvati su disco in pagine di dimensione fissata. Un *buffer-pool* gestisce tali pagine tenendone alcune in *cache*.

Creato originariamente per garantire l'integrità dei dati, questo *overhead* impedisce ai tradizionali DBMS di scalare in maniera tale da gestire contemporaneamente volume dei dati e carico di lavoro.

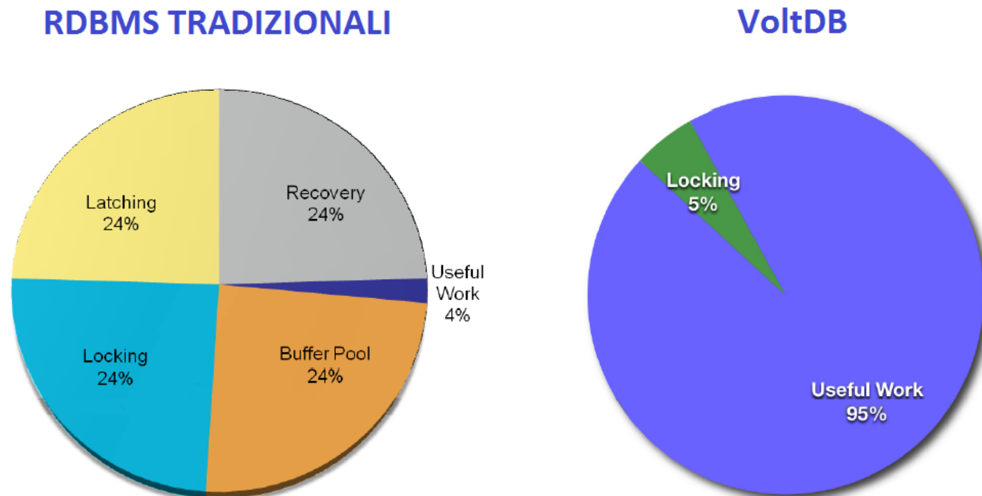

Figura 2.5: *Gestione delle risorse: RDBMS tradizionale vs VoltDB*

L'architettura di VoltDB è creata in maniera da semplificare il processo di *scaling* per poter soddisfare le esigenze delle applicazioni sviluppate. Incrementando il numero di nodi nel *cluster*, incrementa sia il *throughput* conseguentemente all'incremento del numero delle operazioni simultanee in esercizio, sia la capacità di memorizzare dati, conseguentemente all'incremento del numero di partizioni usate per ogni tabella. I dati sono mantenuti in memoria centrale e non su memoria di massa per eliminare l'utilizzo del *buffer manager*; ogni partizione *single-threaded* opera autonomamente eliminando l'uso di *locking* e *latching*. Scalare un database diventa un processo facile, poichè non viene richiesto di cambiare lo schema o il codice dell'applicazione; inoltre si possono aggiungere nodi *on the fly* mentre il DBMS è in esecuzione.

2.3 Le proprietà ACID

Come detto in precedenza, VoltDB nasce come implementazione del progetto H-Store, che, essedo rappresentativo dei sistemi NewSQL, deve garantire le proprietà ACID in contrapposizione a quelli NoSQL, che non le rispettano.

2.3.1 Atomicity, Consistency ed Isolation

Con *Atomicity* si intende che una transazione debba essere un'unità di elaborazione; questo significa che o tutti gli effetti di quest'ultima sono registrati nel DB, o non ne viene nessuno. VoltDB garantisce *Atomicity* in quanto ogni *stored procedure* è considerata una transazione a sè stante, come già affermato.
La proprietà di *Consistency* fa sì che una transazione lasci il DB in uno stato, come il nome suggerisce, consistente, cioè che nessuno dei vincoli venga violato. La consistenza del database, invece, è mantenuta grazie al completo successo o completo fallimento della transazione nel suo complesso.
Da ultimo, *Isolation* è la proprietà per cui ogni transazione viene eseguita in maniera indipendente dalle altre. L'assenza di concorrenza nell'implementazione di VoltDB fa sì che ciò sia sempre verificato.

2.3.2 Durability

Durability è l'ultima delle quattro proprità ACID richieste per garantire accuratezza e affidibilità delle transazioni; essa si riferisce alla capacità di mantenere il database in uno stato consistente e disponibile nonstante possibili errori *software* e/o guasti *hardware*. Gli effetti di una transazione andata a buon fine devono essere persistenti nel tempo; al contrario, transazioni che falliscono o vengono interrotte da *roll-back* non devono influire sull'integrità del database.
Quando una transazione va a buon fine e le modifiche al database vengono apportate, VoltDB ha diversi modi per garantire che i cambiamenti siano permanenti. Se il *server* si spegne per qualsiasi ragione, il database deve essere ripristinato per essere riportato ad uno stato consistente. In base al livello di *durability* che si vuole raggiungere, sono disponibili diverse possibilità.

Snapshot e Command Logging

La prima fra esse è lo *Snapshot* del *Database*, che fornisce una *durability* di livello base. Per *snapshot* si intende una copia su disco del contenuto del database in un dato momento. Lo *snapshot* può essere effettuato manualmente, ma è meglio abilitarne un'esecuzione automatica in fase di avvio del DBMS. Si può configurare la frequenza di tali salvataggi, come parte delle impostazioni di *deployment*; se il *server* dovesse interrompere il proprio funzionamento per una qualsiasi ragione, si dovrebbe usare lo *snapshot* più aggiornato per effettuare *recovery* del sistema prima del *crash*.
Se si ha bisogno di un livello di *durability* maggiore, si può utilizzare il *Command Logging*, ovvero mantenere uno o più file in cui viene registrato un *record* per ogni transazione eseguita con successo. Concetto chiave di tale strategia è quello di mantenere le invocazioni delle transazioni e non le sue conseguenze. Ogni singola *stored procedure* può includere più interrogazioni SQL, che a loro volta possono modificare centinaia, se non addirittura migliaia, di righe. Registrando solo le invocazioni, il *command logs* mantiene al minimo l'impatto che i costi di I/O su disco hanno sulle performance. Usare questa tecnica può ridurre il numero di transazioni perse; soltanto le transazioni non scritte sul *log* al momento del guasto non possono essere recuperate. Esistono due tipologie di implementazione di questa strategia: una asincrona ed una sincrona. La tecnica dell'*Asynchronous Command Logging* provvede a migliorare la durabilità fornendo sia *snapshot* che un *log* di tutte le transazioni avvenute tra uno *snapshot* e l'altro. In questo modo, se un *server* smettesse di funzionare e il DBMS venisse riavviato, non soltanto VoltDB recupererebbe lo stato dell'ultima istantanea, ma potrebbe anche rieffettuare le transazioni presenti nel *log* stesso.

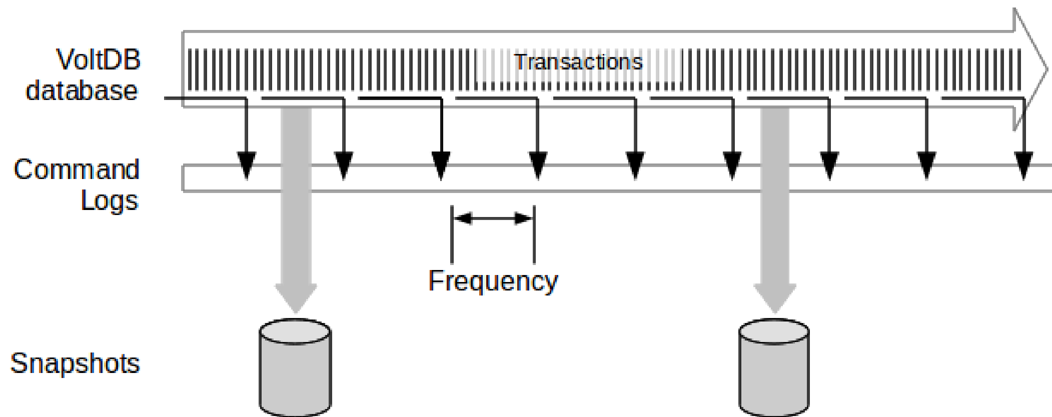

Figura 2.6: Command Logging

La possibilità più completa che garantisce *durability* è il *Synchronous Command Logging* che, come il precedente, tiene traccia di tutte le transazioni fra uno *snapshot* e il seguente ed inoltre aggiorna il *log* dopo il completamento di una transazione, ma prima che venga effettuato il *commit* per essa: in questo modo non esistono transazioni confermate da un *commit* che non siano registrate sul *log*. Unico svantaggio è che sono necessarie le tecnologie più avanzate per quel che riguarda i dischi per poter stare al passo dell'alto livello di *throughput* raggiunto da VoltDB.

2.3.3 Availability

Se da una parte la *durability* assicura persistenza dei dati anche in caso di fallimento, l'*availability* è la capacità di resistere ai guasti di sistema che andrebbero ad intaccare la capacità di funzionamento del DBMS [16]. VoltDB fornisce *K-Safety* e *Network Partition Detection* per il controllo di errori locali nell'hardware; la replicazione dell'intero database (*DataBase Replication*) è usata come protezione contro interruzioni di gravità ed entità maggiori.

K-Safety

La *K-Safety* prevede di duplicare le partizioni in modo tale che se una partizione viene persa o corrotta in qualche modo, il database può continuare comunque a lavorare con le partizioni duplicate. Tutte le copie delle partizioni operano simultaneamente garantendo la consistenza dei dati in maniera continuativa. È possibile configurare il valore parametrico K della proprietà impostandolo al numero dei nodi che il *cluster* può perdere senza compromettere la propria capacità di funzionamento. Come mostrato in figura 2.7 le operazioni procedono ugualmente

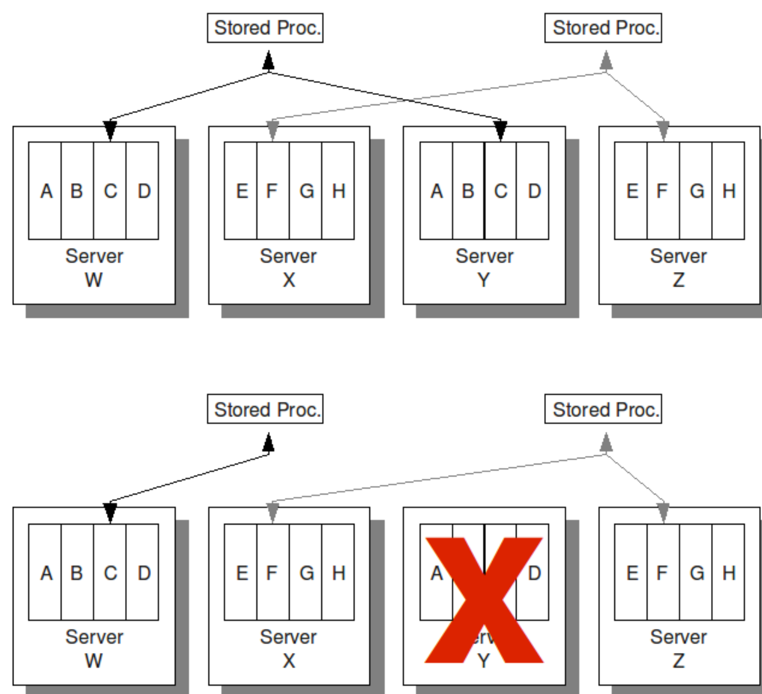

Figura 2.7: K-Safety con K=1

inviando il lavoro ai nodi non coinvolti dal fallimento.

Network Partition Detection

La *Network Partition Detection* lavora insieme alla *K-Safety*; se si presenta un problema di connessione tra i nodi, il *cluster* individua un fallimento di un nodo. È possibile che una o più macchine tentino di continuare separatamente i loro compiti, senza comunicare fra loro: la *Network Partition Detection* assicura che ciò non possa accadere, facendo in modo che solo uno fra le parti del *cluster* sopravviva.

DB Replication

La *DB-Replication* (DR) ha un funzionamento simile a quello della *K-Safety*; piuttosto che replicare partizioni localmente, si replica l'intero *cluster* in remoto. Uno degli usi principali della DR è quello di fornire *disaster recovery.* La DR minimizza il *downtime*, cioè il tempo in cui il *cluster* risulta inattivo, fornendo una replica oppure un sostituto a caldo del DB originale.

Capitolo 3

Caso di studio

In questo capitolo verrà trattata un'analisi prestazionale del software VoltDB su un *cluster* di 6 macchine. Sarà esplorato lo schema logico della base di dati scelta specificando per ciascuna relazione la chiave primaria e relativo partizionamento necessario al corretto funzionamento della stessa. Verrà poi fatta una panoramica sulle componenti software e hardware del *cluster* arrivando infine a presentare *query* contenute nel *benchmark* TPC-H e relativi risultati raggiunti.

3.1 Schema Logico del Database

Il *Transaction Processing Performance Council* (TPC) è un organismo internazionale che progetta *benchmark* standard e ne omologa i risultati. I *benchmark* del gruppo TPC sono ritenuti i più importanti e significativi per la valutazione delle prestazioni dei sistemi DBMS [17]. Per l'analisi prestazionale del DBMS VoltDB, si è scelto di utilizzare il database TPC-H composto da otto relazioni illustrate in figura 3.1.

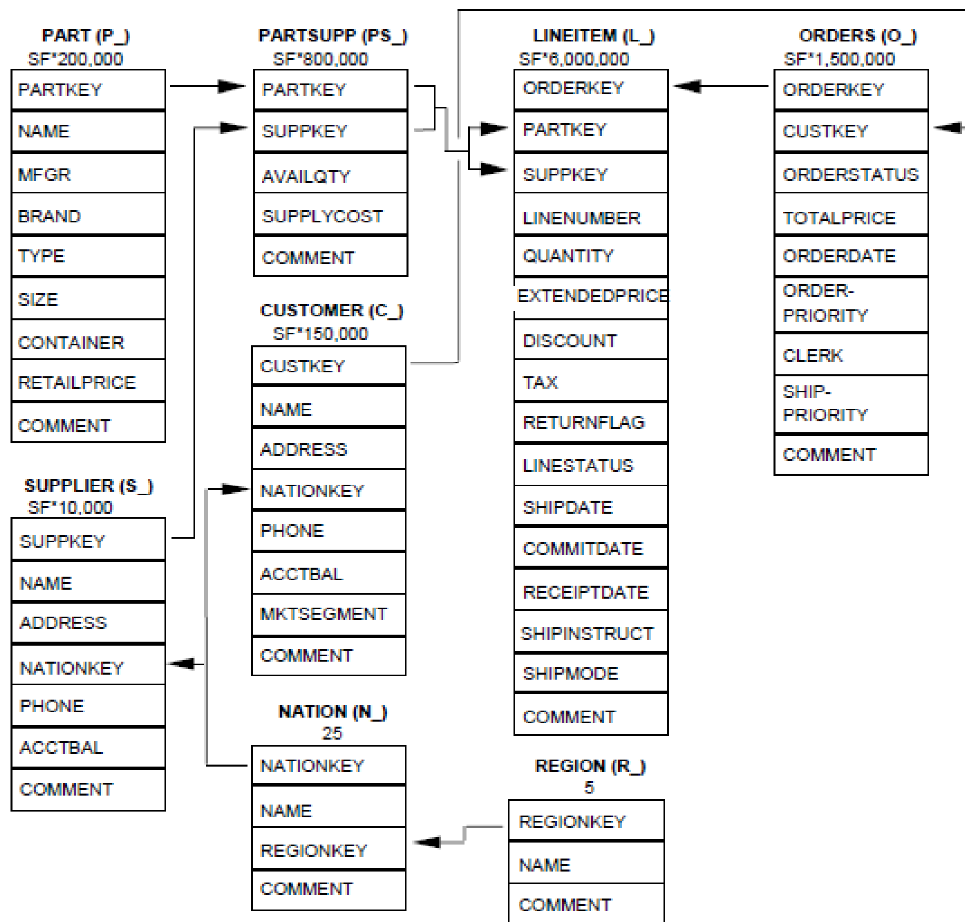

Figura 3.1: Schema TPC-H

Quanto contenuto tra parentesi accanto al nome di ciascuna tabella indica quale sia il prefisso univoco per individuare ogni attributo della specifica relazione in fase di interrogazione. Il numero sotto al nome indica il numero di *record* contenuti nella stessa: SF, invece, indica il fattore di scala che permette di scegliere la dimensione effettiva del database. La punta delle frecce indica la tabelle per cui l'associazione ha cardinalità uno-a-molti. TPC-H non rapprensenta ogni possibile modello di business, ma in generale quello delle imprese che hanno necessità di gestire ordini o vendita di prodotti in tutto il mondo.

Definizione dei vincoli

Le chiavi primarie delle tabelle sono elencate di seguito:

- P_PARTKEY per la tabella PART;
- S_SUPPKEY per la tabella SUPPLIER;
- C_CUSTKEY per la tabella CUSTOMER;
- O_ORDERKEY per la tabella ORDERS;
- N_NATIONKEY per la tabella NATION;
- R_REGIONKEY per la tabella REGION;
- PS_PARTKEY, PS_SUPPKEY per la tabella PARTSUPP;
- L_ORDERKEY, L_LINENUMBER per la tabella LINEITEM;

Il tipo di dato delle chiavi primarie è BIG INT per una questione puramente prestazionale: le chiavi primarie, infatti, sono i campi più interrogati; scegliere una chiave avente un tipo di dato diverso, per esempio VARCHAR, risulterebbe maggiormente dispendioso. A titolo esemplificativo, traducendo in linguaggio SQL, la definizione parziale delle relazioni e dei singoli campi da esse posseduti risulta essere la seguente:

```
CREATE TABLE PART (
        P_PARTKEY          PRIMARY KEY,
        P_NAME             VARCHAR(55),
        P_MFGR             CHAR(25),
        P_BRAND            CHAR(10),
        P_TYPE             VARCHAR(25),
        P_SIZE             INTEGER,
        P_CONTAINER        CHAR(10),
        P_RETAILPRICE      DECIMAL,
        P_COMMENT          VARCHAR(23)
);
```

Per la consultazione dello schema completo tradotto si faccia riferimento all'appendice A. Come accennato nel Capitolo 2, lo schema scritto in formato .sql viene compilato in un *catalog*, ossia in un .jar. È importante sottolineare che il *catalog* dello schema deve essere distribuito su tutti i nodi del *cluster* per poter permettere il corretto funzionamento del database.

Descrizione del *Benchmark*

Il *benchmark* TPC-H comprende una serie di *query ad-hoc* relative al business. Le *query* e i dati del database sono state scelte per essere rilevanti ma comunque mantenendo un sufficiente grado di semplicità di implementazione. Va detto che alcune delle interrogazini sono più complesse dello standard OLTP, potendo così, il linea generale, ottenere un *upperbound*.
Di seguito sono riportate le interrogazioni effettuate come test:

- **Q1**: Questa *query* riporta l'ammontare di quanto fatturato, spedito e restituito.
- **Q2**: Questa *query* quantifica l'incremento delle entrate che sarebbero risultate dalla non applicazione di sconti a livello aziendale in un dato anno e intervallo percentuale.
- **Q3**: Vengono trovati i fornitori che distribuiscono pezzi dati determinati attributi. Un suo uso tipico è determinare se vi sia un numero sufficiente di fornitori per pezzi fortemente ordinati.
- **Q4**: Sono determinati sia quanto bene stia lavorando il sistema di priorità degli ordini che una valutazione della soddisfazione dei clienti.
- **Q5**: Sono selezionati gli ordini non ancora spediti elencati in maniera decrescente rispetto al profitto.
- **Q6**: Vengono identificati i clienti che probabilmente hanno avuto problemi con la merce a loro spedita.
- **Q7**: Viene restituito, dati un continente e un particolare prodotto con relativi formati, i fornitori che lo vendono a prezzo superiore a quello medio.

- **Q8**: In una data nazione sono selezionate le scorte di merce dei fornitori.
- **Q9**: Si determina in media ogni anno quanto guadagno verrebbe perso se gli ordini non fossero molto abbondanti o non fossero relativi a molti prodotti. Questo calcolo potrebbe aiutare a ridurre le spese, concentrando le vendite sulle spedizioni di grandi dimensioni.

3.2 Descrizione del Cluster

Il *cluster* su cui sono stati fatti i test è composto da 6 macchine. VoltDB è supportato solo da sistemi operativi *Linux-based* a 64-bit o *Macintosh OS X*; questo è il motivo che ha portato a scegliere come sistema operativo da installare sulle macchine *Ubuntu GNOME 14.04*. Si ricorda che VoltDB sfrutta l'architettura *shared-nothing*, che necessita di un coordinatore all'interno del *cluster* per poter gestirne il funzionamento globale. Tale compito è svolto da un pc avente le seguenti caratteristiche:

- CPU: Intel(R) Core(TM) i5 CPU M450 @ 2.40 GHz
- RAM: 4 GB
- Hard Disk: 200 GB

I rimanenti 5 computer utilizzati hanno le stesse caratteristiche tecniche:

- CPU: Intel(R) Pentium(R) D CPU 3.40GHz
- RAM: 4 GB
- Hard Disk: 50 GB

Tutti i pc sono collegati in una LAN Ethernet cablata, fisicamente installata nello stesso locale.
Il *cluster* opera logicamente come un'unica entità e per fare ciò occorre che tutte le macchine siano sincronizzate in maniera tale da avere lo stesso orario. Per il corretto funzionamento dell'intero sistema, quindi, VoltDB consiglia di configurare tutti i nodi ad uno stesso *server NTP* (*Network Time Protocol*) locale. Quello che è

stato fatto in pratica in questo caso è sincronizzare il nodo *master* al *server NTP* e successivamente sincronizzare le macchine al coordinatore tramite il servizio NTP.

3.3 Risultati

Per ogni *query* vengono riportati esempi di *output* composti da dati e relativi campi selezionati. Si veda l'Appendice A per le specificità di ogni *query*. Le interrogazioni sono state effettuate su database generati con SF = 4 per le *query* Q1, Q2, Q3; per le restanti si è utilizzato SF = 2.

Q1

Output:

FLAG	**STATUS**	**ΣQNT**	**ΣBASE_PRICE**	**ΣDISC_PRICE**
A	F	151026307	226509091035.52	215184111589.91
N	F	3931226	5887988103.96	5593393984.52
N	O	305935367	458818602903.51	435877453640.93
R	F	151132549	226643425460.80	435877453640.93
μDISC	**μQNT**	**μPRICE**	**ΣCHARGE**	**#ORDER**
0.05	25.50	38249.95	223794237456.68	5921814
0.05	25.54	38250.08	5817216592.91	153934
0.05	25.50	38244.17	453320490008.99	11997087
0.05	25.51	38260.00	223924615047.43	5923769

Tabella 3.1: *Risultati relativi a Q1*

Si fa notare che, nelle tabelle strutturate in questo modo, le righe nella parte superiore continuano nella parte inferiore, solamente per motivi di spazio.

Legenda:

- Σ - *indica la somma rispetto all'attributo. Operazione SQL: SUM().*
- μ - *indica la media rispetto all'attributo. Operazione SQL: AVG().*
- # - *indica un conteggio rispetto all'attributo. Operazione SQL: COUNT().*

Restituite 4 righe in 9,83 secondi.

Q2
Output:

REVENUE
492844188.65

Tabella 3.2: *Risultati relativi a Q2*

Restituita 1 riga in 1,04 secondi.

Q3
Output parziale:

P_BRAND	P_TYPE	P_SIZE	SUPPLIER_CNT
Brand#55	STANDARD POLISHED TIN	36	4
Brand#14	LARGE POLISHED TIN	3	3
Brand#21	SMALL PLATED BRASS	3	3
Brand#22	MEDIUM BURNISHED STEEL	19	3

Tabella 3.3: *Risultati relativi a Q3*

Restituite 28649 righe in 4,94 secondi.

Q4
Output:

O_ORDERPRIORITY	ORDER_COUNT
1-URGENT	23211
2-HIGH	23025
3-MEDIUM	22952
4-NOT SPECIFIED	23075
5-LOW	23150

Tabella 3.4: *Risultati relativi a Q4*

Restituite 5 righe in 0,82 secondi.

Q5

Output parziale:

L_ORDERKEY	REVENUE	O_ORDERDATE	O_SHIPPRIORITY
1668613	939.2019	1994-12-30	0
11104773	933.4764	1995-02-15	0
6167268	900.000	1995-01-19	0
3283971	890.1450	1994-12-26	0

Tabella 3.5: *Risultati relativi a Q5*

Restituite 23407 righe in 1,73 secondi.

Q6

Output parziale:

C_CUSTKEY	C_NAME	REVENUE	C_ACCTBAL
175576	Customer#000175576	904.0251	4834.31
195941	Customer#000195941	901.2165	3144.31
296530	Customer#000296530	885.4755	3757.20
19274	Customer#000019274	872.1600	1793.62
N_NAME	**C_ADDRESS**	**C_PHONE**	**C_COMMENT**
KENYA	aWsvAJ8N BAL...	24-169-579-5785	e slyly above...
INDONESIA	6MW680HnUkho...	19-920-101-7118	gly regular frets...
FRANCE	K76eSEQoEmR	16-977-821-3475	ar deposits. furiously...
CHINA	0hRUUwaFp6l...	28-942-642-4112	s integrate slyly...

Tabella 3.6: *Risultati relativi a Q6*

Restituite 76128 righe in 3,22 secondi.

Q7

Output parziale:

S_ACCTBAL	S_NAME	N_NAME	P_PARTKEY
-925.51	Supplier#000007980	RUSSIA	362961
-929.81	Supplier#000019301	ROMANIA	274287
-929.81	Supplier#000019301	ROMANIA	389262
-963.79	Supplier#000000065	RUSSIA	235053
P_MFGR	**S_ADDRESS**	**S_PHONE**	**S_COMMENT**
Manufacturer#2	xTPaeTWz5YaE0w,MXQ8	32-838-254-1780	counts wake...
Manufacturer#5	2Bp12MkeDYI7I9e	29-339-109-8638	equests use...
Manufacturer#3	2Bp12MkeDYI7I9e	29-339-109-8638	equests use...
Manufacturer#2	BsAnHUmSFArppKrM	32-444-835-2434	l ideas wake...

Tabella 3.7: *Risultati relativi a Q4*

Restituite 622 righe in 3,03 secondi.

Q8

Output parziale:

PS_PARTKEY	VALUE
35615	16459995.21
259008	16389836.01
235521	16094066.50
109933	16024871.36

Tabella 3.8: *Risultati relativi a Q8*

Restituite 10 righe in 1,30 secondi.

Q9
Output:

AVG_YEARLY
786095.20

Tabella 3.9: *Risultati relativi a Q9*

Restituita 1 riga in 2.68 secondi.

3.3.1 Confronti tra MySQL e VoltDB

Si vogliono ora confrontare le prestazioni di VoltDB con quelle dell'RDBMS MySQL. Questo verrà realizzato prendendo in esame due diverse tipologie di analisi: la prima che confronta i tempi di esecuzione per singola *query* [18]; la seconda che considera il numero di transazioni effettuate durante l'esecuzione di interrogazioni gestite parallelamente [19].
Il *benchmark* utilizzato è un test che valuta le prestazioni di sistemi OLTP, simulando l'inserimento di ordini di un'azienda avente più magazzini da gestire. Le transazioni includono l'inserimento, il controllo dello stato e la consegna degli ordini, la registrazione dei pagamenti e il monitoraggio del livello delle scorte presso i magazzini.

I risultati proposti per quanto riguarda la prima analisi sono stati restituiti da MySQL interrogando un database memorizzato su una sola macchina avente le stesse caratteristiche del nodo *master* utilizzato sul cluster per l'analisi di VoltDB. La dimensione della base di dati di MySQL è 1 GB.

	VoltDB	**MySQL**
Q1	9.83	31.08
Q2	1.04	5.31
Q3	4.94	21.83
Q4	0.82	3.03
Q5	1.73	228.40
Q6	3.22	27.45
Q7	3.03	330.50
Q8	1.30	1.11
Q9	2.68	4.86

Tabella 3.10: *Tempi di esecuzione delle query espressi in secondi*

Nella seconda analisi, invece, è stato riportato il numero di transazioni totali effettuate al secondo durante un periodo di esecuzione continua. Le interrogazioni sono state eseguite continuativamente e in parallelo. Le interrogazioni riguardano l'inserimento di nuovi ordini effettuati da un certo numero di clienti diversi, comprendenti un certo numero di magazzini.
Vengono mostrati i risultati ottenuti da MySQL, seguiti poi da quelli ottenuti da VoltDB.

Nodi	**Magazzini**	**Clienti**	**Transazioni al secondo**
1	1	3	232
1	5	5	208
1	3	3	816

Tabella 3.11: *Risultati conseguiti da MySQL*

Nodi	Magazzini	Clienti	Transazioni al secondo
1	12	1	53000
3	36	2	150000
5	60	3	250000
6	72	3	300000
12	144	5	560000

Tabella 3.12: *Risultati conseguiti da VoltDB*

Il numero di transazioni totali effettuate al secondo in Tabella 3.11 è stato ottenuto in oltre 2 minuti di esecuzione continuata; quello in Tabella 3.12 è una media calcolata in oltre 3 minuti di esecuzione continuata.

Dai risultati ottenuti si evince che il punto di forza di VoltDB non è la velocità con cui esegue le *query*, ma il fatto che è in grado di aumentare il *throughput* all'aumentare del numero di *core* presenti all'interno del *cluster*. Le sue performance, infatti, scalano lentamente per *query* eseguite singolarmente, ma quando sono effettuate interrogazioni multiple in parallelo, VoltDB scala proporzionalmente al numero di processori presenti, questo in dimostrazione del fatto che ad ogni *core* viene associato un processo.

3.4 Punti di Debolezza di VoltDB

È doveroso sottolineare, in questa analisi, anche e soprattutto i punti di debolezza evidenziati da VoltDB.
Per quanto riguarda la definizione dello schema logico originale del database, alcuni tipi di dato standard sono stati modificati poichè non riconosciuti: è il caso di DATE e CHAR che sono stati sostituiti rispettivamente con TIMESTAMP e VARCHAR.
VoltDB non ha un pieno supporto per le operazioni sulle date; per esempio, avendo una data scritta nel formato DD-MM-YYYY e volendo effettuare un salto temporale di un certo numero di giorni, la soluzione proposta dal DBMS in esame è quella di trasformare in secondi la data e il numero di giorni desiderato mediante la fun-

zione SINCE_EPOCH(), effettuare l'operazione e successivamente ri-trasformare il dato ottenuto in giorni mediante la funzione TO_TIMESTAMP(). Questa procedura risulta essere ne' la più semplice, ne' la più immedianta.
I dati caricati, sono stati modificati affinchè VoltDB li accettasse. I file .csv nella loro forma standard, infatti, hanno al termine di ogni riga una *pipe* che VoltDB non riconosce; per ovviare a questo problema sono stati opportunamente formattati rimuovendo loro tali *pipe*.
Non è stato possibile eseguire molte interrogazioni; le limitazioni di VoltDB sono molto rigide per quanto riguarda operazioni di *join* e *subquery*. Come già osservato le operazioni di *join* sono supportate solo se fatte sulle chiavi scelte per il partizionamento; *join* su campi diversi da quelli elencati sono possibili solo in caso di replica delle tabelle coinvolte. È evidente notare il forte limite dato dal fatto che, se la memoria centrale totale a disposizione non è sufficiente, alcune tabelle che dovrebbero essere replicate non possono esserlo e, siccome il partizionamento è possibile solo su chiavi primarie, alcune *query* risultano essere del tutto ineseguibili. Un ragionamento simile si può portare a termine anche per quello che riguarda le *subquery*. Il metodo proposto da VoltDB è quello di creare viste e interrogare il DBMS in un secondo momento; operazione, la creazione di viste, che necessita di una quantità di memoria centrale di cui non tutti dispongono. Operativamente quello che è stato fatto durante questa sperimentazione è stato eseguire *in primis* la *query* interna e solo successivamente quella esterna, sommando alla fine i tempi di esecuzione per avere un'idea complessiva delle prestazioni fornite.
Da notare sono anche l'assenza di supporto per costrutti come CHECK, AUTO-INCREMENT e FOREIGN KEY; particolarmente importante da sottolineare è quest'ultimo, senza il quale non è possibile garantire il vincolo di integrità referenziale.
I test effettuati sono stati molteplici: inizialmente si è utilizzato un *cluster* formato da un singolo nodo avente 4 GB di RAM, di cui utilizzabile era più di 3 GB: su questa macchina non è stato possibile caricare un database di dimensione approssimativamente pari ad un 1 GB. Si è allora scelto di aumentare il numero di nodi portandolo a 5: in questo *cluster* la memoria centrale totale è stata in linea di massima di 16 GB e non è stato comunque possibile caricare una base di dati di 4 GB. Con l'aggiunta di una sesta macchina, la memoria totale è stata portata

a circa 20 GB e tale database è stato effettivamente caricato. Va specificato però che, di tutte le *query* scelte per il *benchmark*, solamente 3 sono state effettuate su questo database: per le altre interrogazioni il DBMS non aveva spazio sufficiente in memoria centrale per effettuare le operazioni e contemporaneamente di mantere i dati; motivo, questo, che ha portato a testare il *cluster* di 6 macchine con un database ridimensionato a 2 GB.

Per precisazioni pratiche sulle soluzioni impiegate per permettere al sistema di funzionare si veda l'Appendice B.

Conclusioni

Dopo aver installato il software VoltDB su un *cluster* di computer e aver analizzato le sue prestazioni in fase di interrogazione, si è giunti alla conclusione che VoltDB è adatto per un ridotto e molto specifico settore di mercato per molteplici ragioni. Come spiegato in precedenza, le risorse utilizzate da questo software sono sbilanciate rispetto ai dati che vengono caricati.
Si può concludere quindi che, VoltDB è un prodotto rivolto quasi esclusivamente a quelle aziende che hanno ben chiare e immutabili le molteplici *query* da eseguire. Innegabile infatti è la poca, se non nulla, flessibilità in termini di interrogazioni su dati diversi in diverse tabelle. I test eseguiti in laboratorio mostrano come le macchine utilizzate non siano state appropriate al tipo di analisi che si è svolta; macchine con 4 GB di memoria RAM sono al giorno d'oggi quasi obsolete, ma non è affatto difficile pensare che se un'azienda volesse investire nell'acquisto di calcolatori potrebbe farlo scegliendone con almeno 16 GB di RAM: già su macchine di questo tipo, le risorse disponilibi sarebbero sufficienti a gestire un database di dimenioni non banali.

3.5 Sviluppi futuri

Sicuramente studi ulteriori che possono essere svolti su questo DBMS potrebbero concentrarsi sull'eseguire i test proposti su *cluster* di dimensione maggiore e avente a disposizione maggiore memoria RAM. Si potrebbero osservare in questo modo le prestazioni di VoltDB su database di dimensioni reali.
Potrebbe essere interessante concentrare il lavoro sulla scrittura ed esecuzione di *stored procedure* al fine di verificare se i problemi riscontrati durante l'analisi attuale siano evitabili.

In aggiunta, si potrebbe monitorare l'evoluzione di VoltDB e valutare se tutte le sue limitazioni verranno superate in modo tale da farne uno strumento più versatile.

Appendice A

Benchmark

A.1 Schema Logico

Lo schema logico completo di TPC-H presentato nel Capitolo 3 tradotto in linguaggio SQL è il seguente:

```
CREATE TABLE PART (
        P_PARTKEY        PRIMARY KEY,
        P_NAME           VARCHAR(55),
        P_MFGR           CHAR(25),
        P_BRAND          CHAR(10),
        P_TYPE           VARCHAR(25),
        P_SIZE           INTEGER,
        P_CONTAINER      CHAR(10),
        P_RETAILPRICE    DECIMAL,
        P_COMMENT        VARCHAR(23)
);

CREATE TABLE SUPPLIER (
        S_SUPPKEY        PRIMARY KEY,
        S_NAME           CHAR(25),
        S_ADDRESS        VARCHAR(40),
        S_NATIONKEY      BIGINT NOT NULL,
        S_PHONE          CHAR(15),
```

```
          S_ACCTBAL           DECIMAL,
          S_COMMENT           VARCHAR(101)
);

CREATE TABLE PARTSUPP (
          PS_PARTKEY          BIGINT NOT NULL,
          PS_SUPPKEY          BIGINT NOT NULL,
          PS_AVAILQTY         INTEGER,
          PS_SUPPLYCOST       DECIMAL,
          PS_COMMENT          VARCHAR(199),
          PRIMARY KEY (PS_PARTKEY, PS_SUPPKEY)
);

CREATE TABLE CUSTOMER (
          C_CUSTKEY           PRIMARY KEY,
          C_NAME              VARCHAR(25),
          C_ADDRESS           VARCHAR(40),
          C_NATIONKEY         BIGINT NOT NULL,
          C_PHONE             CHAR(15),
          C_ACCTBAL           DECIMAL,
          C_MKTSEGMENT        CHAR(10),
          C_COMMENT           VARCHAR(117)
);

CREATE TABLE ORDERS (
          O_ORDERKEY          PRIMARY KEY,
          O_CUSTKEY           BIGINT NOT NULL,
          O_ORDERSTATUS       CHAR(1),
          O_TOTALPRICE        DECIMAL,
          O_ORDERDATE         DATE,
          O_ORDERPRIORITY CHAR(15),
          O_CLERK             CHAR(15),
          O_SHIPPRIORITY      INTEGER,
          O_COMMENT           VARCHAR(79)
);
```

```
CREATE TABLE LINEITEM (
        L_ORDERKEY       BIGINT NOT NULL,
        L_PARTKEY        BIGINT NOT NULL,
        L_SUPPKEY        BIGINT NOT NULL,
        L_LINENUMBER     INTEGER,
        L_QUANTITY       DECIMAL,
        L_EXTENDEDPRICE  DECIMAL,
        L_DISCOUNT       DECIMAL,
        L_TAX            DECIMAL,
        L_RETURNFLAG     CHAR(1),
        L_LINESTATUS     CHAR(1),
        L_SHIPDATE       DATE,
        L_COMMITDATE     DATE,
        L_RECEIPTDATE    DATE,
        L_SHIPINSTRUCT   CHAR(25),
        L_SHIPMODE       CHAR(10),
        L_COMMENT        VARCHAR(44),
        PRIMARY KEY (L_ORDERKEY, L_LINENUMBER)
);

CREATE TABLE NATION (
        N_NATIONKEY      PRIMARY KEY,
        N_NAME           CHAR(25),
        N_REGIONKEY      BIGINT NOT NULL,
        N_COMMENT        VARCHAR(152)
);

CREATE TABLE REGION (
        R_REGIONKEY      PRIMARY KEY,
        R_NAME           CHAR(25),
        R_COMMENT        VARCHAR(152)
);
```

A.2 Query svolte

Di seguito verranno illustrate dettagliatamente le *query* presentate nel Capitolo 3 mantenendone il nome e i partizionamenti delle tabelle necessari alla corretta esecuzione delle interrogazioni.

Primo partizionamento:

```
PARTITION TABLE PART ON COLUMN P_PARTKEY;
PARTITION TABLE SUPPLIER ON COLUMN S_SUPPKEY;
PARTITION TABLE PARTSUPP ON COLUMN PS_PARTKEY;
PARTITION TABLE CUSTOMER ON COLUMN C_CUSTKEY;
PARTITION TABLE ORDERS ON COLUMN O_ORDERKEY;
PARTITION TABLE LINEITEM ON COLUMN L_ORDERKEY;
```

Questo schema è adatto al successo del *join* delle *query* che seguono.

Q1: Provvede a produrre un *report* per tutte le parti dei vari ordini spedite in una precisa data. Essa è compresa tra 60 e 120 giorni dalla data contenuta nel DB in cui si è spedita la maggior quantità di merce. La *select-list* è composta da prezzo esteso, prezzo scontato, prezzo scontato più le tasse, quantità media, prezzo medio e sconto medio.

I *record* restituiti sono raggruppati su RETURNFLAG e LINESTATUS e successivamente ordinati in modo crescente.

```
SELECT L_RETURNFLAG, L_LINESTATUS,
SUM(L_QUANTITY) AS SUM_QTY,
SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
AS SUM_DISC_PRICE,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX))
AS SUM_CHARGE,
AVG(L_QUANTITY) AS AVG_QTY,
AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
AVG(L_DISCOUNT) AS AVG_DISC,
COUNT(*) AS COUNT_ORDER
```

```
FROM LINEITEM
WHERE L_SHIPDATE <='1998-09-02'
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG, L_LINESTATUS;
```

Q2: Considera tutte le parti degli ordini spedite in un dato anno aventi uno sconto compreso in un dato intervallo. Verrà restituito l'ammontare di ogni linea d'ordine così come se non le fosse stato applicato tale sconto.

```
SELECT
SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE
FROM LINEITEM
WHERE L_SHIPDATE >= '1994-01-01'
AND L_SHIPDATE < '1995-01-01'
AND L_DISCOUNT BETWEEN 0.06 - 0.01 AND 0.06 + 0.01
AND L_QUANTITY < 24;
```

Q3: Conta il numero di fornitori che riescono a soddisfare particolari richieste espresse clienti; essi sono interessati ad ordinare prodotti in 8 diverse misure, specificando la marca a cui non sono interessati e un fornitore che non abbia ricevuto lamentele di sorta. I risultati sono presentati in maniera crescente per quanto riguarda marca, tipo e formato, e in manierta decrescente rispetto al numero di fornitori del prodotto.

```
SELECT P_BRAND, P_TYPE, P_SIZE,
COUNT(DISTINCT PS_SUPPKEY) AS SUPPLIER_CNT
FROM PARTSUPP, PART
WHERE P_PARTKEY = PS_PARTKEY
AND P_BRAND <> 'Brand#45'
AND P_TYPE NOT LIKE 'MEDIUM POLISHED%'
AND P_SIZE IN (49, 14, 23, 45, 19, 3, 36, 9)
AND PS_SUPPKEY NOT IN (
                    SELECT S_SUPPKEY
                    FROM SUPPLIER
                    WHERE S_COMMENT
                    LIKE '%CUSTOMER%COMPLAINTS%')
GROUP BY P_BRAND, P_TYPE, P_SIZE
ORDER BY SUPPLIER_CNT DESC, P_BRAND, P_TYPE, P_SIZE;
```

Q4: Conta il numero di ordini effettuati in un anno avente almeno un cliente che abbia ricevuto la merce più tardi di quanto pattuito.
La *select-list* è composta da il conteggio degli ordini divisi per livello di priorità ed elencati in ordine crescente sulla base di esso.

```
SELECT O_ORDERPRIORITY,
COUNT(*) AS ORDER_COUNT
FROM ORDERS
WHERE O_ORDERDATE >= '1993-07-01'
AND O_ORDERDATE < '1993-10-01'
AND EXISTS (
                    SELECT *
                    FROM LINEITEM
                    WHERE L_ORDERKEY = O_ORDERKEY
                    AND L_COMMITDATE < L_RECEIPTDATE
)
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY;
```

Secondo partizionamento:

```
PARTITION TABLE PART ON COLUMN P_PARTKEY;
PARTITION TABLE SUPPLIER ON COLUMN S_SUPPKEY;
PARTITION TABLE PARTSUPP ON COLUMN PS_PARTKEY;
PARTITION TABLE ORDERS ON COLUMN O_ORDERKEY;
PARTITION TABLE LINEITEM ON COLUMN L_ORDERKEY;
```

Questo schema è adatto al successo del *join* delle *query* che seguono.
Q5: Recupera la priorità attribuita alle spedizioni e il potenziale ammontare degli ordini. Essi vengono recuperati in ordine crescente rispetto al profitto; tali ordini non devono ancora essere stati spediti rispetto ad una daterminata data.

```
SELECT L_ORDERKEY,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'BUILDING'
AND C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND O_ORDERDATE < '1995-03-15'
AND L_SHIPDATE > '1995-03-15'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE, O_ORDERDATE;
```

Q6: Trova i clienti che hanno determinato una certa perdita di guadagno, per un dato trimestre, avendo restituito quanto acquistato. La *query* considera solo gli ordini effettuati nel trimestre specificato. I clienti sono ordinati sulla base del guadagno che avrebbero portato.
Vengono restituiti: dati personali e *account balance* dei clienti, eventuali commenti e profitto perso.

```
SELECT C_CUSTKEY, C_NAME,
SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE,
C_ACCTBAL, N_NAME, C_ADDRESS, C_PHONE, C_COMMENT
FROM CUSTOMER, ORDERS, LINEITEM, NATION
```

```
WHERE C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND O_ORDERDATE >= '1993-10-01'
AND O_ORDERDATE < '1994-01-01'
AND L_RETURNFLAG = 'R'
AND C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE,
N_NAME, C_ADDRESS, C_COMMENT
ORDER BY REVENUE;
```

Terzo partizionamento:

```
PARTITION TABLE PART ON COLUMN P_PARTKEY;
PARTITION TABLE PARTSUPP ON COLUMN PS_PARTKEY;
PARTITION TABLE CUSTOMER ON COLUMN C_CUSTKEY;
PARTITION TABLE ORDERS ON COLUMN O_ORDERKEY;
PARTITION TABLE LINEITEM ON COLUMN L_ORDERKEY;
```

Questo è il partizionamento richiesto per la seguente *query*.
Q7: Sono trovati, per un dato continente, per ogni pezzo di un certo tipo e con un certo formato, il fornitore che lo ha fornito ad un costo maggiore della media. Se, nello stesso continente, viene fornito lo stesso tipo di pezzo con lo stesso formato allo stesso prezzo, si elencano i fornitori aventi *account balance* maggiore.
La *select-list* è composta da: *account balance*, nome, nazione e dati personali del fornitore, codice del pezzo e l'artigiano che l'ha prodotto.

```
SELECT S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY,
P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM PART, SUPPLIER, PARTSUPP, NATION, REGION
WHERE P_PARTKEY = PS_PARTKEY
AND S_SUPPKEY = PS_SUPPKEY
AND P_SIZE = 15
AND P_TYPE LIKE '%BRASS'
AND S_NATIONKEY = N_NATIONKEY
AND N_REGIONKEY = R_REGIONKEY
```

```
AND R_NAME = 'EUROPE'
AND PS_SUPPLYCOST > (
                SELECT AVG(PS_SUPPLYCOST)
                FROM PARTSUPP, SUPPLIER,
                NATION, REGION
                WHERE P_PARTKEY = PS_PARTKEY
                AND S_SUPPKEY = PS_SUPPKEY
                AND S_NATIONKEY = N_NATIONKEY
                AND N_REGIONKEY = R_REGIONKEY
                AND R_NAME = 'EUROPE')
ORDER BY S_ACCTBAL DESC, N_NAME, S_NAME, P_PARTKEY;
```

Quarto partizionamento:

```
PARTITION TABLE PART ON COLUMN P_PARTKEY;
PARTITION TABLE SUPPLIER ON COLUMN S_SUPPKEY;
PARTITION TABLE PARTSUPP ON COLUMN PS_SUPPKEY;
PARTITION TABLE CUSTOMER ON COLUMN C_CUSTKEY;
PARTITION TABLE ORDERS ON COLUMN O_ORDERKEY;
PARTITION TABLE LINEITEM ON COLUMN L_ORDERKEY;
```

Questo è il partizionamento richiesto per la seguente *query*.

Q8: Si trova, tramite scansione di tutte le scorte disponibili dei fornitori in una data nazione, tutti i pezzi che rappresentano una percentuale significante del valore totale di tutti i pezzi disponibili. La *query* restituisce il numero del pezzo e il valore dello stesso ordinato in maniera decrescente.

```
SELECT PS_PARTKEY,
SUM(PS_SUPPLYCOST * PS_AVAILQTY) AS VALUE
FROM PARTSUPP, SUPPLIER, NATION
WHERE PS_SUPPKEY = S_SUPPKEY
AND S_NATIONKEY = N_NATIONKEY
AND N_NAME = 'GERMANY'
GROUP BY PS_PARTKEY HAVING
SUM(PS_SUPPLYCOST * PS_AVAILQTY) > (
```

```
          SELECT
          SUM(PS_SUPPLYCOST * PS_AVAILQTY) * 0.0001
          FROM PARTSUPP, SUPPLIER, NATION
          WHERE PS_SUPPKEY = S_SUPPKEY
          AND S_NATIONKEY = N_NATIONKEY
          AND N_NAME = 'GERMANY')
ORDER BY VALUE DESC;
```

Quinto partizionamento:

```
PARTITION TABLE PART ON COLUMN P_PARTKEY;
PARTITION TABLE SUPPLIER ON COLUMN S_SUPPKEY;
PARTITION TABLE PARTSUPP ON COLUMN PS_PARTKEY;
PARTITION TABLE CUSTOMER ON COLUMN C_CUSTKEY;
PARTITION TABLE ORDERS ON COLUMN O_ORDERKEY;
PARTITION TABLE LINEITEM ON COLUMN L_PARTKEY;
```

Questo è il partizionamento richiesto per la seguente *query*.

Q9: Considera i pezzi di una determinata marca e con un dato tipo di contenitore e determina la quantità media di ogni pezzo per ogni parte ordinata di ogni ordine (concluso e non) in un database di 7 anni.

```
SELECT
SUM(L_EXTENDEDPRICE) / 7.0 AS AVG_YEARLY
FROM LINEITEM, PART
WHERE P_PARTKEY = L_PARTKEY
AND P_BRAND = 'Brand#23'
AND P_CONTAINER = 'MED␣BOX'
AND L_QUANTITY < (
                  SELECT
                  0.2 * AVG(L_QUANTITY)
                  FROM LINEITEM
                  WHERE L_PARTKEY = P_PARTKEY);
```

Appendice B

Precisazioni Pratiche

Per i test effettuati, è stata utilizzata la versione *trial* dell'*Enterprise Edition* di VoltDB. Sono stati installati i pacchetti Debian *voltdb-ent_4.8-1_amd64.deb* poichè semplificano il processo di installazione sistemando automaticamente i file di VoltDB in cartelle di sistema standard [10].
L'installazione di VoltDB comprende:

- *VoltDB Software & Runtime*: il software VoltDB è distribuito sottoforma di archivi JAVA (file .jar).
- *Example Applications*: sono contenute diverse applicazioni esemplificative atte a dimostrare capacità e performance del DBMS.
- *VoltDB Web Studio*: strumento basato su browser per la visualizzazione e l'interrogazione del database collegata al server.
- *Shell Commands*: script eseguibili che eseguono comuni compiti assegnati a VoltDB.
- *Documentation*: documentazione on-line, tutti i manuali e i *javadoc* che descrivono l'interfaccia Java.

Avendo un file SQL che descrive lo schema, occorre dapprima compilarlo creando un file con estensione .jar utilizzando il comando *compile*, successivamente per creare lo schema si utilizza il comando *create* seguito dal nome del faile .jar.
Per popolare il database sono state caricate le tabelle sui nodi del *cluster* e in

seguito memorizzate nella base di dati mediante il comando *csvloader* che, come suggerisce lo stesso nome, prende in *input* un file con estensione .csv e lo carica in memoria centrale. Ad ogni tabella caricata in memoria sono stati effettuati *snapshot* per garantire *durability*.

Bibliografia

[1] Prasanna V., Nirmala S., *NewSQL - The New Way to Handle Big Data*, http://www.linuxforu.com/2012/01/newsql-handle-big-data/. 2012.

[2] The 451 Group, *NoSQL, NewSQL and Beyond*, https://451research.com/report-long?icid=1651.

[3] Stonebraker M., *New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps*, http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext. 2011.

[4] *An Oracle White Paper - Oracle Real Application Clusters (RAC) 11g Release 2*, November 2010.

[5] Maritato L., *In-Memory database*, Università degli studi di Napoli Federico II, 2013.

[6] hstore.cs.brown.edu/documentation/faq/

[7] *H-Store - Next Generation OLTP DBMS Research*, 2008.

[8] *VLDB 2007 - 33rd International Conference on Very Large Data Bases*, http://www.vldb.org/archives/website/2007/

[9] Michael Stonebraker, *OldSQL vs. NoSQL vs. NewSQL on New OLTP*.

[10] *Using VoltDB V4.2*, April 2014.

[11] *Planning Guide VoltDB V4.3*, April 2014.

[12] *The VoltDB Main Memory DBMS*, 2013.

[13] VoltDB Inc., *High performance, scalable RDBMS for big data and real-time analytics*, TECHNICAL OVERVIEW.

[14] VoltDB Inc., *APPLICATION BRIEF: Partitioning*.

[15] VoltDB Inc., *APPLICATION BRIEF: Durability*.

[16] VoltDB Inc., *APPLICATION BRIEF: Availability*.

[17] Transaction Processing Performance Council, *TPC BENCHMARK^{TM} H*.

[18] http://www.pilhokim.com/index.php?title=Project/EFIM/TPC-H.

[19] https://forum.voltdb.com/showthread.php?8-VoltDB-tpc-c-like-Benchmark-Comparison-Benchmark-Description.