ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E IMPLEMENTAZIONE DI UN SISTEMA DI VERIFICA DELLA CONSISTENZA DEI DATI IN UN DATABASE DISTRIBUITO - HERDDB

Relazione finale in
Ingegneria del Software

Relatore:
Prof. STEFANO RIZZI

Tesi di Laurea di:
HAMADO DENE

ANNO ACCADEMICO 2019–2020
SESSIONE III

PAROLE CHIAVE

HerdDB

Sistema distribuito

Database distribuito

FSM

Consistenza dei dati

Indice

La forza ti viene perché a volte non hai altra scelta. Perché se non lo fai tu, non lo farebbe nessuno per te. Lo fai per chi hai vicino, e lo fai anche per te. Perché è una sfida per dimostrare a te stesso che non ti arrendi, ma che ce la farai."
(Iob Daniela).

Capitolo 1

Introduzione

Per qualsiasi azienda che debba gestire i dati dei propri clienti tramite database è fondamentale, sia legalmente che economicamente, garantire la continuità del proprio business facendo fronte a eventuali malfunzionamenti del sistema. In questo frangente i database distribuiti garantiscono una certa affidabilità e tolleranza ai guasti.

In questo tipo di architettura la memorizzazione dei dati è distribuita su più elaboratori (nodi) connessi tra loro, formando così un sistema distribuito che permette di accedere ai propri dati anche in caso di malfunzionamenti a un nodo. Basti considerare un sistema con database centralizzato dove il nodo che ospita l'applicazione si guasta; in questo caso tutti gli applicativi con cui interagisce smetterebbero di funzionare e questa reazione a catena si tradurrebbe in un blocco della produzione per tutto il lasso di tempo in cui il sistema rimane offline. È facile immaginare come il tempo di guasto sia direttamente proporzionale alla perdita economica dell'azienda.

Nel momento in cui andiamo ad implementare un sistema distribuito, in caso di guasto a un nodo, avremo una continuità al flusso di produzione grazie alla presenza degli altri nodi. Come sappiamo uno degli aspetti più importanti in un database è quello della consistenza dei dati, ovvero che i dati memorizzati devono essere sempre significativi ed effettivamente utilizzabili. In un database distribuito questo concetto diventa ancora più critico poiché essendo i dati replicati su più nodi è necessario garantire la consistenza del dato su tutti i nodi che compongono il sistema distribuito.

Basandosi su questi presupposti, nel seguito del documento in oggetto, verrà descritta e implementata un'applicazione in grado verificare la consistenza

dei dati in un database distribuito su più nodi.Andremo ad introdurre l'algoritmo in un database distribuito HerdDB, il quale è stato sviluppato per lavorare in modalità cluster andando dunque a replicare i dati sui vari nodi.

Capitolo 2

Database

2.1 Introduzione ai database

Un database è una raccolta di informazioni organizzata in modo da poter essere facilmente accessibile, gestita ed aggiornata. Possiamo più semplicemente definirlo come un sistema elettronico che ci permette di accedere, manipolare ed aggiornare facilmente una serie di dati.

In termini tecnici i database possono essere classificati in diversi tipi, in particolare abbiamo:

- RDBMS (Relational Database Management System).
- Non-Relational database (NoSQL).

2.1.1 Relational Database Management System (RDBMS)

In un database relazionale i dati sono archiviati in relazioni tra tabelle sotto forma di tuples (righe) e attributi (Colonne).

Il vantaggio principale di un database relazionale è la sua struttura tabellare, dalle quale i dati possono essere facilmente archiviati, classificati, interrogati e filtrati senza la necessità di riorganizzare le tabelle del database.Questo lo rende una scelta preferita qualora l'applicazione gestisce un insieme di dati ben strutturati. I database relazionali forniscono anche una gestione atomica, consistente e persistente dei dati,dall'inglese ACID ovvero "Atomicity,Consistency,Isolation,Durability".

Uno dei principali svantaggi di un database relazionale è la necessità di

sviluppare attentamente l'architettura prima di aggiungere dei dati. Questo può essere un problema poiché in alcuni casi è difficile prevedere con esattezza la struttura dati che bisogna costruire. Inoltre, la struttura dati si può evolvere nel tempo, il che richiede cambiamenti significativi nel database stesso. Col tempo, l'architettura dovrà essere ottimizzata e questo risulta un processo molto complicato e nella maggior parte dei casi, costoso.

2.1.2 Non-relational database (NoSQL)

Nei database NoSQL i dati sono conservati in documenti detti collections ovvero una collezione di documenti. A differenza dei database relazionali (che utilizzano strutture dati) abbiamo un concetto di chiave-valore. Questi database garantiscono un elevata velocità di esecuzione e di elaborazione di grandi quantità di dati.

L'ovvio vantaggio di un database non relazionale è la capacità di archiviare ed elaborare grandi quantità di dati non strutturati. Di conseguenza, può elaborare qualsiasi tipo di dati senza dover modificare l'architettura. Pertanto, la creazione e la gestione di un database NoSQL è più rapida ed economica.

Uno dei punti di criticità di un database NoSQL è che garantiscono l'atomicità di singole operazioni ma non di una sequenza di operazioni. Come conseguenza, diventa compito dell'applicazione garantire il completamento della transazione in uno stato consistente ed atomico rispetto ad una sequenza di operazioni.

2.1.3 Conclusione

La scelta tra un database relazionale e non relazionale dipende dalle proprie esigenze. Se si lavora con dei dati ben strutturati, più o meno costanti, un database relazionale sarà la scelta migliore. Tuttavia la gestione di vari tipi di dati, porta una maggior manutenzione ed aggiornamenti del database. In tal caso la scelta migliore ricadrebbe su un database NoSQL, che archivia tutti i dati disponibili come file. Risulta meno strutturato rispetto ad uno relazionale, ma richiede meno tempo di manutenzione.

Capitolo 3

Sistemi Distribuiti

3.1 Introduzione ai sistemi distribuiti

Un sistema distribuito è una raccolta di elementi di calcolo autonomi che appare ai suoi utenti come un unico sistema coerente. Questa definizione si riferisce a due caratteristiche dei sistemi distribuiti. Il primo è che un sistema distribuito è una raccolta di elementi informatici ciascuno in grado di comportarsi indipendentemente l'uno dall'altro. Un elemento di calcolo, che generalmente viene chiamato nodo, può essere un dispositivo hardware o un processo software. Una seconda caratteristica è che l'insieme dei nodi appare ai suoi utenti (siano essi persone o applicazioni) come un singolo calcolatore.

3.2 Caratteristiche

- **Insieme di elementi di calcolo autonomi:**
  Un principio fondamentale è che i nodi possono agire indipendente l'uno dall'altro. Solitamente i nodi sono programmati per raggiungere obiettivi comuni, che vengono realizzati scambiando messaggi tra di loro. Un nodo reagisce ai messaggi in arrivo, che vengono quindi elaborati e a loro volta scatenano ulteriori comunicazioni verso gli altri nodi.

- **Sistema unico e coerente:**
  Un sistema distribuito deve apparire come un unico sistema coerente.Questo significa che dovrebbe esistere una visione ad un sistema singolo, ovvero che un utente finale non dovrebbe notare che i processi, i dati e il controllo sono dispersi in una rete di computer.

Raggiungere la visione di un singolo computer solitamente è un compito molto arduo, di conseguenza la coerenza di un sistema distribuito è tale se si comporta in base alle aspettative degli utenti. Più precisamente, in un unico sistema coerente il risultato di una determinata richiesta deve essere uguale in tutti i nodi, indipendente da dove, quando e in che nodo è avvenuto l'interazione con l'utente.

3.3 Vantaggi

- **Scalabilità:**
  I processi di calcolo attraverso un sistema distribuito avvengono indipendentemente l'uno dall'altro. Ciò significa che è possibile aggiungere dei nodi e funzionalità in base alle proprie esigenze.

  Questi sistemi offrono quindi la capacità di ridimensionare in maniera esponenziale la potenza di calcolo in modo relativamente economico.

- **Affidabilità:**
  I sistemi distribuiti creano un esperienza affidabile per gli utenti finali, perché fanno affidamento su centinaia o migliaia di computer relativamente economici che cooperano tra di loro, creando un aspetto esteriore di un singolo computer ad alta potenza.

  In un ambiente centralizzato (a singola macchina) se quest'ultima si guasta, lo stesso vale per l'intero sistema. Mentre in un ambiente distribuito, un guasto ad un nodo non compromette l'intero sistema.

- **Prestazione:**
  Un sistema distribuito può gestire le attività in maniera efficiente poiché i carichi di lavoro e le richieste vengono distribuite su più elaboratori (nodi), permettendo eseguire le operazioni in parallelo.

3.4 Svantaggi

- **Pianificazione:**
  Un sistema distribuito deve decidere quali lavori , quando e dove devono essere eseguiti. Gli scheduler hanno dei limiti, che portano a hardware sottoutilizzati e tempi di esecuzioni imprevedibili.

- **Latenza:**
  Con un interscambio cosi complesso tra elaborazione hardware e chiamate software sulla rete, la latenza può diventare un problema per gli utenti. Questo porta spesso a dover fare dei compromessi tra disponibilità, coerenza e latenza.

- **Osservabilità:**
  Con un sistema distribuito di grandi dimensioni diventa molto difficile raccogliere, elaborare, presentare e monitorare le metriche di utilizzo dell'hardware.

3.5 Esempi di sistemi distribuiti

Sappiamo che un un sistema distribuito utilizza molteplici elaboratori su una rete per completare svariate funzioni. Molti dei sistemi ad oggi utilizzati sono già basati su sistemi distribuiti. Basti pensare che il WWW (World wide web) in sé comprende un insieme di sistemi che cooperano per fornire più rapidamente delle informazioni.
Alcuni esempi dunque includono: Internet, Reti Intranet e peer-to-peer , sistemi di posta elettronica e cosi via.

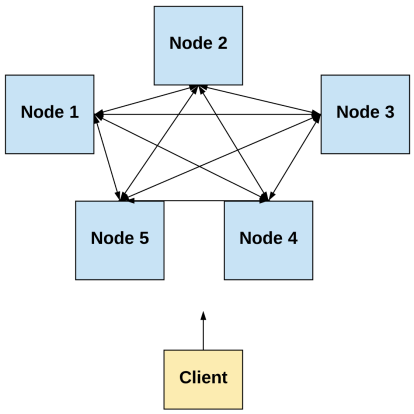

Figura 3.1: Esempio di un sistema distribuito

Capitolo 4

Scalabilità (Scaling)

La scalabilità detto anche ridimensionamento è la capacità di un sistema, di una rete o di un processo di gestire una quantità crescente di lavoro o il suo potenziale di essere ampliato per adattarti alla crescita. Essa può essere valutata utilizzando alcune metriche quali, il volume dei dati, maggiore concorrenza e tasso di interazione più elevato.

4.1 Misure di scalabilità

4.1.1 Volume di dati

Man mano che un'azienda cresce dovrà gestire sempre più dati. Gestire più volumi di dati diventa una attività molto dispendiosa, poiché è necessario cercare,ordinare, leggere dal disco e aggiornare le informazioni in modo efficiente. Con l'avvento dei big data, la memorizzazione di grandi quantità di informazioni è diventato un requisito comune, quindi più dati vengono archiviati più quest'ultimi possono migliorare l'accuratezza delle analisi aziendali.

4.1.2 Concorrenza

Maggior concorrenza significa, più connessioni, thread, messaggi, flussi di dati elaborati in parallelo e gestione contemporanea di più sessioni utente. Se consideriamo le applicazioni Web, la concorrenza indica il numero di utenti che possono interagire con il sistema contemporaneamente senza influire sulla loro esperienza di navigazione. Raggiungere alti livelli di concorrenza diventa difficile perché in un'applicazione basata sul web tutte le richieste vengono elaborate dagli stessi server, quindi gli utenti condividono memoria, CPU, rete e disco.

4.1.3 Interazione

L'interazione misura la frequenza con cui un utente interagisce con il server. Il tasso di interazione può aumentare o diminuire in base al tipo di applicazione.

Per esempio un gioco multiplayer online ha un alto tasso di interazione perché deve scambiare messaggi più volte al secondo tra più nodi dell'applicazione. La sfida più grande dunque è mantenere una bassa latenza con un più alto tasso di interazione tra diversi utenti in tutto il mondo e l'applicazione.

4.2 Tipi di Scalabilità

La scalabilità può essere classificato in Scalabilità verticale e orizzontale.

4.2.1 Scalabilità verticale (Scale-up)

Il ridimensionamento verticale è relativamente semplice, si tratta solo di aggiungere più risorse all'hardware del server, come CPU e memoria, o migliorare le prestazioni del disco cambiandolo in uno più veloce.

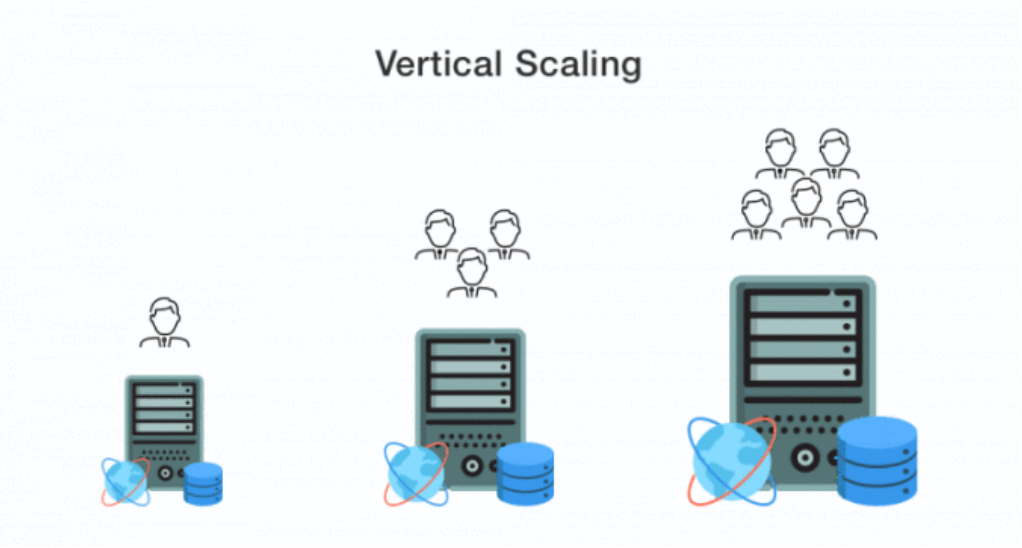

Figura 4.1: Ridimensionamento verticale

Questa strategia è rapida e di solito non richiede alcun cambiamento architettonico, specialmente nel cloud computing, dove è possibile aumentare la capacità di una macchina virtuale in pochi clic.

Tuttavia è possibile raggiungere il limite hardware che può essere utilizzato sullo stesso server, ovvero che non è più possibile aumentare la dimensione della RAM o la quantità di CPU all'infinito.

Un degli svantaggi del ridimensionamento verticale sta nel costo:

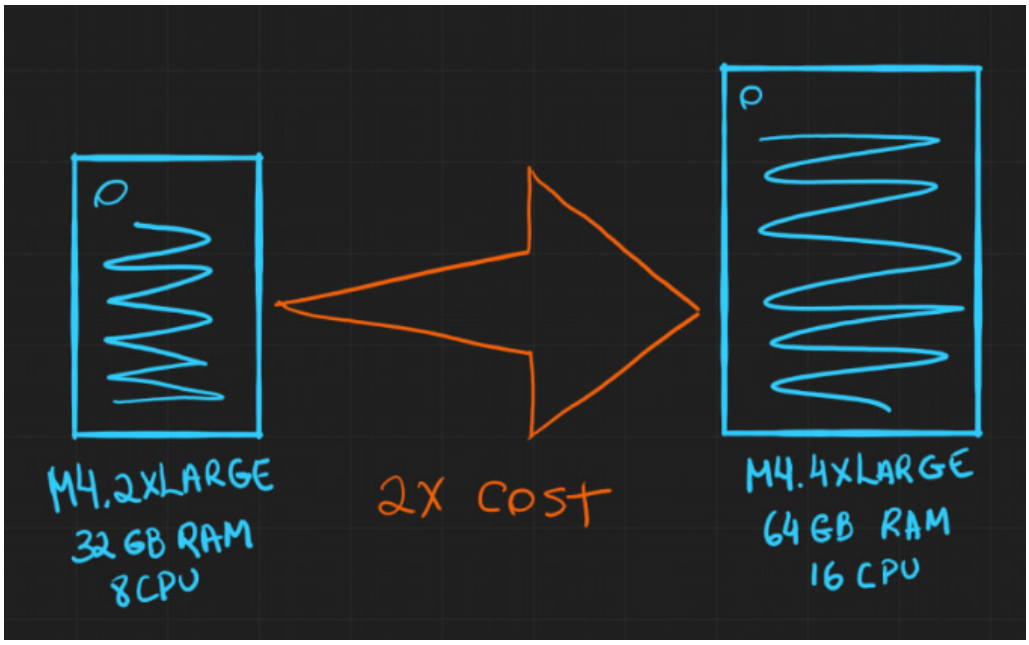

Figura 4.2: Server Amazon AWS

Se consideriamo un AWS EC2 m4.2xlarge che ha 8 CPU,32 GB di RAM con un costo di 302$ al mese, se dobbiamo aggiungere 16GB di RAM bisognerà raddoppiare la quantità di memoria prendendo l'm4.4xlarge con 16CPU e 64GB di RAM che costa esattamente il doppio, ovvero 604$ al mese.

Inoltre centralizzare l'elaborazione su un singolo server non è una buona idea in quanto può essere pericolosa, perché non è una strategia tollerante agli errori. In caso di arresto anomalo del server l'intero sistema non sarà disponibile.

4.2.2 Scalabilità Orizzontale (Scale-out)

Il ridimensionamento orizzontale viene realizzato semplicemente aggiungendo server più semplici anziché acquistare una singola macchina potente.

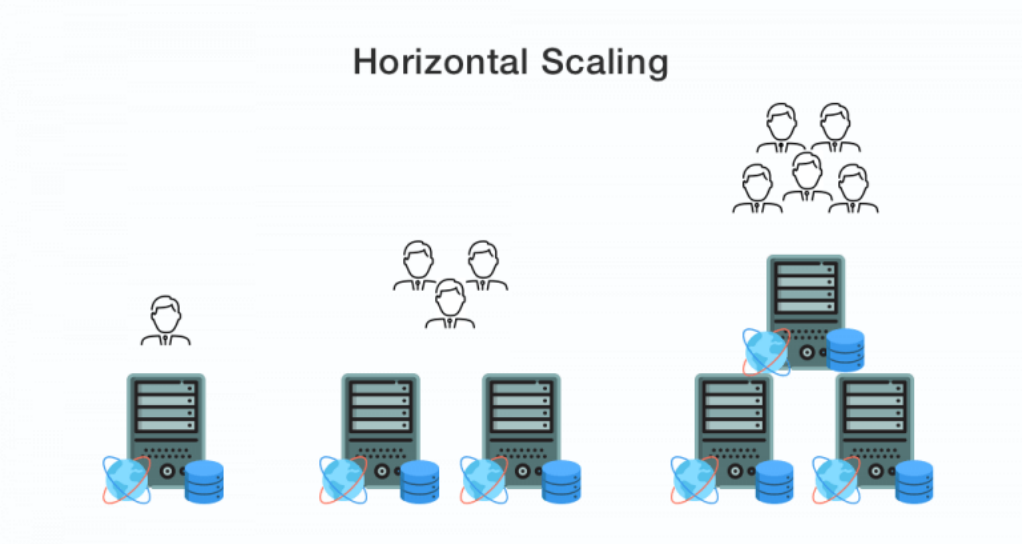

Figura 4.3: Ridimensionamento orizzontale

Molte aziende globali quali Amazon,Netflix,Facebook fanno uso di questa strategia per assistere un gran numero di clienti in tutto il mondo.

Acquistare hardware sempre più potenti non è il modo più economico per servire una base di utenti sempre in costante crescita. Sebbene sia necessario investire di più nell'ingegneria del software per supportare elevati volumi di dati, livelli di concorrenza elevati e un tasso di interazione maggiore. L'investimento iniziale ripaga in una fase successiva, poiché è più economico distribuire il sistema su semplici server con un architettura software sofisticata.

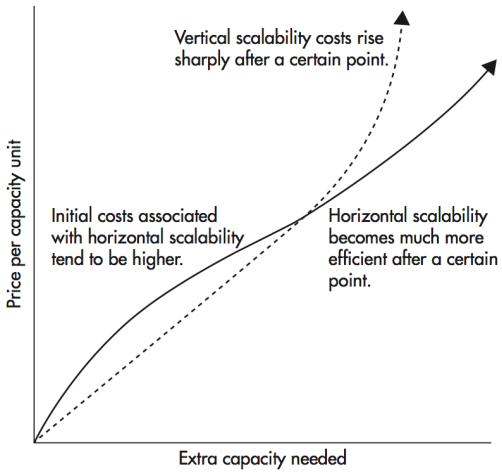

Figura 4.4: Grafico del costo del ridimensionamento verticale ed orizzontale

Capitolo 5

Database distribuiti (DDB)

Un database distribuito (DDB) è un database nel quale gli archivi non sono memorizzati sullo stesso computer ma bensì su più elaboratori o nodi. Il Database in senso fisico può essere dislocato in più computer situati nello stesso luogo, oppure distribuito in una rete di computer connessi tra loro sotto forma di un sistema distribuito.

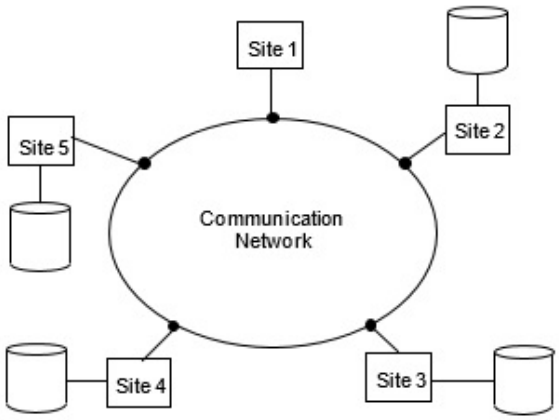

Figura 5.1: Ambiente di database distribuito

5.1 DDBMS

Il DDBMS (distributed database management system) è un'applicazione centralizzata che gestisce un database distribuito come se fosse tutto archiviato su un

singolo nodo. Sincronizza periodicamente tutti i dati e, nei casi in cui più utenti debbano accedere agli stessi dati, garantisce che gli aggiornamenti e le eliminazioni eseguiti sui dati in una posizione si riflettano automaticamente nei dati memorizzati altrove.

5.2 Architettura

I database distribuiti possono essere:

- Omogenei.
- Etorogenei.

5.2.1 Omogeneo

In un sistema di database distribuiti omogeneo, tutti i nodi hanno lo stesso hardware sottostante ed eseguono gli stessi sistemi operativi e applicazioni di database. Appaino all'utente come un unico sistema e sono facilmente progettatili e gestibili.

Per avere un sistema di database distribuito ed omogeneo, le strutture dati in ciascun nodo devono essere identiche o compatibili come per l'applicazione di database utilizzata.

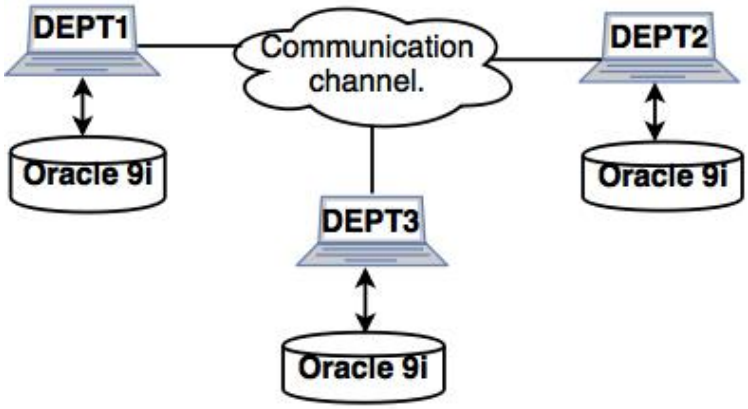

Figura 5.2: Database distribuito omogeneo

5.2.2 Eterogeneo

In un database distribuito eterogeneo, l'hardware, i sistemi operativi e le applicazioni del database possono differire in ciascun nodo. Nodi diversi possono utilizzare schemi e software diversi, sebbene una differenza nello schema possa rendere difficile l'elaborazione delle query e delle transazioni.

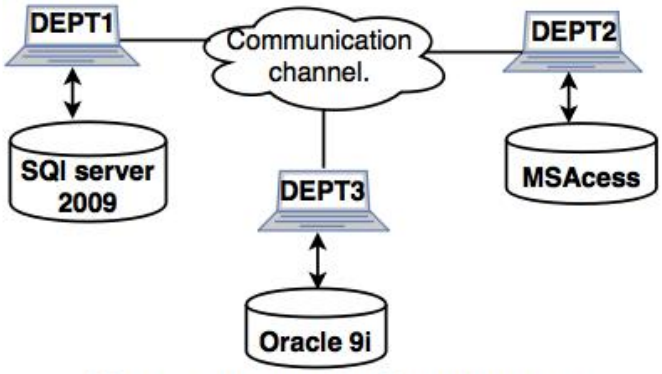

Figura 5.3: Database distribuito eterogeneo

5.3 Archiviazione dei dati distribuiti

Esistono due modi in cui i dati posso essere archiviati nei nodi di un database distribuito:

- Replicazione.
- Frammentazione.

5.3.1 Replicazione

La replicazione è una proprietà del DBMS distribuito che permette di allocare stesse porzioni di database su nodi diversi. In altri termini, dopo la replica tutti i dati vengono memorizzati in modo ridondate su tutti i nodi. Se l'intero database è memorizzato su tutti i nodi, parliamo di un database completamente ridondante.

Il grande vantaggio della replica è che troviamo su tutti i nodi, effettivamente gli stessi dati, quindi eventuali query possono essere eseguiti in parallelo.

Tuttavia i dati devono essere sempre correttamente aggiornati. Qualsiasi modifica effettuata su un nodo deve essere propagata anche su tutti gli altri, altrimenti ciò porterebbe ad un'incoerenza.

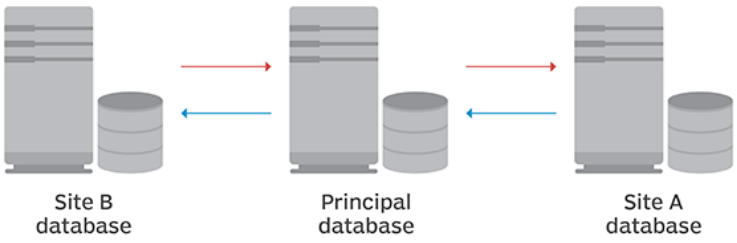

Figura 5.4: Replicazione del database

5.3.2 Frammentazione

In questo approccio i dati sono frammentati, ovvero divisi in parti più piccole. I frammenti vengono distribuiti e memorizzati nei vari siti in cui sono necessari. La distribuzione dei frammenti avviene in modo che possano essere utilizzati per ricostruire i dati originali se uno dei nodi fallisce. Non c'è rischio di perdita dei dati.
La frammentazione a sua volta può essere effettuata usando diversi approcci, in particolare:

- **Frammentazione orizzontale**.
  Nella frammentazione Orizzontale ogni frammento consiste in una sequenza di tuple di una relazione, mantenendo l'intero schema della relazione in ogni frammento. Solitamente in questo caso i dati non vengono replicati, cosi è possibile ricostruire la base di dati semplicemente unendo i frammenti.

- **Frammentazione verticale**.
  Nella frammentazione verticale le relazioni vengono divise per insieme di attributi. In questo caso è necessario replicare la chiave primaria in ogni frammento, cosi da permettere la ricostruzione attraverso un JOIN.

- **Frammentazione ibrida**.
  La frammentazione ibrida si può ottenere eseguendo una frammentazione orizzontale e verticale.

5.4 Vantaggi

- **Sviluppo modulare:**
  I database distribuiti possono essere sviluppati in maniera modulare, questo vuol dire che il sistema può essere espanso aggiungendo nuovi computer e dati locali al nuovo sito (nodo) e collegandoli al sistema distribuito senza interruzioni.

- **Affidabilità:**
  Quando si verificano errori nei database centralizzati, il sistema si arresta completamente. Mentre nei database distribuiti quando si guasta un componente, il sistema continuerà a funzionare a prestazione ridotte fino a quando il guasto non verrà corretto.

- **Riduzione dei costi di comunicazione:**
  Si possono ottenere costi di comunicazione inferiori (Bassa latenza) se i dati si trovano dove vengono maggiormente utilizzati. Ciò non è possibile nei sistemi centralizzati.

5.5 Svantaggi

- **Complessità:**
  Lo svantaggio principale sta nella complessità. Essendo i dati replica, se il software non gestisce in maniera adeguata la replica dei dati, si verificherà un degrado in termini di disponibilità, affidabilità e prestazioni rispetto ad un sistema centralizzato. In tal caso tutti i vantaggi legati ad un database distribuito diventerebbero svantaggi.

- **Sicurezza:**
  In un sistema centralizzato, l'accesso ai dati può essere facilmente controllato. Tuttavia, in un database distribuito l'accesso ai dati replicati deve essere controllato in più posizioni e anche la rete stessa deve essere protetta.

- **Progettazione complessa:**
  Oltre alle difficoltà standard nella progettazione di un database centralizzato, la progettazione di un database distribuito deve tenere conto della frammentazione dei dati, dell'allocazione della frammentazione a siti specifici e della replica dei dati.

- **Controllo dell'integrità più complesso:**
  L'integrità di un database si riferisce alla validità e alla coerenza dei dati memorizzati. L'integrità generalmente è espressa in termini di vincoli, che sono regole di coerenza che il database non è autorizzato a violare. L'applicazione dei vincoli di integrità generalmente richiede l'accesso ad una grande quantità di dati che definiscono i vincoli. nei Database distribuiti, i costi di comunicazione ed elaborazione necessari per applicare i vincoli di integrità sono elevati rispetto ad un sistema centralizzato.

5.6 Apache Cassandra

Apache Cassandra è un sistema di database distribuito open source progettato per l'archiviazione e la gestione di grandi quantità di dati. Può fungere sia da archivio dati operativo in tempo reale per applicazioni transazionali online, sia da database ad alta intensità di lettura per sistemi business intelligenze su larga scala.

E' progettato per avere nodi simmetrici peer-to-peer, anziché nodi principali o denominati, per garantire che non ci possa mai essere un singolo punto di errore. Cassandra partiziona automaticamente i dati su tutti i nodi del cluster, ma l'amministratore ha il potere di determinare quali dati verranno replicati e quante copie dei dati verranno create.

5.6.1 Architettura di Cassandra

Cassandra a differenza dei sui concorrenti presenta un architettura distribuita "ad anello" senza un master-slave. Più precisamente tutti i nodi sono uguali, e non esiste un concetto di nodo principale. La comunicazione tra i vari nodi avviene tramite il protocollo di gossip. Con questo tipo di protocollo ogni nodo comunicherà al suo vicino le ultime novità che ha ottenuto, cosi che in poco tempo tutti i nodi ne saranno a conoscenza. La comunicazione viene fatta solo con alcuni nodi (solitamente max 3 nodi) e non con tutti i nodi in modo da ridurre il carico di rete.

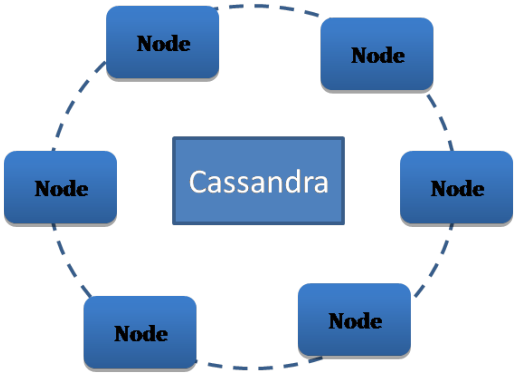

Figura 5.5: Architettura di cassandra

Un cluster cassandra viene visualizzato come un anello, perché utilizza un algoritmo di hashing coerente per distribuire i dati. All'avvio a ciascun nodo viene assegnato un intervallo di token che determina la sua posizione nel cluster. Questi intervalli sono necessari poiché permettono di distribuire i dati in maniera uniforme sull'anello.

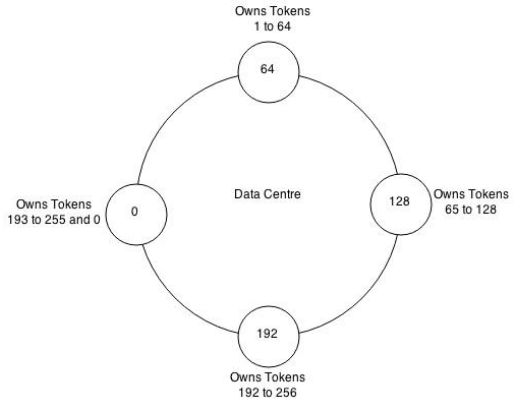

Figura 5.6: Assegnazione dei token

Ciascun nodo è responsabile dei valori del token e di un determinato set di dati determinato dal partizionatore. Un partizionatore è una funziona hash che viene utilizzato per calcolare il token, che verrà utilizzato per determinare il nodo che memorizzerà la prima replica. Dopo aver individuato la prima replica, le repliche successive vengono decide in senso orario.

5.7 Hbase

Apache HBase è un archivio di dati chiave/valore orientato alla colonna creato per essere eseguito su Hadoop Distribuited File System (HDFS).Hadoop è un framework per la gestione di set di dati di grandi dimensioni in un ambiente di elaborazione distribuito.

Hbase è progettato per supportare elevate velocità di aggiornamento delle tabelle e per ridimensionarsi orizzontalmente in cluster di calcolo distribuito. E' noto per fornire una forte coerenza dei dati in lettura e scrittura, che lo distingue dagli altri database NoSQL. Un aspetto importante dell'architettura è l'uso di nodi master per gestire le region server che si occupano di distribuire ed elaborare parti delle tabelle dei dati.

5.7.1 Architettura di Hbase

L'architettura di Hbase consiste principalmente di tre componenti, i Region servers (server di regione), Master server e Zookeeper. In particolare abbiamo:

- **Master Server:**
  L'Assegnazione delle regioni e le operazioni DDL quali creazione ed eliminazioni di tabelle sono gestiti dal Master server. Essi è anche responsabile del monitoraggio di tutte le istanze del RS nel cluster, ed è l'interfaccia per tutte le modifiche ai metadati.

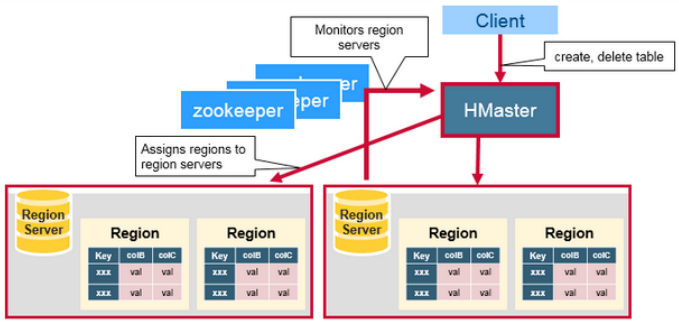

Figura 5.7: Master Hbase

- **Regioni:**
  La tabelle di Hbase sono divise in regioni, dove una regione è delimitato da una chiave di inizio regione e una chiave di fine regione. Quest'ultimi vengono assegnati ai nodi del cluster, chiamati **Region Server (RS)**.

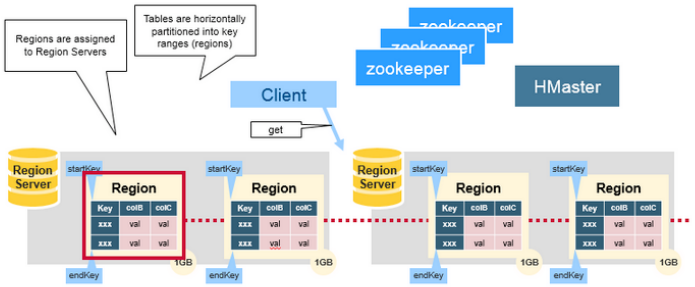

Figura 5.8: Regione Hbase

- **Region Server:**
  Il server delle regione si occupa di gestire le regioni e viene eseguito su su un DataNodes HDFS. I sui compiti principali sono:
  - Comunicare con il client e gestire le operazioni relativi alla manipolazione dei dati.
  - Decidere la dimensione delle regioni.
  - Gestire le richieste di lettura e scrittura per tutte le regioni sottostanti.

- **Hbase MenStore:**
  Il MenStore Hbase è simile alla memoria cache standard. E' un buffer di scrittura in cui Hbase accumula i dati, prima di effettuare una scrittura permanente sul disco. Quando un MenStore accumula abbastanza dati, l'intero set ordinato viene scritto in un file HDFS. Essendo gli HFile immutabili, ad ogni scrittura viene generato un nuovo file.

- **Hbase HFile:**
  Su Hbase dati vengono memorizzati in un HFile che consiste una sequenza ordinata di chiavi-valori . Le scritture dunque sono fatte in maniera sequenziale, e le coppie di chiavi-valori sono memorizzate in ordine crescente. Gli HFile sono immutabili, ed ad ogni scrittura deve esserne generata una nuova.

- **Hbase e Zookeeper:**
  Hbase utilizza Zookeeper come servizio di coordinamento per mantenere lo stato dei server nel cluster. Indica quali server sono attivi e disponibili e fornisce notifiche ad Hbase in caso di errori. Si occupa anche del partizionamento della rete e della comunicazione del client con le regioni.

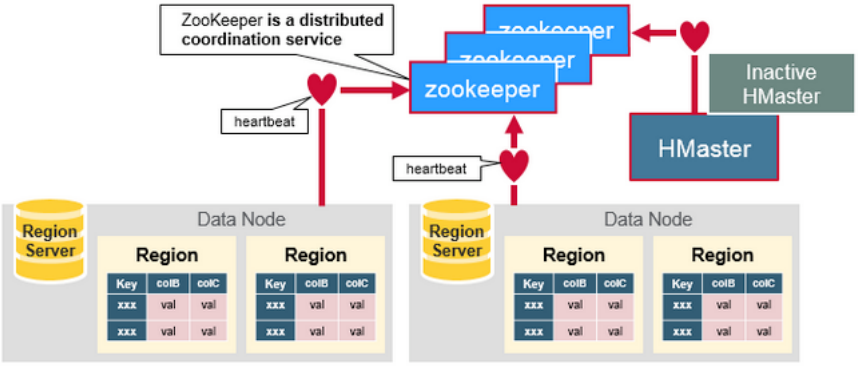

Figura 5.9: Zookeeper su Hbase

5.8 HerdDB

HerdDB è un database distribuito, dove i dati vengono replicati in un cluster di server (ovvero in un sistema distribuito) senza la necessità di uno storage condiviso. La sua architettura è basata sul concetto di database NoSQL, quindi a basso livello è semplicemente un database di tipo chiave-valore con una parte di astrazione SQL, che consente all'utente di sfruttare le sue attuali conoscenze e trasferire applicazioni già esistenti su HerdDB. L'obiettivo di questo progetto è quello di fornire un'applicazione che fosse ottimizzato per "Scritture" veloci con schemi di lettura (Read) e aggiornamenti (Update) basati sulla chiave primaria.

5.8.1 Architettura di HerdDB

I dati all'interno di HerdDB sono organizzati sotto forma di tabelle come in qualsiasi database SQL. Essendo un database distribuito, i dati vengono replicati usando il concetto di replicazione dei dati e le tabelle sono organizzati all'interno di tablespaces.

Un tablespace è un insieme logico di tabelle ed è il concetto architettonico fondamentalmente su cui è costruito la replica dei dati.

Per la replicazione dei dati, HerdDB sfrutta le capacità di Apache Bookkeeper, che fornisce un registro distribuito su cui scrivere le transazioni. Le caratteristiche principali sono:

- Ogni scrittura è garantita come durevole dopo aver ricevuto l'ack da parte di Bookkeeper.
- Viene garantito che ogni replica leggerà solo le entries per la quale l'ack è stato ricevuto.
- Ogni ledger (L'unità di memoria di base di Bookkeeper) può essere scritto una sola volta.

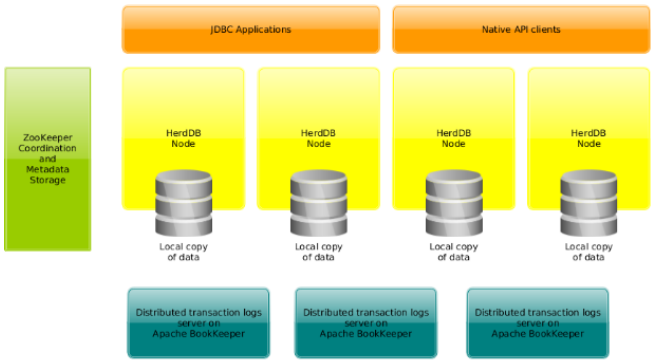

Figura 5.10: Architettura di HerdDB

5.8.2 Replicazione dei dati

In una installazione di tipo cluster, per ogni tablespace viene delegato un nodo leader e un numero di nodi come repliche che vengono denominati follower. Il leader è l'unico che può accettare query, ed è quindi l'unico nodo in grado di modificare i dati di transazione e i metadati dei tablespaces. In altri termini è l'unico nodo che può effettuare manipolazioni ai dati come insert/update.

Per ogni mutazione di dati/metadati, il leader del tablespace scrive un voce sul log delle repliche (Implementato tramite Apache Bookkeeper) e poi aggiorna successivamente la copia locale dei dati. Tramite una funzionalità di Bookkeeper i nodi follower aggiornano continuamente il loro log di transazione attingendosi a quello del leader ed aggiornano anche essi la loro copia locale dei dati.

5.8.3 Concetto di Leader/Follower

Il ruolo di leader è deciso in maniera abbastanza statico, quindi non ci sono elezioni continue. E' possibile per l'amministratore di sistema settare un nodo come leader, attraverso un parametro di configurazione e decidere per ogni tablespace il numero minimo di repliche che deve avere. Se il leader diventa irraggiungibile, dopo un tempo T configurabile dall'amministratore viene nominato leader uno dei nodi follower tramite un algoritmo di leadership election. Quando un nodo parte come leader, deve assicurarsi di avere abbastanza dati, quindi la prima cosa che fa è scaricarsi i dati mancanti da Bookkeeper. Mentre quando una replica viene avviata per la prima volta, deve contattare il leader del tablespace e scaricare tutti i dati e metadati del tablespace.

Per permettere di bilanciare correttamente il carico, è possibile delegare come leader un nodo diverso per ogni tablespace.

5.8.4 Transazione

Una transazione indica una qualunque sequenza di operazioni lecite che, se eseguita in modo corretto, produce una variazione nello stato di una base di dati. In caso di successo, il risultato delle operazioni deve essere permanente o persistente, mentre in caso di insuccesso si deve tornare allo stato precedente all'inizio della transazione [24].

Il nucleo di HerdDB è il log delle transazioni, che è basato su Apache BooKKeeper. In caso di replica, ogni mutazione sui dati viene scritta sul transaction log, e quindi applicata alla versione in memoria della riga corrispondente. Questo registro viene continuamente letto dai follower, in questo modo ogni replica applica le stesse modifiche ai dati e rimane sincronizzata con il suo leader. Ad ogni scrittura sul log viene assegnato un "numero progressivo" che sarà la chiave. Con questa chiave i dati possono essere trasferiti in maniera coerente sulla rete.

5.8.5 Pagine di dati e snapshots

In qualsiasi momento i dati vengono memorizzati in parte sul buffer di memoria ed in parte sul disco. Se la JVM si arresta in modo anomalo azzerando il buffer, non viene perso nessun dato poiché il log della transazioni corrisponde alla verità assoluta e il database può essere recuperato da esso o da un suo snapshot. Quando una riga viene scritta sul disco, gli viene assegnata una pagina di dati, e alla prima mutazione del record, egli viene staccata dalla pagina e quest'ultima viene contrassegnata come pagina sporca. Il checkpoint è il momento in cui viene fatto la manutenzione delle pagine sul disco, in cui tutte le pagine sporche vengono eliminate e ogni record su di esso viene utilizzato per creare nuove pagine insieme ai nuovi records o records aggiornati. I records modificati,inseriti o cancellati nell'ambito delle transazioni non vengono mai scritti su disco e non sono presenti sul buffer principale, in tal modo sulla memoria principale è sempre presente uno snapshot coerente di dati. Per archiviare i dati, ogni transazione utilizza un buffer temporaneo locale.

Durante il checkpoint, l'elenco degli ID delle pagine attive viene scritto sul disco, insieme all'attuale log sequence number. Le pagine vecchie non vengono cancellati istantaneamente, poiché potrebbero far parte di un precedente shapshot valido e possono essere richieste l'esecuzione di una scansione completa della tabella da tale posizione.

5.9 Apache Bookkeeper

La gestione delle repliche di HerdDB è basata su Apache BooKKeeper.

5.9.1 Introduzione a Bookkeeper

Bookkeeper è un servizio che fornisce la memorizzazione persistente di flussi di log entries in sequenze chiamate ledgers. Queste entries vengono poi replicate in vari nodi di un sistema in cluster.
Questo servizio è stato progettato per essere affidabile e resistente ad una vasta gamma di guasti. I bookies possono arrestarsi in modo anomalo, corrompere e scartare dati, ma finché ci sono abbastanza bookies che si comportano correttamente il servizio nel suo insieme continuerà a funzionare.

5.9.2 Concetti e terminologia di Bookkeeper

In Bookkeeper:

- Ogni unità di un log è una entry (una voce)
- I flussi di log entries vengono chiamari ledgers
- I singoli server che memorizzano i ledgers di entries vengono chiamati bookies

Le **entries** di Bookkeeper sono una sequenza di bytes scritte nei ledgers, dove ogni entry ha come campi:

- **Ledger number**
  L'ID del ledger sulla quale è stata scritta la entry.
- **Entry number**
  L'ID che identifica la entry.
- **Last confirmed (LC)**
  L'ID dell'ultima entry che è stata registrata.
- **Data**
  I dati presenti all'interno della entry, scritti dal client ovvero dall'applicazione.
- **Authentication code**
  Il codice di autenticazione del messaggio.

I **ledgers** sono l'unità base di archiviazione in Bookkeeper, e contengono una sequenza di entries. Le entries in Bk vengono scritti in sequenza e al massimo una sola volta. Questo significa che i ledgers hanno una semantica di sola aggiunta ovvero quando una entry viene scritta non può più essere modificata.

La **gestione dell'ordine di scrittura** è responsabilità delle applicazioni **client** che hanno la possibilità di creare e di cancellare i ledgers, leggere le entry e scrivere nuove entries all'interno dei ledgers.

I **bookies** sono singoli server Bookkeeper che gestiscono i ledgers (Più precisamente, frammenti di ledgers). Un **bookie** è un singolo server di archiviazione e si occupa di conservare frammenti di ledgers.

Per ogni ledger L, un **ensemble** è un gruppo di bookies che hanno memorizzato l'entry L. Ogni volta che una entry viene scritto in un ledger, tale entry viene scritto in un sottogruppo di bookies anziché su tutti i bookies.

Bookkeeper richiede un **servizio di archiviazione dei metadati** per archiviare le informazioni relativi ai ledgers e ai bookies disponibili. Per gestire queste attività viene utilizzato **Apache Zookeeper**.

5.9.3 Gestione dei dati nei bookies

I bookies gestiscono i dati in modo strutturato per log, che è stato implementato basandosi su tre tipi di file:

- **Journals:**
  Un journal contiene i log di transazioni di Bookkeeper. Prima di un qualsiasi aggiornamento ad un ledgers , il bookie si assicura che la transazione sia scritta in uno storage non volatile. Viene quindi creato un file journal all'avvio del bookie o quando il file journal precedente raggiungere la dimensione massima.

- **Entry logs:**
  Un entry log gestisce le entries ricevute dai client Bookkeeper. Le entries di ledgers differenti vengono aggregati e scritti in sequenza, mentre i loro offset vengono mantenuti come puntatori ad un ledger cache in modo tale da poter effettuare delle ricerche rapide.

  L'entry log viene creato all'avvio del bookie o quando il file precedente raggiunge la dimensione massima configurata. I vecchi file log delle entries vengono rimossi dal thread del Garbage Collector una volta che non sono associati a nessun ledger attivo.

- **Index files:**
  Per ogni ledger viene creato un index file, che comprende un'intestazione e diverse pagine a lunghezza fissa che registrano gli offset dei dati memorizzati nelle entries logs. Questi files vengono aggiornati costantemente per garantire sempre buone prestazioni.

5.9.4 Aggiunta di entries

Quando un client istruisce un bookie di scrivere una entry in un ledger, vengono effettuati i seguenti passaggi:

- La entry viene aggiunta in sequenza all'entry log.
- L'indice della entry viene aggiornata nella ledger cache.
- Viene aggiunta al journal una transazione corrispondente al tipo di aggiornamento richiesto dal client.
- Viene mandato una risposta al client Bookkeeper.

Per ragioni di performance, l'entry log memorizza le entries in memoria e li raggruppa in bacth, mentre la ledger cache conserva le index pages in memoria e ogni tanto effettua un flush della cache. Il flush viene fatto solitamente in due casi:

- Quando viene raggiunto il limite di memoria della ledger cache, e quindi non c'è più spazio per contenere le index pages più recenti, vengono eliminate quelle sporche dalla cache ed scritti in maniera persistenti nell'index file di riferimento.
- Un thread sincrono in background periodicamente si occupa di effettuare un flush delle index pages dalla ledger cache alle index files.

5.9.4.1 Ledger manager

Un ledger manager gestisce i metadati dei ledgers che sono memorizzati in Zookeeper. Esistono due tipi di ledger managers:

- **Flat ledger manager:**
  Memorizza i metadati dei registri in nodi figlio di un singolo percorso Zookeeper. Il flat ledger manager crea dei nodi sequenziali per garantire l'unicità dell'ID del ledger.
- **Hierarchical ledger manager:**
  Hierarchical ledger manager ottiene innanzitutto un ID univoco globale da Zookeeper utilizzando un Znode di tipo Ephemeral-sequential. L'ID viene spezzato in 3 parti che costituiranno il percorso dove archiviare i metadati del ledger.

5.10 Apache Zookeeper

Apache Zookeeper è un servizio di coordinamento per applicazioni distribuite che consente la sincronizzazione di un cluster. Viene utilizzato per mantenere il

sistema distribuito funzionante come se fosse una singola unità, attraverso le sue funzionalità di sincronizzazione, serializzazione e coordinamento. Può essere pensato come un file system in cui abbiamo degli znodes che memorizzano dati.

5.10.1 Architettura

Zookeeper funziona come un servizio di sincronizzazione replicato con eventuale coerenza. E' robusto, poiché i dati persistenti sono distribuiti tra più nodi (chiamati ensemble) e il client può connettersi ad un server specifico migrando se quel nodo fallisce. Vige il concetto di master, ovvero che un nodo master viene scelto dinamicamente per consenso all'interno dell'ensemble; se il nodo principale ha esito negativo, il ruolo di master passa ad un altro nodo.

Il master è l'autorità per le scritture: in questo modo le scritture sono garantite come persistenti nell'ordine, ovvero sono lineari. Ogni volta che un client scrive nell'ensemble, i server collegati a quel client mantengono i dati insieme al master. Pertanto,la maggior parte dei server saranno sempre aggiornati sui dati. Tuttavia, questo rende impossibile eseguire scritture simultanee. La garanzia delle scritture lineari può essere problematica se Zookeeper viene utilizzato per un carico di lavoro dominante in scrittura. E' un servizio ideale per coordinare gli scambi di messaggi tra vari client, il che comporta una minoranza di scritture a favore di maggior letture.

Il modello dei dati di Zookeeper è simile ad un file system, denominato "Znode". L'unica differenza rispetto ad un file system è che i Znodes hanno anche la capacità di archiviare dati. Il loro compito è quello di mantenere le statistiche della struttura e i numeri di versione per le modifiche ai dati. Esistono vari tipi di Znodes quali:

- **Znodes persistenti:**
  E' il Znode predefinito in Zookeeper. Può essere eliminato solo chiamando la funzione di eliminazione di Zookeeper.

- **Znodes effimeri (Ephemeral Znode):**
  I dati vengono eliminati non appena la sessione viene chiusa o la connessione viene chiusa dal client.

- **Znodes sequenziali:**
  Ad un Znode sequenziale viene assegnato un intero unico crescente. Questo numero viene aggiunto al percorso utilizzato per creare lo Znode. Ov-

vero se il client crea uno Znode sequenziale con il percorso ”/x/x-”, Zookeeper assegna un numero di sequenza, ad esempio 1, e lo aggiunge al percorso. Avremo quindi ”/x/x-1”. Questi tipi di Znode forniscono un modo semplice di creare Znodi con nomi univoci. Offrono anche un modo più immediato per vedere l'ordine di creazione dei Znodi.

Ogni volta che il client deve sapere se qualcosa è cambiato, accede al Znode. Essendo molto costoso in termini di risorse e latenza, per ridurre il polling, Zookeeper lavora sul modello di pubblica iscrizione, in cui ogni Znode pubblica delle API e il client può iscriversi. Ogni volta che viene modificato il contenuto del Znode, tutti i client iscritti a tale API riceveranno una notifica.

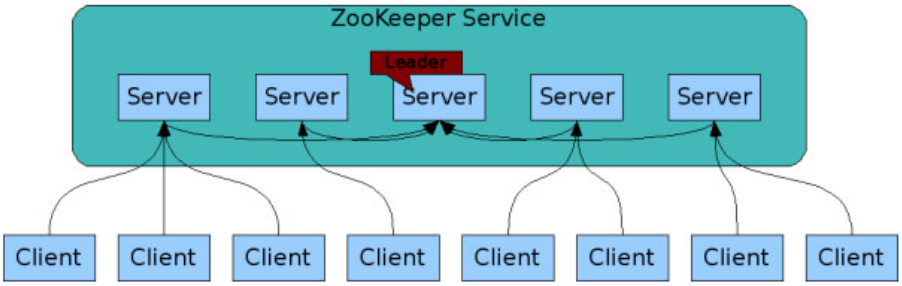

Figura 5.11: Architettura di Zookeeper

Capitolo 6

Consistenza e replicazione nei sistemi distribuiti

6.1 Consistenza

Sappiamo che alcune delle caratteristiche fondamentali che deve avere un database sono Atomicità, Consistenza, Isolamento e Durabilità (ACID) [25]:

- **Atomicità:**
  Significa che la transazione è atomica, Ciò significa che, nel caso di errori, questa deve essere annullata come se non fosse stata mai iniziata.

- **Consistenza:**
  Il sistema prima di iniziare una transazione si trova in uno stato consistente, e al termine della transazione il sistema deve ricondursi ad uno stato sempre consistente. Quindi possiamo immaginarlo come una macchina a stati finiti con uno stato iniziate ”consistente” dove ogni transazione al suo termine deve ricondursi sempre a tale stato.

- **Isolamento:**
  Le transazioni devono essere uno indipendente dall'altro, ovvero che l'eventuale fallimento di una transazione non deve influire sulle altre transazioni che sono in esecuzione nello stesso lasso di tempo.

- **Durabilità:**
  Ogni volta che una transazione richiede un ”commit”, i cambiamenti che sono stati attuati non dovono essere più persi. Quindi per evitare perdite dei dati vengono utilizzati dei registri di log dove vengono scritti tutte le operazioni sul DB.

Con la diffusione dei database distribuiti, che hanno un architettura più complessa e difficile da gestire dato la presenza di più nodi, si è reso necessario trovare un altro approccio differente dal ACID. E' stato dunque introdotto da **Eric Brewer** un nuovo teorema denominato **CAP**. Il teorema di CAP coinvolge i concetti di consistenza, disponibilità e tolleranza di partizioni, che sono caratteristiche desiderabili da parte di un sistema, dalla sua progettazione alla sua implementazione. Brewer afferma che per un sistema informatico a calcolo distribuito è impossibile garantire simultaneamente tutte e tre le caratteristiche di:

- **Consistenza:**
  Nei sistemi distribuiti significa che tutti i client visualizzano gli stessi dati, indipendentemente dal nodo in cui si connettono. Questo significa che ogni volta che i dati vengono scritti su un nodo, devono essere replicati su tutti gli altri nodi del sistema.

- **Disponibilità:**
  Significa che qualsiasi client che effettua una richiesta di dati ottiene una risposta, anche se uno o più nodi sono inattivi.

- **Tolleranza di partizione:**
  Per partizione si intende un interruzione delle comunicazioni all'interno di un sistema distribuito, come per esempio: una connessione persa o temporaneamente ritardata tra due nodi. Un sistema distribuito o cluster per essere tollerante alla partizione deve continuare a funzionare nonostante un numero qualsiasi di interruzioni della comunicazione tra i nodi del sistema.

Abbiamo dunque che i database distribuibili (NoSQL) vengono classificati in base alle due caratteristiche che supportano:

- **Database CP:**
  Un database CP offre consistenza e tolleranza alle partizioni a spese della disponibilità. In questo caso quando si verifica una partizione tra due nodi, il sistema deve arrestare il nodo non coerente fino a quando la partizione non viene risolta.

- **Database AP:**
  Un database AP offre disponibilità e tolleranza alle partizioni a scapito della consistenza. Quando si verifica una partizione, tutti i nodi rimangono disponibili, ma alcuni nodi potrebbero restituire una versione precedente dei dati rispetto ad altri. Quando la partizione viene risolta, i database AP sincronizzano i nodi per riparare tutte le incoerenze nel sistema.

- **Database CA:**
  Un database CA offre coerenza e disponibilità su tutti i nodi. Non può farlo se esiste una partizione tra due nodi nel sistema, e dunque non è in grado di fornire una tolleranza agli errori. Nei sistemi distribuiti è difficile avere un database CA, questo perché il partizionamento non può essere evitato. Tuttavia molti database relazionali, come PostegreSQL, offrono coerenza e disponibilità e possono essere distribuiti su più nodi mediante replica.

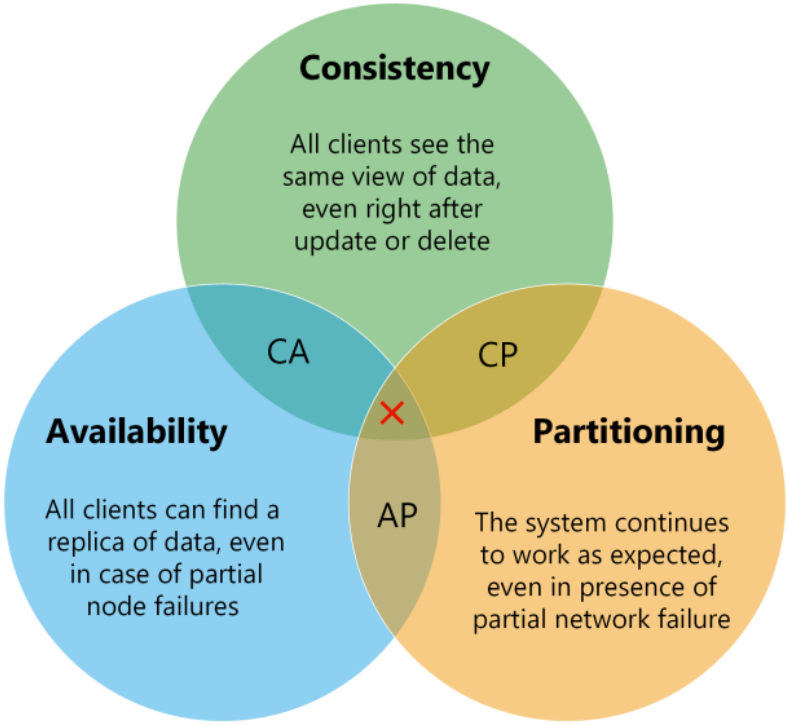

Figura 6.1: Architettura teorema CAP

Essendo il partizionamento un evento non eliminabile, bisogna fare molteplici considerazioni quando si va a scegliere su quali aspetti dare priorità. Se abbiamo a che fare con dei dati che devono essere sempre aggiornati e allineati, anche in caso di un errore di rete allora è necessario avere coerenza (Database CP). Un esempio di caso d'uso è quando si ha a che fare con informazioni finanziarie o informazioni personali. Se invece l'accumulazione dei dati è una priorità allora possiamo fare affidamento ad un DB che garantisce disponibilità (Database AP).

Un tipico esempio è quando si vogliono acquisire i dati comportamentali o le preferenze di un utente. In questi scenari, l'obiettivo sarà quello di acquisire quante più informazioni possibili su ciò che l'utente sta facendo, ma non è fondamentale che il database sia costantemente aggiornato. Deve essere semplicemente accessibile e disponibile anche quando le connessioni di rete non funzionano.

6.2 Replicazione

Se si sceglie di puntare su un DB di tipo CP, allora i dati devono essere replicati su tutti i nodi del sistema distribuito. La replicazione consiste semplicemente nel copiare i dati da un nodo ad un altro nodo, in modo che gli utenti possano accedere agli stessi dati senza alcuna incoerenza. Questa azione comprende anche la duplicazione delle transazioni in real-time, in modo che la replica sia costantemente aggiornata e sincronizzata con l'origine. Le replica dei dati può essere completa , ovvero l'intero database è archiviato in un tutti i nodi o parziale, in cui alcuni frammenti del database utilizzati di frequente vengono replicati e altri non replicati.

6.2.1 Ragioni della replicazione dei dati

La replicazione è un processo che può essere molto utile alle realtà che lavorano con grandi quantità di dati. Uno uso comunque della replica è per il ripristino di emergenza, per garantire che esista sempre un backup accurato in caso di catastrofe, gusto hardware o violazione del sistema in cui i dati vengono compromessi. Avere una replica può anche rendere più rapido l'accesso ai dati, in quanto ogni replica può essere collocata in più aree geografiche in modo tale che l'utente si colleghi sempre alla base dati più vicina, abbattendo i costi di latenza.

6.2.2 Problemi nei sistemi distribuiti

Nonostante l'alta affidabilità e le grandi prestazioni che offre la replicazione, ci sono comunque delle sfide da sostenere per mantenere dati coerenti, quali:

- **Costo della maggiore larghezza di banda:**
  Una rete spesso ha un gran numero di messaggi che fluiscono quando gli utenti si connettono per modificare/leggere o cancellare i dati. Pertanto, la replica dei dati può innalzare i costi di banda.

- **Maggiori requisiti di archiviazioni:**
  Avere più coppie di dati, porta a maggiori costi di archiviazione. Lo spazio richiesto diventa multiplo rispetto ad un sistema centralizzato.

- **Consistenza:**
  Un dei aspetti più critici nell'implementazione di un database basato sulla replica, e quello di riuscire garantire la consistenza dei dati. Per garantire performance e alta affidabilità, le repliche devono essere mantenute sempre in uno stato consistente. Questo implica l'implementazione di un modello di consistenza efficiente, ovvero richiede tecniche e protocolli di sincronizzazione molto complessi.

Capitolo 7

Progetto

Dopo una larga introduzione sui sistemi distribuiti, dei loro principali vantaggi,svantaggi e dopo aver studiato in maniera approfondita i problemi legati alla replicazione e consistenza dei dati, in questo capitolo andremo a studiare come herdDB cerca di garantire la coerenza dei dati utilizzando Bookkeeper e Zookkeeper. Andremo inoltre a proporre un soluzione per eventualmente verificare che i dati presenti sul nodo leader e sui follower siano effettivamente coerenti al 100%.

Ricordiamo che Apache Bookkeeper è un sistema di archiviazione di alto livello progettato per offrire garanzie di durabilità, coerenza e bassa latenza. Inoltre è un servizio scalabile orizzontalmente e quindi a tolleranza d'errore, ottimizzato per carichi di lavoro in tempo reale. Il nodo leader di herdDB esegue il writer di Bookkepper e crea un ledgers . Il ledgers può essere aperto una sola volta e può essere letto tutte le volte che si desidera. Quando il leader finisce di scrivere sul ledgers, lo chiude e quindi non è più possibile riaprirlo in scrittura. Al momento della creazione del ledger, vengono settati tre parametri:

- **Ensemble size (ES):**
  Questo indica il numero di bookie in cui devono essere memorizzati i dati dei ledgers.

- **Write quorum size (WQ):**
  Il numero di coppie per ogni entry.

- **Ack quorum size (AQ):**
  Il numero richiesto di ack utili per considerare la scrittura conclusa con successo.

Se per esempio creiamo un cluster herdDB di 3 macchine (quindi Ensemble size=3) la configurazione ideale sarebbe avere WQ=2, AQ=2. Avremo quindi che ogni scrittura viene replicato due volte (ovvero scritto su almeno due bookie), e per considerare successo la scrittura ci devono essere almeno 2 ack. Avere EQ maggiore di WQ aiuta a migliorare le prestazioni poiché permette di diffondere le scritture e le letture su più bookies. Considerando la nostra configurazione di esempio, se un nodo viene perso avremo comunque due nodi disponibili su cui replicare i dati. In caso di fallimento di un bookie, lo scrittore sceglie un nuovo ensemble (detto anche cambio di gruppo) su cui scrivere i dati, in maniera totalmente trasparente e finché ci sono abbastanza bookies il sistema continuerà a funzionare correttamente. Ad ogni cambio di gruppo, viene iniziato un nuovo segmento di ledger, e i lettori in base ai cambiamenti sui metadati (basato di Zookeeper) sono in grado di connettersi automaticamente ai nuovi bookies. Nel nostro modello basato sulla replicazione abbiamo un solo leader a cui è consentito modificare lo stato di una tabella. Il suo ruolo di leadership deve essere supportato dagli altri nodi del sistema. Per l'elezione del leader vengono sfruttati le funzionalità di zookeeper.

L'uso di Zookeeper non è sufficiente a garantire la coerenza complessiva dei dati, in quanto durante l'elezione, un vecchio leader non ancora morto potrebbe generare delle modifiche concorrenti che porterebbero ad un inconsistenza dei dati. Il ruolo di Bookkeeper in questo situazione risulta fondamentale. Quando un nodo inizia ad agire come un leader, esegue una lettura di recupero dei dati su ogni ledger aperta dal precedente leader. Questa operazione si connette ad ogni bookie che contiene i dati e li segna come fenced (recintati) ed il leader precedente (se risulta ancora vivo) riceve degli errori di scrittura alla sua operazione di modifica successiva. Bookkeeper gestisce tutti i casi, come errori di rete durante il ripristino o concorrenza durante le operazioni. Abbiamo la garanzia che solo una macchina riuscirà nel recupero e quindi potrà iniziare ad applicare le modifiche apportate allo stato del database. Per la gestione dell'elenco dei ledger attivi/innativi e dello stato dei nodi del cluster, viene fatto affidamento a zookeeper.

7.1 Descrizione del problema

Bookkeeper è ottimizzato per un throughput elevato, ma le caratteristiche più importanti riguardano le garanzie di coerenza basato principalmente sulla gestione dei metadati tramite zookeeper e i meccanismi di fencing integrato. Uno dei punti critici è quando devi definire la serie effettiva di voci valida per i lettori,

in particolare l'id dell'ultima voce. Ci sono alcuni meccanismi che entrano in gioco, perché lo scrittore potrebbe fallire e anche la rete potrebbe fallire. Bookeeper garantisce un meccanismo di recupero che permettono di sigillare questo range di entry id validi. Questa operazioni viene chiamata "chiusura del ledger" e consiste nel scrivere su Zookeeper lo stato finale del ledger. Dopo la chiusura, i lettori saranno in grado di leggere l'ultima voce scritta. Ciò permette di mantenere la consistenza dei dati durante le scritture anche in caso di problemi. Questa gestione permette a HerdDB di mantenere un ordine coerente quando un leader effettua una scrittura su bookeeper, e quest'ultimo si occuperà del resto. L'altro punto critico riguarda sopratutto la gestione del follower su herdDB, nonostante Bookkeper ci garantisce coerenza, non abbiamo nessuna certezza che i follower stiano effettivamente applicando correttamente le modifiche. Il follower durante il download dei dati potrebbe perdersi qualche dato a causa della latenza della rete (come la perdita di qualche pacchetto), o potrebbe essere stato un leader precedente e avere ancora in memoria un istruzione che non era riuscito precedentemente a scrivere su disco. Ci sono numerose problematiche che potrebbero portare ad un inconsistenza nelle repliche. La replica è sempre un punto difficile da gestire, poiché potrebbe non corrispondere alla verità.

7.2 Obiettivo

Se riusciamo a capire quando il follower non è in uno stato consistente, potremmo implementare delle misure di recovery automatizzate che permettano di risolvere le incoerenze in maniera poco costosa (sempre se questo è possibile). Il nostro obbiettivo è quello di sviluppare un sistema che ci permetta di verificare il reale stato dei follower. In particolare se effettivamente stanno applicando le modifiche in maniera corretta. Nessuna applicazione è priva di bug, se i follower sono in uno stato inconsistente questo potrebbe essere sintomo di una gestione errata della replica. Se i dati della replica non corrispondono alla verità, avremmo violato alcuni degli aspetti fondamentali della consistenza.

7.3 Analisi

In questa fase dobbiamo analizzare l'effettiva fattibilità della feature. Uno degli aspetti importanti è quello valutare attentamente l'impatto che potrebbe avere la feature in un ambiente di produzione con milioni da dati. Ovvero, quanto può essere costoso effettuare un full tablescan periodico? Che sistema possiamo adottare per non impattare troppo sulle attività principali del DB? A seguito di

queste domande sarebbe opportuno studiare come viene gestita questa attività nei database più famosi basati sulla replica quali Mysql o Hbase. La creazione della checksum comporta inevitabilmente dover effettuare un full table scan, che consiste nel selezionare l'interno set di dati di una tabella. Finché la quantità dei dati non è eccessiva, probabilmente l'impatto di questo scan non sarebbe né evidente né particolarmente bloccante per le attività del database. Dal momento in cui iniziamo a lavorare con una quantità di dati che si aggira attorno ai milioni, uno scan potrebbe essere molto impattante, tenendo occupato il database per diverso tempo. Inoltre è bene evidenziare che nel momento in cui dovremo effettuare il calcolo del checksum, dovremo bloccare tutte le scritture. Questa ha come conseguenza, che se teniamo occupato il DB per 20 minuti, avremo 20 minuti in cui le possibili attività di scrittura (che potrebbero riguardare acquisizione dei dati o altro) non saranno permessi.

7.3.1 Approccio Hbase

Hbase fornisce il comando **hbck** per verificare incongruenze (inconsistenze) tra le sue tabelle. **hbck** deriva dal comando **fsck** di HDFS, utilizzato per verificare le inconsistenze di quest'ultimo. Principalmente il comando verifica la coerenza tra gli stati in memoria del master e dei server della regione e lo stato dei dati in HDFS. Se viene lanciato il comando senza parametri, viene fatto un check di inconsistenza in tutto il cluster. Se non viene rilevata nessuna anomalia, viene riportato l'output "Status ok", in caso contrario viene riportare inconsistenza rilevata.

7.3.2 Approccio Mysql

A differenza di HerDB o Hbase, MySQL non è un database NoSQL ma è un database SQL. La versione cluster è stato accuratamente progettata per offrire una disponibilità del 99,999% garantendo la resilienza ai guasti e la capacità di eseguire la manutenzione programmata senza tempi di inattività. Nonostante non sia un database NoSQL, la sua architettura cluster è simile a quello di HerDB, ovvero che abbiamo un nodo attivo che si occupa di applicare le modifiche ai metadati, e gli altri nodi (passivi) sono delle repliche. Questo implica che deve mantenere un grado di consistenza tra tutti i nodi del cluster. Egli mette a disposizione il comando **mysqldiff** che ci consente di trovare delle incoerenze verificando le differenze tra le tabelle sullo stesso server (database diversi) o su server diversi. In quest'ultimo caso il secondo server potrebbe essere una replica

che contiene effettivamente le stesse tabelle. Con questo comando è possibile verificare se una tabella è stata replicata correttamente.

```
mysqldiff --server1=user@host1 --server2=user@host2
   test:test
```

Nell'esempio viene fatto la differenza del database test che si trova nel server1 con quello che si trova nel server2.

7.3.3 Approccio herdDB

Al di là delle considerazioni fatti su Hbase e Mysql, un idea potrebbe essere quello di sfruttare le attività di checkpoint periodiche che vengono fatti su HerdDB. Può essere un momento ideale per la creazione del checksum dato che durante il checkpoint vengono bloccati tutte le scritture e non sarebbero quindi impattanti per l'attività. Essendo però il checkpoint bloccante (dato che potrebbe richiedere diversi minuti per completarsi) aggiungerci la creazione del checksum aumenterebbe ulteriormente il costo di queste attività (in termini di tempo). Se riprendiamo le considerazioni fatti inizialmente, l'idea non sembra percorribile in quanto potrebbe occupare il DB per troppo tempo. Un altra soluzione è di introdurre lo stesso concetto di Hbase e Mysql, mettendo a disposizione un comando che permetta di verificare la consistenza di una tabella replicato su server diversi. In questo caso, l'uso del comando sarebbe a discrezione del amministratore del sistema, che con opportune valutazioni deciderebbe il momento ideale per utilizzarlo. Possiamo quindi contrassegnare un punto nel tempo (che corrisponde ad una posizione nel log delle transazioni di HerdDB) e forzare il leader e ciascun follower a creare un checksum per ogni tabella e scrivere il risultato su una tabella di sistema o sul log delle transazioni. Il follower al momento della creazione del suo checksum potrebbe in tal caso, fare un confronto del suo checksum con quello del leader e se non corrispondono avremo una chiara situazione di incoerenza.

7.4 Progettazione

In base agli aspetti che abbiamo analizzato nella fase di analisi, e dopo avere fatto le eventuali considerazioni sulla fattibilità possiamo definire in maniera più dettagliata come può essere implementata la feature. In linea massima l'implementazione deve prevedere due comandi che permettano di creare rispettivamente il checksum di una tabella e di una lista di tabelle appartenenti ad uno stes-

so tablespace. Dobbiamo quindi introdurre i comandi **tableconsistencycheck** e **tablespacesconsistencycheck**. Il primo dovrà essere in grado di generare il checksum a partire da una tabella specificata, per esempio:

```
tableconsistencycheck tablename
```

Il secondo dovrà permettere di creare il checksum di una lista di tabelle di un tablespace, ignorando le tabelle di sistema. Questo ci permetterà di lanciare un comando unico qualora vogliamo analizzare tutte le tabelle di un tablespace:

```
tablespacesconsistencycheck tablespacename
```

Quando viene generato il checksum, dobbiamo scrivere il risultato sul log delle transazioni, tenendo conto anche delle informazioni quali: nome del tablespace, nome della tabella, il checksum, il tempo di esecuzione,il numero di record processati e la query utilizzata per lo scan della tabella.

7.4.1 Modelli

La classe CalcitePlanner è il punto d'ingresso delle query, dove vengono mappate a inviate alla classe DDLPlanner.

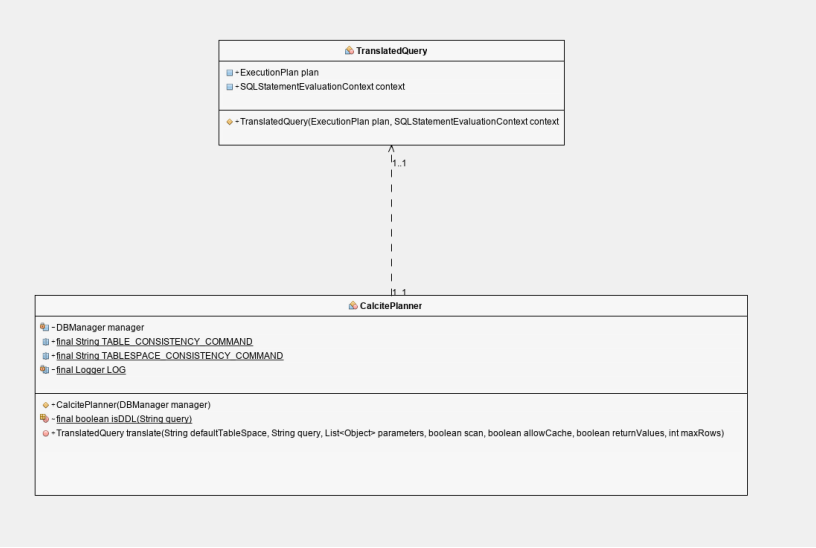

Figura 7.1: Mappatura delle query ricevute via CLI

La Classe DDLPlanner in base alla query ricevuta esegue lo statement corrispondente. Nel nostro caso verrà eseguito lo statement **queryConsistencyCheckStatement**. Quest'ultimo è divisa in due classi: la prima accetta una lista di parametri, in cui può essere specificato oltre al tablespace anche la tabella da analizzare. Mentre la seconda accetta una query che contiene solo il nome di tablespace.

Figura 7.2: Gestione della traduzione delle query SQL

Il DBManager in base allo statement ricevuto, invoca il metodo di creazione e scrittura della digest del TableSpaceManager, passandogli i parametri opportuni, quali tablespace e tabella.

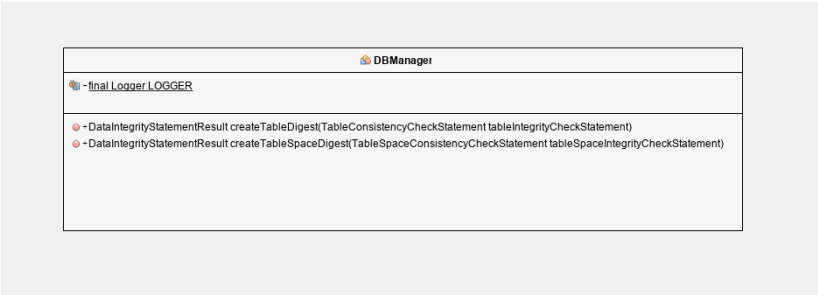

Figura 7.3: Esecuzione degli statement per la creazione del checksum

Queste due classi rappresentano gli statement dei comandi **tableConsistencyCheck** e **tableSpaceConsistencyCheck**. Il primo accetta come parametro il nome di un tablespace, di cui verrà creato un checksum per ogni tabella. Il secondo oltre al tabelespace neccessita il nome di una tabella, di cui verrà generato il checksum.

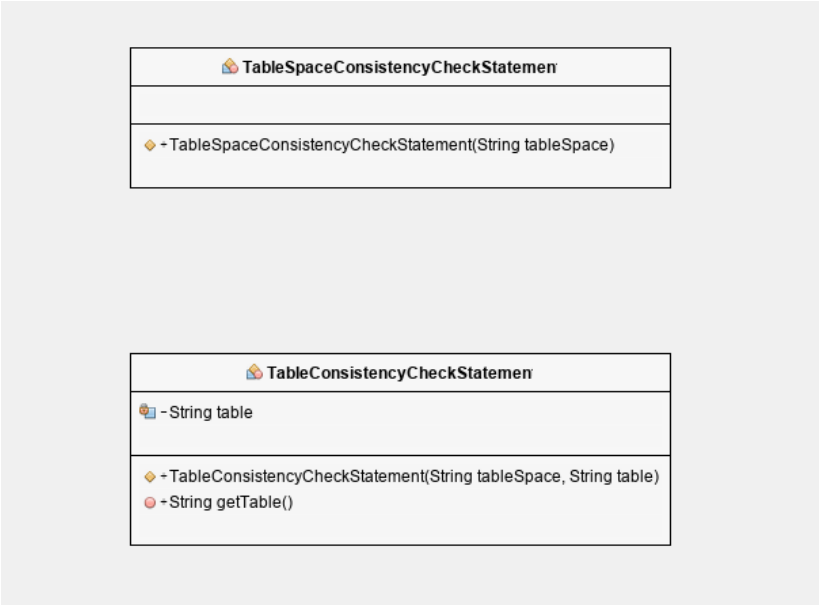

Figura 7.4: Statement

Il TableSpaceManager attraverso l'algoritmo di creazione delle tabella si occupa di creare il checksum e di scriverlo sul log delle transazioni, attraverso il metodo **createAndWriteDigest**. Il metodo **apply** viene eseguito sia dal follower che dal leader. Se il nodo è un follower dovrà applicare la transazione scritta precedentemente dal leader.

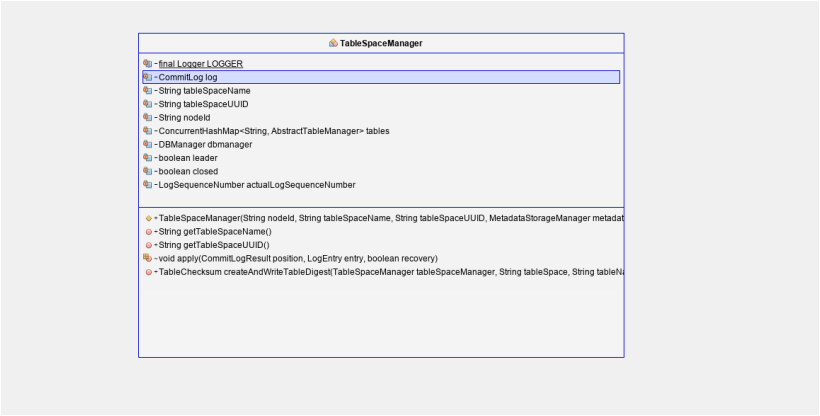

Figura 7.5: Creazione e scrittura del checksum sul log delle transazioni

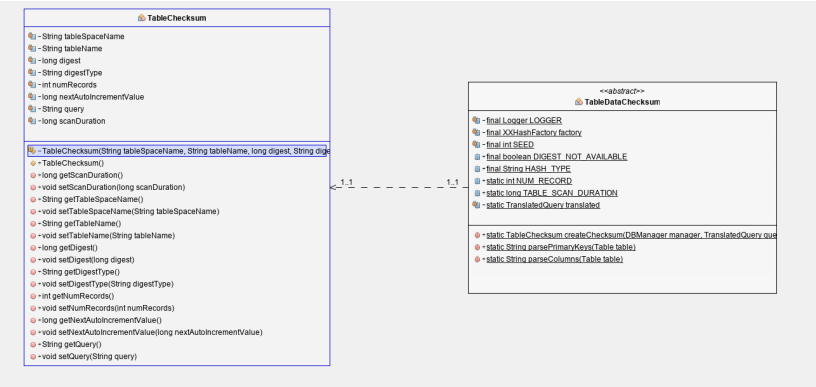

Figura 7.6: Gestione dell'algoritmo per la creazione del Checksum utilizzato dal TableSpaceManager

7.5 Implementazione

7.5.1 Creazione del checksum

Prima di effettuare l'implementazione è stato prima scelto quale algoritmo per la creazione del checksum utilizzare. La scelta è stata presa basandosi su una serie di benchmark pubblicati su github. Abbiamo deciso di adottare l'algoritmo xxHash64, poiché dai bechmarck è risultato l'algoritmo più efficiente e più veloce:

Name	Speed	Quality	Author
xxHash	5.4 GB/s	10	Y.C.
MurmurHash 3a	2.7 GB/s	10	Austin Appleby
Sbox	1.4 GB/s	9	Bret Mulvey
Lookup3	1.2 GB/s	9	Bob Jenkins
CityHash64	1.05 GB/s	10	Pike & Alakuijala
FNV	0.55 GB/s	5	Fowler, Noll, Vo
CRC32	0.43 GB/s	9	
MD5-32	0.33 GB/s	10	Ronald L.Rivest
SHA1-32	0.28 GB/s	10	

Tabella 7.1: Benchmark algoritmi di hashing. Benchmark eseguiti su un sistema Windowns 7 a 32-bit su un sistema Core 2 Duo @3GHz

Il benchmark è stato eseguito utilizzando la suite di test SMhasher che valuta la qualità di collisione, dispersione e qualità delle funzioni hash. Il punteggio di qualità 10 è il punteggio massimo che indica un algoritmo di ottima qualità. Come vediamo dalla tabella, l'algoritmo xxHash ha una velocità nettamente superiori agli altri con 5.4 GB/s.

xxHash offre due versioni: Una versione a 64 bit che offre velocità e dispersioni superiori per i sistemi a 64 bit e una versione a 32 bit più efficiente sui sistemi a 32 bit.

Version	Speed on 64-bit	Speed on 32-bit
XXH64	13.8 GB/s	1.9 GB/s
XXH32 3a	6.8 GB/s	6.0 GB/s

Tabella 7.2: Confronto xxHash64 e xxHash32. Compilanto usando GCC 4.8.2 su Linux Mint 64-bit su un sistema Core i5-3340M @2.7GHz

Per verificare la coerenza dei dati, per ogni tabella viene creato un checksum. Il calcolo del checksum viene eseguito partendo dalla costruzione di una query SQL che effettua un full-table scan della tabella.

```
translated = manager.getPlanner().translate(tableSpace,
   "SELECT "
      + columns
      + " FROM "+ tableName
      + " order by "
      + parsePrimaryKeys(table) ,
         Collections.emptyList(), true, false, false, -1);
```

Per essere sicuri che i dati siano effettivamente nello stesso ordine in tutti i nodi, al momento della creazione del checksum, vengono ordinati per **primaryKey**. Come vediamo nella query, viene fatto una selezione delle colonne della tabella, poiché se facciamo un "select *" non abbiamo nessuna garanzia di come vengono presi i campi. La fase di creazione del checksum viene fatto nel seguente modo:

- Ogni riga della tabella viene consumata, senza dover tenere effettivamente in memoria l'intero set di dati.
- Per ogni riga consumata, si va a serializzare ogni record con il relativo tipo.
- Per ogni riga serializzata viene aggiornato il checksum.
- Al termine avremo un checksum associato a quella specifica tabella.

```
try ( DataScanner scan = manager.scan(statement,
   translated.context,
   TransactionContext.NO_TRANSACTION);){
   StreamingXXHash64 hash64 =
      factory.newStreamingHash64(SEED);
   byte[] serialize;
   long _start = System.currentTimeMillis();

   while(scan.hasNext()){
      NUM_RECORD++;
      DataAccessor data = scan.next();
      Object[] obj = data.getValues();
      Column[] schema = scan.getSchema();
      for(int i=0; i< schema.length;i++){
         serialize =
            RecordSerializer.serialize(obj[i],schema[i].type);
         hash64.update(serialize, 0, SEED);
      }
   }
```

7.5.2 Costruzione query sql

Per avviare il processo della creazione della query, sono stati predisposti due comandi sql **tableconsistencycheck** e **tablespaceconsistencycheck**. La traduzione dei comandi avviene tramite **apache calcite**. Apache calcite è un framework di gestione dei dati dinamico. Contiene dei componenti che compongono un tipico sistema di gestione del database, ma omette alcune funzioni chiave: archiviazione e algoritmi per l'elaborazione dei dati e un repository per l'archiviazione dei metadati. Non contiene un database ma utilizza set di dati in memoria e li elabora utilizzando operatori come groupBy e join.

Quando calcite riceve una query contenente il comando **tableconsistencycheck** o **tablespaceconsistencycheck**, chiama il metodo **rewriteExecuteSyntax(query)** che accetta come paramento la query e ritorna la query sql se corretta.

```
if(query.startsWith(TABLE_CONSISTENCY_COMMAND) ||
   query.startsWith(TABLE_CONSISTENCY_COMMAND.toLowerCase())){
   query = DDLSQLPlanner.rewriteExecuteSyntax(query);
```

```
    return fallback.translate(defaultTableSpace, query,
        parameters, scan, allowCache, returnValues,
        maxRows);
}

if(query.startsWith(TABLESPACE_CONSISTENCY_COMMAND) ||
    query.startsWith(TABLESPACE_CONSISTENCY_COMMAND.toLowerCase())){
    query = DDLSQLPlanner.rewriteExecuteSyntax(query);
    return fallback.translate(defaultTableSpace, query,
        parameters, scan, allowCache, returnValues,
        maxRows);
}
```

I due comandi hanno le seguenti caratteristiche:

- **tableconsistencycheck:**
  Accetta come parametro il nome di una tabella. Per la tabella deve essere specificata il nome del tablespace. Se non viene specificato il tablespace viene utilizzato il tablespace di default **herd**. Genera il checksum relativo alla tabella passato come parametro.

  ```
  tableconsistencycheck tablespace.table
  ```

- **tablespaceconsistencycheck:**
  Accetta come parametro il nome di un tablespace. Per il tablespace specificato viene creato il checksum di ogni tabella contenuta nel tablespace eccetto le tabelle di sistema. Se non viene specificato nessun tablespace, viene restituito errore.

  ```
  tablespaceconsistencycheck tablespace
  ```

7.5.3 Gestione del checksum tra leader e follower

La creazione del checksum viene effettuata inizialmente sul tablespace master. Quando viene lanciata la query, viene chiamata la classe **herddb.data.consistency.TableDataChecksum** che ritorna un oggetto **TableChecksum** contenente i parametri:

- **tableSpaceName:**
  Il nome del tablespace specificato nel lancio del comando.

- **digest:**
  il checksum generato da xxhash64.
- **digestType:**
  il tipo di hash utilizzato. Nel nostro caso ”StreamingXXHash64”.
- **numRecords:**
  Il numero di records elaborati per creazione del checksum.
- **nextAutoIncrementValue:**
  l’ID successivo all’ultimo ID della tabella.
- **query:**
  La query che è stata eseguita.
- **scanDuration:**
  Il tempo impiegato per la creazione del checkum.

L’oggetto **TableChecksum** viene convertito in byte array e salvato sul log delle transazioni.

```
TableChecksum scanResult =
   TableDataChecksum.createChecksum(tableSpaceManager.getDbmanager(),
translated,tableSpaceManager, tableSpace, tableName);
ObjectMapper mapper = new ObjectMapper();
byte[] serialize = mapper.writeValueAsBytes(scanResult);
Bytes value = Bytes.from_array(serialize);
LogEntry entry =
   LogEntryFactory.dataIntegrity(tableName,0, value);
pos=log.log(entry, false);
apply(pos, entry, false);
```

Il follower quando arriva il arriva alla stessa posizione del master, eseguirà la query presente nell’oggetto **TableChecksum** e poi farà il confronto del suo checksum con quello presente all’interno all’oggetto (che è appunto il checksum del leader). La condizione necessaria per passare il check è che il follower abbia esattamente lo stesso checksum è il numero di records processati del leader.

```
case LogEntryType.TABLE_CONSISTENCY_CHECKSUM: {
try {
   TableChecksum check = new
      ObjectMapper().readValue(entry.value.to_array(),
      TableChecksum.class);
```

```
    String tableSpace = check.getTableSpaceName();
    String query = check.getQuery();

    //In recovery mode the follower will have to run the
       query on the transaction log
    if(recovery){
       TranslatedQuery translated =
          manager.getPlanner().translate(tableSpace,query,
          Collections.emptyList(), true, false, false, -1)
          ;
       TableChecksum scanResult =
          TableDataChecksum.createChecksum(manager,translated,this,
          tableSpace, tableNanme);
       long followerDigest =scanResult.getDigest();
       long masterDigest = check.getDigest();
       int masterNumRecords = check.getNumRecords();
       int follNumRecords = scanResult.getNumRecords();
       long table_scan_duration = check.getScanDuration();
       //the necessary condition to pass the check is to
          have exactly the same digest and the number of
          records processed
       if(followerDigest == masterDigest &&
          masterNumRecords == follNumRecords ){
          LOGGER.log(Level.INFO, "Data consistency check
             PASS for TABLE {0} TABLESPACE {1} in {2} ms"
             , new
             Object[]{tableNanme,tableSpace,table_scan_duration});
       }else{
          LOGGER.log(Level.SEVERE, "Data consistency check
             FAILED for TABLE {0} in TABLESPACE {1} after
             {2} ms" , new
             Object[]{tableNanme,tableSpace,table_scan_duration});
       }
    }
}
break;
```

7.6 Performance e testing

Dopo l'implementazione dell'algoritmo sono stati effettuati una serie di test sia funzionali che di performance. Per quanto riguarda i test funzionali sono stati effettuati dei test manuali e predisposti dei test junit nel codice che verificano il corretto funzionamento dei comandi SQL **tableconsistencycheck** e **tablespaceconsistencycheck**. Per i test manuali, abbiamo fatto una build del progetto tramite maven. Lanciando il comando di maven, il progetto viene compilato e viene generato uno zip contenente il jar di herdDB con tutte le sue dipendenze. La build può essere fatta in due modi:

- Con i test, ovvero che il progetto viene compilato solo se tutti i test danno esito positivo:

```
mvn clean install
```

- Senza i test. In questo caso il pacchetto viene compilato senza eseguire i test.

```
mvn clean install -DSkipTests
```

HerdDB può essere eseguito in modalità standalone o in modalità replica. Nella modalità standalone i dati e i metadati e il journal risiedono sul disco locale sul server. Per avviare e gestire il servizio in modalità autonomo basta decomprimere il pacchetto distribuzione,effettuare le dovute configurazioni e lanciare il comando di avvio.

```
bin/service server start
```

Mentre in modalità replica è necessario impostare un cluster Zookeeper e fornire a HerdDB la stringa di connessione a Zookeeper. Successivamente basta impostare la modalità cluster su HerdDB e avviare il servizio.

Per i nostri test, in quanto non abbiamo disponibilità di più macchine faremo i test manuali e di performance su l'installazione standalone. Abbiamo utilizzato una macchina di google cloud con le seguenti caratteristiche:

- **Tipo macchina:** n1-standard-2 (2 vCPU, 7,5 GB di memoria)
- **Sistema operativo:** Centos 7

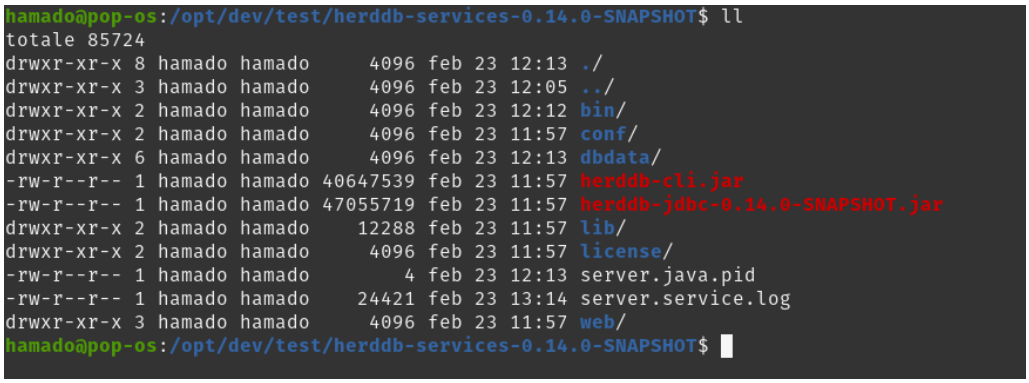

Figura 7.7: Esempio installazione standalone tipica di HerdDB

Per eseguire i comandi SQL, HerdDB mette a disposizione una cli. Per i test manuali, abbiamo creato un tablespace **testc** che contiene una tabella **test** nella quale sono stati inseriti alcuni records.

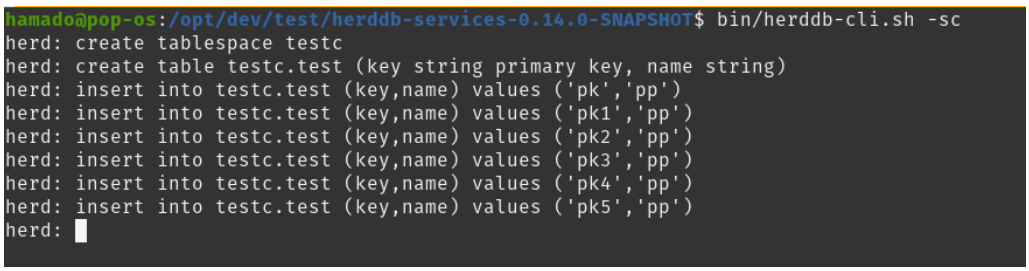

Figura 7.8: Predisposizione ambiente per testare i comandi SQL creati

Per il check della consistenza utilizziamo i due comandi che abbiamo a disposizione:

- **tableconsistencycheck:**
  Che genera il checksum di una tabella specificata.

```
tableconsistencycheck testc.test
```

- **tablespaceconsistencycheck:**
  Che calcola il checksum di tutte le tabelle di un tablespace.

```
tablespaceconsistencycheck testc
```

Il risultato delle digest oltre ad essere scritto sul log delle transazioni viene stampato anche come system out.

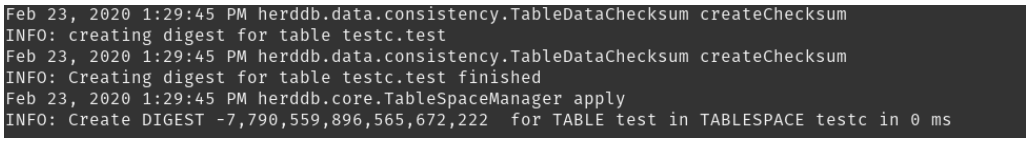

Figura 7.9: Output checksum dopo il lancio del comando

Per quanto riguarda i test di performance, abbiamo valutato con un numero crescente di record nella tabella, il tempo effettivo impiegato per creare il checksum. Per popolare il database con il numero di records desiderato abbiamo utilizzato il tools opensource **YCSB**.

YCSB principalmente è un tools utilizzato per verificare le prestazioni in lettura e scrittura di database NoSQL ed SQL. Per il nostro scopo è stato utilizzato per generare un serie di record casuali all'interno del database. Per eseguire il bench è necessario prima di tutto creare la tabella usertable su HerdDB:

```
CREATE TABLE usertable ( YCSB_KEY VARCHAR(191) NOT NULL,
   FIELD0 STRING, FIELD1 STRING, FIELD2 STRING, FIELD3
   STRING, FIELD4 STRING, FIELD5 STRING, FIELD6 STRING,
   FIELD7 STRING, FIELD8 STRING, FIELD9 STRING, PRIMARY
   KEY (YCSB_KEY))
```

Si creare il file herd.properties che fornisce a YCSB le specifiche per connettersi al database e si copia il JDBC nella cartella jdbc-binding:

```
db.driver=herddb.jdbc.Driver
db.url=jdbc:herddb:server:127.0.0.1:7000
db.user=sa
db.passwd=hdb

cp /PATHTOHERD/herddb-jdbc-*.jar /PATHTOYCSB/jdbc-binding/
```

Per i benchmark di performance del database, ycsb mette a disposizione una serie di workload che eseguono operazioni diverse sul database. Per il nostro scopo andremo solo a caricare i dati sul db, dunque per ogni giro di bench modifichiamo il workloada settando le variabili recordcount e operationcount:

```
# Yahoo! Cloud System Benchmark
# Workload A: Update heavy workload
#  Application example: Session store recording recent
   actions
#
#  Read/update ratio: 50/50
#  Default data size: 1 KB records (10 fields, 100 bytes
   each, plus key)
#  Request distribution: zipfian

recordcount=3000000
operationcount=3000000
workload=site.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.5
updateproportion=0.5
scanproportion=0
insertproportion=0

requestdistribution=zipfian
```

Per il caricamento dei dati si lancia il comando:

```
./bin/ycsb load jdbc -P workloads/workloada -P
   jdbc-binding/herd.properties -cp
   jdbc-binding/herddb-jdbc-*.jar -threads 200 -s >
   workloada_load.txt
```

Numero di record	Tempo (ms)
10000	88
20000	151
30000	201
40000	296
50000	295
100000	635
200000	1029
300000	1661
400000	2128
500000	3148
1000000	5175

Tabella 7.3: Tabella tempo di creazione della checksum

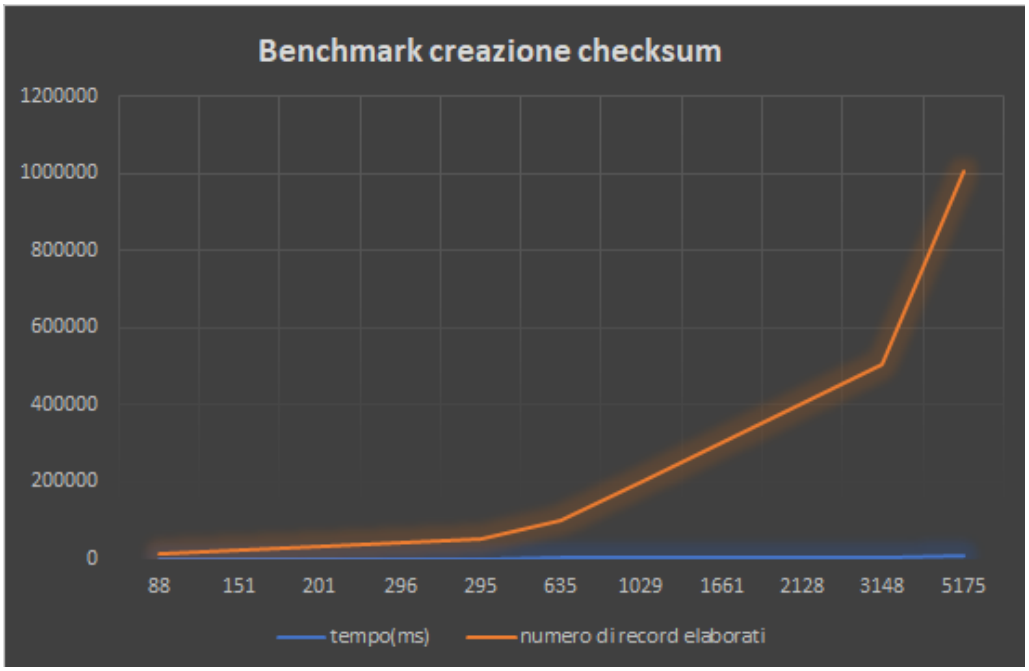

Figura 7.10: Grafico tempo di creazione della checksum

Capitolo 8

Conclusioni

La consistenza dei dati è un argomento molto critico quando si lavora con i sistemi distribuiti, in quanto risulta difficile scrivere un'applicazione senza un singolo punto di errore. Quando si effettua l'implementazione di un software, si cerca di prevenire eventuali problemi scrivendo dei test che possano riprodurre più o meno la situazione reale che si deve analizzare. Questo però non è abbastanza in quanto i problemi informatici possono essere di varia natura e a volte imprevedibili. Considerando banalmente due sistemi operativi diversi, a basso livello ognuno ha una gestione diversa dei suoi processi. Oltretutto una configurazione hardware o di rete diversa può avere un impatto sul comportamento del software dopo essere stato installato. Nell'ambito dei database distribuiti diventa ancora più critico in quanto i meccanismi che per uno sviluppatore sono difficili da prevedere si allargano ulteriormente.

Nell'ambito della consistenza, gli sviluppatori devono cercare di implementare un modello di consistenza adeguato ai requisiti che devono soddisfare. HerdDB in questo ambito implementa una replicazione di tipo "Eventual consistency". L'eventual consistency (consistenza alla fine) è un modello di consistenza in cui le repliche eventualmente diventeranno consistenti. Questo significa che la sincronizzazione non è immediata, e la replica potrebbe essere indietro di qualche records. Solitamente il tempo impiegato della replica per essere alla pari con il leader è quasi impercettibile. Esistono diversi modelli di consistenza che sono adatti ognuna a situazioni diversi. In questo progetto abbiamo implementato un algoritmo che permette di individuale le anomalie tra tra il nodo leader e le repliche che fanno parte del sistema distribuito. In futuro potrebbe essere allargata per correggere le eventuali incoerenza rilevate. L'individuazione potrebbe essere migliorata, trovando un sistema di analisi che non sia invadente per le altre attività del database, per esempio facendo lo

scan della tabella in blocchi ad intervalli regolari configurabili. La correzione potrebbe essere quello di forzare la replica a riscaricare i dati dal leader, in modo da ripartire da un situazione pulita.

Capitolo 9

Appendice

Ringraziamenti

Vorrei dedicare questo ultimo capitolo a tutte le persone che mi hanno sostenuto e che mi sono state vicino, rendendo possibile il raggiungimento di questo traguardo. Sono stati tre anni pieni di sfide e di sacrifici, ma che alla fine mi hanno fatto crescere sotto tanti punti di vista. Questo è un traguardo molto importante per me e per tutta la mia famiglia, qualcosa di unico che tempo fa sembrava solo un miraggio.

In primo luogo vorrei ringraziare il mio relatore Stefano Rizzi per la sua disponibilità nel prendere visione della mia tesi. Ringrazio la mia famiglia, che con tanti sacrifici mi ha portato in questo meraviglioso paese per darmi la possibilità di avere un futuro migliore. Ringrazio tutti i miei amici che mi hanno sostenuto e aiutato ogni singolo giorno. A Mattia che nonostante tutto, ha passato le notti a spiegarmi gli argomenti fatti a lezione. Alla la mia fidanzata Samira, che mi è stata sempre vicina, facendomi sorridere anche nei momenti più difficili. Ringrazio i miei colleghi di lavoro che mi hanno trasmesso la curiosità per questo argomento, in particolare Enrico Olivelli e Alessandro Luccaroni.

Infine ringrazio l'università di Bologna e tutti i docenti che hanno fanno parte di questo percorso trasmettendomi nuove conoscenze.

Bibliografia

[1] Enrico Olivelli. *HerdDB: Distributed Database*. URL: https://github.com/diennea/herddb.

[2] Enrico Olivelli. *HerdDB: Concept and architecture*. URL: https://github.com/diennea/herddb/wiki.

[3] Apache Software Foundation. *Bookkeeper: Concepts and architecture*. URL: https://bookkeeper.apache.org/docs/latest/getting-started/concepts/.

[4] Apache Software Foundation. *Zookeeper: Concepts and architecture*. URL: https://zookeeper.apache.org/doc/r3.5.7/zookeeperInternals.html.

[5] *Zookeeper: What is Apache Zookeeper?* URL: https://www.dezyre.com/article/zookeeper-and-oozie-hadoop-workflow-and-cluster-managers/216.

[6] *Zookeeper: Architecture overview*. URL: http://www.corejavaguru.com/bigdata/zookeeper/architecture.

[7] Apache Software Foundation. *Hbase: Concepts and architecture*. URL: https://hbase.apache.org/book.html#quickstart.

[8] Apache Software Foundation. *Cassandra: Concepts and architecture*. URL: http://cassandra.apache.org/doc/latest/.

[9] Cloudera. *Hbase: Checking consistency in Hbase Tables*. URL: https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/admin_hbase_hbck.html.

[10] Oracle. *Mysql: Identify Differences Among Database Objects*. URL: https://docs.oracle.com/cd/E17952_01/mysql-utilities-1.5-en/mysqldiff.html.

[11] Krishna Nadiminti, Marcos Assuncao e Rajkumar Buyya. «Distributed Systems and Recent Innovations: Challenges and Benefits». In: *InfoNet Magazine* 16 (gen. 2006).

[12] George Coulouris. *Distributed System Concepts and Desing*. Fifth volumes. Addison-Wesley.

[13] Devox. *Distributed Systems in One Lesson by Tim Berglund*. URL: https://www.youtube.com/watch?v=Y6Ev8GIlbxc.

[14] Tech target. *Distributed Database: Definition,Concepts and Architecture*. URL: https://searchoracle.techtarget.com/definition/distributed-database.

[15] DEV. *Distributed System: Vertical vs horinzontal scalability*. URL: https://dev.to/wfelix/scaling-up-vs-scaling-out-vertical-vs-horizontal-scalability-lc3.

[16] Tech target. *Database: Relational Database concepts and Architecture*. URL: https://searchdatamanagement.techtarget.com/definition/relational-database.

[17] Tech target. *Database: Not Only SQL database (NoSQL)*. URL: https://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL.

[18] Clockwise. *NoSQL vs RDBMS: advantages and disadvantages*. URL: https://clockwise.software/blog/relational-vs-non-relational-databases-advantages-and-disadvantages/.

[19] Clockwise. *NoSQL vs RDBMS: advantages and disadvantages*. URL: https://clockwise.software/blog/relational-vs-non-relational-databases-advantages-and-disadvantages/.

[20] Hamzeh Khazaei et al. «How do I choose the right NoSQL solution? A comprehensive theoretical and experimental survey». In: *Journal of Big Data and Information Analytics (BDIA)* 2 (ott. 2015). DOI: 10.3934/bdia.2016004.

[21] Yann Collet. *Extremely fast hash algorithm*. URL: https://github.com/Cyan4973/xxHash.

[22] Enrico Olivelli. *How to Build a Distributed Database with Apache BookKeeper*. URL: https://streamnative.io/blog/tech/2020-02-04-how-to-build-database/.

[23] Wikipedia. *Definizione di transazione*. URL: https://it.wikipedia.org/wiki/Transazione_(basi_di_dati).

[24] Wikipedia. *ACID*. URL: https://it.wikipedia.org/wiki/ACID.

HerdDB

[1] Enrico Olivelli. *HerdDB: Distributed Database*. URL: https://github.com/diennea/herddb.

[2] Enrico Olivelli. *HerdDB: Concept and architecture*. URL: https://github.com/diennea/herddb/wiki.

Apache Software Foundation

[3] Apache Software Foundation. *Bookkeeper: Concepts and architecture*. URL: https://bookkeeper.apache.org/docs/latest/getting-started/concepts/.

[4] Apache Software Foundation. *Zookeeper: Concepts and architecture*. URL: https://zookeeper.apache.org/doc/r3.5.7/zookeeperInternals.html.

[5] *Zookeeper: What is Apache Zookeeper?* URL: https://www.dezyre.com/article/zookeeper-and-oozie-hadoop-workflow-and-cluster-managers/216.

[6] *Zookeeper: Architecture overview*. URL: http://www.corejavaguru.com/bigdata/zookeeper/architecture.

[7] Apache Software Foundation. *Hbase: Concepts and architecture*. URL: https://hbase.apache.org/book.html#quickstart.

[8] Apache Software Foundation. *Cassandra: Concepts and architecture*. URL: http://cassandra.apache.org/doc/latest/.

Consistency

[9] Cloudera. *Hbase: Checking consistency in Hbase Tables*. URL: https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/admin_hbase_hbck.html.

[10] Oracle. *Mysql: Identify Differences Among Database Objects*. URL: https://docs.oracle.com/cd/E17952_01/mysql-utilities-1.5-en/mysqldiff.html.

[20] Hamzeh Khazaei et al. «How do I choose the right NoSQL solution? A comprehensive theoretical and experimental survey». In: *Journal of Big Data and Information Analytics (BDIA)* 2 (ott. 2015). DOI: 10.3934/bdia.2016004.

[21] Yann Collet. *Extremely fast hash algorithm*. URL: https://github.com/Cyan4973/xxHash.

[22] Enrico Olivelli. *How to Build a Distributed Database with Apache BookKeeper*. URL: https://streamnative.io/blog/tech/2020-02-04-how-to-build-database/.

Cluster

[11] Krishna Nadiminti, Marcos Assuncao e Rajkumar Buyya. «Distributed Systems and Recent Innovations: Challenges and Benefits». In: *InfoNet Magazine* 16 (gen. 2006).

[12] George Coulouris. *Distributed System Concepts and Desing*. Fifth volumes. Addison-Wesley.

[13] Devox. *Distributed Systems in One Lesson by Tim Berglund*. URL: https://www.youtube.com/watch?v=Y6Ev8GIlbxc.

[14] Tech target. *Distributed Database: Definition,Concepts and Architecture*. URL: https://searchoracle.techtarget.com/definition/distributed-database.

[15] DEV. *Distributed System: Vertical vs horinzontal scalability*. URL: https://dev.to/wfelix/scaling-up-vs-scaling-out-vertical-vs-horizontal-scalability-1c3.