SQL vs NoSQL vs NewSQL

Alessandro Imbastari
496124

Sommario

Teorema CAP

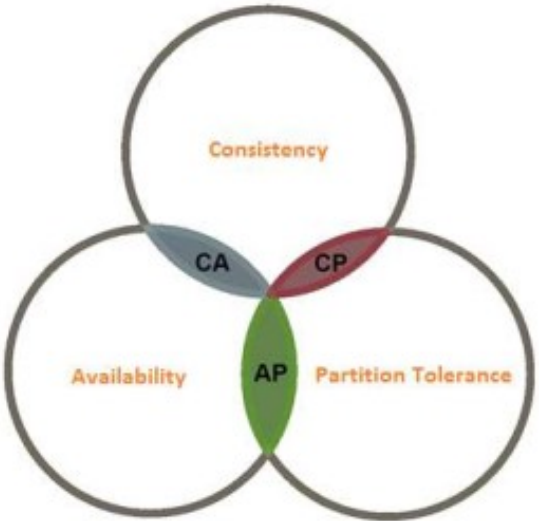

Figura 1: Teorema CAP

Tale teorema afferma che è impossibile per un servizio distribuito essere consistente, disponibile e tollerante alle partizioni nello stesso istante. Solo due di queste proprietà possono coesistere nello stesso momento. Una rappresentazione grafica è mostrata in Figura 1.
Consistency : Si riferisce alla capacità di mantenere il database in uno stato coerente in ogni momento, quindi che tutti i nodi vedano lo stesso stato del database nello stesso istante.

Available : Indica che i dati dovrebbero essere disponibili in ogni momento, quindi che ogni richiesta riceva una risposta.

Partition tolerance : Si riferisce alla capacità del sistema di tollerare le partizioni di rete, quindi il sistema continua a funzionare nonostante arbitrarie perdite di messaggi o malfunzionamenti.

Proprietà ACID vs BASE

Le proprietà ACID sono tipiche dei sistemi SQL, mentre quelle BASE sono tipiche dei sistemi NoSQL.
ACID è l’acronimo di Atomicity, Consistency, Isolation e Durability.

Atomicity : Tutte le operazioni della transazione verranno completate oppure nessuna operazione della transazione verrà eseguita, quindi l'esecuzione di una transazione deve essere per definizione o totale o nulla cioè non sono ammesse esecuzioni parziali.

Consistency : Garantisce che una transazione possa solo portare il database da uno stato valido a un altro. Da notare che è diversa dalla Consistency del teorema di CAP.

Isolation : Ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione.

Durability : Al completamento della transazione, l'operazione non sarà annullata, ma sarà persistente nel tempo.
BASE invece è l’acronimo di Basically Available, Soft state e Eventual consistency.

Basically Available : Indica che il sistema garantisce la disponibilità in termini del teorema di CAP.

Soft state : Indica che lo stato del sistema può cambiare nel tempo.

Eventual consistency : Indica che il sistema diventerà coerente nel tempo, dato che il sistema non riceve input durante quel tempo.

Scalability, Concurrency Control, Replication, Partitioning, Consistency

Queste features permetteranno di confrontare i vari tipi di database.
Scalabilità : Vi sono due tipi di scalabilità, quella verticale, dove si aggiungono risorse ad un nodo, risorse come memoria e CPU, quindi è una scalabilità che ha dei limiti come ad esempio il numero di CPU che la macchina può supportare, questa è tipica nei database tradizionali RDBMS.
Scalabilità orizzontale, non porta all’acquisto di nuove risorse, ma di nuovi nodi, questa è tipica nei sistemi NoSQL e NewSQL.
Da notare che l’acquisto di un nodo di fascia media ha un costo minore confronto all’acquisto di risorse di ultima generazione.

Controllo della concorrenza : Vi sono due tipi di strategie, quella pessimistica, usata nei database tradizionali, che permette un accesso esclusivo al database. Lo scenario è che due o più utenti concorrenti provano ad aggiornare lo stesso record nello stesso istante. Quindi si fa un lock dell’entità che si vuole accedere così gli altri utenti che vogliono accedere alla stessa entità dovranno aspettare che chi li precede finisca il lavoro. Strategia ottimistica, assume che i conflitti sono possibili ma non sono frequenti. Pertanto, prima del commit, ogni transazione verifica se c’è un'altra transazione che ha apportato modifiche alle stesse entità e quindi va in conflitto. Se vengono identificati conflitti, la transazione verrà annullata. Questa strategia può funzionare bene se gli aggiornamenti non sono frequenti e quindi le possibilità di conflitto sono relativamente basse. Questa strategia è tipica nei sistemi NoSQL.

Replicazione : Le repliche offrono una migliore affidabilità, tolleranza ai guasti e durata. Vi sono due tipi di replicazione, Master-Slave e Master-Master mostrate in Figura 2.

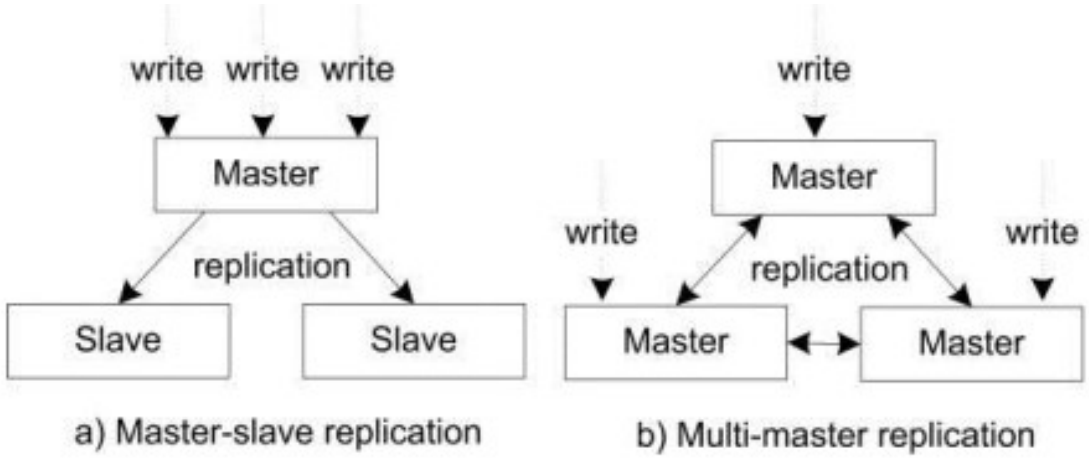

Figura 2

Lo schema Master-Slave (Figura 2.a) è uno schema dove un nodo è definito come Master ed è lui che accetterà e processerà le richieste, quindi i cambiamenti saranno propagati dal Master verso gli Slave. Il Master invia periodicamente degli heartbeat agli Slave, quest'ultimi se non ricevono nessun heartbeat per un determinato periodo di tempo eleggeranno un nuovo Master. In Master-Master (Figura 2.b), più nodi possono processare le richieste di scrittura e l'aggiornamento sarà poi propagato ai nodi rimanenti. Quindi la propagazione può avvenire in varie direzione e non in una sola come in Master-Slave cioè dal Master verso gli Slave.
Inoltre la replicazione può avvenire in modalità sincrona o asincrona. Sincrona, tutte le repliche sono aggiornate in modo sincrono e quindi il sistema non sarà disponibile finché tutti i nodi non hanno fatto il commit dell'operazione. Asincrona, le repliche sono aggiornate in modo asincrono, quindi le letture nelle repliche possono essere inconsistenti per un breve periodo di tempo.

Partizionamento : Vi sono diverse implementazioni di partizionamento, quello orizzontale detto anche sharding e quello verticale. Il primo, memorizza un insieme di record in diversi segmenti (shards) che possono trovarsi su diversi nodi. Le due strategie di sharding più comuni sono il Range Partitioning e Consistent Hashing. Il Range Partitioning assegna i dati alle partizioni che risiedono in server diversi in base agli intervalli di una chiave di partizione. Un server è responsabile dell'archiviazione e della gestione di lettura/scrittura di un intervallo specifico di chiavi. Il vantaggio di questo approccio è l'elaborazione

efficace delle query, poiché le chiavi adiacenti spesso risiedono nello stesso nodo. Tuttavia, questo approccio può causare punti critici e problemi di bilanciamento del carico.

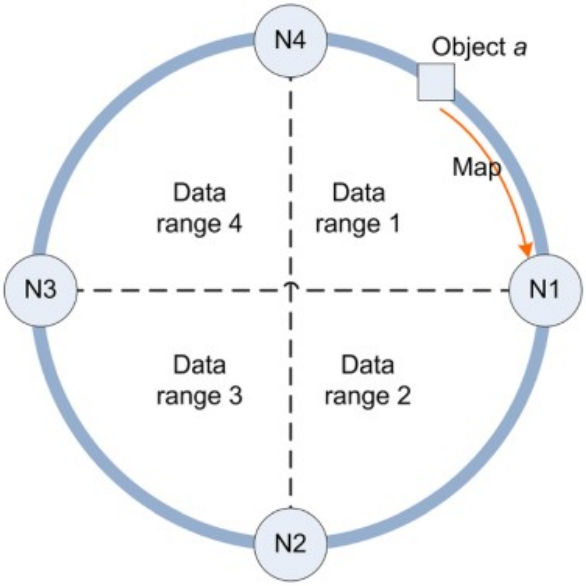

Figura 3: Consistent Hashing

Nel Consistent Hashing, invece, l'insieme di dati è rappresentato come un anello. L'anello è diviso in un numero di intervalli pari al numero di nodi disponibili e ogni nodo è mappato su un punto dell'anello. Per determinare il nodo in cui posizionare un oggetto, il sistema esegue l'hashing della chiave dell'oggetto e trova la sua posizione sull'anello. Nell'esempio in Figura 3, ci sono quattro nodi (N1,...,N4) ed un oggetto "a" che si trova tra N4 ed N1. L'anello verrà percorso in senso orario fino a quando non viene incontrato il primo nodo e l'oggetto viene assegnato a quel nodo. Quindi l'oggetto "a" verrà assegnato ad N1. Ogni nodo è responsabile della regione ad anello tra se stesso e il suo predecessore, quindi N1 è responsabile di "Data range 1". Con questa tecnica la posizione di un oggetto può essere calcolata molto velocemente e non è necessario un servizio di mappatura come nel Range Partitioning. Questo approccio è efficiente anche nel ridimensionamento dinamico: se i nodi vengono aggiunti o rimossi dall'anello, solo le regioni vicine vengono riassegnate a nodi diversi e la maggior parte dei record rimane inalterata ma questo approccio ha un impatto negativo sulle query perché le chiavi adiacenti sono distribuite su nodi diversi. La partizione verticale, invece memorizza un insieme di colonne in diversi segmenti. La strategia di partizionamento dipende fortemente dal modello di dati. Queste strategie sono comuni nei sistemi NoSQL e NewSQL.

Consistenza : La consistenza è fortemente correlata al partizionamento. In generale ci sono due tipi di consistenza, Strong Consistency ed Eventual Consistency. La prima assicura che quando tutte le richieste di scrittura sono confermate gli stessi dati sono visibili a tutte le successive operazioni di lettura. Strong può anche essere letto come immediato e può essere ottenuto con una replicazione sincrona o una completa mancanza di replica, questo comunque porta un aumento di latenza e impatta sulla disponibilità.
In un modello Eventual Consistency, invece le modifiche si propagano alle repliche attraverso il sistema dato un tempo sufficiente. Significa che alcuni nodi

potrebbero essere incoerenti per un certo periodo di tempo, questo è raggiungibile attraverso replicazione asincrona.

OLTP, OLAP e TPC-C

OLTP, OnLine Transaction Processing, è un tipo di elaborazione dei dati che consiste nell'eseguire un numero di transazioni che si verificano contemporaneamente, simulando ad esempio servizi bancari online, acquisti, immissione di ordini o invio di messaggi. OLTP consente l'esecuzione in tempo reale di un numero elevato di transazioni da parte di un numero elevato di persone, mentre OLAP, Online Analytical Processing, di solito comporta l'interrogazione di un database per scopi analitici. Le maggiori differenze tra questi due sono che OLTP richiede query semplici che coinvolgono pochi record e quindi tempi di risposta brevi, mentre in OLAP le query sono complesse ed coinvolgono un numero elevato di record e non è necessario che la risposta sia immediata.
TPC-C è un benchmark per OLTP. È una combinazione di transazioni che simulano le attività trovate in ambienti applicativi OLTP complessi.
Questi ambienti sono caratterizzati da l'esecuzione simultanea di più tipi di transazione, integrità della transazione (proprietà ACID), database costituiti da molte tabelle con un'ampia varietà di dimensioni, attributi e relazioni. La metrica usata misura il numero di ordini elaborati al minuto. Vengono utilizzate più transazioni per simulare l'attività commerciale di elaborazione di un ordine e ogni transazione è soggetta a un vincolo di tempo di risposta. La metrica delle prestazioni per questo benchmark è espressa in transazioni-per-minute-C (tpmC).
In Figura 4 viene mostrato la ricostruzione dello schema del database del benchmark TPC-C utilizzato.
Dove PK sta per PrimaryKey e FK per ForeignKey. La tabella customer contiene 120K record, district 40, history ~120K, item 100K, new_order ~36K, order ~120K, order_line ~1.2M, stock 400K e warehouse 100 record.
La freccia all’interno della tabella indica il campo a cui fa riferimento.

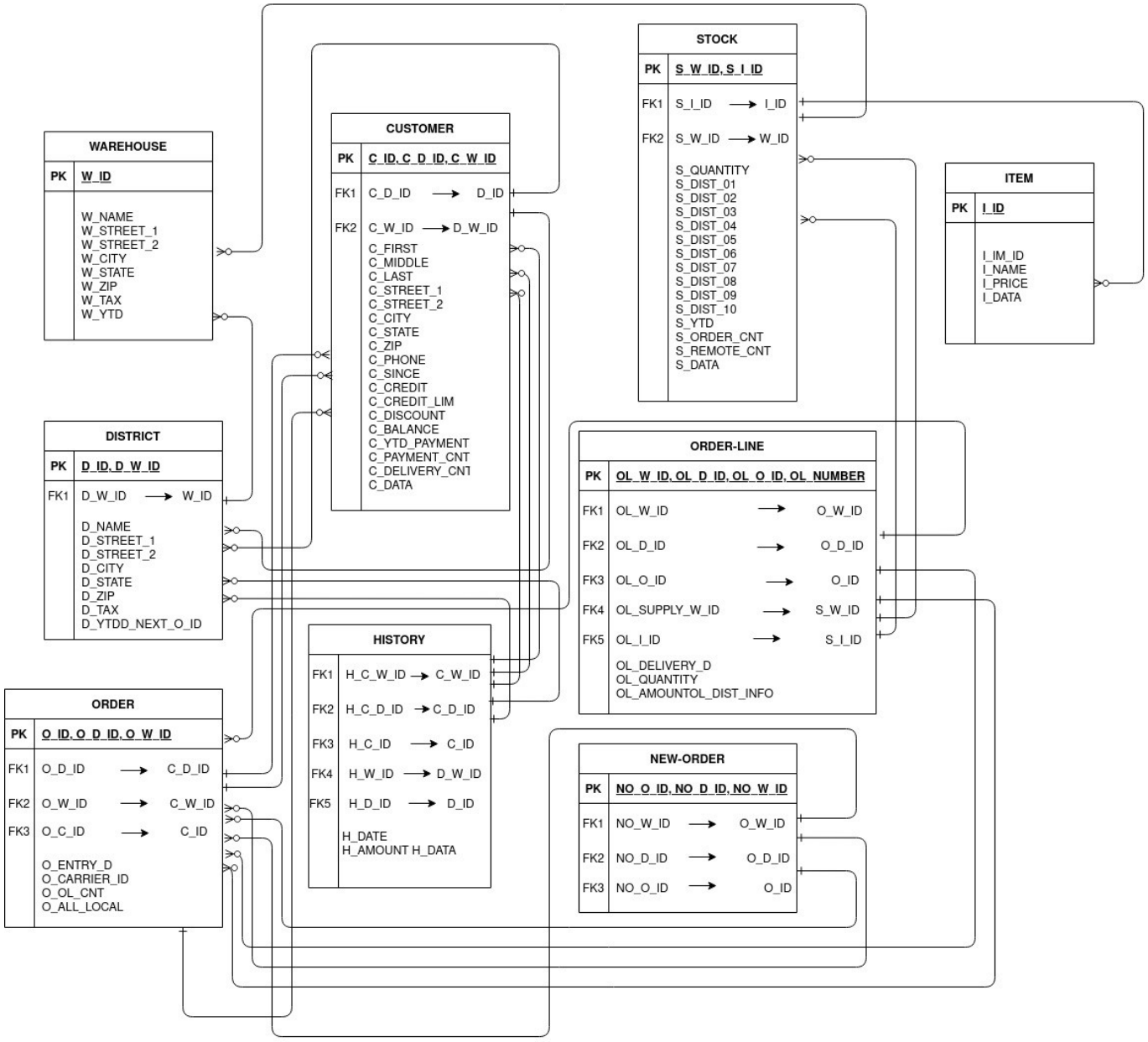
WAREHOUSE
PK W_ID
W_NAME
W_STREET_1
W_STREET_2
W_CITY
W_STATE
W_ZIP
W_TAX
W_YTD
CUSTOMER
PK C_ID, C_D_ID, C_W_ID
FK1 C_D_ID → D_ID
FK2 C_W_ID → D_W_ID
C_FIRST
C_MIDDLE
C_LAST
C_STREET_1
C_STREET_2
C_CITY
C_STATE
C_ZIP
C_PHONE
C_SINCE
C_CREDIT
C_CREDIT_LIM
C_DISCOUNT
C_BALANCE
C_YTD_PAYMENT
C_PAYMENT_CNT
C_DELIVERY_CNT
C_DATA
STOCK
PK S_W_ID, S_I_ID
FK1 S_I_ID → I_ID
FK2 S_W_ID → W_ID
S_QUANTITY
S_DIST_01
S_DIST_02
S_DIST_03
S_DIST_04
S_DIST_05
S_DIST_06
S_DIST_07
S_DIST_08
S_DIST_09
S_DIST_10
S_YTD
S_ORDER_CNT
S_REMOTE_CNT
S_DATA
ITEM
PK I_ID
I_IM_ID
I_NAME
I_PRICE
I_DATA
DISTRICT
PK D_ID, D_W_ID
FK1 D_W_ID → W_ID
D_NAME
D_STREET_1
D_STREET_2
D_CITY
D_STATE
D_ZIP
D_TAX
D_YTDD_NEXT_O_ID
ORDER-LINE
PK OL_W_ID, OL_D_ID, OL_O_ID, OL_NUMBER
FK1 OL_W_ID → O_W_ID
FK2 OL_D_ID → O_D_ID
FK3 OL_O_ID → O_ID
FK4 OL_SUPPLY_W_ID → S_W_ID
FK5 OL_I_ID → S_I_ID
OL_DELIVERY_D
OL_QUANTITY
OL_AMOUNTOL_DIST_INFO
HISTORY
FK1 H_C_W_ID → C_W_ID
FK2 H_C_D_ID → C_D_ID
FK3 H_C_ID → C_ID
FK4 H_W_ID → D_W_ID
FK5 H_D_ID → D_ID
H_DATE
H_AMOUNT H_DATA
ORDER
PK O_ID, O_D_ID, O_W_ID
FK1 O_D_ID → C_D_ID
FK2 O_W_ID → C_W_ID
FK3 O_C_ID → C_ID
O_ENTRY_D
O_CARRIER_ID
O_OL_CNT
O_ALL_LOCAL
NEW-ORDER
PK NO_O_ID, NO_D_ID, NO_W_ID
FK1 NO_W_ID → O_W_ID
FK2 NO_D_ID → O_D_ID
FK3 NO_O_ID → O_ID

Figura 4

SQL

Database SQL detti anche RDBMS sono sistemi di database tradizionali e vengono usati quando si utilizzano dati relazionali, l'integrità dei dati è importante e non si necessita di scalabilità orizzontale, supportano le proprietà ACID. In questi sistemi i dati seguono uno schema predefinito, formato da tabelle e relazioni e permettono solo una scalabilità verticale. Supportano il linguaggio di interrogazione SQL.

NoSQL

Questo tipo di sistemi nascono dalla necessità di gestire una quantità di dati sempre più in crescita, infatti, permettono una scalabilità orizzontale. Sono database non relazionali e supportano le proprietà BASE. Questo tipo di sistemi forniscono database distribuiti ad alte prestazioni, altamente disponibili, progettati per soddisfare le esigenze delle moderne strutture di big data e del cloud per scalabilità e prestazioni. I database NoSQL sono caratterizzati da bassa latenza, prestazioni lineari, replica e condivisione automatica dei dati, gestione semplice, disponibilità di scrittura continua, capacità di gestire dati non strutturati e utilizzo di nodi a basso costo. Presenta alcune limitazioni, ad esempio la mancanza di standard, non supporto del linguaggio SQL, cattive performance di analisi, nessun supporto per gli indici, nessun supporto per le transazioni e sono Eventual Consistency.
Vi sono vari tipi di sistemi NoSQL:

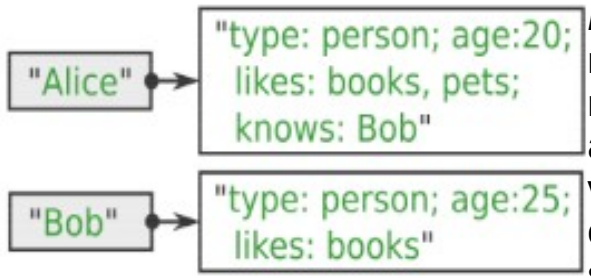

Figura 5: Struttura di un database Key-Value

Key-Value, ogni chiave è unica e identifica in modo non ambiguo dei valori. Quindi sono molto usati per scenari in cui le applicazioni accedono a un insieme di dati utilizzando un valore univoco come chiave, come ad esempio, la memorizzazione di informazioni sulla sessione Web, profili e configurazioni utente. Un esempio di struttura Key-Value è mostrato in Figura 5.

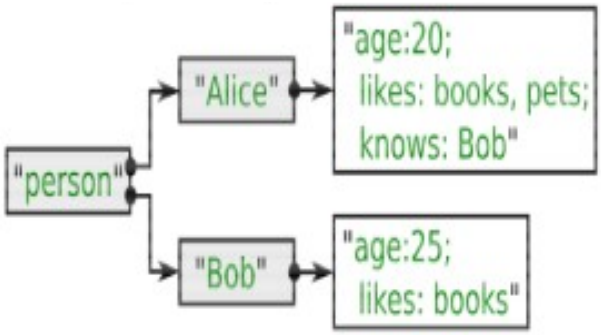

Figura 6: Struttura di un database Wide Column

Wide-Column (esempio mostrato in Figura 6), è una tipologia che spesso viene usata per il processamento del web, dati in streaming e documenti. Possono essere visti come Key-Value, ma la chiave è bidimensionale. Quindi il valore è referenziato da una column key e da una row key. In questi sistemi viene adottato un layout di dati a colonne in modo che ogni colonna venga archiviata separatamente su disco. Spesso supportano il concetto di famiglie di colonne (Column-Family, mostrato in Figura 7) archiviate separatamente. Tuttavia, ciascuna di queste famiglie di colonne contiene in genere più colonne che vengono utilizzate insieme, in modo simile alle tradizionali tabelle di database relazionali. Da notare come risulta più naturale un partizionamento verticale.

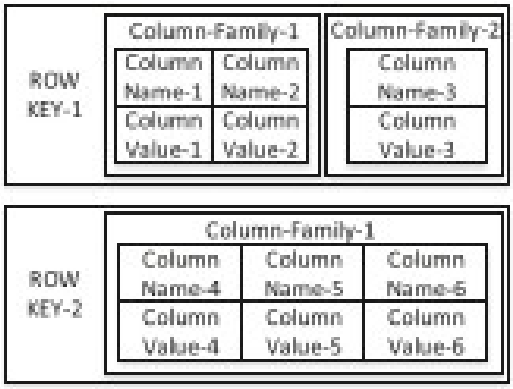

Figura 7: Column-Family

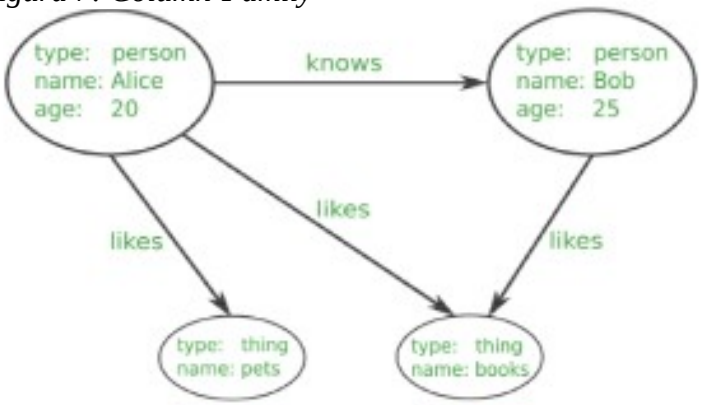

Figura 8: Struttura di un Graph database

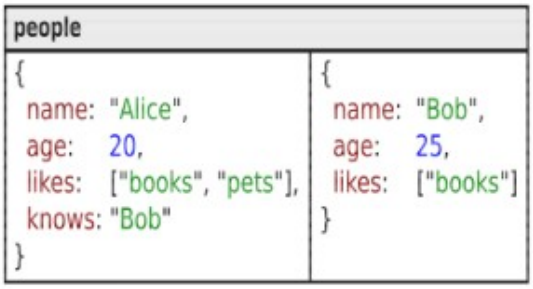

Figura 9: Struttura di un database Document

Graph, utilizza nodi e archi per rappresentare e archiviare i dati. Non seguono uno schema rigido e scalano molto facilmente. Sono utilizzati ad esempio nei Social Network. Un esempio è mostrato in Figura 8.

Document, esempio mostrato in Figura 9, possono essere visti come i Key-Value ma il valore non è completamente opaco e quindi può essere esaminato, questi datastore gestiscono dati che possono essere rappresentati come documenti, che sono strutture dati gerarchiche auto-descrittive che possono

contenere oggetti annidati e attributi ma non richiedono l'adesione a uno schema fisso.

NewSQL

NewSQL è un diverso tipo di sistema di gestione di database relazionali che fornisce le stesse prestazioni scalabili dei sistemi NoSQL per i carichi di lavoro OLTP e mantenendo le garanzie ACID di un tradizionale sistema di database a nodo singolo. NewSQL supporta le proprietà ACID per le transazioni, utilizza SQL per interrogare il database, utilizza un meccanismo di controllo della concorrenza non bloccante che è utile per le letture in tempo reale che non entrano in conflitto con le scritture, ha una architettura shared-nothing, permette una scalabilità orizzontale.
Questi tipi di sistemi sono appropriati per applicazioni che richiedono l'uso di transazioni che manipolano più di un oggetto, o necessitano della Strong Consistency, o anche entrambi. Gli esempi classici sono le applicazioni nel mercato finanziario, dove operazioni come i trasferimenti di denaro devono aggiornare automaticamente due conti e tutte le applicazioni devono avere la stessa visualizzazione del database.
New SQL e Distributed SQL, sono descritti entrambi come soluzioni per combinare i vantaggi di SQL e NoSQL, ma con la differenza che NewSQL è costruito aggiungendo strati di logica distribuita su un database relazionale tradizionale, mentre i Distributed SQL sono costruiti da zero.
In Figura 10 viene mostrato un confronto riassuntivo dei sistemi SQL, NoSQL e NewSQL.

	Old SQL	NoSQL	NewSQL
Relational	Yes	No	Yes
SQL	Yes	No	Yes
ACID transactions	Yes	No	Yes
Horizontal scalability	No	Yes	Yes
Performance/ big volume	No	Yes	Yes
Schema-less	No	Yes	No

Figura 10

Obbiettivi

L’obiettivo principale è quello di fare un confronto sperimentale fra sistemi SQL, NoSQL e NewSQL.
Per fare ciò si sono trovati tre casi d’uso.
Primo caso d’uso :
Per ogni sistema si effettuano analisi direttamente sul database, si è utilizzato il dataset fornito per il primo progetto. Si analizza quindi la velocità di esecuzione delle queries al variare della grandezza del dataset.
Questo verrà fatto sia in modalità cluster che standalone.
Secondo caso d’uso:
Per ogni sistema, si vuole simulare in modo molto semplificato, un sistema come Airbnb quindi si farà un confronto sui tempi di risposta, disponibilità, consistenza e tolleranza alle partizioni. Questo verrà fatto in modalità cluster solo per sistemi NoSQL e NewSQL, per completezza è stato fatto anche in modalità standalone per MySQL, ma l’obbiettivo era vedere come si comportasse il sistema simulando un nodo che falliva, quindi questo è stato possibile farlo solo su cluster.
Terzo caso d’uso :
Implementare il benchmark TPC-C per tutte e tre i sistemi.

Architettura e tecnologie

Tutti gli esperimenti sono stati eseguiti in locale, in particolare su una macchina Lenovo B590 627435G, CPU : i5-3230M 2.60GHz , RAM : 8GB DDR3, SO : Ubuntu 20.04.
Per tutti i casi d’uso sono state scelte le tecnologie MySQL per i sistemi SQL, MongoDB per i sistemi NoSQL e CockroachDB per i sistemi NewSQL, queste sono state poi containerizzate con Docker.
Per tutti casi d’uso la sicurezza del database è stata ignorata per semplificare la riuscita dell’esperimento.
MySQL : È un sistema di database tradizionali, si basa esclusivamente su tabelle relazionali per l'archiviazione dei dati, uso di schemi non flessibili, riduce al minimo le ridondanze tramite la normalizzazione, supporta transazioni ACID e permette una scalabilità verticale. Supporta query in SQL.

MongoDB : È un sistema NoSQL, in particolare di tipo Document. Può essere eseguito su un singolo nodo oppure, se implementato su cluster, quindi distribuito, viene implementata una architettura Master-Slave e la replicazione di default è di tipo asincrono. Offre una scalabilità orizzontale. Non supporta query in SQL. Supporta le proprietà BASE.

In Figura 11 viene mostrata un esempio di architettura di MongoDB implementata su cluster. La stessa architettura verrà poi utilizzata negli esperimenti.

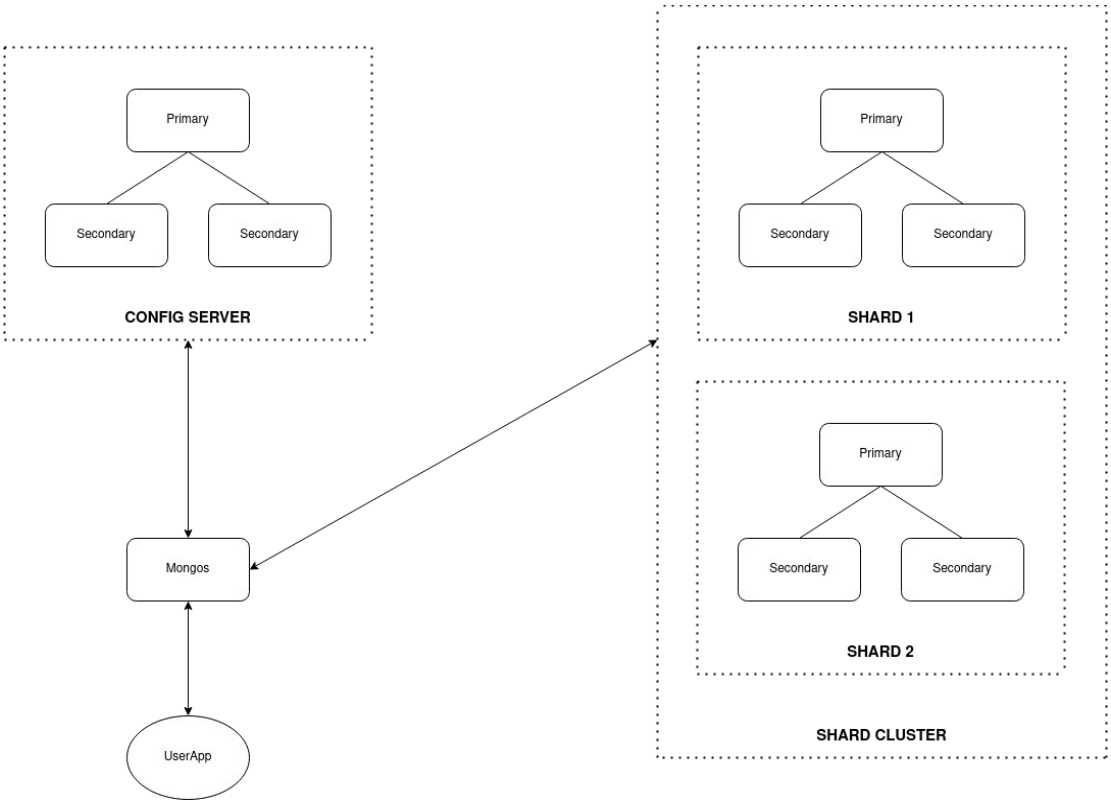

Figura 11: Esempio architettura MongoDB su cluster

Lo sharding è un metodo per distribuire dati su più macchine permettendo cosi una scalabilità orizzontale. Ogni “shard” contiene un sottoinsieme dei dati condivisi. Ogni shard può avere delle “replica set”.
La replication in MonogDB è un insieme di processi mongo che mantengono i stessi dati. Replica sets fornisce quindi ridondandza e alta disponibilità.
Mongos è un query router, fornisce un’interfaccia tra la applicazione client e lo sharded cluster. Quindi non è possibile implementare una architettura master-slave come fatto con gli shard ma si necessiterebbe di un load-balancer e vari mongos.
I server di configurazione salvano i metadati e le impostazioni di configurazione per il cluster.
Quindi il client si interfaccia con “mongos”, che a sua volta chiederà al server di configurazione dove si trovano gli indici ai quale deve accedere, dopo di che farà le ricerche negli shard giusti. Da notare che tutti i nodi primari, secondari e dove risiede mongos si trovano su macchine diverse, in particolare durante l’esperimento, avendolo eseguito su una singola macchina ed utilizzando Docker si trovano su container diversi.

CockroachDB : È un sistema NewSQL, quindi è un RDBMS distribuito, supporta replicazione, quindi se un nodo fallisce i dati continuano a persistere, cioè offre la sopravvivenza dei dati. Può anche essere implementato su un singolo nodo. È conforme alle proprietà ACID. CockroachDB è fortemente consistente. Supporta query in SQL, quindi, gli sviluppatori possono usare schemi, tabelle, righe, colonne e indici. Permette una scalabilità orizzontale. CockroachDB non è l'ideale per transazioni complesse, infatti non viene utilizzato in sistemi OLAP. Implementa un'architettura Master-Master, dove ogni master ha dei nodi replica. Se un master fallisce, i dati persistono nelle repliche.

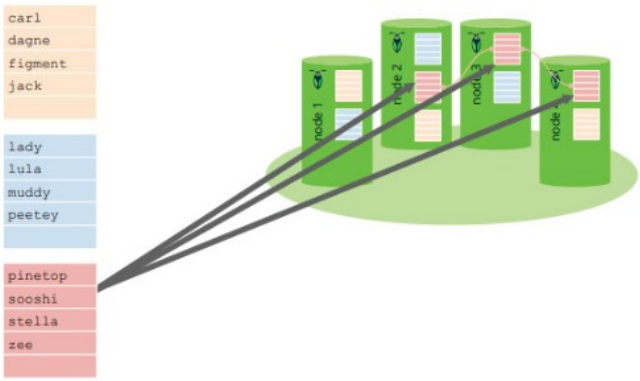

Figura 12: CockroachDB Cluster

In Cockroach un Cluster, è una distribuzione di CockroachDB, che agisce come una singola applicazione logica. Un Nodo, è una macchina individuale dove risiede CockroachDB, se più nodi vengono uniti formano un Cluster. CockroachDB memorizza tutti i dati utente (tabelle, indici, ecc.) e quasi tutti i dati di sistema in una grande mappa ordinata di coppie chiave-valore. Questo spazio delle chiavi è diviso in Ranges, che sono blocchi contigui dello spazio delle chiavi in modo che ogni chiave possa essere sempre trovata in un singolo Range. Inoltre ogni Range viene replicato ed ogni replica risiede su un nodo differente.
Per ogni Range, una delle repliche viene eletta come Leaseholder, ed è quella che riceve e coordina tutte le richieste di lettura e scrittura per quel Range. Viene implementato con un architettura Master-Master quindi qualunque nodo riceva la richiesta funge da "nodo gateway", che elabora la richiesta e risponde al client.
CockroachDB ha una architettura che si può suddividere in vari livelli, come mostrato in Figura 13.

Layer	Order	Purpose
SQL	1	Traduce le query SQL del client in operazioni KV.
Transactional	2	Consente cambiamenti atomici a più entries KV.
Distribution	3	Rappresenta i range KV replicati come una singola entità.
Replication	4	Replica coerente e sincrona dei ranges KV su vari nodi. Questo livello consente anche letture coerenti utilizzando un algoritmo di consenso.
Storage	5	Legge e scrive dati KV su disco

Figura 13: Livelli architettura CockroachDB

Layer SQL : Espone un API SQL quindi le richieste che riceve sono statements SQL. I dati sono letti e scritti dal livello di Storage come coppie key-value (KV). Quindi il layer SQL converte gli statements SQL in operazioni di basso livello KV che saranno inviate al livello Transactional.

Layer Transactional : Implementa il supporto per le transazioni ACID coordinando le operazioni concorrenti. Raggiunge la correttezza utilizzando un protocollo di commit atomico distribuito chiamato Parallel Commits. Quindi questo livello riceve operazioni KV dal livello SQL e controlla il flusso delle operazioni KV inviate al livello Distribution.

Layer Distribution : Fornisce una visione unificata dei dati nel cluster. Per rendere accessibili tutti i dati nel cluster da qualsiasi nodo, CockroachDB archivia i dati in una mappa ordinata monolitica di coppie key-value. Questo key-space descrive tutti i dati nel cluster ed è diviso in Range. La mappa ordinata permette ricerche semplici (è possibile identificare quale nodo è responsabile di una certa porzione dei dati, le query sono in grado di individuare rapidamente dove trovare i dati desiderati.) e scan efficienti (è facile cercare dati all'interno di un range particolare durante lo scan). La mappa ordinata monolitica è composta da due elementi fondamentali, dati di sistema, che includono meta-ranges che descrivono le posizioni dei dati nel cluster; dati utente, che memorizzano i dati della tabella nel cluster.

Layer Replication : Replica i dati tra i nodi e garantisce la coerenza tra queste copie implementando un algoritmo di consenso. Questo algoritmo richiede un quorum di repliche concordi su eventuali modifiche a un range prima che tali modifiche vengano confermate. Il numero di guasti tollerabili è pari a (Fattore di replica - 1)/2. Ad esempio, con la replica 3x, è possibile tollerare un errore; con replica 5x, due errori e così via. Quando un nodo fallisce dopo un periodo di tempo CockroachDB si accorge automaticamente che il nodo ha smesso di funzionare e lavora per ridistribuire i dati per continuare a massimizzare la sopravvivenza. Questo processo funziona anche al contrario: quando nuovi nodi si uniscono al cluster, i dati si ribilanciano automaticamente, assicurando che il carico sia distribuito uniformemente. Quindi questo livello riceve richieste e invia risposte al livello Distribution ed scrive le richieste accettate nel livello Storage.

Layer Storage : Questo livello legge e scrive dati sul disco. Questi dati vengono archiviati come coppie KV su disco utilizzando il motore di archiviazione, che viene trattato principalmente come un'API black-box.

Implementazione

Nel primo caso d’uso, per tutte e tre le tecnologie, si è usato il linguaggio di programmazione Python per interrogare il database. Quindi per ogni tecnologia si sono creati dei container Docker che implementano la tecnologia scelta. Le queries per MongoDB e CockroachDB sono state eseguite sia quando le tecnologie erano implementate su un singolo nodo che in cluster. In particolare il dataset e quindi le queries eseguite sono gli stessi del primo progetto. Il dataset fornito contiene informazioni sull’andamento giornaliero delle azioni sulla borsa del NYSE e NASDAQ. Tale dataset è composto da due file csv. historical_stock_prices.csv contenente circa ventuno milioni di righe, dove ogni riga riporta informazioni sul prezzo di chiusura ed apertura, prezzo minimo, massimo, volume e data di un’azione. Il secondo file, historical_stocks.csv, contenente poco più di seimila righe, dove ogni riga riporta informazioni sul nome dell’azienda, il settore, l’industria, etc...
Le queries dovevano essere tre, ne è stata implementata solo una, la prima, che doveva essere in grado di generare, per ciascuna azione: la data della prima quotazione, la data dell’ultima quotazione, la variazione percentuale della quotazione, il prezzo massimo e quello minimo, tutto ordinato per valori decrescenti della data dell’ultima quotazione.
Si è deciso di implementare solo la prima query che anche essendo la più semplice non ha prodotto risultati soddisfacenti.
Nel secondo caso d’uso si è utilizzato JMeter per effettuare i test di carico. JMeter è un progetto Apache che può essere usato per verificare, analizzare e misurare le prestazioni di vari servizi, quindi può essere utilizzato per simulare un carico pesante su un server o un gruppo di server. Nel progetto vengono riportati i file di configurazioni per ogni tecnologia presa in considerazione.
L’idea era quella di simulare un semplice sistema di prenotazioni, dunque un utente può prenotare più stanze. In Figura 14 viene mostrato lo schema usato per CockroachDB, mentre in Figura 15 un esempio dello schema usato per MongoDB, dove “utenti”, “stanze” e “prenotazioni” sono collections.
L’architettura di CockroachDB in questo caso prevede tre nodi e un server proxy che fa da load balancer. Per MongoDB invece è la stessa mostrata in Figura 11.

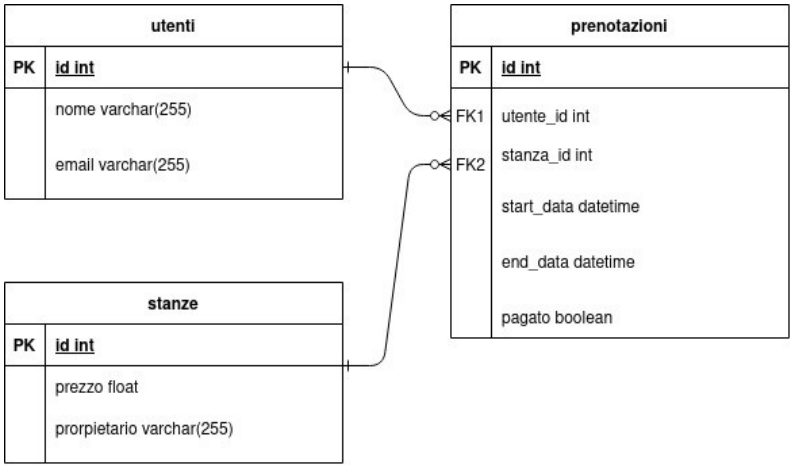

Figura 15: Schema CockroachDB

```
#utenti
{
  id: 1,
  nome: "alessandro",
  email: "test@test.com"
}

#stanze
{
  id: 1,
  prezzo: 100,
  proprietario: "Luigi"
}

#prenotazioni
{
  id: 1,
  id_utente: 1,
  id_stanza: 1,
  start_date: "02/02/2020",
  end_date: "05/02/2020",
  pagato: true
}
```

Figura 14: Schema MongoDB

Nel terzo caso d'uso, non si è implementato da zero il benchmark TPC-C ma si sono utilizzati progetti già creati e testati, in particolare per CockroachDB si sono seguite le linee guida sulla documentazione ufficiale (link), essi offrono già un workload TPC-C, quindi sono stati creati tre container (nodi), si è importato il dataset TPC-C per poi eseguire il benchmark. Per MongoDB invece, si è usato PyTPCC, un framework basato su Python (scaricabile dal link py-tpcc), lo stesso usato nel paper ufficiale scaricabile dal sito di MongoDB. Da notare che però non si usa Python3 ma la versione precedente, in particolare si è usato Python2.7, anche in questo caso si è usato MongoDB in modalità cluster con l'architettura mostrata in Figura 11.
Per completezza si è voluto testare TPC-C anche per MySQL, ma implementato solo su un singolo nodo. Quindi è stato scaricato il progetto "tpcc-mysql" scaricabile dal link tpcc-runner, sono state quindi seguite le indicazioni riportate in questo repository, in particolare viene scaricata un'immagine docker che implementa il benchmark TPC-C come riportato nel repository seguente tpcc-mysql, tale immagine docker però implementa MySQL versione 5, quindi è stata creata ed usata una nuova immagine aggiornata.
Tutti i benchmark sono stati eseguiti con cento warehouse, un minuto di ramp up e per una durata di dieci minuti.
Ulteriori informazioni, come ad esempio, i comandi necessari per eseguire i test sono reperibili all'interno del progetto.

Risultati

Di seguito vengono riportati i risultati ottenuti in tutte e tre i casi d'uso.
Primo caso d'uso: Di seguito viene riportato il tempo di esecuzione della query al variare della grandezza del dataset e della tecnologia. Nelle tabelle seguenti, dove si legge "null" vuol dire che il processo è stato ucciso perché impiegava troppo tempo. Quindi è stato creato un container per ogni tecnologia da analizzare e attraverso un programma scritto in Python sono state eseguite le query. In Tabella 1 è possibile visualizzare i tempi di risposta delle query quando le tecnologie sono state implementate in modalità single-node. In Tabella 2 invece, quando implementate in modalità cluster. Nel progetto è possibile trovare anche una implementazione di CockroachDB multi-regione HA i cui risultati non sono stati riportati dato che erano nettamente peggiori ad un semplice Cluster. Tali risultati sono comunque visibili all'interno del progetto.

Tabella 1: Tempi di risposta modalità StandAlone

#Record\Tecnologia	MySQL (sec)	MongoDB (sec)	CockroachDB (sec)
1K	5.34	0.40	9.66
100K	2200.44	8.50	7.41
1M	null	480.62	11.97
10M	null	null	64.46
~21M	null	null	145.82

Tabella 2: Tempi di risposta modalità Cluster

#Record\Tecnologia	MongoDB (sec)	CockroachDB (sec)
1K	2.01	13.37
100K	18.19	13.69
1M	670.88	18.53
10M	null	61.57
~21M	null	152.84

Secondo caso d’uso
Si è utilizzato JMeter per effettuare test di carico, questo è stato fatto per le tecnologie MongoDB e CockroachDB implementate in modalità cluster.
CockroachDB : La configurazione JMeter prevede tre client che contemporaneamente eseguono queries che prevedono la creazione, aggiornamento, inserimento e selezione di record e tabelle. Ogni query verrà ripetuta per cento volte. Le queries sono undici, quindi, le queries totali eseguite saranno 3300. Il cluster è formato da tre nodi e un proxy-server con load balancer di tipo round robin e ogni nodo ha le sue repliche. Quando un nodo fallisce di default vengono attesi 5 minuti prima che viene considerato morto, questo tempo è stato impostato al minimo possibile, quindi a 1 minuto e 15 secondi.
MongoDB : La configurazione JMeter è la stessa, solo che due client eseguono sette queries differenti e un client sei. Ogni query viene ripetuta 165 volte e così facendo si eseguiranno 3300 queries.
Con tutte e due le tecnologie durante la fase di test è stato simulato un nodo che falliva. Di seguito vengono riportati i risultati ottenuti dei tempi di risposta delle richieste effettuate. Inoltre, con CockroachDB è stato registrato un Error Rate (percentuale dei test falliti) dello 0.15%, un Throughput di 25.1 richieste per secondo gestite. Invece, per MongoDB è stato registrato un Error Rate dello 0% e un Throughput di 19 richieste al secondo. Comunque, con nessuna tecnologia sono stati riscontrati errori nel database, cioè tutti gli inserimenti e modifiche sono state eseguite correttamente.

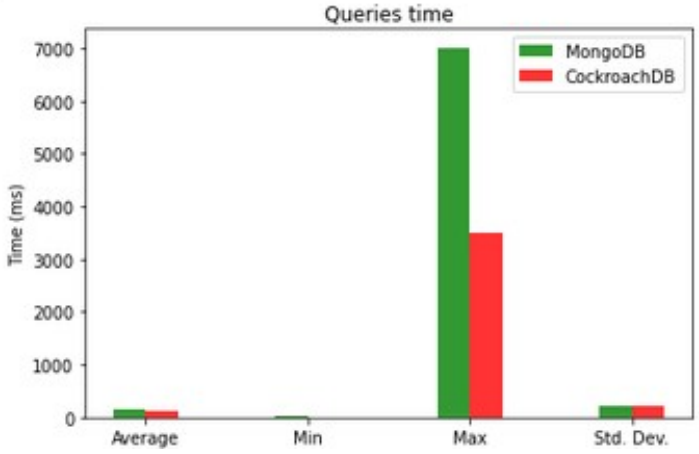

Figura 16: Tempi di risposta

Terzo caso d’uso quindi test con il benchmark TPC-C, sono state eseguite operazioni sulle tabelle : DELIVERY, NEW_ORDER, ORDER_STATUS, PAYMENT e STOCK_LEVEL. Il test è stato eseguito per una durata di dieci minuti. Di seguito i risultati ottenuti per le tecnologie scelte.
CockroachDB, l’output ottenuto riporta un tempo di risposta medio di 816.3 ms ed 1184.6 tpmC.
MongoDB, l’output ottenuto non è molto dettagliato, vengono riportate solo le transazioni al secondo che sono di 359.76 txn/sec.

Secondo gli standard TPC-C, il valore tpmC viene calcolato in base al numero di transazioni New Order processate nel modello standard.
Quindi tpmC e txn/sec non sono paragonabili. I risultati in formato grezzo, ma più dettagliati sono visibili all'interno nel progetto.
Invece, per MySQL l'unico ad essere implementato in modalità standalone, sono state registrate 54.2 tpmC.
Nel progetto è possibile trovare i file di configurazione JMeter per MySQL, ma i risultati non sono riportati perché per il secondo caso d'uso l'idea era simulare un nodo che falliva e poichè non si è implementato MySQL in modalità cluster non è stato possibile simularlo.

Osservazioni

Si può notare che riguardo al primo caso d'uso CockroachDB è stato l'unico ad riuscire a completare tutte le query, inoltre è possibile osservare che per le tecnologie implementate su cluster, siccome i nodi sono implementati come istanze Docker che risiedono su una singola macchina non si è ottenuto un guadagno computazionale ma in realtà si è aggiunta solo la latenza quindi si può notare un aumento dei tempi di risposta. Questo aumento risulta essere maggiore su MongoDB confronto a CockroachDB.
Invece, riguardo al secondo caso d'uso, in particolare nella Figura 16 si possono notare i tempi di risposta massimi abbastanza elevati, questi tempi probabilmente si sono registrati nel momento in cui i nodi hanno fallito.
Per confermare tale teoria sono stati quindi rieseguiti i test senza far fallire nessun nodo ed i tempi di risposta massimi si sono abbassati.
Il fatto che CockroachDB dia dei tempi di risposta inferiori alle altre tecnologie è normale, dato che è nato per i carichi di lavoro OLTP e quindi deve dare risposte immediate.
Per tutti i casi d'uso si è notato la semplicità di implementazione e di utilizzo di CockroachDB, infatti non necessita di nessuna configurazione al contrario di MongoDB che se implementato su un cluster, si dovranno configurare i server di configurazione, le repliche, gli shard e mongos, invece con CockroachDB non si necessita di tutto ciò dato che si configura automaticamente, infatti anche l'aggiunta di un nodo (dopo che il cluster è inizializzato) risulta essere molto semplice, inoltre nella fase di test CockroachDB e MongoDB non necessitano di configurare nessun utente e quindi lasciare da parte la sicurezza al contrario di MySQL.
Non si è riusciti ad eseguire un'analisi adeguata delle performance in termini di disponibilità, consistenza e tolleranza alle partizioni. Questo perché l'esperimento è stato eseguito su una singola macchina con l'ausilio dei

container. Spesso accadeva che la macchina non rispondesse per ore e quindi si è provveduti ad un riavvio manuale. CockroachDB è l'unica tecnologia che non ha bloccato la macchina in nessuno dei casi d'uso. Il benchmark TPC-C in realtà per avere un risultato realistico doveva essere eseguito per almeno tre ore, invece la sua durata è stata di dieci minuti per il motivo sopra descritto.