POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Progettazione e sviluppo di un sistema per l'analisi di scalabilità di basi di dati NewSQL

Un approccio pratico alla validazione delle prestazioni di CockroachDB e NuoDB

Relatore
prof. Paolo Garza

Candidato
Gianluca Tutolo

LUGLIO 2019

Indice

Elenco delle figure V

Elenco delle tabelle VI

1 Introduzione 1
- 1.1 Scopo della tesi . . . 1
- 1.2 Motivazioni . . . 2
- 1.3 Struttura e guida alla lettura . . . 2

2 Stato dell'arte 3
- 2.1 Sistemi distribuiti . . . 3
  - 2.1.1 Vantaggi e svantaggi . . . 3
  - 2.1.2 Scalabilità . . . 4
  - 2.1.3 Teorema CAP . . . 5
- 2.2 Basi di dati NewSQL . . . 5
  - 2.2.1 Proprietà ACID . . . 5
  - 2.2.2 Caratteristiche delle basi di dati NewSQL . . . 6
- 2.3 Modello ad attori . . . 6
  - 2.3.1 Akka . . . 7
  - 2.3.2 Motivazioni ed utilizzi . . . 9
- 2.4 Docker . . . 9
  - 2.4.1 Docker compose . . . 9
  - 2.4.2 Vantaggi di Docker . . . 10
  - 2.4.3 Utilizzo di Docker nel progetto . . . 10
- 2.5 R . . . 10
  - 2.5.1 Utilizzo di R nel progetto . . . 10
- 2.6 YCSB (Yahoo! Cloud Serving Benchmark) . . . 11
  - 2.6.1 Workload . . . 11
  - 2.6.2 Benchmark . . . 11
  - 2.6.3 Componenti del framework . . . 12
  - 2.6.4 Utilizzo . . . 12
  - 2.6.5 Limitazioni . . . 12

3 Basi di dati in analisi 13
3.1 CockroachDB . . . 13
3.1.1 Terminologia . . . 13
3.1.2 Funzionamento . . . 14
3.1.3 Architettura . . . 16
3.2 NuoDB . . . 21
3.2.1 Terminologia . . . 22
3.2.2 Processi in NuoDB . . . 22
3.2.3 Architettura . . . 23
3.2.4 Modello dei dati . . . 24
3.2.5 Funzionamento . . . 25

4 Framework 29
4.1 Introduzione . . . 29
4.2 Architettura . . . 29
4.2.1 Web server . . . 30
4.2.2 Coordinator . . . 31
4.2.3 Worker . . . 32
4.3 Cluster di test . . . 34
4.4 Modello dei dati . . . 35
4.4.1 Dati utilizzati per i benchmark . . . 35
4.4.2 Parametri di configurazione . . . 36
4.4.3 Risultati dei benchmark . . . 38
4.5 Funzionamento . . . 41
4.5.1 Creazione del carico di lavoro . . . 41
4.5.2 Ciclo di vita di un benchmark . . . 42
4.5.3 Coordinator come macchina a stati finiti . . . 44
4.5.4 Worker: dettagli e funzionamento . . . 45
4.6 Risultati . . . 47
4.6.1 Tecnica per l'elaborazione dei risultati . . . 47
4.6.2 Analisi dei risultati . . . 48

5 Conclusioni 56
5.1 Risultati ottenuti . . . 56
5.1.1 CockroachDB e NuoDB a confronto . . . 56
5.2 Casi d'uso . . . 59
5.2.1 Utilizzi dei benchmark . . . 59
5.2.2 Utilizzi del framework . . . 59
5.3 Sviluppi futuri . . . 59

A Tabelle dati benchmark 61
A.1 Operazioni di lettura di singole righe 61
A.2 Operazioni di scrittura 62
A.3 Operazioni di lettura con join (1) 63
A.4 Operazioni di lettura con join (2) 64
A.5 Operazioni miste 65

Bibliografia 66

Elenco delle figure

2.1 Rappresentazione del meccanismo degli attori 7

3.1 Schema di un'operazione di scrittura in CockroachDB[13] 15
3.2 Architettura NuoDB[12] 23
3.3 Schema di accesso a NuoDB[11] 26

4.1 Illustrazione semplificata dell'architettura del sistema 30
4.2 Dettaglio attori che supportano il coordinator 32
4.3 Dettaglio worker e connection manager 34
4.4 Schema delle tabelle 35
4.5 Coordinator come FSM 44
4.6 Latenze per operazioni di sola lettura di singole righe 49
4.7 Latenze per operazioni di sola scrittura 50
4.8 Latenze per operazioni di lettura con join (2 righe selezionate) . . . 51
4.9 Latenze per operazioni di lettura con join (300 righe selezionate) . . 53
4.10 Latenze per operazioni miste 54

Elenco delle tabelle

4.1 Tabella dei risultati per operazioni di sola lettura di singole righe . 49
4.2 Tabella dei risultati per operazioni di sola scrittura 51
4.3 Tabella dei risultati per operazioni di lettura con join (2 righe selezionate) 52
4.4 Tabella dei risultati per operazioni di lettura con join (300 righe selezionate) 53
4.5 Tabella dei risultati per operazioni miste 54

A.1 Tabella dati per operazioni operazioni di sola lettura di singole righe 61
A.2 Tabella dati per operazioni operazioni di sola scrittura 62
A.3 Tabella dati per operazioni operazioni di lettura con join (2 righe selezionate) 63
A.4 Tabella dati per operazioni operazioni di lettura con join (300 righe selezionate) 64
A.5 Tabella dati per operazioni operazioni miste 65

Capitolo 1

Introduzione

In questo primo capitolo vengono presentati lo scopo della tesi, la sua struttura e le motivazioni che hanno condotto alla realizzazione del progetto.

1.1 Scopo della tesi

Questa tesi si pone l'obiettivo di analizzare le prestazioni di due diverse basi di dati distribuite, appartenenti alla categoria dei sistemi NewSQL. In particolare le basi di dati in questione sono: CockroachDB e NuoDB.

Prima di poter confrontare questi diversi prodotti, si è reso necessario studiarli e comprenderne le architetture, in modo da riuscire ad averne una visione più completa.
Solo dopo aver compreso le differenze e le analogie tra le due basi di dati, si è potuto procedere alla valutazione delle loro prestazioni, effettuando diversi *benchmark*[1].

A tal fine si è scelto di sviluppare uno strumento in grado di generare diversi *workload*[2] e di raccogliere misurazioni dei tempi di risposta della base dati in analisi. Tale software è stato pensato come un framework estensibile: utilizzabile con diverse basi di dati e diverse tipologie di carichi di lavoro. Si è inoltre deciso di realizzarlo con un'architettura distribuita, in maniera da poter realmente simulare l'invio di molte interrogazioni in parallelo alla base di dati, senza i limiti dovuti all'utilizzo di un singolo elaboratore.

---

[1]Per ora si può intendere il termine "benchmark" come un test appositamente studiato per effettuare una valutazione di prestazioni. Una definizione più completa verrà fornita nella sezione 2.6.2.

[2]Il termine "workload" è qui utilizzato con il significato di insieme di operazioni. Per una definizione più dettagliata del termine si veda la sezione 2.6.1.

La realizzazione di tale framework è diventata, di conseguenza, il secondo obiettivo cardine del progetto di tesi.

1.2 Motivazioni

La scelta di affrontare tali argomenti nel progetto di tesi è stata condizionata dalla volontà di conoscere nuove tecnologie, riguardanti le basi di dati, e di approfondire le conoscenze sui sistemi distribuiti.

In particolare, la possibilità di progettare da zero e realizzare un software distribuito (estensibile e di moderata complessità) ha avuto un forte impatto sulla decisione del lavoro di tesi.
Un ulteriore stimolo è stato quello di dover utilizzare un linguaggio di programmazione precedentemente sconosciuto.
L'apprendimento del linguaggio Scala, ed il suo utilizzo in un ambiente distribuito, sono stati visti come elementi concretamente utili in un futuro contesto lavorativo.

L'approfondimento delle basi di dati della famiglia NewSQL è stata ritenuta utile per comprendere al meglio le tecnologie in via di sviluppo, e le differenze con quelle esistenti e di maggior utilizzo (database SQL e NoSQL).
Inoltre, si è ritenuto interessante lo studio delle prestazioni di queste basi di dati e l'approfondimento delle tecniche per realizzare tali analisi.

1.3 Struttura e guida alla lettura

Con questa introduzione, la tesi si compone in totale di cinque capitoli.
Dopo aver analizzato le motivazioni del progetto ed i relativi obiettivi (capitolo 1), vengono descritte le tecnologie utilizzate per la realizzazione del framework, oltre ai concetti fondamentali dei sistemi distribuiti e delle basi di dati NewSQL (capitolo 2).
Il capitolo 3 approfondisce nel dettaglio le due basi di dati, appartenenti a questa categoria, oggetto di studio. Il framework, realizzato allo scopo di analizzare le prestazioni dei due database, è ampiamente analizzato nel capitolo 4.
La tesi si conclude con l'analisi e la discussione dei risultati ottenuti, proponendo sviluppi futuri per il progetto oltre a delle possibili applicazioni (capitolo 5).

Capitolo 2

Stato dell'arte

In questo capitolo si introducono alcuni concetti generici, che costituiscono il punto di partenza per poter analizzare il progetto di tesi.
Si descrivono, inoltre, le tecnologie utilizzate per sviluppare il software di analisi, descritto nel dettaglio nel capitolo 4.

2.1 Sistemi distribuiti

Un sistema distribuito è un insieme di componenti, eseguiti su computer connessi attraverso una rete, che comunicano tra di loro unicamente attraverso scambi di messaggi[5, 16].

2.1.1 Vantaggi e svantaggi

Rispetto ai sistemi centralizzati, quelli distribuiti hanno alcuni notevoli vantaggi:

- **Affidabilità**: dovuta alla possibilità di ridondare ogni componente.
  In caso del guasto di una macchina, ne esiste un'altra nel sistema che può svolgere le stesse funzioni, in modo da non intaccare il funzionamento globale.

- **Eterogeneità**: sia l'hardware che il software dei componenti di un sistema distribuito possono essere profondamente diversi. Questo poiché ognuno di essi è indipendente dall'altro e la comunicazione avviene unicamente attraverso un protocollo di messaggi.

- **Scalabilità**: insieme all'affidabilità, è spesso la caratteristica che conduce alla decisione di utilizzare un sistema distribuito. Rappresenta la capacità del sistema di poter gestire un aumento del carico di lavoro, eventualmente con un incremento delle risorse (generalmente hardware) disponibili. Si veda 2.1.2 per maggiori dettagli.

- **Trasparenza**: caratteristica che consente all'utente di vedere il sistema distribuito come un singolo elaboratore, senza nessuna percezione dei suoi componenti.
- **Economicità**: generalmente, a parità di capacità computazionale, un sistema distribuito ha un prezzo di alcuni ordini di grandezza inferiore rispetto a quello di un sistema centralizzato.

Nonostante i numerosi vantaggi, i sistemi distribuiti hanno anche diversi svantaggi. In gran parte questi sono legati alla maggior complessità, sia hardware che software, del sistema. Altri svantaggi sono legati alla difficoltà di garantire la comunicazione tra i componenti e la loro sicurezza[1].

2.1.2 Scalabilità

Come già accennato in precedenza, la scalabilità è la capacità di un sistema, una rete o un processo di gestire un aumento del carico di lavoro[4].

In particolare un sistema si definisce scalabile se vi è una dipendenza lineare tra le sue risorse e la capacità di gestire il carico di lavoro. Questo significa, ad esempio, che un aumento della memoria disponibile deve coincidere con un aumento proporzionale della sua capacità di output.
Se per avere un incremento delle capacità del sistema sono necessarie moltissime risorse (quindi il rapporto tra le due cose è fortemente non lineare), allora questo non è scalabile.

Scalabilità orizzontale e scalabilità verticale

Esistono due principali tipi di scalabilità, che dipendono dal modo in cui le risorse vengono aggiunte al sistema per aumentarne le capacità.

La scalabilità verticale consiste nell'aumentare le risorse hardware di una singola macchina. Questo significa incrementarne la capacità di elaborazione, mediante l'aggiunta di componenti hardware migliori.
Generalmente la scalabilità verticale è fortemente limitata da un aumento non lineare dei costi dei componenti.

La scalabilità orizzontale è relativa all'aggiunta di nuovi elaboratori nel sistema distribuito, che vengono connessi, mediante la rete, a quelli già presenti.

[1] Si noti che la sicurezza è un problema che riguarda sia i singoli componenti del sistema, sia i messaggi che questi si scambiano.

Oggigiorno si tende a preferire un sistema che supporti la scalabilità orizzontale, che in termini di costi è più vantaggiosa rispetto a quella verticale.

2.1.3 Teorema CAP

In informatica teorica, il teorema CAP, noto anche come teorema di Brewer, afferma che è impossibile per un sistema informatico distribuito fornire simultaneamente tutte e tre le seguenti garanzie [7]:

- **Coerenza** (*Consistency*): ogni operazione di lettura riceve come risposta il dato più aggiornato o un errore
- **Disponibilità** (*Availability*): ogni richiesta riceve una risposta, che potrebbe non contenere la versione più aggiornata del dato
- **Tolleranza alla partizione** (*Partition tolerance*): il sistema continua a funzionare anche in seguito a perdite di messaggi o guasti tra i nodi del sistema distribuito

Nel caso delle basi di dati distribuiti, in seguito al teorema CAP, sono nate soluzioni alternative al problema. Alcune tecnologie in questo settore sono AP, garantiscono quindi la disponibilità dei dati e la tolleranza alle partizioni di rete, senza però fornire garanzie sulla coerenza dei dati. Molte basi di dati distribuite sono invece CP, ovvero preferiscono garantire la coerenza dei dati piuttosto che la loro disponibilità in caso di guasti.

2.2 Basi di dati NewSQL

NewSQL è una classe di sistemi per la gestione delle basi di dati relazionali (*Relational Database Management Systems* RDBMS), che tenta di fornire la stessa scalabilità dei sistemi NoSQL per il processamento di dati transazionali (*Online Transaction Processing* OLTP), garantendo però, a differenza di questi ultimi, le proprietà ACID dei database tradizionali[14].

Il termine "NewSQL" è stato utilizzato per la prima volta nel 2011 dall'analista Matthew Aslett, per discutere della nuova generazione di DBMS[1].

2.2.1 Proprietà ACID

ACID è l'acronimo di Atomicità (*Atomicity*), Consistenza (*Consistency*), Isolamento (*Isolation*) e Durabilità (*Durability*). Queste sono le proprietà che ogni transazione deve possedere, affinché possa operare in modo corretto.

- **Atomicità**: la transazione deve essere indivisibile nella sua esecuzione: l'esecuzione deve essere o totale o nulla (non possono esserci condizioni intermedie).

- **Consistenza**: lo stato della basi di dati deve essere coerente e consistente sia prima che dopo l'esecuzione della transazione, che perciò non deve portare a violazioni di vincoli d'integrità.

- **Isolamento**: ogni transazione deve essere completamente isolata dalle altre ed eseguita in modo indipendente da esse. Il successo o fallimento di una transazione non deve influire sull'esito delle altre.

- **Durabilità** (o **persistenza**): dopo che una transazione è stata completata con successo ed ha richiesto il *commit*, le modifiche apportate alla base dati non devono essere perse. A tale scopo vengono utilizzati dei file di log, per tenere traccia delle operazioni eseguite.

2.2.2 Caratteristiche delle basi di dati NewSQL

Tutte le basi di dati NewSQL supportano SQL e sono database relazionali (RDBMS), con supporto alle proprietà ACID.
A differenza dei tradizionali RDBMS offrono la possibilità di essere eseguiti in un contesto distribuito, quindi su più macchine diverse e, grazie alla cosiddetta *shared-nothing architecture*[2], supportano la scalabilità orizzontale.

Generalmente NewSQL offre la possibilità di accedere ai dati sia come tabelle (come nelle basi di dati SQL), sia come coppie chiave-valore o come documenti (come nei database NoSQL).

2.3 Modello ad attori

Il *modello ad attori* è un paradigma di programmazione per la computazione concorrente. L'unità fondamentale di tale modello è l'*attore*, un agente attivo che recita un ruolo sulla base di una sceneggiatura[9].

[2]La shared-nothing architecture è un architettura tipica di molti sistemi distribuiti, in cui ogni nodo non condivide alcuna risorsa fisica (né memoria né disco) con gli altri e possiede un sottoinsieme dei dati (generalmente replicati su più nodi diversi). Questo tipo di architettura elimina i *single point of failure* e consente una facile scalabilità orizzontale. Di contro introduce una maggior complessità delle applicazioni, che devono gestire eventuali conflitti dovuti ad aggiornamenti dello stesso dato su nodi diversi.

In questo paradigma di programmazione, gli attori sono entità indipendenti che comunicano tra di loro mediante lo scambio di messaggi immutabili. Ogni attore del sistema possiede un nome univoco e opera in modo concorrente agli altri, in maniera totalmente asincrona. Gli attori possiedono una *mailbox* nella quale vengono accodati i messaggi e dalla quale vengono estratti e processati. L'ordine di arrivo di tali messaggi è indeterminato, a causa dell'asincronismo dell'invio. L'elaborazione di un messaggio dipende da come è definito il *behavior* dell'attore e può portare a tre diverse conseguenze:

- invio di un messaggio ad un altro attore
- creazione di un attore
- cambiamento dello stato interno all'attore stesso

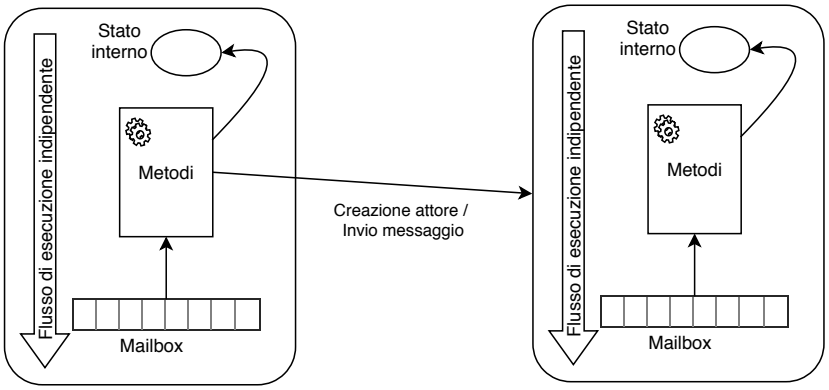


Figura 2.1: Rappresentazione del meccanismo degli attori

La figura 2.1 mostra quanto appena descritto e mette in evidenza come ogni attore possieda il proprio flusso di esecuzione indipendente da quello degli altri. Si evince, inoltre, che lo stato interno di un attore non può essere modificato direttamente dagli altri, ma è solo concesso richiederne un cambiamento attraverso lo scambio di messaggi asincroni.

2.3.1 Akka

Akka è un toolkit open-source che semplifica la realizzazione di applicazioni concorrenti e distribuite su JVM, implementando il modello ad attori.

Ogni attore è creato e supervisionato dal proprio attore padre. La supervisione si può quindi definire parentale, nel senso che ogni attore è responsabile di monitorare eventuali fallimenti negli attori figli che ha generato.

Akka possiede una struttura modulare, di cui il *core* fornisce il supporto al modello ad attori. Moduli aggiuntivi forniscono, ad esempio, supporto per lo sviluppo di applicazioni web e cluster.

Akka cluster

Akka cluster è un modulo che consente lo sviluppo di un sistema di attori (*actor system*) su più nodi.
L'architettura che ne risulta è ti tipo peer-to-peer e fornisce un servizio decentralizzato, senza un singolo punto di rottura (*single point of failure*).
Akka cluster utilizza un protocollo di gossip[3] ed uno strumento di *failure detector* automatico per verificare lo stato dei nodi che compongono il cluster.

Ogni membro del cluster è identificato da una tupla composta da: hostname, porta ed un identificativo univoco.
Attori e nodi sono disaccoppiati: su un nodo membro del cluster potrebbe non esserci nessun attore in esecuzione, oppure più di uno.

Akka HTTP

Akka HTTP è un modulo che implementa completamente sia il lato server che il lato client dello stack HTTP. Questa implementazione è stata realizzata sopra i moduli fondamentali di akka: *akka-actor* e *akka-stream*.

Questo modulo non rappresenta un *framework*[4] completo per la realizzazione di un'applicazione web, ma piuttosto un'insieme di librerie. Di conseguenza, Akka HTTP è pensato per sviluppare applicazioni in cui l'interfaccia HTTP funge da strato di integrazione e non da punto centrale.

Akka FSM

Akka offre ai programmatori la possibilità di sviluppare gli attori come delle macchine a stati finiti (FSM: *Finite State Machine*).

[3]Un protocollo con lo scopo di propagare le informazioni tra i diversi membri di un'architettura peer-to-peer.

[4]Un framework è un'architettura logica di supporto, che è possibile utilizzare come intelaiatura per la realizzazione di applicazioni. Un framework è realizzato con diverse scelte implementative già prese, che velocizzano e semplificano lo sviluppo del software da parte del programmatore, ma lo forzano a determinate decisioni architetturali.

Una macchina a stati finiti può essere descritta come un insieme di relazioni del tipo:

$$Stato(S) \times Evento(E) \rightarrow Azioni(A), Stato(S')$$

Queste relazioni possono essere interpretate come: *Se ci troviamo nello stato S e si verifica l'evento E, dovremo compiere le azioni A ed effettuare una transizione allo stato S'*[10].

Quando il comportamento di un attore è abbastanza complesso e articolato, torna molto utile la possibilità di descriverlo ed implementarlo come una FSM. In questo caso, gli eventi che causano le transizioni da uno stato all'altro sono generalmente messaggi mandati da altri attori.

2.3.2 Motivazioni ed utilizzi

Il disaccoppiamento del mittente dai messaggi inviati è uno dei principali vantaggi del modello ad attori, che consente la comunicazione asincrona[8].
Tale modello, inoltre, semplifica la programmazione concorrente e si presta molto bene alla traduzione di alcuni domini. In particolare nei casi in cui siano facilmente distinguibili dei ruoli da svolgere e le interazioni tra questi.
Il framework sviluppato (capitolo 4) rientra in questo caso. Nel progetto si hanno diversi ruoli, facilmente traducibili in attori, che interagiscono tra di loro.

2.4 Docker

Docker è una piattaforma per lo sviluppo, lo spostamento e l'esecuzione di applicazioni basate su container.

Un container è un ambiente isolato, basato sulla virtualizzazione software, generalmente utilizzato per ospitare una singola applicazione.
Se ogni applicazione è eseguita in un proprio container, è possibile ospitarle tutte su una stessa macchina, senza rischi di conflitti di dipendenze o di configurazioni. Un container, inoltre, facilita lo spostamento dell'applicazione che ospita su una nuova macchina, evitando di dover riconfigurare l'ambiente di esecuzione.

2.4.1 Docker compose

Docker compose consente di definire ed eseguire applicazioni complesse (costituite da più componenti su container diversi) in modo semplice, utilizzando un singolo file YAML. Permette di gestire l'ordine in cui i servizi vengono avviati, la rete che deve connetterli ed altre configurazioni.

2.4.2 Vantaggi di Docker

I principali vantaggi di Docker sono: velocità, portabilità, scalabilità, rapidità di distribuzione e densità[3].

La velocità deriva dalla leggerezza dei container che, di conseguenza, richiedono poco tempo per essere costruiti, mentre la portabilità si è già discussa accennando a a come i container possano facilmente essere spostati su macchine diverse, essendo degli ambienti isolati.
La scalabilità è una caratteristica che dipende sia dalla portabilità che dalla possibilità che Docker offre di poter essere eseguito su ogni macchina Linux.
La rapidità di distribuzione deriva dalla standardizzazione di Docker e dal fatto che ogni applicazione, inserita in un container, si porta dietro tutte le sue dipendenze.
Infine con densità si intende l'utilizzo efficiente delle risorse fisiche da parte di Docker e la conseguente possibilità di eseguire più container su uno stesso host.

2.4.3 Utilizzo di Docker nel progetto

Nel corso dello sviluppo dell'applicazione di benchmark, Docker è stato fondamentale per poterne testare il funzionamento.
Prima di installare il software della base di dati NewSQL da testare su diversi nodi fisici, si sono utilizzati dei container per averne delle istanze in locale, in modo da semplificare il testing e lo sviluppo dell'applicazione.
Questi diversi container sono stati gestiti mediante l'utilizzo di Docker compose.

Oltre all'utilizzo nelle fasi preliminari del progetto, docker è stato anche fondamentale per la creazione di un ambiente per l'elaborazione dei risultati. Tale ambiente contiene tutti i software necessari per generare, a partire dall'output del programma, un confronto dei dati raccolti e una loro rappresentazione.

2.5 R

R è un pacchetto di servizi software per la manipolazione dei dati, il calcolo e la visualizzazione grafica[17]. Include, inoltre, un vero e proprio linguaggio di programmazione, che consente di scrivere script da eseguire per l'elaborazione dei dati.

2.5.1 Utilizzo di R nel progetto

Come verrà spiegato in seguito (si veda 4.4.3), l'output del software realizzato è un JSON contenente, tra le altre cose, una rappresentazione ad istogramma dei dati raccolti. Questo formato è particolarmente adatto per il trasporto dei dati

mediante il protocollo HTTP, ma non è altrettanto idoneo ad una rapida comprensione degli stessi.
Per questo motivo si è scelto di utilizzare script bash e R per generare rappresentazioni più esplicative. In particolare si è utilizzato R per creare un istogramma vero e proprio, partendo da quello contenuto nel JSON.

2.6 YCSB (Yahoo! Cloud Serving Benchmark)

YCSB è un framework sviluppato con lo scopo di valutare le prestazioni di diverse basi di dati di tipo chiave-valore e cloud. Fornisce, inoltre, una serie di workload (si veda 2.6.1) utili a tale scopo e la possibilità di generarne di nuovi.

2.6.1 Workload

Il termine "workload" viene utilizzato per indicare un carico di lavoro, composto da diverse operazioni, da far eseguire al sistema in analisi, allo scopo di misurarne le prestazioni.

Nel caso specifico dell'analisi di una base di dati, il workload definisce un'insieme di interrogazioni da eseguire. Queste comprendono sia operazioni di lettura che di scrittura, con una certa distribuzione.
Il workload definisce anche i dati iniziali su cui verrà poi eseguito il benchmark. Generalmente la fase di preparazione ed inizializzazione della base dati (fase di *loading*) non è soggetta alle misurazioni.

2.6.2 Benchmark

Arrivati a questo punto è fondamentale definire il concetto di "benchmark". In generale, un benchmark è un test appositamente studiato per valutare le prestazioni di un dispositivo o l'efficacia di un processo tecnico o di uno strumento finanziario, in rapporto a uno standard di riferimento. In ambito informatico, l'operazione di *benchmarking* è generalmente effettuata mediante software sviluppati appositamente a tale scopo e generalmente misura il tempo necessario affinché un dispositivo o un programma esegua determinate operazioni.

Un benchmark può essere realizzato mediante l'esecuzione di uno o più workload, misurando il tempo e le risorse necessarie al loro completamento.
Generalmente i workload sono eseguiti utilizzano scenari diversi, in modo da consentire un'analisi più accurata del sistema.

2.6.3 Componenti del framework

YCSB è composto sostanzialmente da due componenti:

- il client YCSB, che rappresenta un generatore di workload estensibile,
- un insieme di scenari di workload, già pronti per essere eseguiti dal generatore.

2.6.4 Utilizzo

Per poter eseguire un workload, utilizzando YCSB, sono necessari diversi passaggi.

1. Configurare la base dati da testare, in particolare creando le tabelle utilizzate dai workload.
2. Scegliere l'opportuno strato di interfaccia al database. Questo è costituito da una classe Java, con il compito di eseguire le interrogazioni generate dal client di YCSB. Questa classe di interfaccia deve ricevere diversi parametri, che generalmente sono passati da linea di comando all'esecuzione del framework.
3. Scegliere il workload che si vuole eseguire.
4. Scegliere i parametri per l'esecuzione, come ad esempio il numero di thread da utilizzare per interagire con la base dati.
5. Eseguire la fase di *loading*, in cui vengono caricati i dati utili allo svolgimento del workload nel database.
6. Eseguire la sezione *transaction* del workload, cioè l'esecuzione vera e propria con relativa misurazione della latenza delle operazioni.

2.6.5 Limitazioni

Anche se YCSB può essere idoneo per una prima analisi delle prestazioni di una base di dati, per effettuare benchmark con un maggior grado di libertà non è la scelta migliore. Oltre a questo se si desidera che le interrogazioni siano mandate alla base di dati da più macchine in parallelo (e non semplicemente da più thread), allora si deve optare per qualcosa di diverso.

Per queste ragioni si è sviluppato un framework di benchmark nuovo, con la possibilità di essere eseguito in un contesto distribuito.
Il progetto sviluppato prende ispirazione da YCSB, ma rende più semplice la personalizzazione dei workload da far eseguire alla base di dati e offre la possibilità di essere utilizzato in un cluster con diversi nodi.

Capitolo 3

Basi di dati in analisi

Questo capitolo presenta nel dettaglio le due basi di dati NewSQL, analizzati durante il progetto di tesi.
Per un'introduzione generale alle basi di dati NewSQL si rimanda alla sezione 2.2.

3.1 CockroachDB

CockroachDB è una base di dati distribuita di tipo relazionale (SQL), che appartiene alla categoria dei sistemi NewSQL (si veda 2.2). In quanto base di dati distribuita, CockroachDB viene generalmente utilizzato in un cluster, ovvero in un sistema in cui più nodi cooperano per garantire maggiori prestazioni, scalabilità, disponibilità e tolleranza ai guasti.
Dal punto di vista del teorema CAP (si veda 2.1.3) CockroachDB garantisce la consistenza dei dati e la tolleranza alle partizioni: si può quindi definire come CP.

CockroachDB è un software open-source, progettato per essere eseguito in un ambiente cloud e per essere resistente ai guasti[6].

3.1.1 Terminologia

Prima di procedere ad un'analisi più dettagliata del funzionamento di CockroachDB e della sua architettura, è necessario comprendere alcuni termini che saranno utilizzati in seguito.

- *Nodo*: macchina individuale sulla quale è in esecuzione CockroachDB.
- *Cluster*: l'intero sistema, costituito da più nodi, che agisce come una singola applicazione logica.
- *Range*: CockroachDB memorizza tutti i dati (tabelle, indici, alcuni dati di sistema, ecc.) in una grande mappa ordinata, formata da coppie chiave-valore.

Lo spazio delle chiavi è suddiviso in "range" in modo che ogni chiave sia collocata sempre in un solo range. Quando un range raggiunge la dimensione di 64 MiB, viene splittato in due range.

- *Replica*: di default CockroachDB replica ogni range tre volte (tale configurazione può essere modificata) e memorizza ogni replica su un nodo diverso, in modo da aumentare la tolleranza ai guasti.

- *Leaseholder*: per ogni range, una delle repliche è il "leaseholder" ed ha il compito di coordinare tutte le richieste di lettura e scrittura rivolte a tale range. Il leaseholder è sempre aggiornato e quando riceve una richiesta di lettura, la serve immediatamente (senza la necessità di coordinare le altre repliche).

- *Raft leader*: per ogni range, una delle repliche è il leader per le richieste di scrittura. Tale leader ha un ruolo fondamentale nel protocollo Raft, che serve ad assicurare il consenso delle repliche in fase di scrittura. Nella maggior parte dei casi il Raft leader coincide con il leaseholder.

- *Raft log*: ogni replica tiene un file di log su disco, in cui sono memorizzate, in ordine cronologico, tutte le operazioni di scrittura su cui hanno raggiunto un accordo.

3.1.2 Funzionamento

In CockroachDB ogni nodo ha un comportamento simmetrico, quindi è totalmente indifferente a quale di essi viene mandata la richiesta. Questo rende la base di dati idonea ad essere integrata con un *load balancer*, cioè un elemento in grado di ridistribuire il carico di lavoro equamente tra i diversi nodi.
Come già accennato nella sezione 3.1.1, i dati sono memorizzati come coppie chiave-valore in range con una dimensione massima di 64 MiB, ognuno dei quali di default è replicato 3 volte su nodi diversi. Questo consente di evitare la perdita di dati, nel caso in cui un nodo si guastasse.

Le richieste alla base di dati possono essere fatte mediante un'API SQL che utilizza il dialetto PostgreSQL. Quando un nodo riceve una richiesta relativa ad un range, la inoltra al nodo leaseholder per quel range (possono essere più di uno se l'interrogazione coinvolge più range). Nel caso di operazioni di lettura il leaseholder può servire direttamente il client, senza contattare le altre repliche, in quanto detiene sempre la copia più aggiornata del range. Quando invece viene richiesta un'operazione di scrittura, deve essere utilizzato il protocollo *Raft* (un protocollo di consenso) per coordinare tutte le repliche.

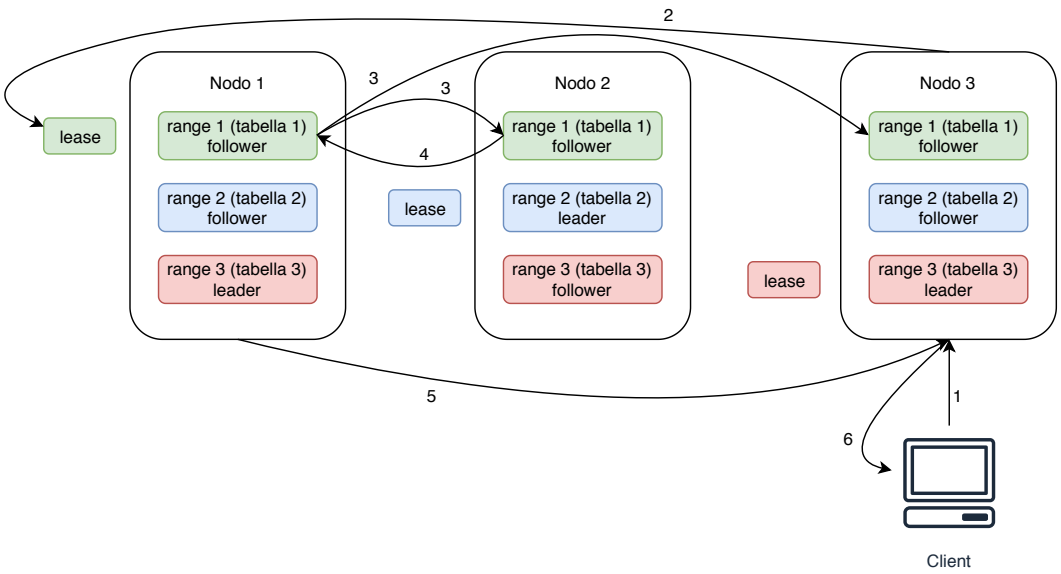


Figura 3.1: Schema di un'operazione di scrittura in CockroachDB[13]

Nella figura 3.1 è illustrato il meccanismo con cui CockroachDB gestisce una richiesta di scrittura.

1. Il nodo 3 riceve una richiesta di scrittura per la tabella 1.

2. La richiesta è inoltrata al nodo 1, poiché è il leaseholder per il range coinvolto.

3. Il leaseholder coincide (come quasi sempre avviene) con il Raft leader, quindi il nodo 1 appende nel suo Raft log l'operazione di scrittura e notifica il tutto alle altre repliche.

4. Quando una replica ha appeso nel suo Raft log l'operazione di scrittura, lo notifica al Raft leader. Ed appena la maggioranza dei nodi ha raggiunto il consenso, la transazione viene committata. Nell'esempio è il nodo 2 il primo a rispondere al Raft leader, mentre il nodo 3 committerà l'operazione poco dopo.

5. Il nodo 1 notifica il successo dell'operazione al nodo 3.

6. A sua volta il nodo 3 notifica l'avvenuta scrittura al client che ha fatto la richiesta.

Nel caso di una richiesta di lettura, il procedimento è molto più semplice. Il nodo che riceve l'interrogazione la inoltra al leaseholder, che gli fornirà subito la risposta da dare al client (senza la necessità di coinvolgere le altre repliche).

3.1.3 Architettura

CockroachDB ha un'architettura suddivisa in strati, ognuno dei quali vede gli altri come delle *black-box API* (cioè come delle scatole nere). Gli strati (*layer*) sono:

- *SQL layer*: espone l'interfaccia SQL agli sviluppatori e converte le interrogazioni ricevute in operazioni basate su chiave-valore. In questo strato avviene la pianificazione logica e fisica necessaria per l'esecuzione della query.

- *Transactional layer*: è lo strato responsabile di fornire supporto alle transazioni ACID (vedi 2.2.1), coordinando operazioni distribuite. Ogni *statement* deve essere seguito dal commit.

- *Distribution layer*: fornisce una visione unificata dei dati del cluster. Affinché questi siano accessibili da ogni nodo, CockroachDB li memorizza in una grande mappa ordinata chiave-valore. Questa mappa monolitica consente rapidi *lookup* (è possibile identificare velocemente dove si trova il dato desiderato) e *scan* efficienti (l'ordinamento permette di trovare facilmente dati in un certo range).

- *Replication layer*: è lo strato responsabile di copiare i dati tra i nodi, garantendo la consistenza attraverso un protocollo di consenso (il protocollo Raft). Il numero minimo di nodi affinché il protocollo funzioni è 3. Il sistema è in grado di sopportare perdite di nodi (dovute a guasti degli stessi o a problemi di rete) e di continuare a funzionare correttamente, a condizione che il numero di nodi offline sia al massimo pari a $(\textit{fattore di replicazione} - 1)/2$. Questo significa che, con la configurazione di base, il sistema è in grado di tollerare la perdita di un nodo, ma il fattore di replicazione è un parametro modificabile.

- *Storage layer*: si occupa di leggere e scrivere i dati su disco. Ogni nodo di CockroachDB contiene almeno uno *store*, specificato in fase di avvio del nodo stesso, che rappresenta la posizione in cui memorizzare i dati.

SQL layer

CockroachDB implementa una buona parte dello standard ANSI SQL, consentendo anche l'utilizzo della semantica delle transazioni ACID.
Le interrogazioni sono veicolate attraverso il protocollo PostgresSQL, per cui esistono diversi driver compatibili.

Dopo la ricezione di un'interrogazione lo strato SQL esegue diverse operazioni di pianificazione e codifica.

1. *Parsificazione*: ogni interrogazione viene analizzata e confrontata con un file, che descrive la sintassi supportata da CockroachDB. Successivamente, viene convertita da stringa ad un albero di sintassi astratta (AST).

2. *Pianificazione logica*: l'AST viene convertito in un piano di esecuzione di alto livello.
   In questa fase avviene un'analisi semantica che consente di verificare la validità dell'interrogazione e, mediante un algoritmo di ricerca, vengono valutati diversi possibili modi di eseguire la query. Tra questi viene scelto quello a minor costo[1].

3. *Pianificazione fisica*: vengono selezionati i nodi che parteciperanno all'esecuzione dell'interrogazione, sulla base della località dei range coinvolti[2].

4. *Esecuzione*: i nodi selezionati ricevono parte del piano fisico e lo utilizzano per la computazione di un frammento dell'interrogazione. Questo compito è svolto dai *logical processor*, creati da CockroachDB sui nodi coinvolti. I processor comunicano tra di loro, in maniera tale che i risultati combinati dell'interrogazione siano mandati al nodo che l'ha ricevuta.
   È importante sottolineare che in questa fase avviene anche una codifica[3] dei dati da stringhe a byte, su cui lavorano gli strati successivi. I risultati sono poi riconvertiti da byte a stringhe.

Nello strato SQL è anche presente un'ottimizzazione per le interrogazioni che coinvolgono più range. Quest'elemento architetturale prende il nome di *DistSQL*. Nel caso una query sia compatibile con DistSQL, ogni nodo può ritornare al coordinatore (colui che ha ricevuto l'interrogazione) dei risultati intermedi più compatti, al posto dell'intero set di righe. Il coordinatore potrà, in seguito, aggregare tali risultati intermedi. Questo riduce la quantità di dati scambiati tra i nodi ed il tempo necessario per servire l'interrogazione.

Transactional layer

CockroachDB da moltissima importanza alla consistenza. Per tale motivo il transactional layer è responsabile di gestire tutti gli statement come transazioni.

---

[1] Il piano logico scelto può essere osservato con il comando *EXPLAIN*.

[2] Il piano fisico può essere richiesto con il comando *EXPLAIN(DISTSQL)*.

[3] Poiché i dati sono memorizzati come coppie chiave-valore, si utilizza una codifica che preserva l'ordinamento lessicografico delle chiavi.

Per processare richieste concorrenti e garantire la consistenza, CockroachDB si affida a MVCC (*Multi-Version Concurrency Control*), cioè il controllo della concorrenza multiversione. Questo protocollo consiste nel non sovrascrivere i dati vecchi, ma di crearne nuove versioni, ognuna delle quali contraddistinta da una marca temporale (*timestamp*). Periodicamente un *garbage collector* si occupa di cancellare i dati obsoleti.

Il transactional layer esegue diversi passaggi per garantire la consistenza e la correttezza delle transazioni.

1. Quando deve essere eseguita un'operazione di scrittura, per prima cosa devono essere create due particolari strutture dati, utili al completamento della transazione. La prima è il *transaction record*, che tiene traccia dello stato corrente della transazione (che può essere *PENDING*, *COMMITTED* oppure *ABORTED*) ed è memorizzato nel primo range coinvolto nell'operazione di scrittura. La seconda struttura dati si chiama *write intent* ed è un valore provvisorio utilizzato dal MVCC che contiene anche un puntatore al transaction record.

2. CockroachDB verifica lo stato corrente nel transaction record: se è ABORTED, la transazione deve essere ricominciata da capo. In caso contrario, essa passa in stato COMMITTED e viene notificato al client il successo dell'operazione.

3. Dopo che la transazione si è conclusa con il commit, tutti i write intent devono essere risolti. Questo significa che devono essere convertiti in valori MVCC, rimuovendo il puntatore al transaction record.
   Nel caso in cui la transazione sia ABORTED, i write intent sono semplicemente ignorati, mentre se lo stato è PENDING significa che sono presenti dei conflitti da risolvere, prima che essa possa diventare COMMITTED.

Per comprendere al meglio i passaggi appena descritti, è necessario chiarire quali sono i conflitti che possono dover essere risolti, prima che la transazione passi allo stato COMMITTED.
Questi sono di due tipi:

- scrittura/scrittura: quando due transazioni pendenti creano un write intent per la stessa chiave,

- scrittura/lettura: se un'operazione di lettura incontra un write intent (per la chiave da leggere) con un timestamp minore del suo.

I conflitti appena descritti possono essere risolti in diversi modi.
Nel caso le transazioni avessero priorità diverse, quella di minor importanza viene

abortita.
Nel caso di una lettura che trova un write intent per la stessa chiave, si può tentare di spostare in avanti la marca temporale della scrittura[4].
Infine, quando nessuna delle precedenti soluzioni è applicabile, la transazione arrivata per ultima viene messa in una coda, in attesa che quella con cui vi è conflitto termini (non importa se con un COMMIT o un ABORT).
Nel caso di deadlock tra transazioni (che avviene quando entrambe sono bloccate da un write intent dell'altra), una a caso viene abortita.

Distribution layer

Come già accennato in precedenza, l'elemento chiave del distribution layer è la grande mappa ordinata in cui sono memorizzate le informazioni. Tale struttura contiene principalmente due tipi di elementi:

- i dati di sistema, memorizzati nei *meta range*, che mantengono le informazioni inerenti alla posizione degli altri dati nel cluster;
- i dati utente, che comprendono le tabelle con le relative righe.

L'accesso ai meta range e la loro replicazione avviene normalmente come per gli altri range.
Queste informazioni di sistema sono memorizzate con un indice a due livelli. Il primo è *meta1* ed indirizza il secondo livello: *meta2*, che consente di localizzare i dati del cluster. Ogni nodo possiede le informazioni necessarie per localizzare il *meta1* ed ha una cache in cui memorizzare i *meta2* a cui ha fatto accesso, in modo da velocizzare utilizzi futuri. Quando scopre che il range che ha in cache non è più valido per una certa chiave, lo aggiorna con una normale lettura.
Quando un nodo riceve una richiesta, controlla la cache e verifica se le chiavi coinvolte sono nel *meta2* memorizzato. In caso contrario utilizza il software *gRPC*[5] per comunicare con gli altri nodi e ricavare i dati utili.
I meta range sono memorizzati nella mappa, come coppie chiave-valore, più nel dettaglio hanno la seguente struttura:

```
metaX/successorKey -> indirizzo_leaseholder, [indirizzi altre repliche]
```

[4]Quest'operazione di *push* del timestamp è possibile solo se la transazione in questione ha un livello di isolamento *snapshot*, che nel caso di CockroachDB è dato dall'utilizzo di MVCC.

[5] gRPC è un framework moderno e open-source per chiamate a procedure remote (RPC: *Remote Procedure Call*) che può essere eseguito ovunque. Consente ad applicazioni client e server di comunicare in modo trasparente, e rende più facile costruire sistemi collegati[2].

In questa rappresentazione, *metaX* rappresenta il livello del meta range (*meta1* o *meta2*), mentre *successorKey* è la chiave del primo elemento non presente nel range (questo rende più efficienti le operazioni di scan).

Anche le righe delle tabelle sono memorizzate come coppie chiave-valore, e hanno la seguente struttura per la chiave:

```
/<id tabella>/<id indice>/<valori colonne indicizzate>
```

Il valore contiene, invece, tutti i valori delle colonne non indicizzate.

Ogni range contiene al suo interno dei metadati che prendono il nome di *range descriptor*. Questo contiene: l'identificativo del range, gli indirizzi delle sue repliche (con specificato il leaseholder) e lo spazio delle chiavi che memorizza.
Date le informazioni che contiene, il range descriptor deve essere aggiornato al verificarsi di uno di questi tre eventi: cambiamenti dei membri del gruppo Raft relativo al range (descritto nel Replication layer), split del range o cambiamento del leaseholder. Per questioni di prestazioni, anche i range descriptor vengono salvati in cache.

Se un range raggiunge la dimensione massima (di default 64 MiB) viene splittato in altri due range; questa regola non si applica a *meta1* che non può essere suddiviso.

Replication layer

CockroachDB è in grado di accorgersi quando un nodo termina e di ridistribuire i dati in modo tale da massimizzare la *survivability* (cioè la capacità di resistere a guasti senza perdita di dati). Oltre a questo, è anche capace di effettuare un ribilanciamento dei dati quando dei nuovi nodi si aggiungono al cluster.

Il cuore del replication layer è il protocollo Raft, un algoritmo di consenso utilizzato per salvare in modo sicuro i dati su più macchine e garantire che esse concordino sullo stato attuale (anche in caso di guasti).
Tutti i nodi che possiedono una replica di un determinato range vengono organizzati in un gruppo Raft. Ogni replica interna al gruppo ha il ruolo di *leader* (longevo ed eletto dal protocollo) o di *follower*. Il leader coordina tutte le operazioni di scrittura che coinvolgono il range (per cui si è formato il gruppo Raft) e manda periodicamente un *heartbeat*[6] ai follower. Se per un certo periodo di tempo i follower

[6]L'heartbeat è un messaggio mandato periodicamente per segnalare a chi lo riceve che il mittente è ancora vivo. Nel caso del protocollo Raft il leader lo utilizza per notificare ai follower che c'è ancora.

non ricevono nessun heartbeat, diventano automaticamente candidati e procedono ad eleggere un nuovo leader.
Quando un'operazione di scrittura riceve il quorum ed è committata dal Raft leader, essa viene appesa nel Raft log. Questo file di log contiene tutte le operazioni su cui il gruppo Raft ha raggiunto il consenso, pertanto può essere utilizzato per portare un nodo che si trova in uno stato non aggiornato a quello corrente.

MVCC consente di creare uno *snapshot* (cioè un'istantanea) di una qualunque replica, creando una copia di tutti i suoi dati ad un certo timestamp. Questi snapshot vengono scambiati durante le operazioni di ribilanciamento tramite gRPC. Dopo che lo snapshot è stato caricato, si devono applicare tutte le modifiche presenti nel Raft log con un timestamp successivo a quello dello snapshot.

È importante ricordare che il leaseholder di un range è l'unico nodo a poter servire le richieste di lettura e a proporre operazioni di scrittura su tale range. Per questo motivo le prestazioni migliorano quando esso coincide con il Raft leader.

Storage layer

Lo storage layer è lo strato con il compito di scrivere su disco le modifiche committate (riportate nel Raft log) e di leggere i dati richiesti. Questi sono salvati come coppia chiave-valore utilizzando *RocksDB*[7], di cui ogni store possiede tre istanze. Queste servono per memorizzare: il Raft log, i dati temporanei (ad esempio computazioni intermedie di interrogazioni distribuite) e i restanti dati del nodo. Le tre istanze di RocksDB condividono una cache.

Tutti i dati MVCC sono salvati in RocksDB e periodicamente vengono compattati da un garbage collector.

3.2 NuoDB

Anche NuoDB, come CockroachDB (3.1) è una base dati distribuita relazionale che appartiene alla categoria dei sistemi NewSQL (2.2), progettata per essere utilizzata in ambiente cloud.

[7]RocksDB è una base di dati NoSQL embedded (cioè strettamente legata ad un software applicativo per accedervi) che memorizza in modo persistente coppie chiave-valore. È stato sviluppato da Facebook, è scritto in C++ ed è completamente open-source.

NuoDB supporta le transazioni ACID e opera in modo simile ad un in-memory database (IMDB)[8].

3.2.1 Terminologia

Prima di proseguire con l'analisi dell'architettura di NuoDB e del suo funzionamento, è importante chiarire alcuni termini che verranno utilizzati in seguito.

- *Host*: rappresenta un nodo del sistema distribuito, sul quale girano uno o più processi di NuoDB.
- *Domain*: è la collezione di host che formano il sistema distribuito. Un domain può contenere uno o più database.

3.2.2 Processi in NuoDB

NuoDB prevede diversi tipi di processo. Su ogni host possono essere eseguite una o più istanze di questi processi.

- *Broker*: è responsabile di gestire l'accesso al transaction engine ed allo storage manager. Ogni domain deve avere almeno un broker per essere definito. Su ogni host vi può essere al più un'istanza del processo broker.
- *Agent*: ha il compito di gestire gli altri processi in esecuzione e di comunicarne lo stato al broker. Su ogni host deve sempre esserci un processo tra broker ed agent. Poiché le ultime versioni di NuoDB non supportano più quest'ultimo, ogni nodo possiede un broker.
- *Transaction engine (TE)*: è responsabile di fornire l'acceso ad un database, di processare le interrogazioni SQL, di cachare dati e di coordinare le transazioni con gli altri processi.
  Il transaction engine lavora unicamente in memoria ed è l'elemento chiave dell'architettura di NuoDB. Infatti la configurazione minima di un domain è rappresentata da un unico host, con un broker ed un TE. Tale sistema non fornisce però la proprietà della persistenza dei dati.
- *Storage manager (SM)*: è l'elemento responsabile di garantire la persistenza dei dati, eseguendo operazioni di scrittura e lettura su disco.

[8]Un in-memory database è DBMS che gestisce i dati nella memoria principale, garantendo minori tempi di accesso ai dati. Dovendo comunque garantire la persistenza, queste basi dati salvano su disco informazioni utili in caso di guasti, generalmente nel formato di file di log e checkpoint.

Tutti i processi descritti sono paritari (*peer*) e ne risulta un'architettura *peer-to-peer*[9]. In questo modo non è necessaria la presenza di un coordinatore, che rappresenterebbe un single point of failure e richiederebbe una configurazione diversa per l'host con tale mansione.
Quando un processo viene avviato, effettua una mutua autenticazione con quelli già in esecuzione, utilizzando SRP (*Secure Remote Password protocol*). I diversi peer comunicano tra di loro, su canali cifrati, mediante l'utilizzo di un semplice protocollo di coordinamento peer-to-peer.

3.2.3 Architettura

NuoDB possiede un'architettura distribuita, divisa in due strati fondamentali: quello transazionale (*transactional tier*) e quello di memorizzazione (*storage management tier*). A questi si aggiunge un ulteriore componente adibito all'amministrazione degli altri.

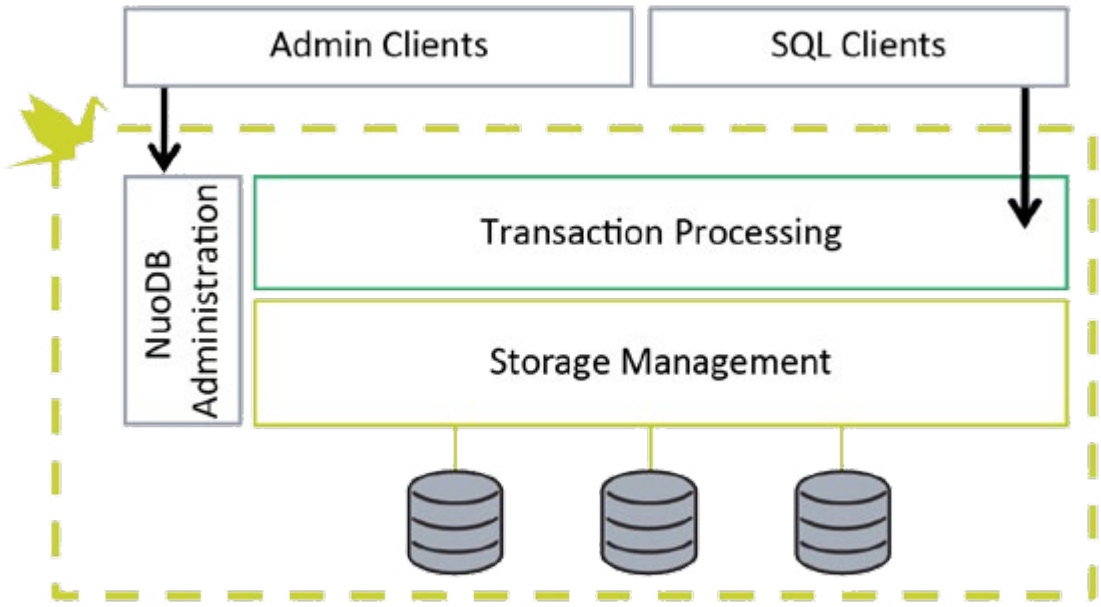


Figura 3.2: Architettura NuoDB[12]

La relazione tra i diversi strati architetturali e le tipologie di processi in NuoDB (3.2.2) è abbastanza evidente. L'insieme dei processi broker e agent costituisce lo

[9]Termine che denota un'architettura in cui tutti i nodi sono equivalenti (peer). Si contrappone all'architettura *client/server*.

strato amministrativo e di gestione, i TE formano il livello transazionale, mentre gli SM rappresentano lo storage management tier.

Architettura a tre livelli

La divisione architetturale in tre strati è ciò che consente la scalabilità orizzontale di NuoDB, e ne costituisce l'elemento chiave.

Le basi di dati tradizionali presentano un forte accoppiamento tra la rappresentazione in memoria dei dati e la loro scrittura su disco. Questo limita le prestazioni a causa del throughput limitato dei dischi, e rende difficile scalare.
NuoDB ha deciso di separare la gestione della durabilità (persistenza dei dati) dalle altre tre proprietà ACID. Per realizzare questo obiettivo, ha attribuito allo strato transazionale il compito di garantire atomicità, consistenza e isolamento ("ACI"); mentre ha riservato la mansione di fornire la durabilità ("D") allo strato di memorizzazione.

Broker e strato di gestione

Lo strato di gestione è costituito da un insieme di processi broker: uno per ogni host.
Il broker è responsabile del nodo su cui è in esecuzione. In particolare monitora i processi TE ed SM attivi e le risorse disponibili.

Ogni broker ha una visione completa del domain. Questo significa che vede tutti gli altri broker, tutti i processi di NuoDB in esecuzione (TE e SM), tutti i database ed i principali eventi che li coinvolgono.
Per come sono implementati, tutti i broker hanno la stessa conoscenza del domain e le stesse capacità di gestirlo. Questo evita che vi sia un single point of failure.

I broker sono gli elementi che offrono al client la visione della basi di dati distribuita come un'unica entità logica. Inoltre svolgono anche il ruolo di *load balancer*, cioè distribuiscono il carico di lavoro tra i vari transaction engine disponibili, sulla base di regole stabilite in fase di configurazione[10].

3.2.4 Modello dei dati

NuoDB al suo interno memorizza i dati in oggetti chiamati: "Atomi". Ogni dato, indice e metadato è memorizzato sotto forma di Atomo.

[10]Generalmente queste regole devono tenere conto della posizione geografica dei nodi fisici

Gli Atomi rappresentano frammenti di un database, su cui devono essere garantite le proprietà ACID. La loro dimensione può variare nel tempo.
Il transaction engine, tra gli altri compiti, deve mappare contenuti SQL in Atomi e viceversa.

Atomi Catalogo

NuoDB prevede una tipologia particolare di Atomi, che non memorizza dati inerenti ad un database. Queste strutture dati prendono il nome di Atomi Catalogo, ed hanno il compito di conservare le informazioni necessarie a risolvere ed indirizzare gli altri Atomi.
Quando un TE viene avviato deve popolare l'Atomo Catalogo root (*root Catalog Atom* o *Master Catalog*), che consente di scoprire tutti gli altri elementi della base di dati.

3.2.5 Funzionamento

Dopo aver analizzato l'architettura di NuoDB ed il suo modello di memorizzazione dei dati, si può ora procedere con un analisi più dettagliata del suo funzionamento.

Schema di accesso

Per prima cosa occorre analizzare come avviene l'accesso ad un database da parte di un client.

1. Quando un'applicazione cerca di connettersi alla base dati, si rivolge al broker in esecuzione su un host. Questo conosce tutti i TE presenti nel domain, grazie alle informazioni che scambia con i broker in esecuzione sugli altri nodi.

2. Sulla base delle informazioni in suo possesso, il broker decide qual'è il TE migliore per l'applicazione e glielo comunica.

3. Il client si connette al TE fornitogli dal broker, che potrebbe anche risiedere su un host diverso.
   Da questo momento in poi la comunicazione avviene direttamente tra client e TE, senza più l'intervento del broker.

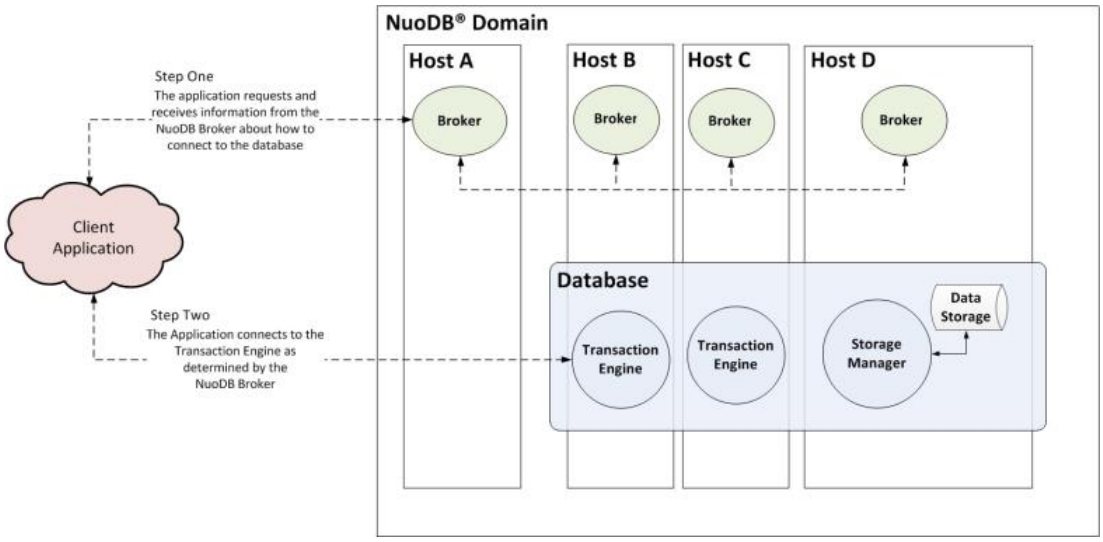


Figura 3.3: Schema di accesso a NuoDB[11]

Caching

In NuoDB la cache svolge un ruolo fondamentale.
Dopo che un transaction engine ha accettato una connessione da un client SQL, esegue l'interrogazione sui dati in cache. Nel caso in cui i dati necessari non siano presenti nella cache (*cache miss*), il TE valuta[11] qual'è il peer migliore a cui richiederli. Questo potrebbe essere un altro transaction engine che possiede le informazioni necessarie in cache, oppure uno storage manager che le ha salvate su disco.
Dopo aver identificato il peer migliore, in possesso dei dati necessari, questi vengono scambiati mediante il protocollo di coordinazione peer-to-peer.

NuoDB utilizza un meccanismo di *cache on-demand*, cioè solo gli Atomi richiesti sono salvati in cache.
In pratica la popolazione della cache avviene sempre in seguito alla richiesta di un client SQL. Nello specifico quando vengono inseriti dei nuovi dati e quando un'operazione genera un cache miss.
Un cambiamento della cache di un TE scatena un aggiornamento dell'Atomo Catalogo, che deve riflettere tale modifica.

[11]La valutazione del peer migliore viene eseguita mediante un'apposita funzione di costo.

MVCC e Chairman

Come CockroachDB, anche NuoDB si affida al protocollo MVCC (*Multi-Version Concurrency Control*) per garantire la consistenza anche in caso di conflitti. In questo modo ogni operazione di aggiornamento o cancellazione genera semplicemente una nuova versione dei dati, che resta in uno stato pendente fino al commit della transazione associata.

L'utilizzo di questo protocollo garantisce un livello di isolamento snapshot, che evita conflitti quando vengono eseguite operazioni di lettura, anche se è in corso un aggiornamento dei dati coinvolti.
Quando avviene che più transazioni vogliano aggiornare lo stesso dato, il protocollo MVCC non è più sufficiente ad evitare il conflitto e diventa necessaria una mediazione.
Per risolvere il problema NuoDB sceglie un TE, che possiede in cache l'Atomo su cui si è verificato il conflitto, e lo nomina *Chairman* per quello specifico Atomo. Per ogni Atomo esiste un TE noto che ha il compito di Chairman, cioè svolge il ruolo di mediatore quando si verificano dei conflitti riguardanti quello specifico blocco di dati.
Si noti che quando si verifica un cache miss in un transaction engine, se nessun altro TE possiede i dati richiesti in cache, questi vengono recuperati da uno storage manager ed il TE diventa automaticamente Chairman per quell'Atomo. Quando un Chairman non è più disponibile (ad esempio a causa di un guasto del nodo), ne viene eletto uno nuovo.
Per minimizzare i conflitti il protocollo MVCC versiona i singoli record e non interi Atomi.

Persistenza dei dati

Come già detto in precedenza tutti i dati sono memorizzati in Atomi, ognuno dei quali possiede un identificativo univoco ed è memorizzato come coppia chiave valore.

Di default ogni storage manager possiede una copia completa della base di dati. Tuttavia è possibile configurare a piacere come partizionare i dati, quante repliche crearne e dove memorizzarle fisicamente.

Uno storage manager, oltre a scrivere i dati del database, può mantenere (se la relativa opzione viene abilitata) un *journal* con tutte le modifiche dei dati. Queste sono espresse semplicemente come differenze tra le varie versioni e sono relativamente piccole.

Il journal consente un recupero più rapido dei dati quando si verificano guasti o quando è necessario un riavvio dello storage manager.

Il recupero di un dato da parte di un TE viene effettuato valutando il costo dell'operazione, mediante un'apposita funzione. In questo modo, se sono disponibili due SM con lo stesso dato, verrà consultato quello più veloce a fornire la risposta (ad esempio per ragioni hardware o di distanza dei nodi).

Protocollo di commit

Per garantire tutte le quattro proprietà ACID, il transaction engine che sta gestendo una transazione deve coordinarsi con uno storage manager. Infatti mentre il TE garantisce atomicità, consistenza ed isolamento; è lo SM a fornire la durabilità. In questo contesto, il coordinamento che avviene tra TE e SM prende il nome di protocollo di commit.

NuoDB permette di personalizzare, a seconda delle esigenze, il protocollo di commit, in modo da poter scegliere il miglior compromesso tra persistenza e prestazioni. Queste si possono massimizzare richiedendo che la transazione venga considerata committata con successo, dopo che il TE ha mandato dei messaggi asincroni agli SM. In questo caso la durabilità non è garantita, perché potrebbe succedere che per un guasto nessun storage manager riceva il messaggio del TE.
L'alternativa a questa soluzione prevede di specificare quanti SM devono aver confermato la scrittura persistente del dato, prima che la transazione possa considerarsi conclusa con successo.

Capitolo 4

Framework

Questo capitolo analizza nel dettaglio il lavoro svolto durante il progetto di tesi. Partendo dalla descrizione del framework sviluppato, si arriva alla presentazione e discussione dei risultati, ottenuti utilizzando tale software.

4.1 Introduzione

Lo scopo del progetto è lo sviluppo di un software distribuito, in grado di essere eseguito su più nodi, per misurare le prestazioni di diverse basi dati di tipo NewSQL. Successivamente ci si pone l'obiettivo di confrontare i risultati ottenuti e di valutare il possibile impiego di tali database, soprattutto in un contesto IoT.

4.2 Architettura

Il framework è stato sviluppato interamente in Scala, facendo ampio uso del modello ad attori, con il supporto di Akka.

L'applicazione è idonea ad essere eseguita in un contesto cluster, dove ogni nodo ha uno o più ruoli. Questi devono essere passati al lancio del programma come argomenti della JVM e vengono registrati nella configurazione di Akka.

Queste funzioni, definite in fase di avvio, sono svolte da attori Akka che comunicano tra di loro mediante scambi di messaggi. Dato che ogni nodo può avere più ruoli, è possibile che più attori diversi ed indipendenti coesistano sulla stessa macchina.
I possibili ruoli sono:

- *web-server*
- *coordinator*

- *worker*

All'avvio del software, ogni nodo verifica se possiede il ruolo web-server ed in caso positivo crea un attore Akka che svolge tale funzione. Lo stesso procedimento è ripetuto per la mansione di coordinator. La funzione di worker viene invece gestita successivamente, quando è richiesto un nuovo benchmark.

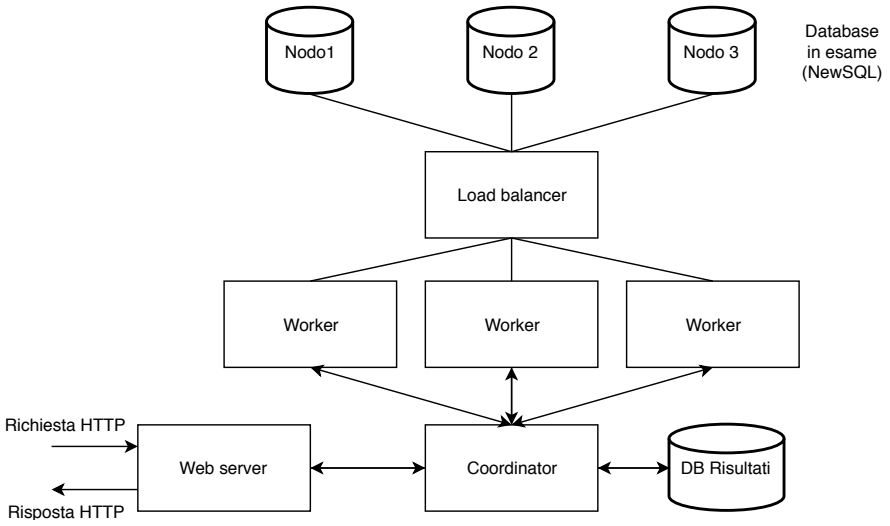


Figura 4.1: Illustrazione semplificata dell'architettura del sistema

La figura 4.1 rappresenta uno schema semplificato dell'architettura del framework. Il client manda richieste al web server, che le inoltra al coordinator, il quale scambia messaggi con i worker. Quest'ultimi mandano le interrogazioni alla base di dati in esame, passando attraverso un load balancer. Il coordinator ha anche il compito di scrivere i risultati dei benchmark su un'altra base di dati, in modo da poterli recuperare quando richiesto.

4.2.1 Web server

I nodi del cluster con il ruolo di server web espongono un'interfaccia HTTP REST per controllare l'applicazione.
In particolare offrono la possibilità di avviare un nuovo benchmark e di recuperare quelli effettuati, permettendo di filtrarli all'occorrenza.

Il server web è stato realizzato utilizzando il modulo *Akka HTTP* del toolkit Akka.
Dopo essere stato creato e avviato dal processo principale, l'attore inizia ad ascoltare all'indirizzo e porta specificati, in attesa di richieste HTTP.

Ogni richiesta scatena l'invio di un messaggio all'attore coordinator, la cui risposta viene poi inoltrata al client.

Lo scambio di dati tra il web server ed il client avviene mediante file JSON. In particolare quando si vuole avviare un nuovo benchmark, è necessario inviare, mediante una POST, un file JSON con la configurazione desiderata (si veda 4.4.2 per maggiori dettagli). Anche il risultato restituito dal server su richiesta è rappresentato come JSON. Il modello dati del risultato è trattato in maggior dettaglio in 4.4.3.

4.2.2 Coordinator

Il coordinator ha il compito di gestire i worker, di coordinare lo svolgimento del processo di benchmark, di salvare i risultati e di recuperarli quando richiesti dal web server.

Per poter svolgere correttamente il proprio dovere, e garantire la consistenza, il coordinator è realizzato come un attore *singleton*. Questo significa che vi è un solo attore di questo tipo in tutto il cluster.
In realtà più nodi possono avere il ruolo di coordinator, ma verrà comunque creato un solo attore nel sistema.
Nel caso in cui il nodo su cui si trova l'attore dovesse fallire o essere spento, un altro con il ruolo di coordinator prenderebbe il suo posto.

Questo è sicuramente il ruolo più complesso del sistema, per questo motivo è costituito da più attori.
Quello principale (chiamato semplicemente coordinator) viene creato, esattamente come il web server, dal processo *main*. Esso svolge il compito di avviare il benchmark quando richiesto dal client, di gestire i processi worker e di aggregare e memorizzare i risultati.
Questo attore ne crea a sua volta altri due:

- il *member manager*: con la funzione di restare in ascolto degli eventi che si verificano nel cluster;
- il *serve request actor*: responsabile di rispondere ai messaggi del web server scatenati da delle richieste HTTP (ad eccezione della POST).

I due attori appena illustrati sono figli del coordinator e per questo sono anche loro dei singleton.

Member manager

Il member manager è il più semplice tra gli attori che costituiscono il coordinator. Il suo unico compito è verificare quando dei nodi con ruolo worker si uniscono al cluster o lo abbandonano e notificare all'attore padre l'evento.
Sulla base dei messaggi del member manager, il coordinator aggiorna una struttura dati che tiene traccia degli indirizzi dei nodi worker.

Serve request actor

Il serve request actor svolge la funzione di rispondere ai messaggi mandati dal web server, facendo da tramite con la base di dati[1] in cui sono salvati i benchmark. Ad ogni messaggio ricevuto corrisponde un'interrogazione alla base dati; il risultato è inoltrato, in un nuovo messaggio, al server web.

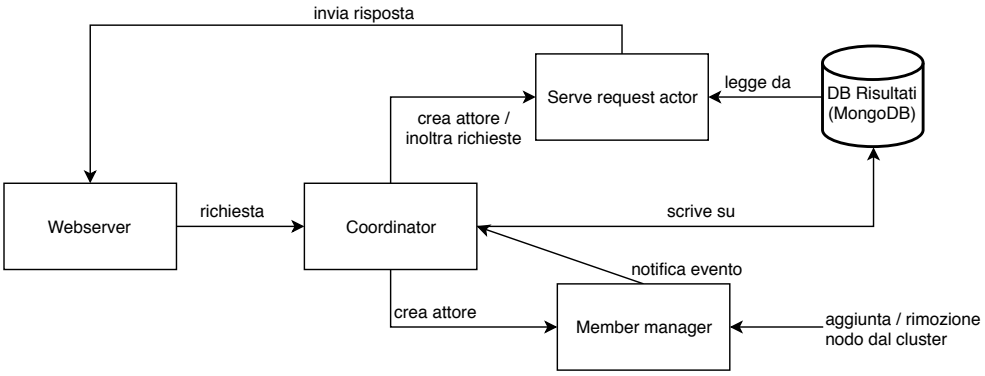


Figura 4.2: Dettaglio attori che supportano il coordinator

La figura 4.2 mostra uno schema del funzionamento degli attori figli del coordinator e le loro interazioni con il resto del sistema.

4.2.3 Worker

I nodi con ruolo worker hanno il compito di eseguire le interrogazioni alla base di dati che si sta testando, raccogliere statistiche ed inoltrarle al coordinator.

Mentre per il coordinator ed il web server esiste un rapporto uno a uno tra ruolo ed attore che lo svolge, per il worker non è così. Infatti è possibile che su un nodo con ruolo worker vengano eseguiti più attori con tale mansione.

[1] Si noti che gli esiti dei benchmark sono salvati su una base dati che non ha nulla a che vedere con quella in esame. La sua funzione è unicamente quella di rendere persistenti ed accessibili i risultati ottenuti.

Il numero di processi (attori) di questo tipo, su una singola macchina, è regolato da una configurazione presente nel JSON, mandato dal client quando richiede l'avvio di un benchmark.

Come si è accennato in precedenza, gli attori worker non sono creati all'avvio del programma. È compito del coordinator istanziarli quando viene richiesto un nuovo benchmark e fermarli quando questo termina. In particolare esso conosce tutti gli indirizzi dei nodi con ruolo worker grazie al member manager, e su ognuno di questi crea tanti attori worker quanti richiesti dal client nel JSON di configurazione.
Questa tecnica permette di cambiare facilmente il numero di processi adibiti all'interrogazione della base di dati in esame, similmente a come in YCSB si configura la quantità di thread (si veda 2.6.4).

Quando il coordinator crea gli attori worker, sullo stesso nodo istanzia anche un altro attore: il *connection manager*. Il coordinator fa in modo che ogni attore worker creato conosca il connection manager del nodo corrispondente.

Connection manager

Il connection manager conosce il numero massimo di connessioni disponibili e le fornisce ai worker quando gliele richiedono.

Il suo principio di funzionamento è il seguente.

1. Quando viene creato, inizializza una lista vuota di connessioni alla base di dati ed un contatore.

2. All'occorrenza, un worker manda un messaggio al connection manager, per richiedere una connessione.

3. Se il numero di connessioni create, indicate dal contatore, non ha ancora raggiunto il massimo (configurabile all'avvio del programma) l'attore utilizza un *factory method*[2] per generarne una nuova e la restituisce al worker che ha fatto richiesta.

4. Se il numero di connessioni create, indicate dal contatore, ha raggiunto il massimo, il connection manager può soltanto attingere a quelle disponibili nella lista (rimuovendone una). Se questa è vuota, significa che al momento non sono disponibili connessioni.

[2]Il metodo factory è quello che viene definito un *design pattern*, ovvero una soluzione ad un problema ricorrente. In questo caso specifico, il metodo factory ha il compito di istanziare la connessione alla giusta base di dati, utilizzando la classe concreta appropriata.

5. Quando un worker restituisce una connessione, la lista viene aggiornata.

In caso tutte le connessioni siano già in uso ed un worker ne chiedesse una, il connection manager notifica la situazione all'attore che ha fatto richiesta e questo ritenta dopo un certo intervallo di tempo.

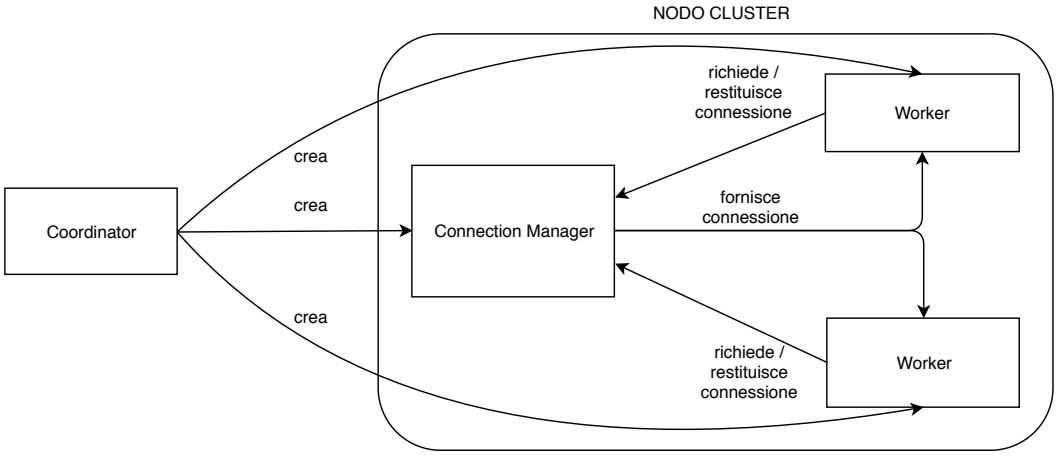


Figura 4.3: Dettaglio worker e connection manager

La figura 4.3 mostra i rapporti ed i messaggi scambiati tra gli attori worker di un nodo ed il relativo connection manager.
Nello schema si è assunto che il JSON di configurazione richiedesse due processi worker su ogni macchina con ruolo idoneo.

4.3 Cluster di test

Le basi di dati NewSQL sotto analisi (CockroachDB e NuoDB) ed il framework sviluppato sono stati installati su delle macchine AWS (*Amazon Web Services*).
Più nel dettaglio si sono utilizzate:

- Tre istanze *m5x.large*, per i nodi delle basi di dati distribuite, con le seguenti caratteristiche:
  - 4 CPU
  - 16 GiB RAM
  - 20 GiB GP2 SSD dedicati al sistema operativo
  - 100 GiB GP2 SSD dedicati ai dati ed ai log
  - Centos7 come sistema operativo

- Un'istanza *t3.medium*, per eseguire il programma di benchmark, con le seguenti caratteristiche:
  - 2 CPU
  - 4 GiB RAM
  - 20 GiB GP2 SSD dedicati al sistema operativo
  - 100 GiB disco magnetico HDD per dati e log del software
  - Centos7 come sistema operativo

Avendo a disposizione un solo nodo per eseguire il software di benchmark sviluppato, gli sono stati assegnati tutti i ruoli descritti in precedenza (web server, coordinator e worker).

4.4 Modello dei dati

Per poter comprendere il funzionamento dell'applicazione è importante analizzare il formato dei dati: sia quelli in ingresso che quelli in uscita.

4.4.1 Dati utilizzati per i benchmark

Tutti workload sono stati eseguiti utilizzando dati sintetici[3] volti a simulare quelli provenienti dal mondo IoT.
Più di preciso si sono create tre tabelle relative a: *dispositivi*, *misurazioni* e *periodi di validità*.
Lo schema di queste relazioni, in particolare il numero di campi che lo compongono, è stato reso customizzabile mediante il JSON di configurazione del benchmark.

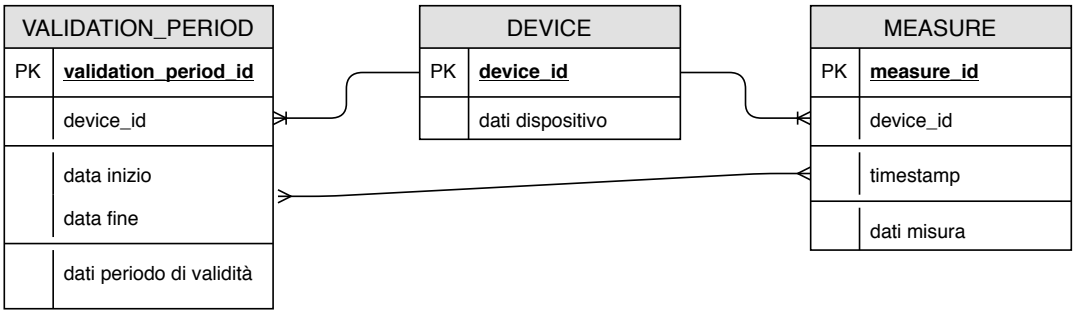


Figura 4.4: Schema delle tabelle

[3]Con *dato sintetico* si intende un dato generato artificialmente attraverso del codice.

La figura 4.4 mostra lo schema delle tabelle prese in esame per tutti i benchmark effettuati. I nomi dei campi non rispettano esattamente quelli utilizzati, ma sono stati modificati nello schema per rendere più evidenti le relazioni tra le tabelle.
La relazione tra il timestamp delle misure ed il range temporale nel periodo di validità è del tipo molti a molti, ma per un particolare dispositivo in un determinato momento esiste sempre un solo periodo di validità. Questo significa che se si esegue la join delle tre tabelle, utilizzando l'identificativo del dispositivo, e si impone che il timestamp delle misure sia nel range del periodo di validazione, si ottiene una tabella con una riga per ogni misura.

4.4.2 Parametri di configurazione

Come si è già accennato, l'avvio di un benchmark viene richiesto dal client mediante l'invio di un file JSON di configurazione. Tale file contiene i seguenti campi:

- *workload id*: un identificativo del tipo di workload che si vuole eseguire. Attualmente il tipo di carico di lavoro che si è interessati ad analizzare è quello legato all'ambito IoT, ma questo campo consente un'estensibilità futura;
- *numero di righe della tabella dei dispositivi*;
- *misurazioni per dispositivo*: il numero di righe della tabella delle misurazioni è espresso in funzione di quelle dei dispositivi, mediante un coefficiente intero;
- *periodi di validità per dispositivo*: il numero di righe della tabella dei periodi di validità è espresso in funzione di quelle dei dispositivi, mediante un coefficiente intero;
- *numero di campi delle tabelle*: per ogni tabella è possibile esplicitare il numero di colonne che deve avere. In caso tale configurazione non venga data, si utilizza un valore di default;
- *numero di operazioni da eseguire*;
- *operazioni di lettura*: il numero di operazioni di lettura, espresso come percentuale del numero di operazioni complessivo. Le restanti saranno scritture;
- *operazioni di lettura con join*: il numero di operazioni di lettura con join delle tre tabelle, espresso come percentuale delle operazioni di lettura totali;

- *dimensione delle scan*: il numero di righe da leggere è espresso come percentuale della dimensione delle tabelle. Se questo valore è zero, significa che si richiedono solo letture puntuali[4];

- *numero massimo di righe da inserire*: valore che identifica il massimo numero di righe da inserire nelle operazioni di scrittura durante l'esecuzione del benchmark (non ha effetto sulla fase di caricamento della base dati). Il numero di righe inserite è casuale, ma varia tra uno ed il valore di questo parametro;

- *dimensione in byte dei campi*: permette di modificare la dimensione in byte dei campi (uguale per tutti) in modo da poter regolare lo spazio occupato dalle tabelle;

- *numero minimo di nodi worker*: parametro che consente di bloccare l'esecuzione del benchmark se nel cluster non vi sono almeno un minino di nodi con ruolo worker;

- *processi worker per nodo*: identifica il numero di attori worker che devono essere creati su ogni nodo con il ruolo idoneo. In pratica questo parametro permette di fissare il numero di richieste parallele alla base di dati;

- *numero di intervalli dell'istogramma*: le misurazioni dei tempi di risposta del database vengono salvate in un istogramma ad intervalli. Questo parametro serve a specificare il numero di intervalli desiderati;

- *massimo valore dell'istogramma*: specifica il massimo valore per gli intervalli dell'istogramma. Tutte le misurazioni che eccedono tale valore sono raccolte in un intervallo speciale;

- *skip della fase di caricamento*: è un parametro booleano opzionale che, se attivo, consente di saltare la fase di caricamento e preparazione della base di dati, per passare direttamente a quella di esecuzione del workload. Affinché questo sia possibile, esistono diverse condizioni che devono essere soddisfatte. In particolare le tabelle su cui si vuole eseguire il benchmark devono essere le stesse attualmente disponibili (come numero di righe, quantità di campi e dimensione degli stessi);

- *cleanup*: è un parametro booleano che stabilisce se le tabelle devono essere cancellate alla fine del benchmark. Solo se questa opzione non era attiva per il benchmark precedente, è possibile richiedere lo skip della fase di caricamento di quello attuale;

[4]Con "puntuale" si intende la lettura per chiave primaria di una tabella, che genera in uscita una sola riga come risultato (o nessuna se la chiave specificata non è presente).

- *parametri per connessione alla base di dati tramite JDBC*: sono tre diversi parametri che comprendono: la stringa di connessione, lo username e la password.

4.4.3 Risultati dei benchmark

I risultati dei benchmark sono stati memorizzati in una base di dati NoSQL, più in particolare in MongoDB. L'istanza di questa base dati viene eseguita sulla stessa macchina dove gira il codice.
Prima di proseguire nello spiegare questa scelta per la memorizzazione dei benchmark, è importante dare qualche breve nozione su MongoDB.

MongoDB

MongoDB è una base di dati non relazionale (NoSQL) di tipo documentale, infatti i dati sono salvati con un formato simile al JSON.
MongoDB è fatto di database che contengono collezioni. Una collezione è una raccolta di documenti. Ogni documento è composto da campi. Le collezioni possono essere indicizzate, il che migliora le prestazioni di ricerche e ordinamenti[15].

Un aspetto importante di Mongo è che ogni collezione è senza schema, perciò i documenti che contiene possono avere campi diversi uno dall'altro.
Inoltre ogni campo può contenere al suo interno un altro documento.

Modello del benchmark

Dopo aver esposto in breve il modo in cui sono organizzati i dati su MongoDB, si può illustrare con maggior semplicità la scelta di tale base di dati per l'archiviazione degli esiti dei benchmark, ma prima si deve analizzare il modello adottato per la memorizzazione di questi dati.

Per ogni workload eseguito, sono state memorizzate le seguenti informazioni in un documento:

- *benchmark id*: un identificativo univoco associato al benchmark. È rappresentato da un UUID (*Universally Unique Identifier*), cioè da un valore su 128 bit in formato esadecimale;
- *timestamp di inizio*: la marca temporale del momento in cui il benchmark è stato avviato;
- *timestamp di fine*: la marca temporale del momento in cui il benchmark si è concluso;

- *file di configurazione*: il JSON di configurazione che è stato usato per il benchmark. Questa informazione è utile per poter conoscere i parametri che hanno portato a determinati risultati;

- *stato del sistema*: una stringa con le informazioni relative al numero di nodi del cluster e ai loro ruoli nel momento in cui il benchmark è stato eseguito;

- *stato attuale* e *lista degli stati*: ogni benchmark in un determinato momento si può trovare in uno di quattro possibili stati. Questi due campi rappresentano sia lo stato attuale, sia la lista di quelli attraverso cui è passato. Ogni stato è un documento costituito da tre campi: il nome dello stato, il timestamp in cui il benchmark ha raggiunto tale condizione e l'id dello stesso (ripetuto per semplificare alcune interrogazioni);

- *risultato*: memorizza in un altro documento il risultato del workload eseguito. Il nome di ogni campo rappresenta l'etichetta di una barra dell'istogramma (ottenuto ragruppando le misurazioni in un certo numero di intervalli), mentre il valore associato è un array di due elementi contenente il numero di misurazioni che rientrano in quell'intervallo e la loro somma;

Scelta di Mongo

Come si evince dalla struttura appena analizzata, MongoDB si adatta perfettamente alla memorizzazione dei dati con questo formato.

La possibilità di archiviare il JSON di configurazione, senza necessità di una elaborazione particolare, e di avere una struttura con documenti annidati, rendono Mongo idoneo allo scopo.
Torna inoltre utile anche la grande flessibilità dei documenti all'interno della collezione. Infatti a seconda dello stato in cui si trova il benchmark, non tutte le informazioni saranno presenti in ogni momento. Ad esempio la marca temporale di fine, così come il risultato, saranno presenti solo alla conclusione dell'esecuzione del workload.

Istogramma dei risultati

Si è già più volte detto che gli esiti di un benchmark sono riportati in un istogramma, che in questo contesto non è un grafico, ma una struttura dati.
Prima di passare all'analisi del funzionamento del software (4.5), è importante capire la tecnica utilizzata per l'aggregazione delle misurazioni.

L'istogramma viene costruito nel seguente modo:

1. Vengono letti dal JSON di configurazione il numero di intervalli ed il massimo valore dell'istogramma (già introdotti in 4.4.2).

2. Si calcola la dimensione di ogni intervallo eseguendo il rapporto tra i due valori.

3. Viene costruita una mappa chiave-valore, dove la chiave è l'etichetta dell'intervallo, mentre il valore è una tupla (coppia di valori) contenente il numero di elementi in quell'intervallo e la loro somma.
   La chiave della mappa rappresenta l'estremo inferiore dell'intervallo (incluso nello stesso), mentre l'estremo superiore (escluso) è rappresentato dalla chiave successiva. L'ultimo intervallo non possiede l'estremo superiore ed è infinito a destra.

4. Ogni misurazione viene confrontata con le chiavi della mappa e, identificato l'intervallo corretto, vengono aggiornati il contatore e l'accumulatore.

Risposta restituita al client

Nella risposta che il web server restituisce al client, non vi è semplicemente il documento di Mongo che rappresenta il risultato. A questo vengono aggiunte ulteriori informazioni statistiche, calcolate sull'istogramma.

Dei dati aggiunti nella risposta il più significativo è probabilmente il tempo medio per operazione. Il calcolo di tale valore è reso possibile dalla struttura dati utilizzata per l'istogramma, che memorizza, oltre al conteggio, anche la somma degli elementi di ogni intervallo.
Per rendere più significativa la media, è stata calcolata anche la deviazione standard utilizzando la seguente formula:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N}}$$

Nel caso specifico, N è il numero di intervalli dell'istogramma, x_i è la media all'interno dello specifico intervallo (per ognuno dei quali l'istogramma memorizza somma e conteggio dei valori), mentre $\overline{x}$ è la media complessiva, cioè il tempo medio per operazione calcolato in precedenza.

Un altra informazione molto importante, che viene aggiunta alla risposta, è il *throughput* medio. Questo dato è calcolato in termini di operazioni al secondo, sfruttando nuovamente la struttura utilizzata per l'istogramma.

Ai dati statistici appena citati, se ne aggiungono altri due: il 50° percentile (cioè la mediana) ed il 99° percentile.
Non avendo mai a disposizione tutti i dati[5], questi due valori vengono stimati partendo dall'istogramma. In particolare, individuato l'intervallo sotto il quale ricade il 50% (o 99%) dei dati, viene utilizzata la media interna a tale intervallo. Questa approssimazione è tale che per infiniti intervalli si ottengono i valori esatti di questi due dati statistici.

4.5 Funzionamento

Dopo aver trattato l'architettura del sistema (4.2) ed il modello dati utilizzato (4.4), è giunto il momento di concentrarsi sul funzionamento del software di benchmark.

4.5.1 Creazione del carico di lavoro

Uno degli elementi chiave del software sviluppato è sicuramente il modo in cui vengono generate tutte le operazioni destinate alla base di dati.

Questa funzionalità è implementata mediante due componenti: il *workload generator* ed il *workload materializer*. Questi sono stati pensati come interfacce, in modo da essere estensibili per compatibilità con diverse tipologie di carichi di lavoro. Di ognuna di esse è stata realizzata un'implementazione concreta, idonea per l'ambito IoT.

Workload generator

Il workload generator ha il compito di creare lo scheletro delle operazioni da mandare alla base di dati.
Riceve come parametro una parte, abbastanza estesa, del file JSON di configurazione: per l'esattezza tutta la sezione contenente informazioni relative alle tabelle e alle operazioni da svolgere.

Con le informazioni in suo possesso, il generator crea delle strutture dati che rappresentano operazioni di diverse tipologie, rispettando le proporzioni richieste dal client. Queste strutture dati contengono unicamente le informazioni fondamentali per poter eseguire l'operazione corrispondente.
Nel caso di un'operazione di creazione la struttura conterrà lo schema della tabella;

[5]Ogni worker vede solo i risultati delle operazioni che lui ha eseguito, mentre il coordinator riceve solo degli aggregati (istogrammi).

per una lettura sarà sufficiente il nome della relazione; per una scrittura servirà sia il nome che il numero di righe da inserire.

Come si può intuire, questo modo di modellare le operazioni sulla base di dati è estremamente leggero dal punto di vista della memoria occupata da ciascuna di esse.

Workload materializer

Il workload materializer ha il compito di generare concretamente i dati da inserire nelle tabelle e di ricavare i valori da utilizzare nei filtri delle interrogazioni.
Per svolgere il suo compito, riceve gli stessi parametri del generator, ma i suoi metodi, per essere invocati, richiedono diversi argomenti.

La funzione responsabile di generare i dati da inserire nelle tabelle restituisce un *iteratore*[6], in modo che in memoria sia necessario mantenere solo una riga per volta. Oltre a tale funzione, il workload materializer fornisce metodi per la generazione della chiavi, da utilizzare per le letture. In particolare offre sia una funzione che restituisce una singola chiave, per la selezione di un'unica riga, sia una che ne produce due, per letture di blocchi di dati contigui.
Il materializer tiene anche traccia dei nomi dei campi da utilizzare per le join.

4.5.2 Ciclo di vita di un benchmark

L'esecuzione di un benchmark passa attraverso diversi stati. Si vuole ora analizzare il modo in cui il software gestisce la richiesta di avvio di un benchmark e come questo si svolge.

1. Il client manda, attraverso l'API REST, esposta dal web server, un file JSON di configurazione per richiedere l'avvio di un nuovo benchmark. Questo file viene opportunamente validato e si verifica che tutti i parametri necessari per l'esecuzione del carico di lavoro siano opportunamente settati.

2. Se il file ricevuto è corretto, il sistema genera un UUID, lo assegna al nuovo benchmark e lo restituisce come risposta al client.

3. A questo punto viene creato un documento in MongoDB, riferito al nuovo benchmark, in cui sono memorizzati: l'identificativo, il timestamp di inizio, il JSON di configurazione, lo stato del cluster al momento della richiesta e

[6] Un "iteratore" (*Iterator*) è uno strumento per accedere agli elementi di una collezione uno alla volta, mediante due metodi principali: *hasNext* che permette di sapere se c'è ancora qualcosa da leggere e *next* che restituisce l'elemento successivo.

lo stato del benchmark stesso. Questo si trova al momento nella condizione *pending*: non è ancora stato avviato, ma si trova in attesa.

4. La fase successiva prevede alcuni controlli, relativi a dei parametri del file di configurazione, che solo il coordinator può eseguire[7]. Il motivo è che il coordinator è l'unico componente a conoscere, grazie all'attore figlio member manager, tutti gli indirizzi dei nodi con ruolo worker.
   Quest'informazione viene utilizzata per verificare se è possibile rispettare la volontà dell'utente, riguardo al minimo numero di nodi worker per l'esecuzione del carico di lavoro. In caso contrario lo stato del benchmark passa a *failed*.

5. Se tutti i controlli sono stati superati, il coordinator istanzia il workload generator e lo utilizza per l'inizializzazione della base dati, creando le tabelle.

6. A questo punto, il coordinator crea tanti attori worker su ogni nodo con ruolo idoneo, quanti richiesti dal client nel file di configurazione. Oltre a questi, crea anche un connection manager, responsabile di gestire le connessioni al database, e lo comunica ai worker.

7. Dopo queste operazioni preliminari, il coordinator genera (sempre grazie al workload generator) le operazioni di inserimento necessarie per il riempimento della base di dati e le distribuisce equamente tra i worker.
   Il benchmark passa a questo punto alla fase di *loading*.

8. Ogni worker esegue le operazioni di inserimento, generando i dati con il supporto del workload materializer.
   Al termine del lavoro assegnatogli, il worker manda un messaggio al coordinator.

9. Quando tutti i worker hanno ultimato la fase di caricamento dei dati iniziali nel database, si può passare alla fase successiva.
   Il coordinator utilizza nuovamente il workload generator per ottenere le operazioni da distribuire tra i worker.
   Il benchmark passa quindi alla fase di *running*, ovvero l'esecuzione vera e propria.

10. Ogni worker riceve la sua parte di operazioni da elaborare e la esegue, con il supporto del workload materializer.
    Le singole interrogazioni alla base di dati sono eseguite da dei metodi che calcolano il tempo di risposta e lo restituiscono al worker. Questo, dopo aver eseguito tutte le operazioni, avrà una grande collezione di tempi che

[7] La validazione del JSON, menzionata in precedenza, è eseguita non appena questo viene ricevuto, quindi avviene internamente al web server.

potrà aggregare in un istogramma (come illustrato in 4.4.3) da mandare al coordinator.

11. Man mano che i worker terminano di eseguire il loro compito, il coordinator riceve gli istogrammi parziali e provvede ad aggregarli. Quando anche l'ultimo worker ha completato le operazioni assegnategli, il coordinator può calcolare il risultato finale e scriverlo su MongoDB.
    Il benchmark passa quindi allo stato *completed*.

Si noti che se durante l'esecuzione del benchmark si verifica un errore, questo viene notificato dal worker al coordinator, che provvede a fermare le operazioni ancora in corso e a cambiare lo stato in *failed*.

4.5.3 Coordinator come macchina a stati finiti

Dopo aver analizzato il ciclo di vita di un benchmark, si può forse intuire come l'attore coordinator possa essere modellato come una macchina a stati finiti. Questo infatti possiede uno stato interno, che cambia sulla base dei messaggi ricevuti dagli altri attori del sistema e della condizione attuale.

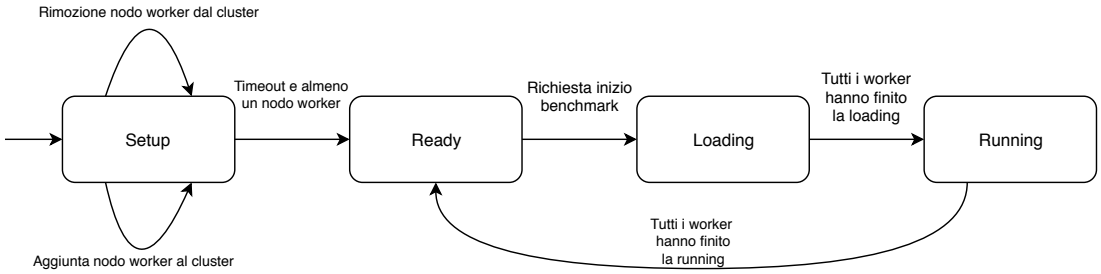


Figura 4.5: Coordinator come FSM

La figura 4.5 mostra gli stati che modellano il comportamento dell'attore coordinator e gli eventi che portano a transizioni da uno stato all'altro.

Il coordinator parte da uno stato di *setup*, in cui aspetta che i nodi con ruolo worker si aggreghino al cluster. Il tempo di attesa è un parametro, modificabile al lancio del programma.
Allo scadere del timer, se almeno un nodo con ruolo worker si è unito al cluster, il coordinator passa allo stato di *ready*. Durante questa transizione si porta dietro una lista dei nodi con ruolo worker.

Il coordinator resta nella fase ready fino a quando non riceve un messaggio, da parte del web server, con la richiesta di avviare un benchmark. A questo punto avviene quanto descritto nella sezione 4.5.2.
Terminata l'operazione assegnatagli, il coordinator ritorna allo stato di ready, pronto a servire nuove richieste.

Un benchmark può essere avviato unicamente quando il coordinator si trova nella fase ready. In tutti gli altri stati, in caso di nuove richieste, verrà notificato che l'attore non è pronto.
Tutti gli altri tipi di messaggi, che il web server può mandare al coordinator, sono gestiti dal serve request actor (di cui si è parlato nella sezione 4.2.2). Questo si occupa di fare da tramite tra l'interfaccia REST e l'API di MongoDB, consentendo di leggere i dati salvati anche quando il coordinator è impegnato nell'esecuzione di un benchmark.

Per modellare l'attore come la macchina a stati finiti appena illustrata, il coordinator è stato realizzato con il supporto di Akka FSM (si veda 2.3.1).

4.5.4 Worker: dettagli e funzionamento

Si è già discusso il ruolo svolto dagli attori worker nella sezione 4.2.3, ma è ora necessario chiarirne il funzionamento.
Quando un attore worker viene istanziato dal coordinator, riceve come parametri tre elementi:

- il riferimento allo stesso coordinator;
- il riferimento al connection manager responsabile dello specifico nodo;
- un numero intero chiamato *seed*.

I riferimenti agli altri attori consentono una comunicazione diretta. Si noti che è il coordinator a creare il connection manager per un nodo ed a "presentarlo" ad ogni worker. A questi fornisce anche il proprio riferimento, in modo che i processi possano notificargli gli esiti delle operazioni richieste[8].

[8]Senza il riferimento esplicito al coordinator, un worker potrebbe unicamente rispondere ai messaggi che questo gli invia, ma non avrebbe l'opportunità di prendere l'iniziativa nella comunicazione.

Il seed, infine, viene passato dal worker al workload materializer, che lo utilizza per la generazione delle chiavi dei record da inserire nelle tabelle. Questo particolare numero intero viene generato dal coordinator, in quanto è l'unico a conoscere tutti gli attori worker.

Heartbeat

Prima ancora che possa iniziare a ricevere e rispondere ai messaggi, l'attore worker appena creato deve programmare l'invio periodico di un messaggio di *heartbeat* al coordinator. L'heartbeat è un messaggio particolare, con l'unico scopo di notificare che il worker è ancora "vivo" e sta lavorando.
Se per un certo intervallo di tempo il coordinator non riceve l'heartbeat da un worker, lo considera "morto" e, dopo aver fermato il processo corrispondente, ne crea uno nuovo a cui assegna le stesse operazioni da svolgere. Si noti che tale operazione non è sempre possibile; in questi casi si preferisce considerare fallito il benchmark, piuttosto che eseguirlo con un numero di worker inferiori a quelli richiesti dal client.

L'invio periodico di questo messaggio di heartbeat viene effettuata mediante lo *scheduler*, messo a disposizione da Akka. Questo permette di mandare un messaggio, ad uno specifico attore, ad intervalli di tempo regolari. Una volta configurato, restituisce un *Cancellable*, che consente di fermare l'invio periodico di messaggi.

Elaborazione delle richieste

Subito dopo che l'attore worker è stato creato, riceve dal coordinator un messaggio contenente le operazioni da eseguire.
Per poter continuare il proprio lavoro l'attore ha bisogno di una connessione alla base di dati; per questo motivo invia un messaggio al connection manager, che gliene fornisce una[9].

Una volta ottenuta la connessione, il worker può iniziare ad utilizzarla per eseguire le operazioni che gli sono state assegnate.
Nel caso in cui queste fossero molto numerose[10], il worker le suddivide in gruppi. Ognuno di questi viene processato come un messaggio diverso, in maniera tale che il worker possa sbloccarsi ad intervalli regolari. Quando un attore sta gestendo un messaggio non può elaborarne altri, per questo motivo è importante che il tempo che il worker dedica ad un singolo messaggio non sia eccessivo.

[9] I dettagli di questo scambio di messaggi tra worker e connection manager sono già stati illustrati nella sezione 4.2.3.

[10] Anche in questo caso la soglia è un parametro che può essere opportunamente settato all'avvio.

L'elaborazione di ogni operazione, richiesta dal coordinator, si traduce in un'interrogazione alla base di dati da parte del worker. Questa genera come output o un errore o un valore in millisecondi, che rappresenta il tempo di esecuzione.
Quando vi è un errore, il worker notifica il fallimento al coordinator. Diversamente tutte le misurazioni delle latenze vengono aggregate in un istogramma parziale.
È compito del coordinator mettere insieme tutti gli istogrammi parziali, ricevuti dai diversi worker.

Terminate tutte le operazioni assegnategli, il worker restituisce la connessione al connection manager.

4.6 Risultati

Il formato dei risultati prodotti dal framework (si veda 4.4.3) non è di immediata comprensione, per questo motivo i dati raccolti sono stati rielaborati.

4.6.1 Tecnica per l'elaborazione dei risultati

L'elaborazione dei benchamark effettuati (memorizzati come JSON) è stata realizzata mediante il semplice utilizzo di script bash e R. Il procedimento che segue è stato applicato ad ogni benchmark.

In un primo momento si sono estratte le informazioni di maggior interesse dal JSON, utilizzando un semplice script bash che sfrutta jq[11], e le si sono esportate in formato CSV.
Successivamente, partendo da questa nuova rappresentazione dei risultati, si è realizzato un istogramma sfruttando uno script R (si veda 2.5). In questo grafico, l'asse x rappresenta le latenze, espresse in millisecondi, misurate effettuando le operazioni sulla base di dati. L'asse y contiene invece il numero di operazioni, espresse su una scala logaritmica.
Le latenze sono divise in intervalli con estremo inferiore incluso ed estremo superiore escluso. Il numero di tali intervalli dipende da un parametro del file di configurazione. Anche la massima latenza misurata è definita da un parametro del JSON, inizialmente fornito dal client all'avvio del benchmark. L'ultimo intervallo è aperto a destra e raccoglie tutte le operazioni, il cui tempo di esecuzione è maggiore o uguale al massimo stabilito dall'utente.
Complessivamente l'istogramma fornisce per ogni intervallo il numero di operazioni il cui tempo di esecuzione ricade al suo interno.

[11] Un processatore di JSON semplice e leggero, utilizzabile da linea di comando.

Dopo aver prodotto per ogni benchmark il CSV e l'istogramma, si è generato un ulteriore file per il confronto dei risultati ottenuti dalle diverse basi di dati.
Sempre attraverso uno script bash, si è prodotto un ulteriore CSV contenente sia una tabella riassuntiva delle configurazioni di ogni benchmark, sia un confronto delle prestazioni di CockroachDB e NuoDB per ogni carico di lavoro.
In particolare il file riporta, per ogni benchmark, la latenza media di NuoDB e di CockroachDB e la relativa variazione percentuale, calcolata con la seguente formula:

$$variazione\ percentuale = \left[\left(\frac{latenza_{NuoDB}}{latenza_{CockroachDB}} \cdot 100\right) - 100\right] \% \tag{4.1}$$

Oltre questi dati, il CSV contiene anche il throughput medio di CockroachDB, quello di NuoDB e la variazione percentuale dei due valori.

4.6.2 Analisi dei risultati

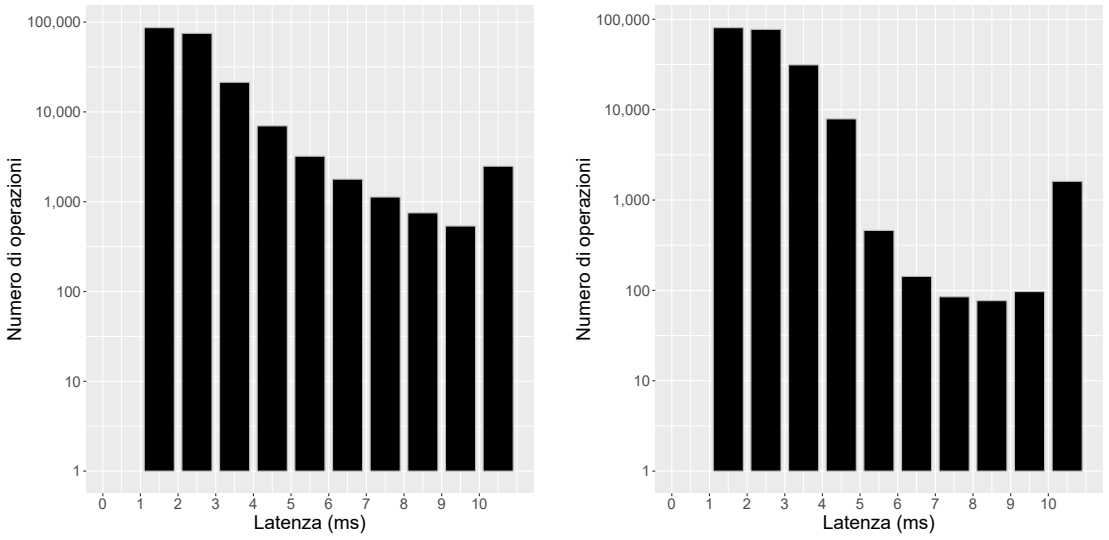
Si analizzeranno ora alcuni dei benchmark più significativi, utilizzandoli per effettuare un confronto tra CockroachDB e NuoDB.

Tutti i benchmark che si andranno ad esaminare sono stati effettuati nelle stesse condizioni.

- La base di dati NewSQL è stata installata su tre nodi.
- Le interrogazioni sono state effettuate da dieci processi worker, istanziati sulla stessa macchina fisica, utilizzando altrettante connessioni al database.
- Per entrambe le basi di dati è stato richiesto un fattore di replicazione pari a tre. Questo significa che ogni dato è replicato su ogni nodo (essendo anche questi tre).
- Le tabelle utilizzate hanno le seguenti dimensioni:
  - *dispositivi*: 200 righe e 4 colonne;
  - *misurazioni*: 300.000 righe e 5 colonne. Con 1500 misurazioni per ogni dispositivo;
  - *periodi di validità*: 10.000 righe e 6 colonne. Con 50 periodi di validità per ogni dispositivo.
- La dimensione complessiva della base di dati utilizzata per i benchmark è di circa 40 GiB.
- È stato creato un indice sulla tabella dei periodi di validità, per velocizzare le operazioni di join.

Operazioni di lettura di singole righe

Il primo workload in analisi è costituito da 200.000 operazioni di lettura per chiave primaria. Di conseguenza ogni interrogazione seleziona una singola riga della tabella coinvolta.
Le operazioni sono state equamente distribuite tra le tre tabelle descritte nella sezione 4.4.1.



(a) *CockroachDB* (b) *NuoDB*

Figura 4.6: Latenze per operazioni di sola lettura di singole righe

Come si può osservare dagli istogrammi riportati nella figura 4.6, le prestazioni delle due basi di dati NewSQL sono simili per le operazioni di lettura di singole righe, anche se NuoDB ha una latenza media leggermente inferiore.

	Latenza media (ms)	Throughput medio (ops/s)
CockroachDB	2,035	491,341
NuoDB	1,935	516,837
Variazione percentuale[12]	−4,933%	+5,189%

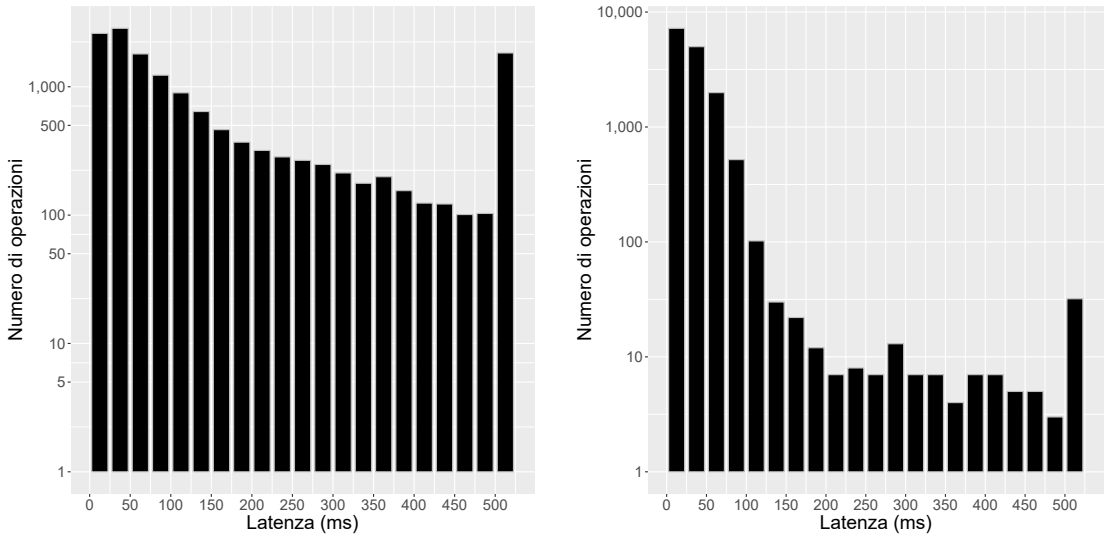
Tabella 4.1: Tabella dei risultati per operazioni di sola lettura di singole righe

[12]Calcolata utilizzando la formula 4.1.

La tabella 4.1 conferma quanto affermato in precedenza. Per le operazioni di sola lettura per chiave primaria, le prestazioni di NuoDB sono leggermente migliori rispetto a quelle di CockroachDB; sia per la latenza che per il throughput.

Operazioni di scrittura

Per il prossimo benchmark sono state effettuate 15.000 operazioni di scrittura. Il numero di righe inserite da ogni interrogazione varia da uno a cinque in maniera casuale.
Anche in questo caso le interrogazioni sono state equamente distribuite tra le tre tabelle



(a) *CockroachDB* (b) *NuoDB*

Figura 4.7: Latenze per operazioni di sola scrittura

In questo caso gli istogrammi della figura 4.7 mostrano che NuoDB è più performante di CockroachDB nelle operazioni di inserimento. Ciò è dimostrato dalla presenza, nel secondo istogramma, di un maggior numero di operazioni nei primi intervalli.
Come si può notare la scala dei valori sull'asse y è diversa tra le due figure, mettendo il risalto come per NuoDB molte più operazioni ricadano nei primi due intervalli di latenze.

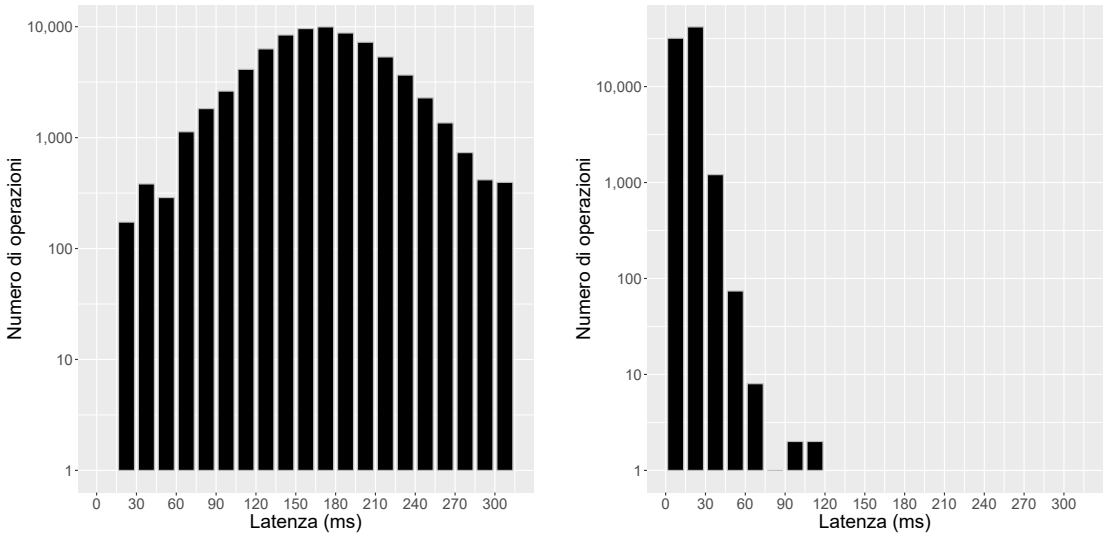
	Latenza media (ms)	Throughput medio (ops/s)
CockroachDB	210,361	4,754
NuoDB	34,664	28,849
Variazione percentuale	−83,522%	+506,863%

Tabella 4.2: Tabella dei risultati per operazioni di sola scrittura

La tabella 4.2 evidenzia le differenze di prestazioni tra le due basi di dati NewSQL, per quanto riguarda le operazioni di scrittura. Un dato che spicca particolarmente è la variazione percentuale tra il throughput medio di NuoDB e quello di CockroachDB. A parità di tempo, in media, NuoDB esegue sei volte il numero di scritture effettuate da CockroachDB.

Operazioni di lettura con join (1)

Si analizzano ora le prestazioni delle due basi di dati NewSQL per un carico di lavoro composto da 75.000 operazioni di lettura, che richiedono la join delle tre relazioni.
Le interrogazioni selezionano unicamente due righe della tabella risultante dalle due inner join.



(a) *CockroachDB* (b) *NuoDB*

Figura 4.8: Latenze per operazioni di lettura con join (2 righe selezionate)

Gli istogrammi della figura 4.8 mostrano la notevole differenza di prestazioni tra CockroachDB e NuoDB. Quest'ultimo ha tempi di esecuzione medi notevolmente migliori per questo tipo di workload. Tale fatto è riscontrabile dal maggior numero di operazioni nella parte sinistra dell'istogramma, dove le latenze sono minori.

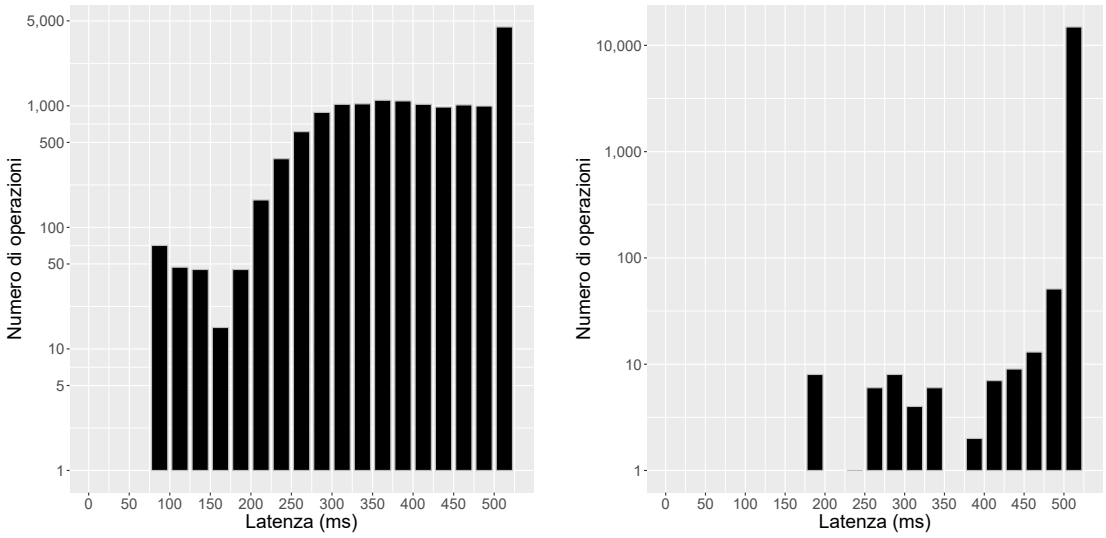
	Latenza media (ms)	Throughput medio (ops/s)
CockroachDB	168,909	5,92
NuoDB	15,871	63,007
Variazione percentuale	−90,604%	+964,243%

Tabella 4.3: Tabella dei risultati per operazioni di lettura con join (2 righe selezionate)

La tabella 4.3 ribadisce quanto appena detto e mette in risalto come le latenze ed il throughput delle due basi di dati differiscano di un ordine di grandezza, per questo tipo di workload.

Operazioni di lettura con join (2)

Il prossimo workload è simile a quello appena analizzato, ma mette in risalto i cambiamenti delle prestazioni delle due basi di dati all'aumentare del numero di righe selezionate.
In questo caso le operazioni effettuate sono state 15.000 letture con join delle tre tabelle, che selezionano 300 righe.



(a) *CockroachDB* (b) *NuoDB*

Figura 4.9: Latenze per operazioni di lettura con join (300 righe selezionate)

Negli istogrammi della figura 4.9 è osservabile un cambiamento notevole delle prestazioni di NuoDB rispetto a quanto visto per il benchmark precedente (figura 4.8).

	Latenza media (ms)	Throughput medio (ops/s)
CockroachDB	431,248	2,319
NuoDB	1138,067	0,879
Variazione percentuale	+163,901%	−62,107%

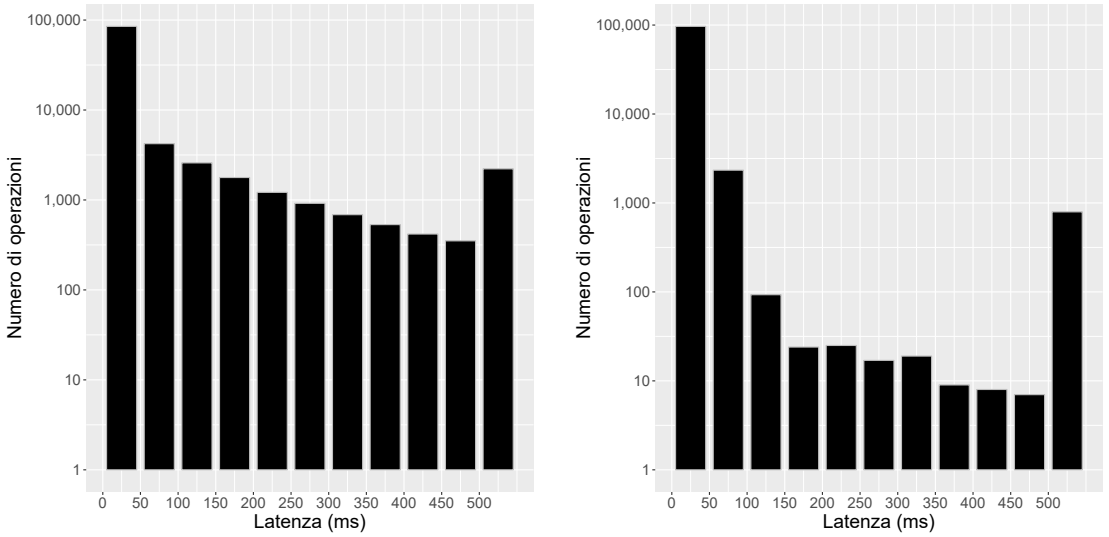
Tabella 4.4: Tabella dei risultati per operazioni di lettura con join (300 righe selezionate)

La tabella 4.4 mette ancora più in risalto come, all'aumentare del numero di righe selezionate, CockroachDB abbia tempi di esecuzione migliori rispetto a NuoDB.

Operazioni miste

L'ultimo benchmark in analisi è stato eseguito con un workload, costituito da operazioni di diverse tipologie. In particolare sono state effettuate 100.000 interrogazioni, di cui 80% letture e 20% scritture.
L'1% delle letture ha richiesto la join delle tre tabelle, producendo in uscita due

righe. Le restanti richieste di lettura sono state effettuate tutte per chiave primaria, restituendo un unico record come output. Le scritture, infine, sono state effettuate inserendo un numero di righe variabile da uno a cinque.
Come per gli altri workload analizzati, tutte le operazioni sono state distribuite equamente tra le tre tabelle.



(a) *CockroachDB* (b) *NuoDB*

Figura 4.10: Latenze per operazioni miste

Come si può osservare dagli istogrammi della figura 4.10, anche per un carico di lavoro costituito da operazioni diverse, NuoDB offre prestazioni migliori.

	Latenza media (ms)	Throughput medio (ops/s)
CockroachDB	50,888	19,651
NuoDB	46,693	21,417
Variazione percentuale	−8,244%	+8,986%

Tabella 4.5: Tabella dei risultati per operazioni miste

La tabella 4.5 consente di osservare che le differenze di latenza e throughput tra le due basi di dati, per questo workload, non sono notevoli. Nonostante ciò, NuoDB resta comunque la scelta migliore per questo specifico insieme di operazioni.

Si può affermare che la motivazione principale delle migliori prestazioni di NuoDB, in questo benchmark, sia legata al numero di righe selezionate dalle operazioni di

lettura.
Questo workload prevedeva che la maggior parte delle operazioni (l'80%) fossero letture, che selezionavano una o due righe. Come già osservato nei benchmark precedenti, questo tipo di operazioni è eseguito con maggior efficienza da NuoDB. Se i record selezionati dalle operazioni di lettura fossero stati maggiori, avremmo potuto osservare tempi migliori di esecuzione da parte di CockroachDB.

Capitolo 5

Conclusioni

In questo ultimo capitolo vengono utilizzati i risultati (appena presentati nella sezione 4.6) per effettuare un confronto tra le due basi di dati NewSQL.
Vengono inoltre forniti dei possibili casi d'uso per il progetto sviluppato e delle indicazioni su possibili sviluppi futuri.

5.1 Risultati ottenuti

Il percorso di tesi affrontato ha portato, in primo luogo, allo sviluppo del framework descritto nel capitolo 4. Questo software distribuito, in grado di effettuare benchmark con workload facilmente configurabili, è stato utilizzato allo scopo di raggiungere l'obiettivo inizialmente prefissato: confrontare le prestazioni di CockroachDB e NuoDB.

5.1.1 CockroachDB e NuoDB a confronto

Sulla base dei risultati presentati nella sezione 4.6, si può procedere con un confronto finale tra le due basi di dati NewSQL, oggetto di tesi.

Le caratteristiche di CockroachDB possono essere riassunte come segue.

- Vantaggi:
  - Sicuramente il primo grande vantaggio di CockroachDB è di essere open source. Di contro NuoDB fornisce gratuitamente solo una *Community Edition*, con diverse limitazioni.
  - Un altro punto a favore di CockroachDB è la presenza di una documentazione più vasta e di un forum di supporto, che semplificano l'installazione e la configurazione del prodotto.

  - CockroachDB offre anche un'interfaccia grafica ricca di informazioni e statistiche di utilizzo. Questa interfaccia è stata molto utile durante i benchmark per monitorarne l'andamento.
  - Per quanto riguarda le prestazioni, CockroachDB è risultato migliore di NuoDB nelle operazioni di lettura che coinvolgono numerosi record contigui, ovvero nelle scansioni per chiave primaria.

- Svantaggi:

  - In generale le prestazioni di CockroachDB sono risultate inferiori a quelle di NuoDB, in particolare per le operazioni di join e di scrittura.
  - CockroachDB non offre un load balancer integrato, per questo motivo è stato necessario l'utilizzo di quello fornito a pagamento da AWS. Senza il load balancer non sarebbe stato possibile distribuire equamente il carico di lavoro tra i nodi disponibili.
  - Si sono inoltre riscontrate diverse problematiche nell'esecuzione di operazioni di join senza un apposito indice.
    Quando si effettuavano tali tipi di interrogazioni, tra tabelle di grosse dimensioni, si verificava un utilizzo eccessivo di memoria. Il risultato era che il processo di CockroachDB veniva ucciso dall'*out of memory killer*[1]. Per questo motivo tutti i benchmark sono stati eseguiti con un indice sulla tabella relativa ai periodi di validità. Tale indice nel caso di NuoDB ha migliorato i tempi di esecuzione, ma non era strettamente necessario per il corretto funzionamento.

Dopo aver esaminato vantaggi e svantaggi di CockroachDB, si può ora procedere alla valutazione dell'altra basi di dati NewSQL, analizzata nel corso della tesi: NuoDB.

- Vantaggi:

  - NuoDB garantisce mediamente prestazioni migliori di CockroachDB, in particolare per le operazioni di join e scrittura.
  - La presenza di un load balancer integrato, configurabile con diverse politiche per la distribuzione del carico, è un altro grande vantaggio offerto da NuoDB.

- Svantaggi:

[1] L'out of memory killer è un processo utilizzato dal kernel linux, quando la memoria disponibile diminuisce ad un livello critico. Il suo compito è selezionare uno o più processi da "killare", in base a quanta memoria questi stanno utilizzando (anche altri fattori influenzano la scelta del processo).

- Uno dei grandi difetti di NuoDB è il fatto di non essere open source. La *Community Edition*, fornita gratuitamente, ha troppe limitazioni anche solo per effettuare dei benchmark seri ed è quindi impossibile da utilizzare in produzione. In particolare questa versione consente di avere solo tre transaction engine ed un unico storage manager. La seconda limitazione si traduce nell'impossibilità di avere la replicazione dei dati, con conseguente assenza di tolleranza ai guasti.
  Tutti i benchmark riportati in precedenza (sezione 4.6) sono stati effettuati con l'*Enterprise Edition* di NuoDB, cioè la versione completa, a pagamento, senza le limitazioni appena segnalate.

- La documentazione fornita da NuoDB è risultata spesso confusionaria ed a tratti incompleta. Questo ha influenzato i tempi di installazione in maniera negativa, anche a causa della mancanza di un forum di supporto.

- NuoDB, a differenza di CockroachDB, non fornisce un'interfaccia grafica per il monitoraggio della base di dati.
  Questa mancanza, anche se non grave, ha reso più scomodo l'osservazione dell'andamento dei benchmark.

- Un altro difetto molto importante, già menzionato in precedenza, è il calo di prestazioni di NuoDB all'aumentare del numero di righe selezionate nelle operazioni di scan.

Dopo aver analizzato i principali pregi e difetti delle due basi di dati NewSQL, esaminate durante il percorso di tesi, l'autore ritiene opportuno fornire un proprio parere a riguardo.

Nonostante mediamente NuoDB offra prestazioni migliori rispetto a CockroachDB, quest'ultimo rappresenta probabilmente la scelta migliore tra le due basi di dati. Il fatto che sia open source, e che fornisca una documentazione più vasta, lo rendono più idoneo nel caso si volesse adottare una soluzione NewSQL. L'unico caso in cui si ritiene che NuoDB sia la scelta più appropriata è uno scenario nel quale siano necessarie migliori prestazioni nelle join ed in scrittura.
Tuttavia entrambe queste base di dati non sono ancora sufficientemente mature per un utilizzo intensivo in produzione. Di conseguenza, nel caso si avesse necessità di una buona scalabilità, la scelta migliore è quella di rinunciare al modello relazionale a favore di un database NoSQL. Un'alternativa potrebbe essere la valutazione delle prestazioni di altre basi di dati NewSQL, più mature rispetto a CockroachDB e NuoDB.

5.2 Casi d'uso

Come già riportato all'inizio di questo capitolo, si può affermare che i risultati del progetto di tesi siano due. Da un lato si è sviluppato il framework descritto nel capitolo 4; dall'altro si è utilizzato tale software per effettuare dei benchmark di CockroachDB e NuoDB.
A questo punto si possono analizzare i possibili utilizzi di questi risultati.

5.2.1 Utilizzi dei benchmark

I benchmark effettuati sulle due database NewSQL consentono di conoscere come queste basi di dati si comportino con diversi carichi di lavoro.
Le statistiche raccolte rendono possibile un confronto con i dati relativi ad altri database di utilizzo più comune; questo permette una scelta consapevole del prodotto più idoneo alle esigenze.

Si ritiene importante ricordare che le basi di dati appartenenti alla categoria NewSQL sono una tecnologia relativamente recente e di conseguenza non sono state ancora largamente collaudate. Per tale motivo questa tesi può essere utilizzata per avere maggiori dettagli sulle prestazioni, l'architettura ed il comportamento (in diverse situazioni e con diversi carichi di lavoro) di due prodotti, appartenenti a questa categoria di basi di dati.

5.2.2 Utilizzi del framework

Si analizzeranno ora i possibili impieghi del framework, sviluppato allo scopo di effettuare i benchmark di CockroachDB e NuoDB.

Al momento il software può solo essere impiegato per effettuare ulteriori test delle due basi dati NewSQL e per ricavarne gli esiti. Questi benchmark possono essere effettuati sfruttando diversi nodi per mandare le interrogazioni al database che si desidera testare. Come già ripetuto più volte, i workload sono configurabili facilmente con il file JSON, inviato per richiedere l'avvio del benchmark.

5.3 Sviluppi futuri

Il framework è stato sviluppato in modo da essere facilmente estensibile all'utilizzo con altre basi di dati, purché queste supportino JDBC. Di conseguenza in futuro possono essere scritte le classi necessarie alla gestione di altre basi di dati, implementando le interfacce già presenti.

Tra le estensioni che potrebbero essere implementante, una di particolare interesse sarebbe la possibilità di generare anche carichi di lavoro OLAP (*On-Line Analytical Processing*) e non unicamente OLTP (*On-Line Transaction Processing*)[2]. Questa funzionalità consentirebbe di valutare, tra diverse basi di dati in analisi, quale offra prestazioni migliori nel caso, occasionalmente, fossero necessarie interrogazioni particolarmente complesse. Si deve comunque tenere presente che per questo tipo di carico di lavoro esistono i data warehouse, progettati appositamente a tale scopo.

Un'altra funzionalità, che potrebbe essere aggiunta al framework, è la possibilità di simulare un partizionamento della rete o la caduta di nodi della base di dati. In questo modo sarebbe possibile osservare i cambiamenti delle prestazioni al verificarsi di guasti. Questo aspetto può essere di particolare interesse per una scelta più consapevole della base di dati da adottare per una specifica applicazione.

Come spiegato nella sezione 4.5.1, il workload generator e materializer sono due interfacce, di cui è stata realizzata un'implementazione concreta adatta a simulare un ambito IoT. In futuro potrebbe essere utile sviluppare altre implementazioni di queste interfacce, in maniera tale da poter eseguire benchmark con carichi di lavoro appartenenti a contesti diversi.

Oltre all'implementazione di nuove funzionalità per il framework, in futuro si potrebbero effettuare ulteriori benchmark su CockroachDB e NuoDB. In particolare si potrebbero effettuare dei test aumentando sia il numero di nodi a disposizione dell'applicazione, sia quelli su cui viene installato la base di dati.
Un'altra possibilità è quella di ripetere i benchmark con database di maggiori dimensioni.

[2]La differenza principale tra un carico di lavoro OLAP ed uno OLTP è che il primo è costituito da interrogazioni più complesse (ad esempio con calcolo di aggregati ed utilizzo di GROUP BY), mentre il secondo è formato da interrogazioni più semplici su cui devono essere garantite le proprietà tipiche delle transazioni.

Appendice A

Tabelle dati benchmark

Questa appendice riporta, in forma tabulare, i dati ottenuti dai diversi benchmark, analizzati nella sezione 4.6 (in cui sono rappresentati sotto forma di istogramma).

A.1 Operazioni di lettura di singole righe

La seguente tabella mostra i dati raccolti durante il benchmark di 200.000 operazioni di lettura per chiave primaria (selezione di una singola riga).

Intervalli di latenze (ms)	Numero operazioni CockroachDB	Numero operazioni NuoDB
[0; 1)	0	0
[1; 2)	86707	80835
[2; 3)	75024	77469
[3; 4)	21382	31315
[4; 5)	6993	7906
[5; 6)	3211	461
[6; 7)	1783	143
[7; 8)	1128	85
[8; 9)	751	77
[9; 10)	536	97
[10; ∞)	2485	1612
	200.000	200.000

Tabella A.1: Tabella dati per operazioni operazioni di sola lettura di singole righe

A.2 Operazioni di scrittura

In questa sezione sono riportati i dati inerenti al benchmark di 15.000 operazioni di scrittura. Si ricorda che il numero di righe inserite varia da uno a cinque in modo casuale.

Intervalli di latenze (ms)	Numero operazioni CockroachDB	Numero operazioni NuoDB
[0; 25)	2604	7216
[25; 50)	2851	4996
[50; 75)	1801	1990
[75; 100)	1230	520
[100; 125)	896	102
[125; 150)	641	30
[150; 175)	463	22
[175; 200)	370	12
[200; 225)	319	7
[225; 250)	284	8
[250; 275)	267	7
[275; 300)	248	13
[300; 325)	213	7
[325; 350)	177	7
[350; 375)	199	4
[375; 400)	155	7
[400; 425)	124	7
[425; 450)	122	5
[450; 475)	101	5
[475; 500)	103	3
[500; ∞)	1832	32
	15.000	15.000

Tabella A.2: Tabella dati per operazioni operazioni di sola scrittura

A.3 Operazioni di lettura con join (1)

Sono qui riportati, in forma tabulare, i dati relativi al benchmark eseguito con un workload composto da 15.000 operazioni di join.
In tutte le interrogazioni sono state selezionate solamente due righe dalla tabella, risultante dalle due inner join.

Intervalli di latenze (ms)	Numero operazioni CockroachDB	Numero operazioni NuoDB
[0; 15)	0	31887
[15; 30)	173	41818
[30; 45)	382	1208
[45; 60)	288	74
[60; 75)	1128	8
[75; 90)	1825	1
[90; 105)	2624	2
[105; 120)	4127	2
[120; 135)	6303	0
[135; 150)	8411	0
[150; 165)	9610	0
[165; 180)	9945	0
[180; 195)	8768	0
[195; 210)	7241	0
[210; 225)	5334	0
[225; 240)	3662	0
[240; 255)	2280	0
[255; 270)	1356	0
[270; 285)	732	0
[285; 300)	416	0
[300; ∞)	395	0
	75.000	75.000

Tabella A.3: Tabella dati per operazioni operazioni di lettura con join (2 righe selezionate)

A.4 Operazioni di lettura con join (2)

La tabella riporta i dati relativi al benchmark composto da 15.000 join, con selezione di 300 record.

Intervalli di latenze (ms)	Numero operazioni CockroachDB	Numero operazioni NuoDB
[0; 25)	0	0
[25; 50)	0	0
[50; 75)	0	0
[75; 100)	71	0
[100; 125)	47	0
[125; 150)	45	0
[150; 175)	15	0
[175; 200)	45	8
[200; 225)	168	0
[225; 250)	367	1
[250; 275)	613	6
[275; 300)	883	8
[300; 325)	1029	4
[325; 350)	1038	6
[350; 375)	1111	0
[375; 400)	1097	2
[400; 425)	1029	7
[425; 450)	977	9
[450; 475)	1017	13
[475; 500)	996	51
[500; ∞)	4452	14885
	15.000	15.000

Tabella A.4: Tabella dati per operazioni operazioni di lettura con join (300 righe selezionate)

A.5 Operazioni miste

Questi dati fanno riferimento al benchmark eseguito con un workload composto da 100.000 interrogazioni di diverse tipologie:

- 80% letture
  - 1% join delle tre tabelle, con selezione di due righe
  - 99% letture per chiave primaria (singola riga in uscita)
- 20% scritture (numero di righe inserite variabile da uno a cinque).

Intervalli di latenze (ms)	Numero operazioni CockroachDB	Numero operazioni NuoDB
[0; 50)	85080	96667
[50; 100)	4223	2337
[100; 150)	2583	93
[150; 200)	1776	24
[200; 250)	1215	25
[250; 300)	918	17
[300; 350)	686	19
[350; 400)	532	9
[400; 450)	418	8
[450; 500)	351	7
[500; ∞)	2218	794
	100.000	100.000

Tabella A.5: Tabella dati per operazioni operazioni miste

Bibliografia

[1] Matthew Aslett. *What we talk about when we talk about NewSQL*. 451 Group, 2011.

[2] The gRPC Authors. *Frequently Asked Questions about gRPC*. 2019. URL: `http://grpc.io/faq/`.

[3] Babak Bashari Rad, Harrison Bhatti e Mohammad Ahmadi. «An Introduction to Docker and Analysis of its Performance». In: *IJCSNS International Journal of Computer Science and Network Security* 173 (mar. 2017), p. 8.

[4] André B. Bondi. «Characteristics of Scalability and Their Impact on Performance». In: *Proceedings of the 2nd International Workshop on Software and Performance*. 2000.

[5] George Coulouris, Jean Dollimore, Tim Kindberg e Gordon Blair. *Distributed Systems. Concepts and Design*. A cura di Addison-Wesley. 2012.

[6] Klint Finley. «CockroachDB is the resilient cloud software built by ex-Googlers». In: *Wired UK* (2014).

[7] Seth Gilbert e Nancy Lynch. «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services». In: *SIGACT News* (2002).

[8] Carl Hewitt. «Actor Model for Discretionary, Adaptive Concurrency». In: *CoRR* (2010).

[9] Carl Hewitt, Peter Bishop e Richard Steiger. «A Universal Modular ACTOR Formalism for Artificial Intelligence». In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. 1973.

[10] Lightbend Inc. *FSM*. 2019. URL: `https://doc.akka.io/docs/akka/2.5/fsm.html`.

[11] NuoDB Inc. *NuoDB At a Glance*. 2019. URL: `http://doc.nuodb.com`.

[12] NuoDB Inc. *Technical white paper: NuoDB Architecture*. 2018. URL: `go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf`.

[13] Cockroach Labs. *Reads and Writes in CockroachDB*. 2019. URL: `http://www.cockroachlabs.com/docs`.

[14] Andrew Pavlo e Matthew Aslett. «What's Really New with NewSQL?» In: *SIGMOD Record* 45.2 (2016).

[15] Karl Seguin. *Il Piccolo Libro di MongoDB*. 2015.

[16] Andrew S. Tanenbaum e Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2ª ed. Pearson Prentice Hall, 2007.

[17] W. N. Venables, D. M. Smith e the R Core Team. *An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics*. 2019.