Distributed Database using YugabyteDB and CockroachDB

Advanced Databases (INFO-H-415)

Sony Shrestha (000583110)

Aayush Paudel (000583200)

Shofiyyah Nadhiroh (000583951)

MD Kamrul Islam (000583487)

UNIVERSITÉ LIBRE DE BRUXELLES

ÉCOLE POLYTECHNIQUE DE BRUXELLES

Fall 2023

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Distributed Database

A distributed database is a database system that spans across multiple computers or nodes, allowing them to work together to store, manage, and retrieve data. Unlike a centralized database, where all data is stored on a single machine, a distributed database distributes data across multiple nodes in a network. This distribution of data across multiple nodes helps in enhancement of performance, scalability, and fault tolerance.

Some of the popular databases for distributed data storage include:

1. YugabyteDB
2. CondensationDB
3. Citus
4. Apache Cassandra
5. Apache HBase
6. ETCD
7. CockroachDB

1.2. Centralized Database vs Distributed Database

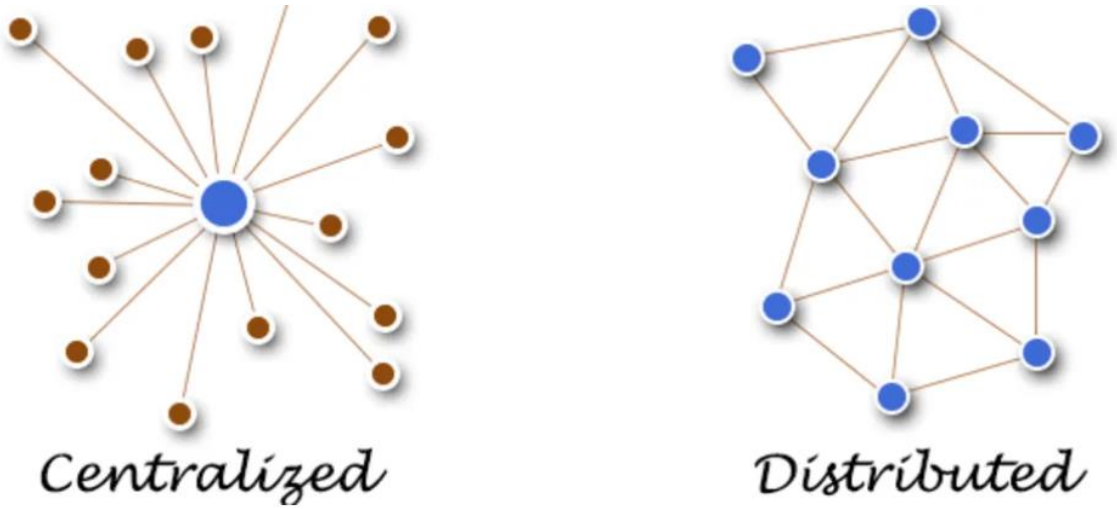


Fig 1.1: Centralized and Distributed Database

Centralized Database	Distributed Database
This is the type of database where a single database is located at one location in the network.	This is the type of database that contains two or more database files located at different locations in the network.
Managing, updating and taking in backups of data is easier because there is only one database file.	As there are multiple database files in a distributed database, it requires time to synchronize data.
In this type of database, it requires time to access data because multiple users might be accessing database files at the same time.	Time taken to retrieve data is higher because the data is fetched from the nearest database file.
If the database fails. Users do not have access to a database.	If one database fails, users can still access other database files.
It has more data consistency and it provides the complete view to the user.	It can have data replications and there can be some data inconsistency.

Table 1.1: Comparison between Centralized and Distributed Database

1.3. Features of Distributed Database

Distributed databases have several key features that distinguish them from traditional, centralized databases. Some of them are explained below.

1. **Distribution of Data:** The main characteristic of Distributed Database is the distribution of data across multiple nodes or servers. Each node typically stores a subset of the database, allowing for horizontal scaling and improved performance.
2. **Fault Tolerance:** Distributed databases are designed to provide fault tolerance by replicating data across multiple nodes. In the event of a node failure, the system can continue to operate, and data remains accessible from other nodes.

3. **High Availability:** Due to data replication and the ability to distribute workload, distributed databases can offer high availability. Users can access the database even if some nodes are temporarily unavailable.
4. **Parallel Processing:** Distributed databases enable parallel processing of queries and transactions by distributing data and computation across multiple nodes. This can lead to improved performance, especially for read-intensive workloads.
5. **Data Partitioning:** Data is often partitioned or fragmented across nodes based on specific criteria. Horizontal partitioning involves dividing rows of a table, while vertical partitioning involves dividing columns. This allows for more efficient storage and retrieval.
6. **Scalability:** Distributed databases support scalability by allowing the addition of new nodes to the system. This can be achieved through both horizontal scaling (adding more nodes) and vertical scaling (increasing resources on individual nodes).
7. **Consensus Protocols:** Distributed databases often employ consensus protocols, such as Paxos or Raft, to achieve agreement among nodes on the state of the system. Consensus is crucial for ensuring consistency and fault tolerance.
8. **Autonomy:** Nodes in a distributed database system may operate autonomously, having control over their subset of data and local operations. Autonomy allows for more flexible management and optimization at the local level.

1.4. Type of Distributed Database

1. Homogenous Distributed Database
2. Heterogeneous Distributed Database

1.4.1. Homogeneous Distributed Database

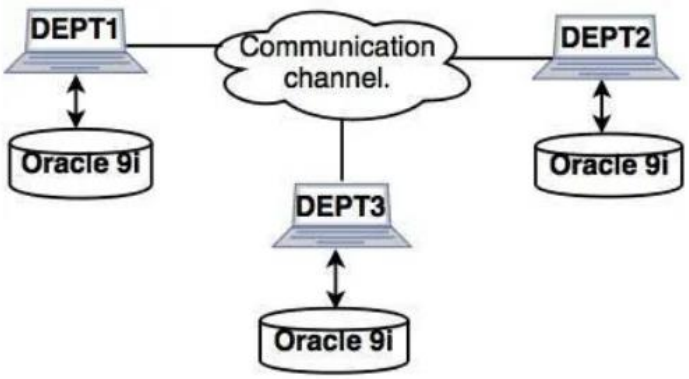


Fig 1.2: Homogenous Distributed Database

A homogenous distributed database is a network of identical databases stored on multiple sites. The sites have the same operating system, DDBMS, and data structure, making them easily manageable. Homogeneous databases allow users to access data from each of the databases seamlessly.

1.4.2. Heterogeneous Distributed Database

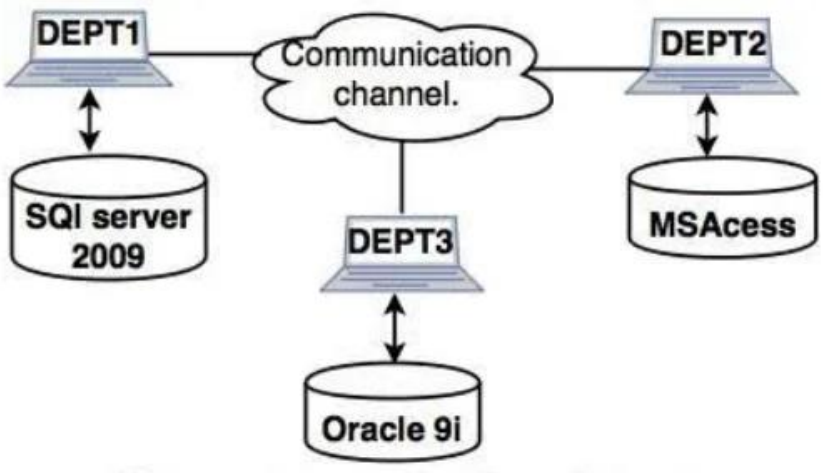


Fig 1.3: Heterogeneous Distributed Database

A heterogeneous distributed database uses different schemas, operating systems, DDBMS, and different data models.

In the case of this type of distributed database, a particular site can be completely unaware of other sites causing limited cooperation in processing user requests. Due to this limitation, translations are required to establish communication between sites.

1.5. Distributed Data Storage

There are 2 ways in which data can be stored on different sites. They are:

1. Replication
2. Fragmentation

1.5.1 Replication

In this approach of storing data, the entire data is stored redundantly across multiple sites. In other words, systems maintain multiple copies of data. Some key advantages of replication include:

1. **Parallel Processing:** With replicated data, multiple nodes can simultaneously handle read queries, improving response times for read-intensive workloads.
2. **Load Balancing:** Replication allows distributing read and write operations across multiple nodes, reducing the load on any single node. This enables better utilization of resources and improved overall system performance.
3. **High Availability:** Replication provides redundancy by creating copies of data on multiple nodes. If one node fails, others can continue serving requests, ensuring high availability and minimal disruption.
4. **Data Durability:** Copies of data on different nodes act as backups, helping to prevent data loss in case of hardware failures, disasters, or other unforeseen events.
5. **Read Scalability:** Replication supports the scaling of read operations by distributing them across multiple nodes. This allows the system to handle increased read loads without a proportional increase in resource requirements.
6. **Reduced Latency:** Replicating data across geographically distributed nodes allows users to access data from a location that is physically closer to them, reducing network latency and improving response times.

However, it has certain disadvantages as well which includes:

1. **Storage Costs:** Replicating data across multiple nodes increases storage requirements. While this may be necessary for fault tolerance and performance, it also incurs additional storage costs.
2. **Write Latency:** While replication can improve read performance, it may introduce latency in write operations. Ensuring that updates are propagated to all replicas can result in delays for write-intensive workloads.
3. **Eventual Consistency:** Maintaining consistency across replicated copies can be challenging, especially in scenarios where updates are made to different copies simultaneously.
4. **Management Overhead:** Managing a system with replicated data adds complexity to configuration, maintenance, and troubleshooting. Ensuring consistency, dealing with conflicts, and monitoring the health of multiple replicas require additional administrative effort.

1.5.2. Fragmentation

In this approach, the relations are fragmented (i.e., they're divided into smaller parts) and each of the fragments is stored in different sites. The prerequisite for fragmentation is to make sure that the fragments can later be reconstructed into the original relation without losing data.
There are two main types of fragmentation.

1. Horizontal Fragmentation
2. Vertical Fragmentation

1.5.2.1. Horizontal Fragmentation

This type of fragmentation, commonly referred to as splitting by rows, involves dividing a table's rows into subsets, where each subset is stored on a different database server or node. This is often done for load balancing or to improve query performance.

1.5.2.2. Vertical Fragmentation

This type of fragmentation, commonly referred to as splitting by columns, involves dividing a table's columns into subsets, where each subset is stored on a different database server or node. This is done to optimize storage or improve query performance.

Fragmentation is useful since it avoids the creation of duplicate data, which prevents data inconsistency.

1.6. Advantages of Distributed Database

Now-a-days, as data has become an indispensable part of our lives, Distributed databases are at the center of any organization’s data architecture. End-users interacting with a web service or mobile application may not notice a distributed database in operation, but it is the distributed database working hard behind the scenes that power many of these use-cases.

Some key advantages of distributed database include:

1. **Parallel Processing:** Distributed databases can distribute data and processing across multiple nodes, allowing for parallel execution of queries. This can lead to improved performance, especially for read-intensive operations.
2. **Horizontal Scaling:** Distributed databases can be easily scaled horizontally by adding more nodes to the system. This enables the database to handle increased workloads and growing amounts of data.
3. **Redundancy and Fault Tolerance:** With data distributed across multiple nodes, distributed databases can provide high availability. If one node fails, others can continue to serve requests, ensuring continuous access to data.
4. **Global Access:** Distributed databases can be designed to store data across geographically dispersed locations. This enables users to access data from locations that are physically closer to them, reducing latency and improving performance.
5. **Isolation of Failures:** Failures on one node typically do not affect the entire system. Isolating failures to specific nodes contribute to fault tolerance and system reliability.

6. **Efficient Resource Utilization:** Distributed databases allow for the efficient use of resources across multiple nodes. This can lead to better utilization of computing power, storage, and network bandwidth.
7. **Data Partitioning:** Distributing data across nodes simplifies data management tasks. Each node can be responsible for a specific subset of data, making it easier to organize, update, and maintain.
8. **Isolation of Sensitive Data:** Distributed databases allow for the isolation of sensitive or confidential data to specific nodes, contributing to enhanced security. Access controls can be applied at the node level.

1.7. Disadvantages of Distributed Database

While distributed databases offer various advantages, it also comes with certain disadvantages and challenges that need careful consideration. Some of the disadvantages of distributed databases include:

1. **Complexity of Design:** Designing a distributed database system can be complex, requiring careful consideration of factors such as data distribution, replication, and consistency. This complexity increases with the scale and heterogeneity of the distributed environment.
2. **Data Consistency:** Ensuring data consistency across distributed nodes is a significant challenge. Coordinating updates and transactions to maintain consistency can introduce overhead and complexities, impacting system performance.
3. **Coordination Overhead:** The need for coordination between distributed nodes can lead to increased overhead. Implementing consensus protocols and coordination mechanisms may introduce latency and affect the overall system performance.
4. **Security Concerns:** Distributing data across multiple nodes introduces security challenges. Ensuring consistent and robust security measures across all nodes is crucial to prevent unauthorized access, data breaches, or other security issues.
5. **Data Fragmentation:** Data fragmentation, the division of data across multiple nodes, can lead to challenges in query optimization and management. Efficiently organizing and retrieving fragmented data may require additional effort.

6. **Cost of Implementation:** Implementing and maintaining a distributed database system can involve higher initial costs. The need for specialized hardware, software, and skilled personnel may contribute to increased implementation costs.
7. **Difficulty in Debugging and Troubleshooting:** Identifying and resolving issues in a distributed environment can be more challenging than in a centralized system. Debugging and troubleshooting distributed systems often require specialized tools and expertise.

2. YugabyteDB

2.1. Introduction

YugabyteDB is an open-source, distributed SQL database designed for high-performance, scalability, and resilience. It is built on a Google Spanner-inspired architecture and combines features of traditional relational databases with the benefits of NoSQL databases. It features automatic sharding for horizontal scalability, transparent multi-region deployments, and compatibility with the PostgreSQL query language, making it easy for developers to integrate with existing tools and applications. With support for on-premises, cloud, and hybrid deployments, YugabyteDB aims to provide a robust and versatile solution for modern distributed database needs. YugabyteDB, apart from being consistent and partition tolerant, supports distributed ACID transactions, auto-sharding, and auto-balancing. Besides PostgreSQL-compatible SQL API, it supports the Apache Cassandra Query Language (CQL). It makes use of a customized version of RocksDB, called DocDB, as its underlying distributed storage engine. DocDB is a log-structured merge-tree (LSM) based "key to object/document" store.

Yugabyte was first developed as closed source commercial software. However, by 2019, its founders realised the power of the open-source community and made it available under the Apache 2.0 license (Wikipedia, YugabyteDB, n.d.). It was developed by former Facebook software engineers.

2.2. Top Level Architecture

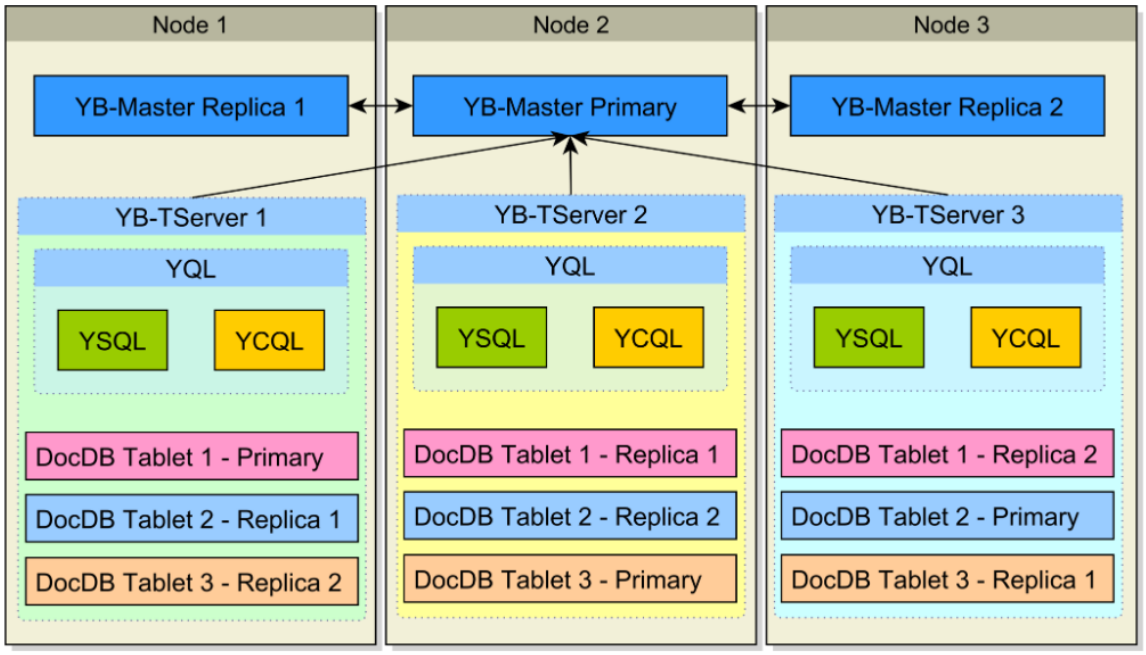


Fig 2.1: Top Level Architecture of YugabyteDB

YugabyteDB uses auto-sharding by default, so data is scattered automatically across multiple nodes (Mihalcea, 2023). However, unlike the Single-Primary replication, all nodes can be used both for reading and writing data.

In order to allow all database nodes to accept writes, one node becomes the Leader (Primary) of a given table record while the other nodes become Followers (Replicas) of that table row.

To support this design architecture, YugabyteDB makes use of YB-Master service that can locate the node that stores the Leader shard for the record we want to change.

2.2.1. YugabyteDB YB-Master

The YB-Master service stores the table metadata and the mapping between the table rows and the nodes that store the Leader Tablet and the Replica Tablets that store the entry for that particular table row.

Knowing where the Leader and Followers are stored, the YB-Master service provides load balancing, as well as auto-sharding.

In order to support fault tolerance, the YB-Master service is run on three nodes, one called YB-Master Leader while others two being YB-Master Followers. The Leader election is done using the Raft consensus protocol.
If a cluster has more than three nodes, the other database nodes that don't host the YB-Master service will just cache the database metadata locally.

2.2.2. YugabyteDB YB-TServer

Unlike the YB-Master where only three database nodes store YB-Master service, in YB-TServer, each database node stores a YB-TServer service, which is responsible for executing queries and statements and storing the underlying data.

The YB-TServer has two main components:

1. The YQL query engine
2. The DocDB storage engine

2.2.2.1. YQL Query Engine

The YQL query engine provides two APIs:

- **YSQL** – SQL query engine based on PostgreSQL
- **YCQL** – a semi-relational API based on Cassandra Query Language

Similar to the PostgreSQL query parser, the YugabyteDB YSQL query engine provides:

1. **The Parser** - It compiles the SQL query into an Abstract Syntax Tree query model
2. **The Optimizer** - It is responsible for generating the Execution Plan
3. **The Executor** - It is responsible for executing the Execution Plan against the data storage engine

2.2.2.2. DocDB Storage Engine

YugabyteDB makes use of a distributed document storage called DocDB which stores the table records as documents in a LSM (log-structured merge) tree structure.

In YugabyteDB, a database table is split into multiple shards, each of which are called Tablets. These Tablets are distributed across multiple nodes. One node will store the Leader Tablet, while the other nodes will store the Followers for that particular Tablet.

When updating a table record, YugabyteDB locates the node holding the Leader Tablet and first applies the change to the Leader Tablet. Then it replicates the change to one Tablet Replica synchronously and to the other Followers asynchronously.

2.3. Features

1. **System Architecture**

YugabyteDB uses shared-nothing system architecture. A table will be split into multiple tablets. Depending on the replication factor, each tablet has its corresponding number of replicas (tablet peers) across different nodes.

2. **Compression**

Relying on RocksDB, YugabyteDB's storage engine is responsible for converting every supported data format (i.e., documents, CQL rows) to key-value pairs and storing them in RocksDB. The way in which data compression is accomplished in YugabyteDB depends on how it is done in RocksDB, which uses Dictionary Compression.

3. **Concurrency Control**

YugabyteDB makes use of MVCC and a variant of OCC for concurrency control. Under a distributed environment, it uses Two-Phase Commit with Early Acknowledgement. When a transaction wants to modify a number of rows, it first writes "provisional" records of each modified row into the target tablet storing the row. These records cannot be seen by the client unless the transaction commits. If conflicts occur when writing these records, the transaction will restart and

abort. Otherwise, the transaction commits and notifies success to the client. After that, the "provisional" records are applied and cleaned asynchronously.

4. Data Model

YugabyteDB's storage engine, DocDB, is based on RocksDB. Unlike RocksDB, DocDB is a "key to object/document" store instead of a "key to value" store. Values in DocDB can be primitive types as well as object types (e.g., lists, sorted sets, and sorted maps) with arbitrary nesting.

5. Foreign Keys

Foreign keys can be created with the CREATE TABLE and ALTER TABLE commands.

6. Indexes

YugabyteDB supports log-structured merge-tree (LSM) and generalized inverted (GIN) indexes. These indexes are based on YugabyteDB's DocDB storage and are similar in functionality to PostgreSQL's B-Tree and GIN indexes, respectively.

7. Joins

YugabyteDB SQL API supports the different types of joins such as cross join, inner join, right outer join, left outer join, and full outer join and join algorithms such as Nested Loop Join, Hash Join, Sort-Merge Join.

8. Logging

YugabyteDB uses the Raft distributed consensus algorithm for replication, so all the changes to the database will be recorded in Raft logs, which can be used during recovery.

9. Query interface

YSQL is a PostgreSQL code-compatible API, accessed via standard PostgreSQL drivers using native protocols. It exploits the native PostgreSQL code for the query layer and replaces the storage engine with calls to the pluggable query layer.

YCQL is a Cassandra-like API, accessed via standard Cassandra drivers using the native protocol port of 9042. In addition to the 'vanilla' Cassandra components, YCQL is augmented with features

like Transactional consistency. Unlike Cassandra, Yugabyte YCQL is transactional, JSON data types supported natively and Tables can have secondary indexes.

10. Storage Architecture

YugabyteDB is a disk-oriented database management system. However, as its storage engine is implemented as a log-structured merge-tree (LSM), some of the data will be stored in memory before flushed out to disk.

11. Storage Model

YugabyteDB's storage model depends on RocksDB, which uses Static Sorted Table (SST) format.

2.4. Raft Consensus Protocol

The Raft algorithm for the leader election process ensures that only one leader is active at any given time, and the system achieves consensus on the state of the distributed log (Wikipedia, Raft (algorithm), n.d.). The use of timeouts, voting, and log replication mechanism helps Raft maintain a robust and fault-tolerant distributed system.

Each server participating in the protocol can only take one of the following roles at a time:

1. **Follower**

   Followers only respond to RPCs, but they do not initiate any communication.

2. **Candidate**

   Candidates start a new election, incrementing the term, requesting a vote, and voting for themselves. Depending on the outcome of the election, candidates become leader, follower, or restart these steps (within a new term). Only a candidate with a log that contains all committed commands can become leader.

3. **Leader**

   The leader sends heartbeats (empty AppendEntries RPCs) to all followers, thereby preventing timeouts in idle periods.

All of these roles have a randomized time-out, on the elapse of which all roles assume that the leader has crashed and they convert to be candidates, triggering a new election and incrementing the current term.

The diagram below shows the state change of servers in the Raft Consensus algorithm.

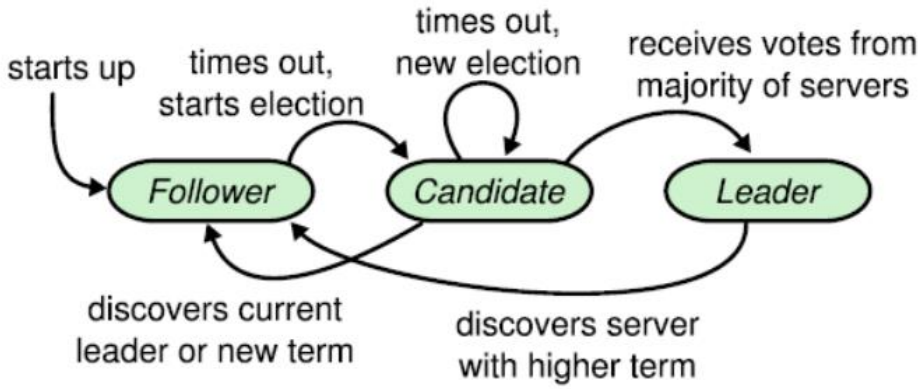


Fig 2.2: State Change of servers in Raft Consensus Algorithm

The brief overview of Raft Consensus Algorithm is provided below.

1. **Initialization**
    a. In the beginning, all nodes in the Raft cluster start as followers.
    b. A follower is a passive node that simply responds to requests from leaders or candidates.
2. **Timeouts**
    a. Each follower has a timeout called the "election timeout."
    b. If a follower doesn't receive any heartbeat (communication) from a leader or candidate within this timeout period, it assumes that there is no current leader and starts the election process.
3. **Election Request (Timeout Occurs)**
    a. When a follower's election timeout occurs without receiving any communication, it transitions to the candidate state and increments its current term.
    b. The candidate then votes for itself and sends RequestVote RPCs (Remote Procedure Calls) to all other nodes in the cluster.
4. **Voting Process**
    a. Nodes respond to RequestVote RPCs and cast their votes.

b. A node will vote for a candidate if it has not voted for another candidate in the current term and if the candidate's log is at least as up-to-date as its own.

5. **Winning the Election**
    a. If a candidate receives votes from a majority of nodes, it becomes the leader for the current term.
    b. The candidate then sends out AppendEntries RPCs to replicate its log entries across the cluster.
6. **Heartbeats**
    a. Once a leader is elected, it periodically sends out heartbeat messages to all followers to maintain its leadership status.
    b. If a follower doesn't receive a heartbeat or any other communication from the leader within a certain timeout, it may start a new election.
7. **Log Replication**
    a. The leader is responsible for replicating its log entries to all followers. This ensures that all nodes in the cluster have a consistent log and can make consistent decisions.

2.5. Log-Structured Merge-Tree (LSM tree)

LSM is a data structure designed for efficient storage and retrieval of key-value pairs, often used in storage systems, databases, and distributed file systems. LSM trees are particularly well-suited for write-intensive workloads.

A simple version of LSM Trees comprises 2 levels of tree-like data structure,

1. Memtable, which resides completely in memory
2. SStables, which are stored in disk

New records are inserted into the Memtable T0 component. If the insertion causes the T0 component to exceed a certain size threshold, a contiguous segment of entries is removed from T0 and merged into SSTable T1 on disk.

LSM trees are optimized for write-heavy scenarios, making them suitable for scenarios where frequent write operations occur, such as in databases and storage systems.

2.5.1. Components

1. **Memtable**
    a. The Memtable is an in-memory data structure where write operations are initially stored. It is typically implemented as a sorted data structure like a skip list or a red-black tree.
    b. Write operations are fast when performed in memory.
2. **SSTables (Sorted String Tables)**
    a. Data from the Memtable is periodically flushed to disk in the form of SSTables.
    b. SSTables are immutable, meaning that once they are written to disk, they cannot be modified. This property simplifies the storage and retrieval process.
3. **Compaction**
    a. LSM trees periodically perform a process called compaction to merge and compact SSTables. During compaction, overlapping SSTables are merged to create a new, consolidated SSTable.
    b. Compaction helps reduce the number of SSTables, improves read performance, and reclaims disk space.

2.5.2. Read and Write Operation in LSM

2.5.2.1. Read Operation

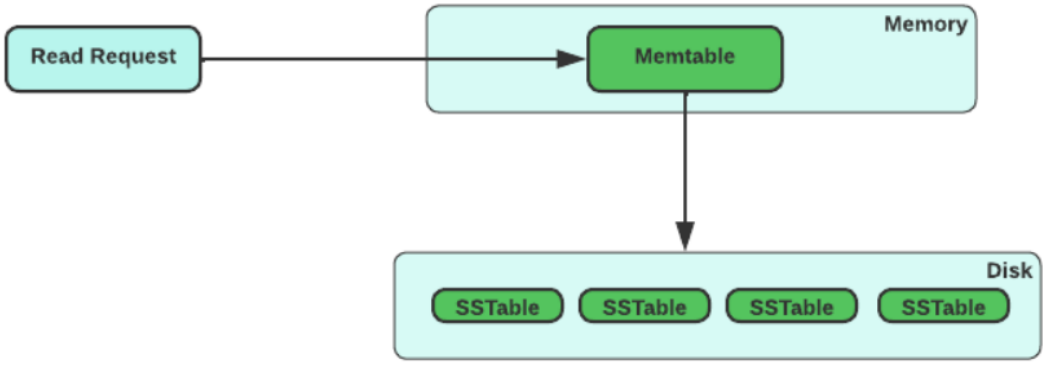


Fig 2.3: Simple LSM Tree

1. Read operations involve searching both the Memtable and SSTables.

2. The data is first checked in the Memtable, and if not found, the SSTables are sequentially searched. The sorted nature of SSTables makes range queries efficient.

2.5.2.2. Write Operation

1. When a write operation occurs, data is first written to the Memtable for fast, in-memory writes.
2. When the Memtable becomes full or reaches a certain threshold, it is flushed to disk as a new SSTable.

2.5.3. Advantages of LSM

LSM provides write efficiency and compaction which can be summarized as below.

1. **Write Efficiency:** LSM trees are optimized for write-intensive workloads because write operations are initially performed in memory, and disk writes are sequential.
2. **Compaction:** Periodic compaction helps maintain efficient storage and retrieval over time.

3. CockroachDB

3.1. Introduction

CockroachDB is an open-source distributed relational database system. It is designed to be highly scalable and resilient, providing strong consistency and ACID (Atomicity, Consistency, Isolation, Durability) guarantees across a distributed architecture (Guy Harrison). The name "Cockroach" reflects the system's ability to survive and adapt to failures, similar to how cockroaches are known for their resilience.

The goal of CockroachDB is to provide a distributed SQL database that combines the best features of traditional relational databases with the scalability and fault tolerance required for modern, cloud-native applications. It is often used in scenarios where high availability, scalability, and consistency are critical requirements.

3.2. Core Features

In the ever-evolving realm of modern applications, the need for robust, scalable, and resilient database solutions has never been more pressing. Amidst this dynamic landscape, CockroachDB emerges as a frontrunner, offering a distributed SQL database that seamlessly amalgamates scalability, resilience, and a familiar PostgreSQL-compatible interface. Unlike traditional databases that struggle to cope with increasing workloads, CockroachDB effortlessly scales horizontally by adding more nodes to its cluster, making it an ideal choice for applications that demand both high availability and scalability.

CockroachDB's resilience extends beyond mere scalability; it is designed to withstand the harshest of environments, ensuring data integrity and availability even in the face of node, network, or datacenter failures (Wikipedia, CockroachDB, n.d.). Its distributed architecture replicates data across multiple nodes, safeguarding against data loss and ensuring continuous operation, even amidst disruptions. Moreover, CockroachDB upholds the principles of ACID transactions, guaranteeing atomicity, consistency, isolation, and durability, making it an impeccable choice for applications that require strict data consistency and reliability.

CockroachDB's capabilities extend beyond traditional data boundaries, enabling geo-partitioning and multi-region deployments, catering to applications with a global user base, optimizing performance, and minimizing latency for users worldwide. Leveraging PostgreSQL compatibility, CockroachDB seamlessly integrates with existing PostgreSQL skills and tools, minimizing the learning curve for developers and ensuring a smooth transition from PostgreSQL to CockroachDB.

3.3. Architecture

3.3.1. The CockroachDB Cluster Architecture

CockroachDB operates as a collection of independent servers, each managing its own local storage. Unlike traditional database clusters with designated primary nodes, CockroachDB features symmetrical nodes, eliminating single points of failure. Data is partitioned into ranges and replicated for redundancy. Clients interact with CockroachDB servers using the PostgreSQL protocol, often facilitated by load balancers for efficient connection distribution. Within each range, a leaseholder node assumes responsibility for data access, ensuring data consistency across replicas.

The diagram demonstrates how data is retrieved from a CockroachDB cluster. A client initiates a request (1), which is intercepted by a load balancer acting as a proxy (2). The load balancer forwards the request to an appropriate node within the cluster (3). This node, designated as the gateway, communicates with the leaseholder node for the specific data range (4). The leaseholder node, responsible for managing access to the data range, retrieves the requested data and sends it back to the gateway node. The gateway node then forwards the retrieved data to the client, completing the data retrieval process.

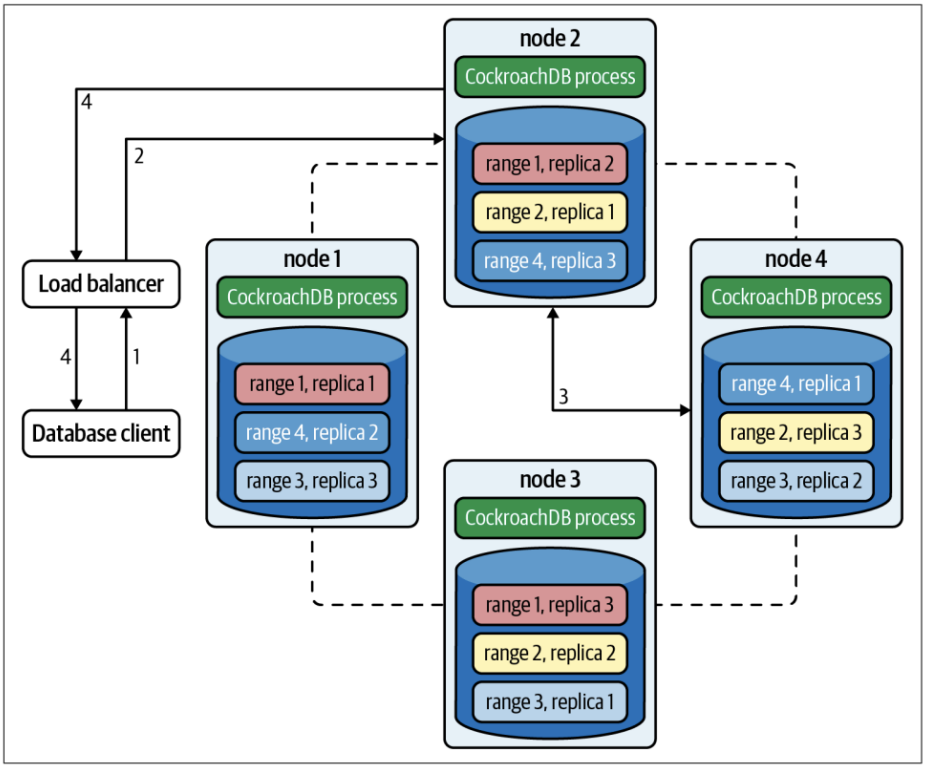


Fig 3.1: CockroachDB Cluster Architecture

3.3.2. Ranges and Replicas

CockroachDB's underlying data organization relies on a KV storage system. Each table is divided into ranges, similar to shards or shard chunks found in other database architectures. The primary key acts as the KV key, and values represent row data. For non-unique indexes, keys are combined, while unique indexes employ index keys as keys and primary keys as values for those keys. The responsibility for managing ranges lies with leaseholder nodes, which often serve as Raft leaders, ensuring consistent data replication across multiple nodes.

3.3.3. SQL Layer

The SQL layer is the topmost layer of the CockroachDB architecture and is responsible for handling all SQL-related tasks, such as parsing SQL statements, optimizing queries, and executing transactions. It acts as an interface between the user and the underlying storage engine, translating

SQL statements into low-level key-value operations that can be understood and processed by the lower layers.

3.3.3.1. Tables as Represented in the KV Store

Each entry in the KV store has a key based on the following structure:
/<tableID>/<indexID>/<IndexKeyValues>/<ColumnFamily>

For a base table, the default indexID is “primary”.

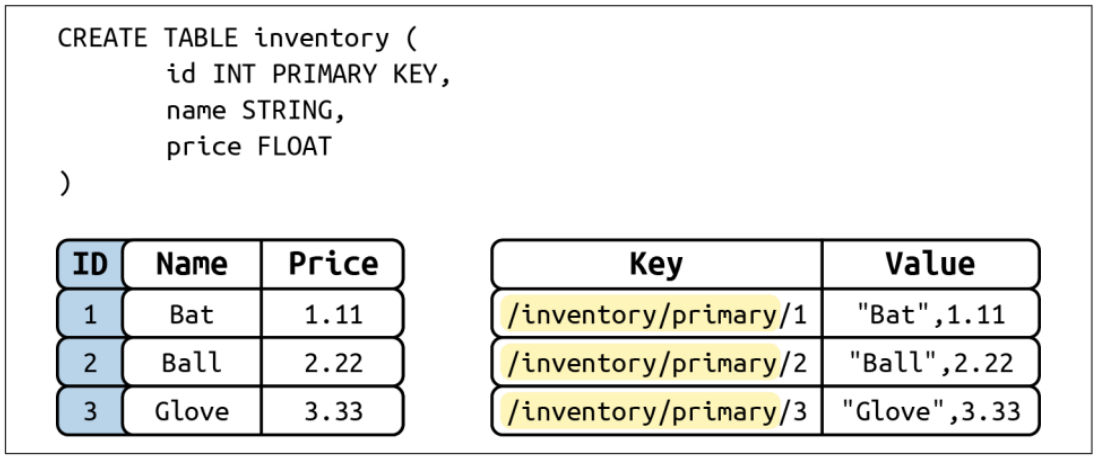


Fig 3.2: CockroachDB Table

3.3.3.2. Column Families

The values of all the columns in a table are consolidated in the value part of a single key-value (KV) entry. Nevertheless, it is feasible to instruct CockroachDB to store sets of columns in distinct key-value entries by utilizing column families. Every column family within a table will be assigned its own key-value entry.

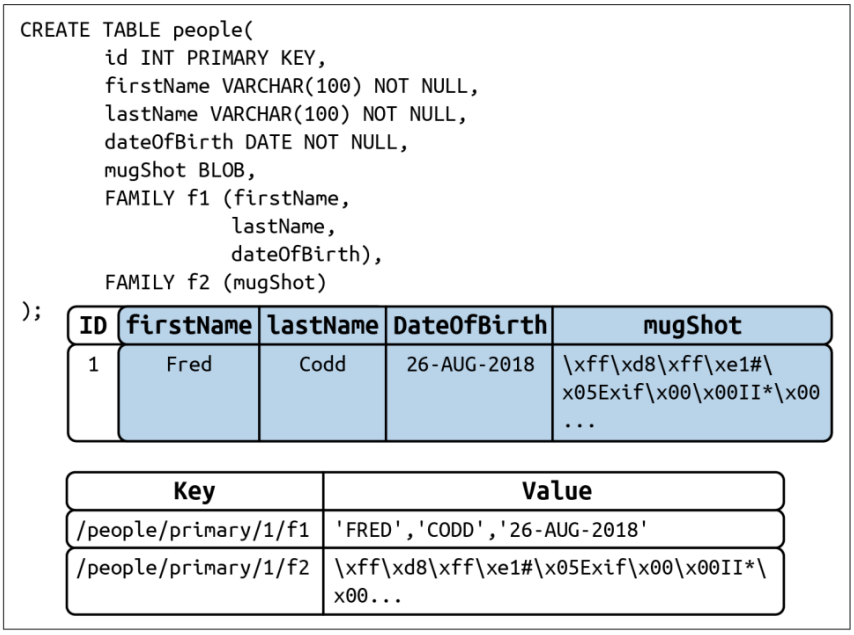


Fig 3.3: CockroachDB Column

3.3.3.3. Indexes in the KV Store

Indexes are represented using a comparable key-value structure. A non-unique index key comprises the table and index name, the key-value, and the main key-value. By default, a non-unique index does not have a specific "value". An example of a non-unique index is shown in the figure below.

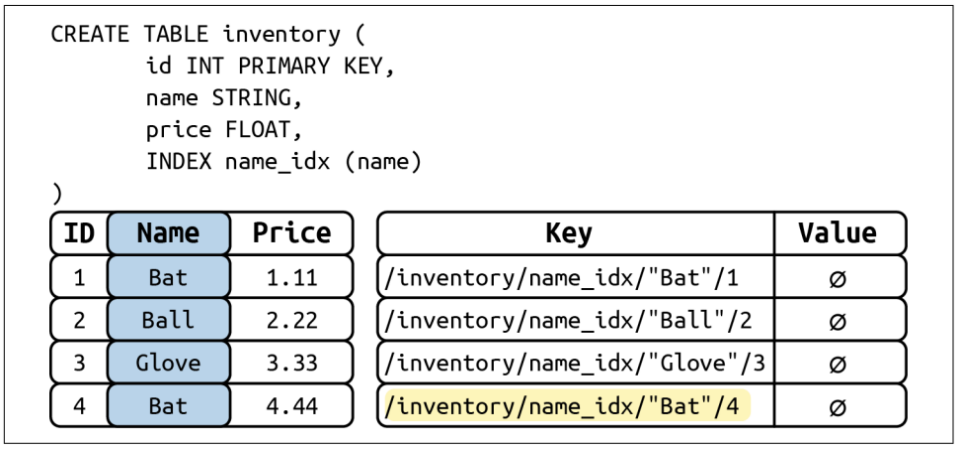


Fig 3.4: CockroachDB Indexes

When using a unique index, the KV value is automatically set to the value of the primary key by default. If the name field in the inventory database used in earlier examples was unique, a unique index on the name field is depicted in the image below.

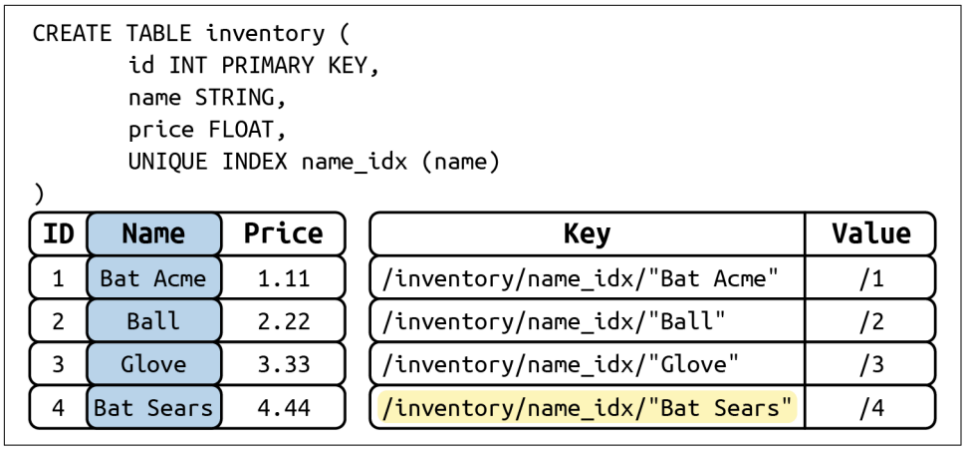


Fig 3.5: CockroachDB Unique Index

3.3.3.4. Inverted Indexes

CockroachDB allows for the definition of columns as either arrays or JSON documents. Inverted indexes enable efficient searches on values contained within arrays or JSON documents. In this instance, the key-values consist of the JSON path and value, together with the primary key, as depicted in the image below.

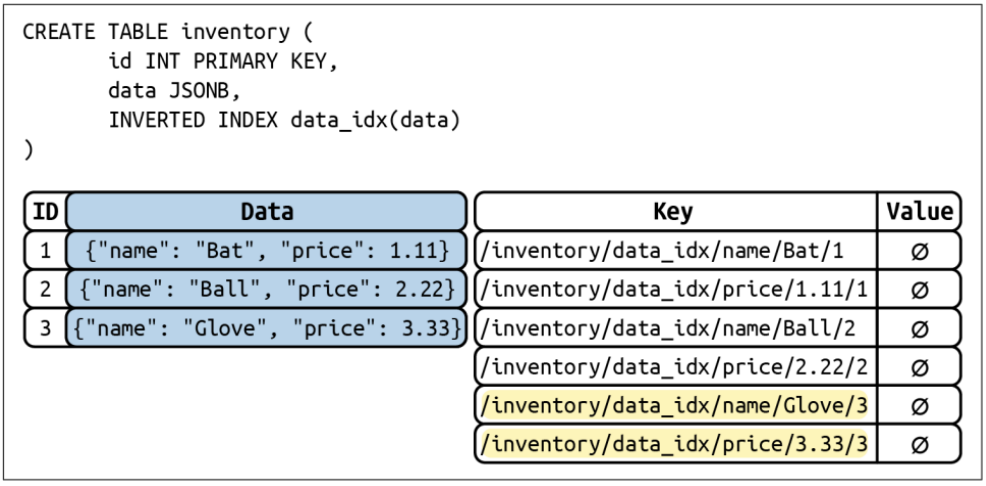


Fig 3.6: CockroachDB Inverted Index

3.3.3.5. The STORING Clause

The STORING clause in the CREATE INDEX statement enables the inclusion of extra columns in the value section of the KV index structure. Adding these extra columns can optimize a query that involves a projection (such as a SELECT list) that only includes those columns and the index keys.

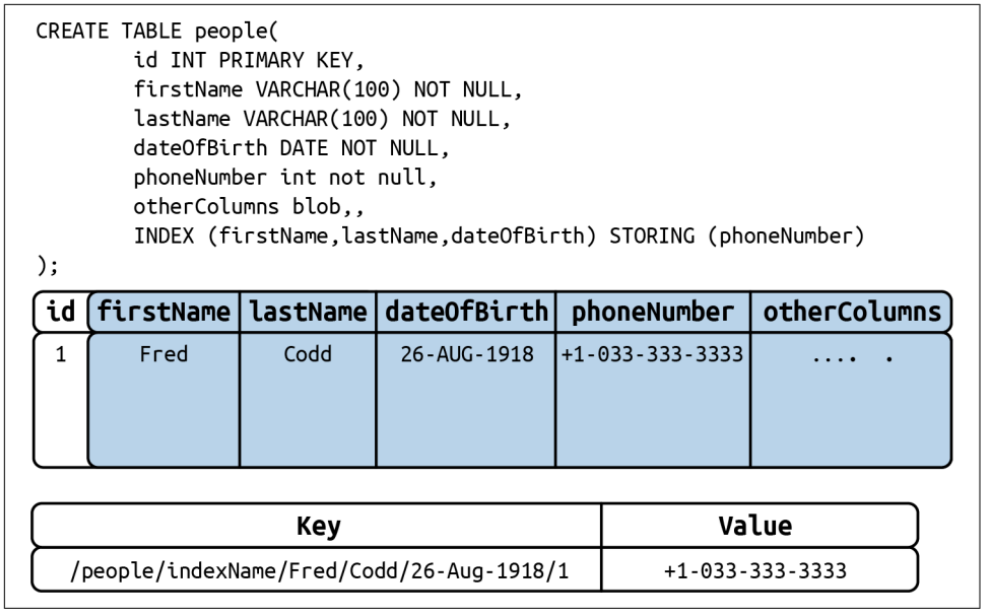


Fig 3.7: CockroachDB Storage Clause

3.3.4. Transaction Layer

The transaction layer in CockroachDB is responsible for implementing ACID (Atomicity, Consistency, Isolation, and Durability) transactions, ensuring data integrity and consistency across multiple nodes. It manages the lifecycle of transactions, coordinates concurrent operations, and resolves conflicts to maintain data integrity.

3.3.5. Distribution Layer

The distribution layer in CockroachDB is responsible for partitioning data across multiple nodes in the cluster, ensuring scalability and high availability. It manages the placement and replication of data, ensuring that data is accessible even if some nodes fail.

3.3.6. Replication Layer

The replication layer in CockroachDB is responsible for replicating data changes across multiple nodes in the cluster, ensuring data consistency and availability in a distributed environment. It utilizes the Raft consensus protocol to maintain a single, consistent view of the data across all nodes.

3.3.7. Storage Layer

The storage layer in CockroachDB serves as the foundation for data storage and retrieval, handling the physical persistence of data on disk. It provides a low-level interface for the replication layer to read and write data.

4. Yahoo Cloud Serving Benchmark (YCSB)

4.1. Introduction

The Yahoo Cloud Serving Benchmark (YCSB) is an open-source database benchmarking suite and a critical analytical component of cloud-based database management system (DBMS) evaluation (Chant, n.d.). It allows users to comparatively measure how various modern SQL and NoSQL DBMS perform simple database operations on generated datasets.

The result returned by YCSB benchmark includes:

1. Runtime
2. Throughput [ops/sec]
3. Latency (avg)
4. Latency (min)
5. Latency (max)
6. Latency (95th percentile)
7. Latency (99th percentile)

These aggregated performance KPIs serve as the foundation for analysis of capabilities and performance of different databases.

4.2. Phases in YCSB

YCSB supports two phases: Load and Run. In the Load phase the initial data records are written into the database, i.e. it is a 100% insert workload. In the Run phase the defined mix of CRUD operations is executed.

4.2.1. Load Phase Command Line

```
bin/ycsb.sh load jdbc -P workloads/workloada -P db.properties -p operationcount=10000 -p recordcount=10000
```

4.2.2. Run Phase Command Line

```
/bin/ycsb.bat run jdbc -P workloads/workloada -P db.properties -p operationcount=10000 -p recordcount=10000
```

4.3. Configurable Parameters

Number of parameters can be configured while performing these tests. Some of them are explained below.

1. **threadcount:** Number of parallel threads
2. **recordcount**: Number of initial records to be loaded into database
3. **operationcount**: Number of operations to be performed during run test (default = 1000)
4. **fieldcount**: Number of database fields of an entry (default = 10)
5. **fieldlength**: Length of each database field (default = 500)
6. **readallfields**: true = all fields are read (default); false = only one field is read (key)
7. **readproportion**: Read portion of the workload (0 - 1) i.e. Number of read operations to be performed during run test
8. **writeproportion**: Write portion of the workload (0 - 1) i.e. Number of write operations to be performed during run test
9. **updateproportion**: Update portion of the workload (0 - 1) i.e. Number of update operations to be performed during run test
10. **scanproportion**: Scan portion of the workload (0 - 1) i.e. Number of scan operations to be performed during run test.

4.4. YCSB Workload

The YCSB workloads are labeled from A to F, and each workload has a specific mix of database operations. The brief overview of the YCSB workloads with proportion of different operations performed is provided below.

Workload A	Update heavy workload	50 % read 50% write (not necessarily the same records)
Workload B	Read mostly workload	95 % read 5 % write (not necessarily the same records)
Workload C	Read only	100% Read
Workload D	Read latest workload	5% insert 95 % read
Workload E	Short ranges	95 % scan 5 % insert
Workload F	Read-modify-write	Forces read of the records to be modified and updated

Fig 4.1: YCSB Workloads

1. Workload A (Update Heavy)
    - Read: 50%
    - Update: 50%
    - Insert: 0%
    - Scan: 0%
    - Delete: 0%

2. Workload B (Read Mostly)
    - Read: 95%
    - Update: 5%
    - Insert: 0%
    - Scan: 0%
    - Delete: 0%

3. Workload C (Read Only)
    - Read: 100%
    - Update: 0%
    - Insert: 0%
    - Scan: 0%
    - Delete: 0%

4. Workload D (Read Latest)
    - Read: 95%
    - Update: 5%
    - Insert: 0%
    - Scan: 0%
    - Delete: 0%

5. Workload E (Short Range Scan)
    - Read: 50%
    - Update: 0%
    - Insert: 0%
    - Scan: 50%
    - Delete: 0%

6. Workload F (Read-Modify-Write)
    - Read: 50%
    - Update: 50%
    - Insert: 0%
    - Scan: 0%
    - Delete: 0%

4.5. Workload Configuration

In the configuration for workload definition, we need to define number of records to be loaded, number of operations to be performed along with proportion of read, update, scan, insert and delete operation to be performed based on our use case.

The sample snip showing workload configuration is provided below.

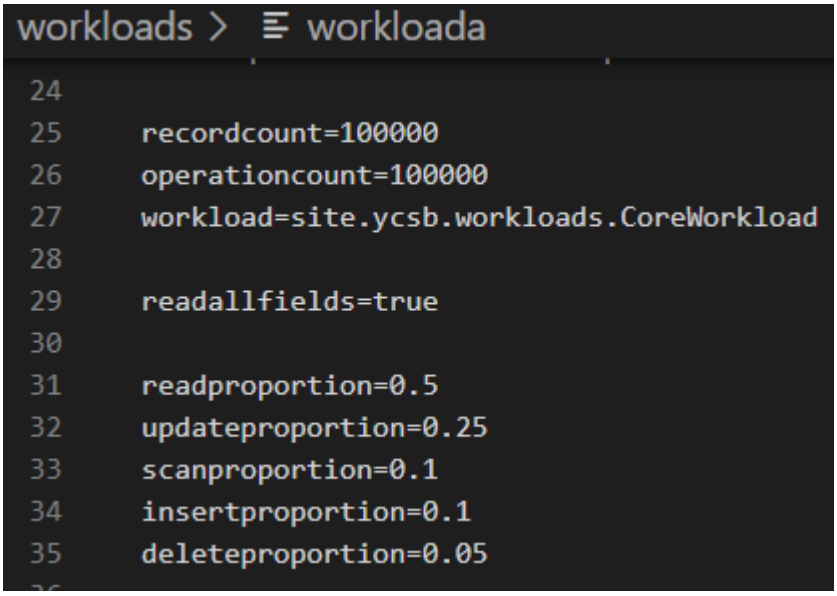


Fig 4.2: Workload Configuration File

5. Implementation

5.1. Multi-Node YugabyteDB Cluster Setup

5.1.1. Step 1: Single-Node Cluster Setup

In order to set up a single node cluster in YugabyteDB, following steps were followed (Choudhury, 2020).

1. Download and Extract YugabyteDB

```
wget https://downloads.yugabyte.com/yugabyte-2.2.0.0-darwin.tar.gz tar xvfz yugabyte-2.2.0.0-darwin.tar.gz && cd yugabyte-2.2.0.0/
```

2. Configure Loopback IP Addresses

```
sudo ifconfig lo0 alias 127.0.0.2
sudo ifconfig lo0 alias 127.0.0.3
```

3. Start Single Node

```
./bin/yugabyted start
```

4. Review Cluster Status
    a. Access the YB-Master admin UI at https://127.0.0.1:7000/
    b. Access the YB-TServer admin UI at https://127.0.0.1:9000/

5. Connect and Load Sample Data

```
./bin/ysqlsh CREATE DATABASE yb_demo; \c yb_demo; \i share/schema.sql \i share/products.sql \i share/users.sql \i share/orders.sql \i share/reviews.sql
```

5.1.2. Step 2: Multi-Node Cluster Expansion

1. Start Node 2

```
bin/yugabyted start --base_dir=/home/yugabyte-2.2.0.0/data2 --listen=127.0.0.2 --join=127.0.0.1
```

2. Verify Cluster Expansion

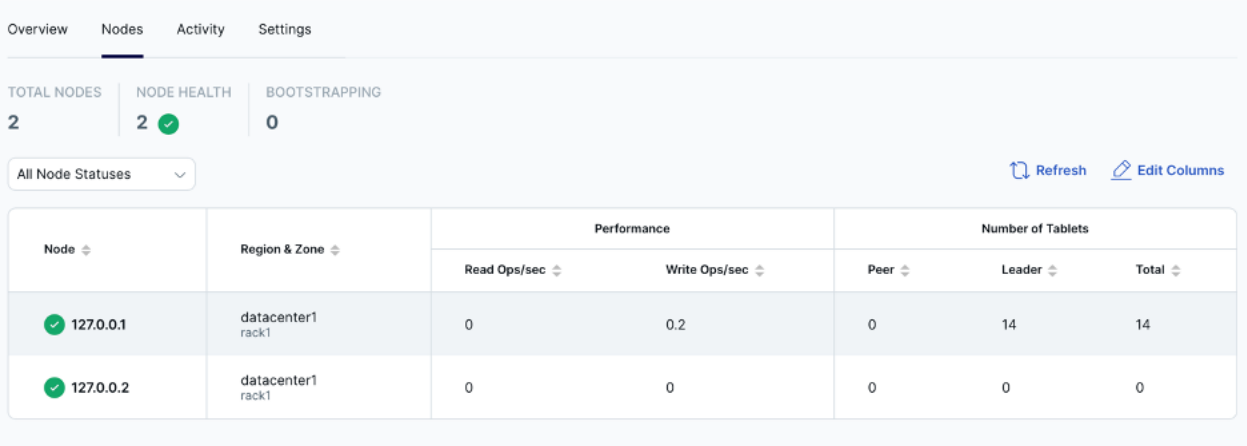


Fig 5.1: Cluster setup in YugabyteDB

5.2. Multi-Node CockroachDB Cluster Setup

5.2.1. Prerequisites

1. Make sure CockroachDB is installed.
2. For testing or development, consider running a single-node cluster.
3. Note: Running multiple nodes on a single host is for testing and not suitable for production (Labs, n.d.).

5.2.2. Step 1: Generate Certificates

1. Create Directories

```
mkdir certs my-safe-directory
```

2. Create CA Certificate and Key Pair

```
cockroach cert create-ca \ --certs-dir=certs \ --ca-key=my-safe-directory/ca.key
```

3. Create Node Certificate and Key Pair

```
cockroach cert create-node \ localhost \ $(hostname) \ --certs-dir=certs \ --ca-key=my-safe-directory/ca.key
```

4. Create Client Certificate and Key Pair

```
cockroach cert create-client \ root \ --certs-dir=certs \ --ca-key=my-safe-directory/ca.key
```

5.2.3. Step 2: Start the Cluster

1. Start First Node

```
cockroach start \ --certs-dir=certs \ --store=node1 \ --listen-addr=localhost:26257 \ --http-addr=localhost:8080 \ --join=localhost:26257,localhost:26258,localhost:26259 \ --background
```

2. Start Second Node

```
cockroach start \ --certs-dir=certs \ --store=node1 \ --listen-addr=localhost:26257 \ --http-addr=localhost:8080 \ --join=localhost:26257,localhost:26258,localhost:26259 \ --background
```

3. Initialize the cluster

```
cockroach init --certs-dir=certs --host=localhost:26257
```

5.2.4. Step 3: Use the Built-in SQL Client

1. Run SQL Client on Node 1

```
cockroach sql --certs-dir=certs --host=localhost:26257
```

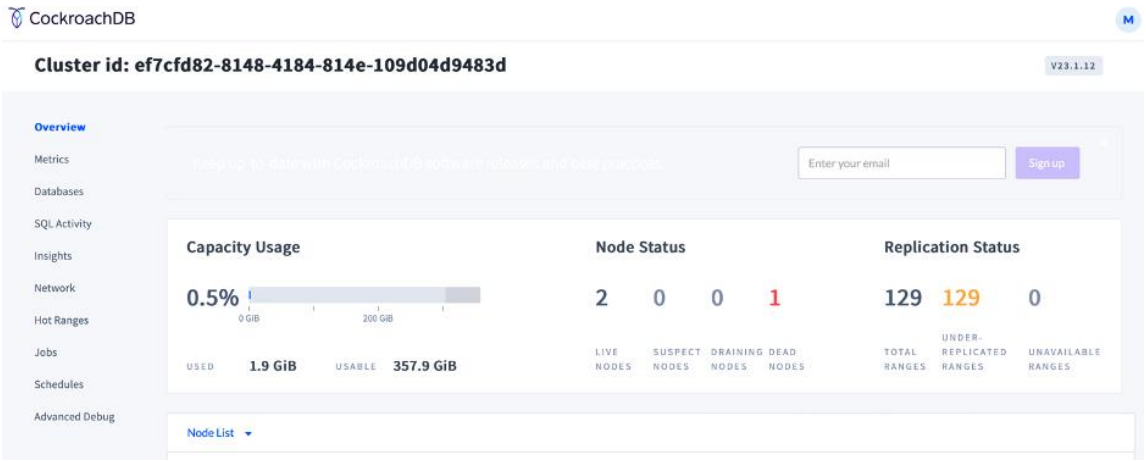


Fig 5.2: Cluster setup in CockroachDB

5.3. Multiple Instances of MySQLDB Setup

5.3.1. Step 1: Install MySQL in your local machine

1. Make sure that MySQL is installed on your local machine.

5.3.2. Step 2: Install second instance of MySQL in different port

1. Create new data directory folder

```
mkdir mysqlInstance2
cp /source_folder/my.cnf /destination/mysqlInstance2/
```

2. Edit the my.cnf file

```
Change port=3306 to port=3310
Change datadir to the path where my.cnf file was copied to
Change server_id=10
```

3. Initialize the mysqld with data directory path

```
mysqld -initialize-insecure -basedir= '/source_folder/bin/mysqld' -datadir= '/destination/mysqlInstance2/'
```

4. Install MySQL Instance as service

```
mysqld -install MySQLServer2 -defaults-file='/destination/mysqlInstance2/my.cnf'
sudo service MySQLServer2 start
```

5. Access mysql from different port

```
mysql -u root -p 3310
```

5.4. YCSB Setup

5.4.1. Step 1: Download YCSB

```
curl -O --location https://github.com/brianfrankcooper/YCSB/releases/download/0.17.0/ycsb-0.17.0.tar.gz tar xfvz ycsb-0.17.0.tar.gz cd ycsb-0.17.0
```

5.4.2. Step 2: Set Up a Database to Benchmark

YugabyteDB, CockroachDB and MySQLDB were installed.

5.4.3. Step 3: Configure the Properties File

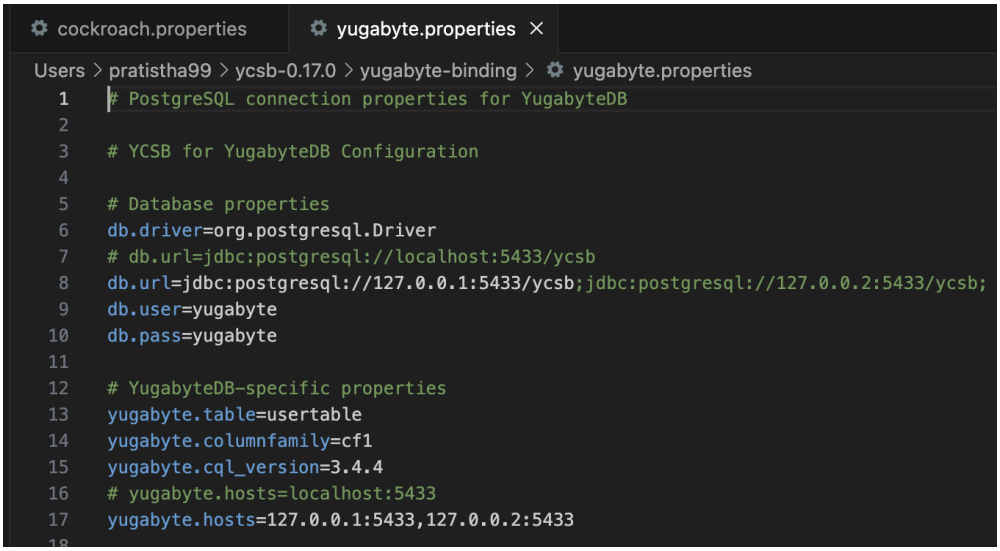


Fig 5.2: Configuring properties file for YugabyteDB

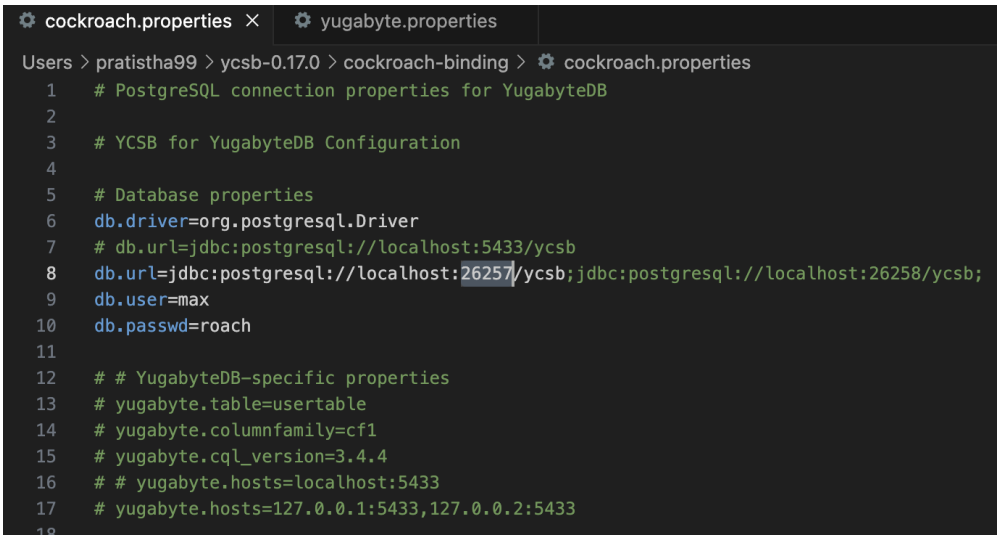


Fig 5.3: Configuring properties file for CockroachDB

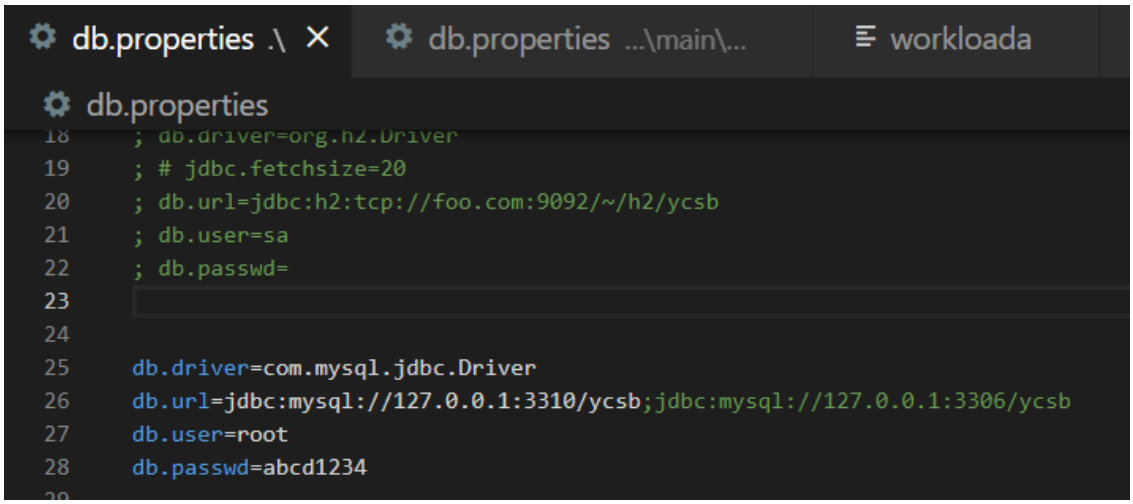


Fig 5.4: Configuring properties file for MySQLDB

5.4.4. Step 4: Run YCSB Command for YugabyteDB

5.4.4.1. Load Phase

Following command can be used to load data into YugabyteDB before starting JDBC workload (Yugabyte, n.d.).

```
Users/john/ycsb-0.17.0/bin/ycsb.sh load jdbc -s -P /Users/john/ycsb-0.17.0/yugabyte-binding/yugabyte.properties -P /Users/john/ycsb-0.17.0/workloads/workloada -p recordcount=1000 -p operationcount=1000
```

Output of Load Phase

```
[CLEANUP: Count=32, Max=1168, Min=42, Avg=156.03, 90=202, 99=1168, 99.9=1168, 99.99=1168] [INSERT: Count=10000, Max=237823, Min=369, Avg=4612.08, 90=8703, 99=43487, 99.9=124287, 99.99=144511]
[OVERALL], RunTime(ms), 1943
[OVERALL], Throughput(ops/sec), 5146.68039114771
[TOTAL_GCS_G1_Young_Generation], Count, 2
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 5
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.2573340195573855
[TOTAL_GCS_G1_Concurrent_GC], Count, 0
```

```
[TOTAL_GC_TIME_G1_Concurrent_GC], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Concurrent_GC], Time(%), 0.0
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 2
[TOTAL_GC_TIME], Time(ms), 5
[TOTAL_GC_TIME_%], Time(%), 0.2573340195573855
[CLEANUP], Operations, 32
[CLEANUP], AverageLatency(us), 156.03125
[CLEANUP], MinLatency(us), 42
[CLEANUP], MaxLatency(us), 1168
[CLEANUP], 95thPercentileLatency(us), 223
[CLEANUP], 99thPercentileLatency(us), 1168
[INSERT], Operations, 10000
[INSERT], AverageLatency(us), 4612.0777
[INSERT], MinLatency(us), 369
[INSERT], MaxLatency(us), 237823
[INSERT], 95thPercentileLatency(us), 14167
[INSERT], 99thPercentileLatency(us), 43487
[INSERT], Return=OK, 10000
```

5.4.4.2. Run Phase

```
/Users/john/ycsb-0.17.0/bin/ycsb.sh run jdbc -s \
-P /Users/john/ycsb-0.17.0/yugabyte-binding/yugabyte.properties \
-P workloads/workloada \
-p recordcount=10000 \
-p operationcount=10000
```

Output of Run Phase

READ: Count=4958, Max=540159, Min=262, Avg=5242.64, 90=7947, 99=38111, 99.9=423679, 99.99=540159] [CLEANUP: Count=32, Max=1323, Min=85, Avg=147.78, 90=138, 99=1323, 99.9=1323, 99.99=1323] [UPDATE: Count=5042, Max=477439, Min=356, Avg=5553.73, 90=8115, 99=41343, 99.9=374271, 99.99=425983]
[OVERALL], RunTime(ms), 2332
[OVERALL], Throughput(ops/sec), 4288.164665523156
[TOTAL_GCS_G1_Young_Generation], Count, 1
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 4
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.17152658662092624
[TOTAL_GCS_G1_Concurrent_GC], Count, 0
[TOTAL_GC_TIME_G1_Concurrent_GC], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Concurrent_GC], Time(%), 0.0
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 1
[TOTAL_GC_TIME], Time(ms), 4
[TOTAL_GC_TIME_%], Time(%), 0.17152658662092624
[READ], Operations, 4958
[READ], AverageLatency(us), 5242.641185962081
[READ], MinLatency(us), 262
[READ], MaxLatency(us), 540159
[READ], 95thPercentileLatency(us), 13199
[READ], 99thPercentileLatency(us), 38111
[READ], Return=OK, 4958
[CLEANUP], Operations, 32
[CLEANUP], AverageLatency(us), 147.78125
[CLEANUP], MinLatency(us), 85
[CLEANUP], MaxLatency(us), 1323
[CLEANUP], 95thPercentileLatency(us), 194
[CLEANUP], 99thPercentileLatency(us), 1323
[UPDATE], Operations, 5042
[UPDATE], AverageLatency(us), 5553.733042443475

```
[UPDATE], MinLatency(us), 356
[UPDATE], MaxLatency(us), 477439
[UPDATE], 95thPercentileLatency(us), 13447
[UPDATE], 99thPercentileLatency(us), 41343
[UPDATE], Return=OK, 5042
```

5.4.5. Step 5: Run YCSB Command for CockroachDB

5.4.5.1. Load Phase

```
/Users/john/ycsb-0.17.0/bin/ycsb.sh load jdbc -s \
-P /Users/john/ycsb-0.17.0/cockroach-binding/cockroach.properties \
-P workloads/workloada \
-p recordcount=10000 \
-p operationcount=10000
```

Output of Load Phase

```
[CLEANUP: Count=32, Max=2553, Min=61, Avg=318.88, 90=534, 99=2553, 99.9=2553,
99.99=2553] [INSERT: Count=10000, Max=136319, Min=9744, Avg=15294.65, 90=18415,
99=25711, 99.9=35871, 99.99=38879]
[OVERALL], RunTime(ms), 6249
[OVERALL], Throughput(ops/sec), 1600.2560409665546
[TOTAL_GCS_G1_Young_Generation], Count, 3
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 8
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.12802048327732438
[TOTAL_GCS_G1_Concurrent_GC], Count, 0
[TOTAL_GC_TIME_G1_Concurrent_GC], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Concurrent_GC], Time(%), 0.0
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
```

```
[TOTAL_GCs], Count, 3
[TOTAL_GC_TIME], Time(ms), 8
[TOTAL_GC_TIME_%], Time(%), 0.12802048327732438
[CLEANUP], Operations, 32
[CLEANUP], AverageLatency(us), 318.875
[CLEANUP], MinLatency(us), 61
[CLEANUP], MaxLatency(us), 2553
[CLEANUP], 95thPercentileLatency(us), 954
[CLEANUP], 99thPercentileLatency(us), 2553
[INSERT], Operations, 10000
[INSERT], AverageLatency(us), 15294.6528
[INSERT], MinLatency(us), 9744
[INSERT], MaxLatency(us), 136319
[INSERT], 95thPercentileLatency(us), 21375
[INSERT], 99thPercentileLatency(us), 25711
[INSERT], Return=OK, 10000
```

5.4.5.2. Run Phase

```
Users/john/ycsb-0.17.0/bin/ycsb.sh run jdbc -s \
-P /Users/john/ycsb-0.17.0/cockroach-binding/cockroach.properties \
-P workloads/workloada \
-p recordcount=10000 \
-p operationcount=10000
```

Output of Run Phase

```
READ: Count=4968, Max=92031, Min=134, Avg=1957.05, 90=1847, 99=31295, 99.9=83327,
99.99=92031] [CLEANUP: Count=32, Max=2245, Min=106, Avg=336.75, 90=452, 99=2245,
99.9=2245, 99.99=2245] [UPDATE: Count=5032, Max=483071, Min=8376, Avg=25304.58,
90=22447, 99=353023, 99.9=432383, 99.99=478207]
[OVERALL], RunTime(ms), 6079
```

```
[OVERALL], Throughput(ops/sec), 1645.0074025333115
[TOTAL_GCS_G1_Young_Generation], Count, 2
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 5
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.08225037012666557
[TOTAL_GCS_G1_Concurrent_GC], Count, 0
[TOTAL_GC_TIME_G1_Concurrent_GC], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Concurrent_GC], Time(%), 0.0
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 2
[TOTAL_GC_TIME], Time(ms), 5
[TOTAL_GC_TIME_%], Time(%), 0.08225037012666557
[READ], Operations, 4968
[READ], AverageLatency(us), 1957.0509259259259
[READ], MinLatency(us), 134
[READ], MaxLatency(us), 92031
[READ], 95thPercentileLatency(us), 7175
[READ], 99thPercentileLatency(us), 31295
[READ], Return=OK, 4968
[CLEANUP], Operations, 32
[CLEANUP], AverageLatency(us), 336.75
[CLEANUP], MinLatency(us), 106
[CLEANUP], MaxLatency(us), 2245
[CLEANUP], 95thPercentileLatency(us), 548
[CLEANUP], 99thPercentileLatency(us), 2245
[UPDATE], Operations, 5032
[UPDATE], AverageLatency(us), 25304.58187599364
[UPDATE], MinLatency(us), 8376
[UPDATE], MaxLatency(us), 483071
[UPDATE], 95thPercentileLatency(us), 25743
[UPDATE], 99thPercentileLatency(us), 353023
[UPDATE], Return=OK, 5032
```

5.4.6. Step 6: Run YCSB Command for MySQLDB

5.4.6.1. Load Phase

```
/Users/john/ycsb-0.17.0/bin/ycsb.sh load jdbc -s \
-P /Users/john/ycsb-0.17.0/jdbc-binding/db.properties \
-P workloads/workloada \
-p recordcount=10000 \
-p operationcount=10000
```

Output of Load Phase

```
[OVERALL], RunTime(ms), 3625
[OVERALL], Throughput(ops/sec), 275.86206896551727
[TOTAL_GCS_PS_Scavenge], Count, 0
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 0
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.0
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 0
[TOTAL_GC_TIME], Time(ms), 0
[TOTAL_GC_TIME_%], Time(%), 0.0
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 1168.0
[CLEANUP], MinLatency(us), 1168
[CLEANUP], MaxLatency(us), 1168
[CLEANUP], 95thPercentileLatency(us), 1168
[CLEANUP], 99thPercentileLatency(us), 1168
[INSERT], Operations, 1000
[INSERT], AverageLatency(us), 2952.12
[INSERT], MinLatency(us), 1995
[INSERT], MaxLatency(us), 23295
[INSERT], 95thPercentileLatency(us), 4731
```

```
[INSERT], 99thPercentileLatency(us), 9583
[INSERT], Return=OK, 1000
```

5.4.6.2. Run Phase

```
Users/john/ycsb-0.17.0/bin/ycsb.sh run jdbc -s \
-P /Users/john/ycsb-0.17.0/jdbc-binding/db.properties \
-P workloads/workloada \
-p recordcount=10000 \
-p operationcount=10000
```

Output of Run Phase

```
[OVERALL], RunTime(ms), 2334
[OVERALL], Throughput(ops/sec), 428.4490145672665
[TOTAL_GCS_PS_Scavenge], Count, 0
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 0
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.0
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 0
[TOTAL_GC_TIME], Time(ms), 0
[TOTAL_GC_TIME_%], Time(%), 0.0
[READ], Operations, 480
[READ], AverageLatency(us), 360.95416666666665
[READ], MinLatency(us), 222
[READ], MaxLatency(us), 2483
[READ], 95thPercentileLatency(us), 510
[READ], 99thPercentileLatency(us), 692
[READ], Return=OK, 480
[CLEANUP], Operations, 1
```

```
[CLEANUP], AverageLatency(us), 1129.0
[CLEANUP], MinLatency(us), 1129
[CLEANUP], MaxLatency(us), 1129
[CLEANUP], 95thPercentileLatency(us), 1129
[CLEANUP], 99thPercentileLatency(us), 1129
[UPDATE], Operations, 520
[UPDATE], AverageLatency(us), 2866.026923076923
[UPDATE], MinLatency(us), 2150
[UPDATE], MaxLatency(us), 18463
[UPDATE], 95thPercentileLatency(us), 4611
[UPDATE], 99thPercentileLatency(us), 7319
[UPDATE], Return=OK, 520
```

6. Comparison between YugabyteDB and CockroachDB

6.1. Similarities between YugabyteDB and CockroachDB

YugabyteDB's sharding, replication, and transactions architecture is similar to CockroachDB in that both are inspired by the Google Spanner design paper. Additionally, both use Raft as the distributed consensus replication algorithm and RocksDB as the per-node storage engine. Both of them were released under the Apache 2.0 open-source license.

6.2. Difference between YugabyteDB and CockroachDB

Yugabyte and CockroachDB are both distributed SQL databases designed to provide high availability, fault tolerance, and scalability. While they share similarities in terms of their goals and functionality, there are differences between the two in terms of architecture.

In case of CockroachDB, tables and indexes are broken down into ranges, while they are divided into tablets in case of YugabyteDB. In both databases, these Ranges/Tablets are replicated via Raft mechanism. Distributed transactions are provided across arbitrary ranges/tablets. Replicas of Tablets are stored data in RocksDB in YugabyteDB while CockroachDB has built their own open source KV store called Pebble in order to store replicas of Ranges.

The SQL tables in Yugabyte use **hash partitioning** by default. Rows are spread evenly across a cluster by hashing each row key. By contrast, Cockroach uses **range partitioning** where rows are clustered together in chunks that are sorted by key.

In CockroachDB, automatic adjustments to the workload involve the dynamic splitting, merging, and rebalancing of ranges across the entire cluster. Each SQL table is comprised of one or more ranges, with range partitioning used to organize data so that adjacent rows in a table or index become neighboring records in the ranges. These ranges are indexed, allowing for efficient scanning of tables and indexes in the order defined by the indexed columns.

Contrastingly, in Yugabyte, tablets automatically rebalance within the cluster in response to node outages, but they lack inherent capabilities for splitting or merging. When creating a table in Yugabyte, the number of tablets is set at the time of table creation and remains unalterable afterward. The default number of tablets for a table is twice the number of tablet servers in the cluster, although this can be configured using extensions to the CREATE TABLE syntax. Notably, there is a current limit of 50 tablets per table in Yugabyte, although the specific reasons for this limitation are not clearly defined.

When choosing between YugabyteDB and CockroachDB, it's essential to consider specific requirements such as consistency models, deployment options, and ease of integration with existing tools and applications. Additionally, evaluating the performance characteristics and community support for each database can help inform the decision based on the needs of a particular use case.

6.3. Benchmarking Result: Load Phase

6.3.1. Run Time

Run Time during load phase measures the amount of time needed to load data into the database. The better the database performs, the lower its runtime is expected to be.

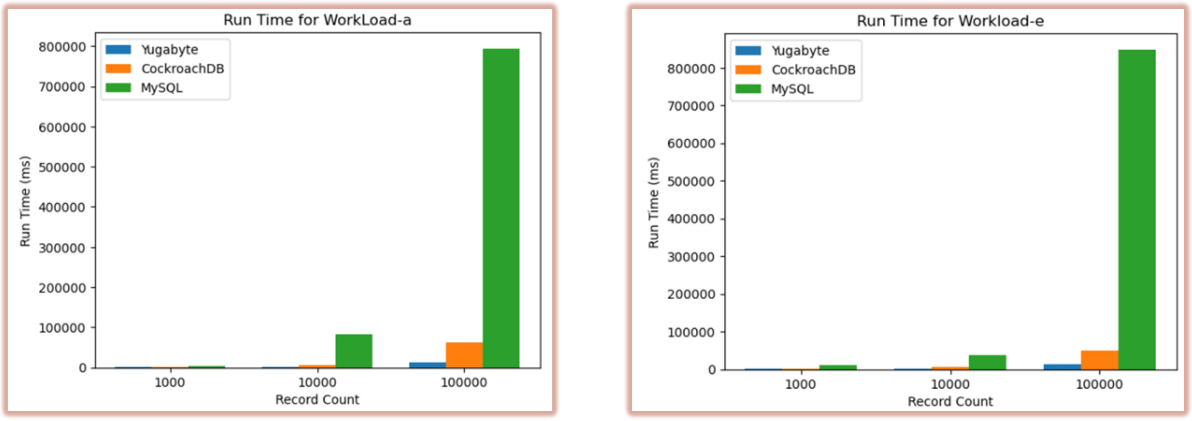


Fig 6.1: Run Time for workload-a and workload-e

The two bar charts depict the runtime in milliseconds for workload-a and workload-e across three different databases: CockroachDB, MySQLDB, and YugabyteDB. We have compared the performance for three record counts (rc_1000, rc_10000, rc_100000).

YugabyteDB

1. As the number of records increases, the time required to load data into the database increases as expected.

CockroachDB

1. As the number of records increases, the time required to load data into the database increases as expected for both workloads.

MySQLDB

1. As the number of records increases, the time required to load data into the database increases as expected.

In conclusion, Yugabyte takes the least amount of time to load data into the database, followed by CockroachDB and MySQL for all record counts.

6.3.2. Throughput Distribution

Throughput during load phase measures the number of insert operations performed per second. The better the database performs, the higher its throughput is expected to be.

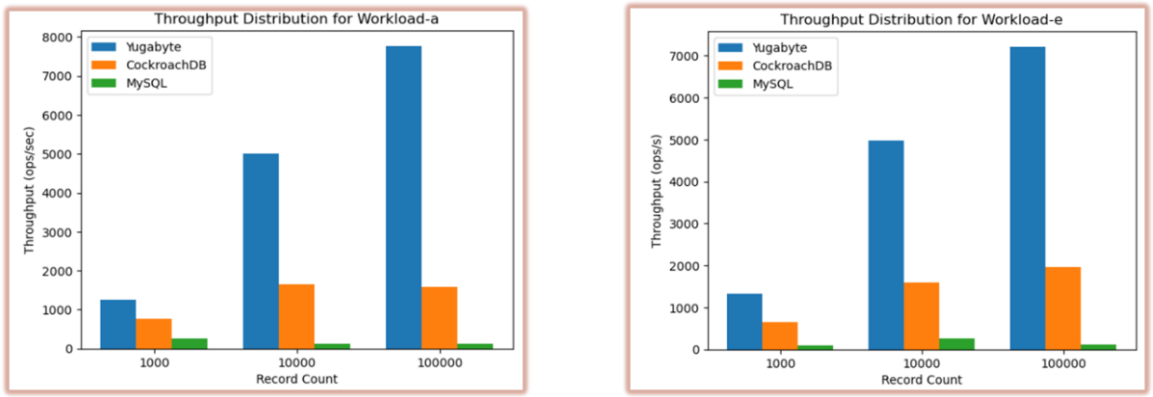


Fig 6.2: Throughput Distribution for workload-a and workload-e

The two bar charts depict the throughput in operations per second for workload-a and workload-e across three different databases: CockroachDB, MySQLDB, and YugabyteDB. We have compared the performance for three record counts (rc_1000, rc_10000, rc_100000).

YugabyteDB

1. As the number of records has increased, the number of operations performed per second has increased for both workloads.

CockroachDB

1. As the number of records has increased, the number of operations performed per second has increased for workload-e while it has fluctuated for workload-a.

MySQLDB

1. As the number of records has increased, the number of operations performed per second has fluctuated for both workloads.

In conclusion, Yugabyte seems to be the best performer in terms of Throughput i.e. it performs a large number of operations per second in comparison to CockroachDB and MySQL holding second and third position respectively.

6.3.3. Insert Latency

Insert Latency measures the time delay between the initiation of an insert operation and its completion. The better the database performs, the lower its insert latency is expected to be.

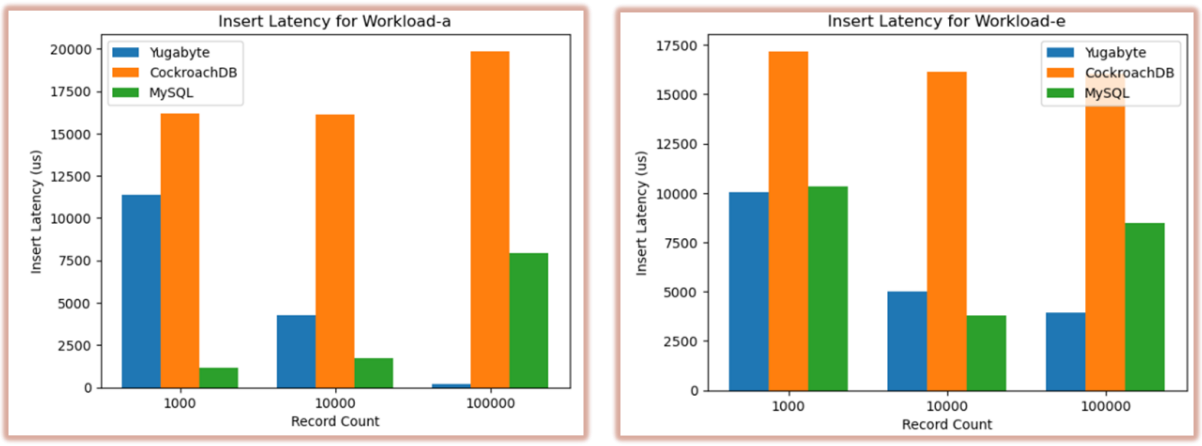


Fig 6.3: Insert Latency for workload-a and workload-e

The two bar charts depict the insert latency in microseconds for workload-a and workload-e across three different databases: CockroachDB, MySQLDB, and YugabyteDB. We have compared the performance for three record counts (rc_1000, rc_10000, rc_100000).

YugabyteDB

1. As the number of records has increased, the insert latency has increased for both workloads.

CockroachDB

2. As the number of records has increased, the insert latency for both workloads have fluctuated.

MySQLDB

3. As the number of records has increased, the insert latency for workload-a has increased while it has fluctuated for workload-e.

In conclusion, CockroachDB seems to have the highest insert latency while the insert latency is least for MySQL for 1000, 10000 records in workload-a and 10,000 records in workload-e and YugabyteDB has least latency for 10000 records in workload-a and 1,000 and 100,000 records in workload-e.

6.4. Benchmarking Result: Run Phase

6.4.1. Run Time

Run Time (during run phase) measures the amount of time needed to execute different operations in the database (read, scan, update, delete). The better the database performs, the lower its runtime is expected to be.

6.4.1.1. Run Time for Workload-a

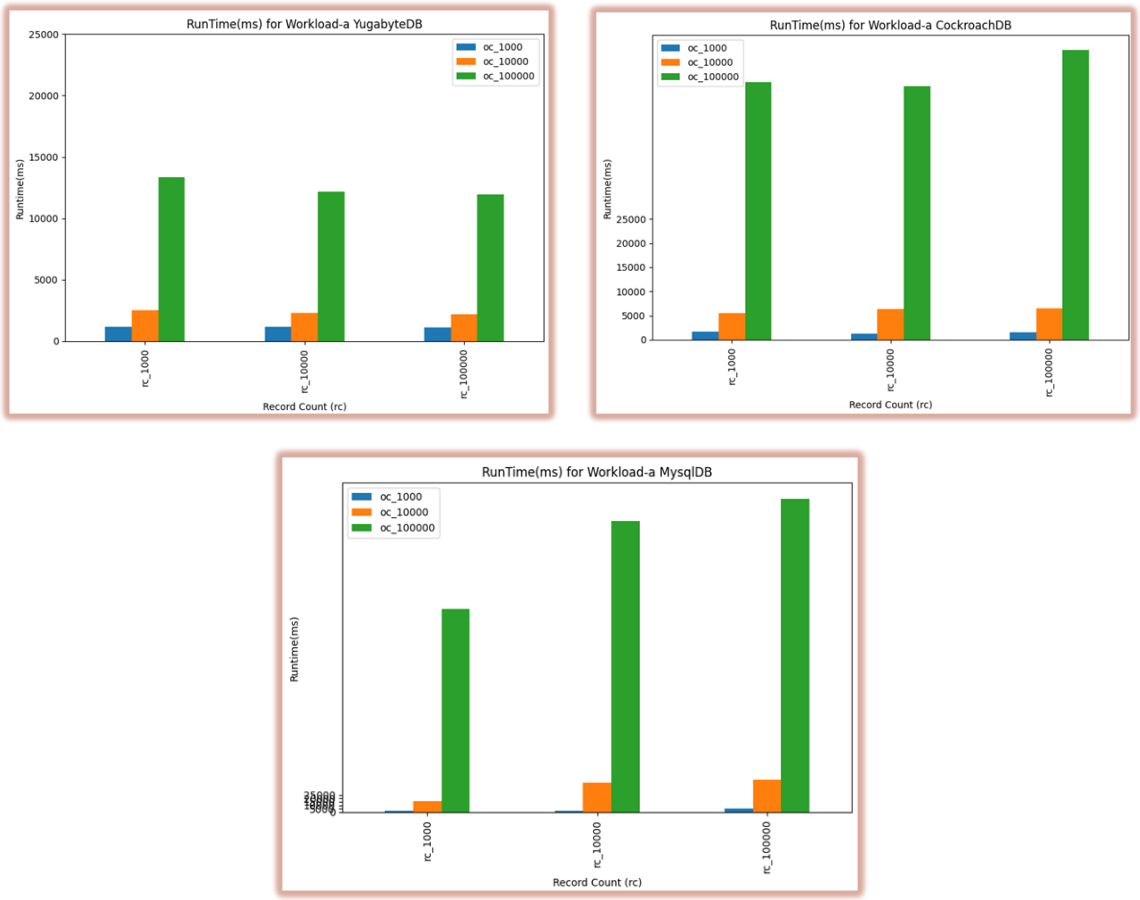


Fig 6.4: Run Time for workload-a

The three bar charts represent the total runtime in milliseconds for workload-a on three different databases: CockroachDB, MySQLDB, and YugabyteDB. The runtime is shown across three levels

of record count (rc_1000, rc_10000, rc_100000) and three different operation counts (oc_1000, oc_10000, oc_100000). Here's an analysis of the runtime performance for each database:

YugabyteDB

1. Exhibits an increase in runtime from rc_1000 to rc_10000 but not as dramatically as the other two databases.
2. At rc_100000, the runtime for oc_10000 and oc_100000 is quite similar, indicating a possible plateau in scalability for very high operation counts.

CockroachDB

1. Shows a consistent pattern where the runtime significantly increases with the record count.
2. The operation count has a notable effect on runtime, especially at higher record counts (rc_10000 and rc_100000), where oc_100000 has a much lower runtime compared to oc_10000.

MySQLDB

1. Indicates a similar trend to CockroachDB, where runtime rises with the record count.
2. However, for MySQLDB, the runtime decreases for oc_100000 compared to oc_10000 at rc_100000, suggesting better handling of larger batches of operations.

Comparative Observations

1. **At rc_1000**: The runtime is relatively low for all databases and operation counts. However, CockroachDB and YugabyteDB have similar runtimes, which are higher than that of MySQLDB.
2. **At rc_10000**: All databases show an increased runtime, with CockroachDB having the highest and MySQLDB the lowest. YugabyteDB's runtime is in the middle but closer to that of MySQLDB.
3. **At rc_100000**: The runtime for CockroachDB and MySQLDB spikes, particularly for oc_10000. However, MySQLDB's runtime for oc_100000 drops back down, suggesting it can handle high operation counts more efficiently. YugabyteDB's runtime remains consistent between oc_10000 and oc_100000, suggesting a limit to scalability.

Overall Insights

1. **YugabyteDB:** While the runtime increases with record count, it does not do so as sharply as CockroachDB, and it shows a potential limit to scalability at the highest record and operation counts.
2. **CockroachDB:** Seems to have scalability issues as the number of records increases, especially at the highest operation count.
3. **MySQLDB:** Shows the best scalability with record count increases, especially at the highest operation count, which is indicative of efficient large-scale operation handling.

In scenarios where runtime is a critical factor, MySQLDB seems to offer the best performance for large-scale operations, while CockroachDB and YugabyteDB may require optimization or configuration adjustments to improve performance at scale.

6.4.1.2. Run Time for Workload-e

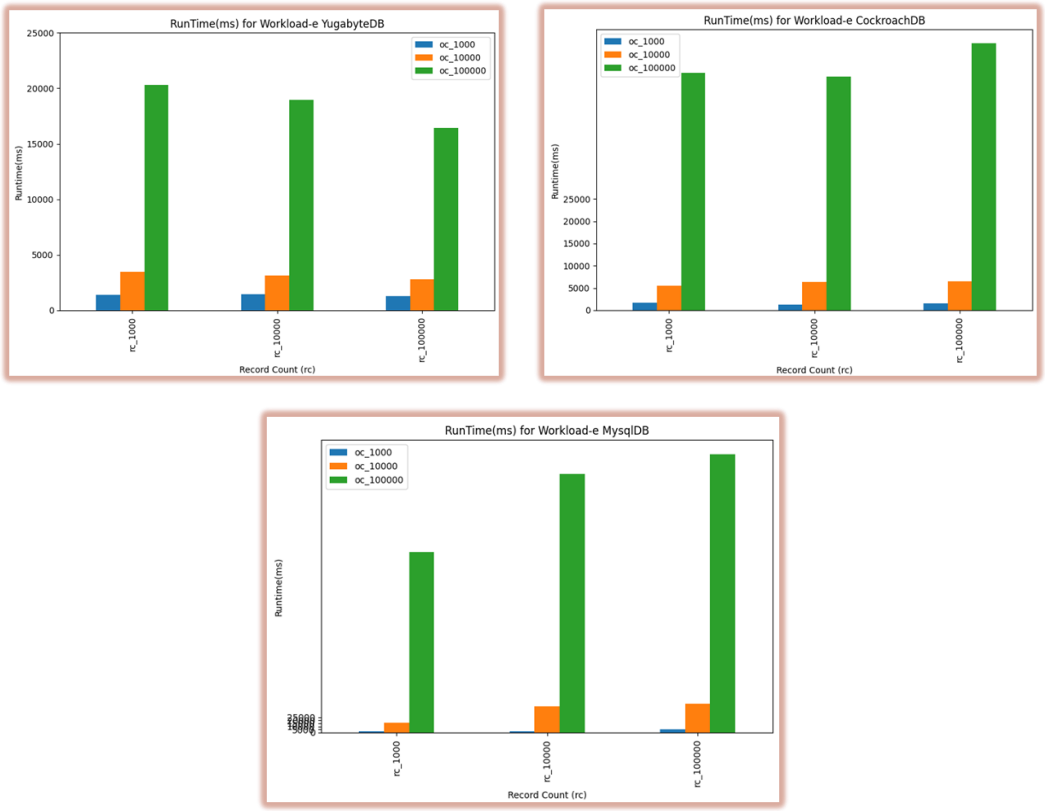


Fig 6.5: Run Time for workload-e

The three bar charts illustrate the total runtime in milliseconds for executing workload-e on three different databases: YugabyteDB, MySQLDB, and CockroachDB. The runtime is measured across three levels of record count (rc_1000, rc_10000, rc_100000) and three different operation counts (oc_1000, oc_10000, oc_100000). We have compared the runtime performance for each database.

YugabyteDB

1. Exhibits the highest runtime at the rc_10000 record count for oc_10000, indicating a possible performance issue at this specific operation and record count scale.
2. The runtime decreases for the oc_100000 operation count at rc_10000, which might suggest better efficiency in handling larger batches of operations.
3. At rc_100000, the runtime increases again for oc_10000 and oc_100000, suggesting scalability challenges at the highest record count.

CockroachDB

1. The runtime increases with the record count, showing the highest runtime at rc_100000 for oc_10000.
2. Unlike the other two databases, CockroachDB does not show a decrease in runtime for oc_100000 at rc_100000; instead, the runtime remains relatively high, which may point to scalability limits.

MySQLDB

1. Shows a similar pattern to YugabyteDB, with the highest runtime at the rc_10000 record count for oc_10000.
2. At rc_100000, there is a notable decrease in runtime for oc_100000 compared to oc_10000, which may indicate better performance when handling very large numbers of operations.
3. The runtime across different operation counts at rc_1000 and rc_100000 is significantly lower than that of rc_10000, showing a non-linear scaling behavior.

Comparative Observations

1. **At rc_1000:** All three databases have relatively low runtimes, indicating efficient handling of smaller workloads.
2. **At rc_10000:** YugabyteDB and MySQLDB exhibit a peak in runtime for oc_10000, but this is reduced at oc_100000. CockroachDB also shows a high runtime for oc_10000 but does not demonstrate a significant reduction at oc_100000.
3. **At rc_100000:** All databases experience an increase in runtime from rc_10000. MySQLDB manages to reduce the runtime for oc_100000, unlike YugabyteDB and CockroachDB, which show increased runtimes for higher operation counts.

Overall Insights

1. **YugabyteDB**: YugabyteDB shows potential performance issues at mid-level record counts with higher operation counts but improves when the number of operations is further increased.
2. **CockroachDB**: CockroachDB's runtime suggests scalability concerns at higher record counts, especially since it does not show improved runtime performance at the highest operation count.
3. **MySQLDB**: The scaling behavior of MySQLDB appears to be the most optimal, particularly at the highest record and operation counts, where it manages to reduce runtime despite increased load.

In scenarios where the total runtime is a critical factor, and especially at larger scales, MySQLDB seems to offer the best performance, whereas YugabyteDB and CockroachDB may require careful consideration and potentially additional optimizations to manage larger workloads efficiently.

6.4.2. Throughput Distribution

Throughput Distribution measures the number of operations (transactions) performed per second. The better the database performs, the higher its throughput is expected to be.

6.4.2.1. Throughput Distribution for Workload-a

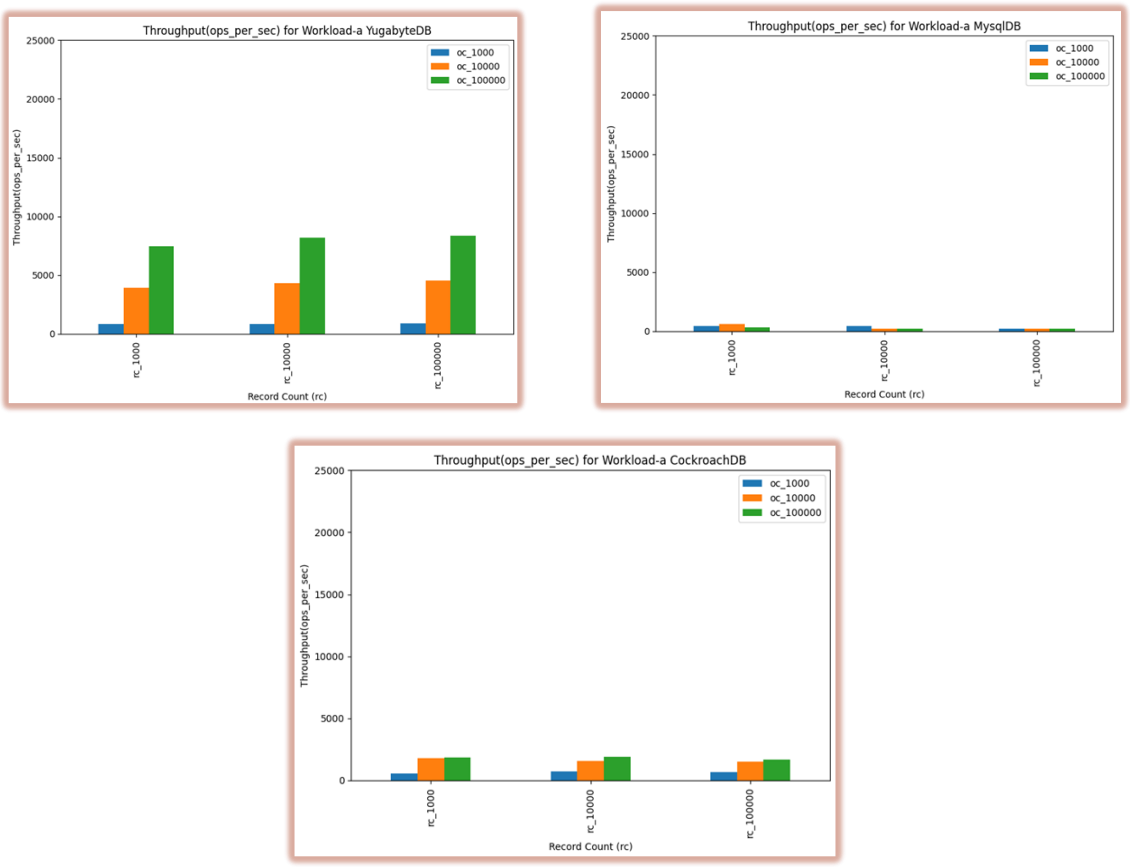


Fig 6.6: Throughput Distribution for workload-a

The three bar charts represent the throughput in operations per second for workload-a across three different databases: CockroachDB, MySQLDB, and YugabyteDB. The throughput is measured at different levels of operation counts (oc_1000, oc_10000, oc_100000) and record counts (rc_1000, rc_10000, rc_100000).

YugabyteDB

1. The throughput increases from oc_1000 to oc_10000 across all record counts, but it does not increase further at oc_100000. Instead, it either remains similar or decreases slightly.
2. At rc_1000 and rc_10000, the highest throughput is observed at oc_10000. At rc_100000, the throughput is almost the same for oc_10000 and oc_100000.

CockroachDB

1. The throughput for CockroachDB increases with the operation count at lower record counts (rc_1000 and rc_10000). However, at the highest record count (rc_100000), the throughput decreases slightly when the operation count increases to oc_100000.
2. The highest throughput for CockroachDB is observed at oc_10000 for rc_10000.

MySQLDB

1. Shows a remarkably consistent throughput across all record counts and operation counts, with only minimal variations.
2. The throughput is not significantly affected by the increase in the number of operations, which could indicate a bottleneck that limits the scalability of the database for this workload.

Comparative Observations

1. **At rc_1000**: All databases have lower throughput for oc_1000, which increases for oc_10000. CockroachDB and YugabyteDB show a decrease or plateau in throughput at oc_100000, while MySQLDB remains consistent.
2. **At rc_10000**: YugabyteDB shows the highest throughput at oc_10000, whereas CockroachDB and MySQLDB have similar throughputs across oc_10000 and oc_100000.
3. **At rc_100000**: CockroachDB and YugabyteDB exhibit a decrease in throughput at oc_100000 compared to oc_10000, suggesting some scalability issues. MySQLDB maintains a consistent throughput, although low.

Overall Insights

1. **YugabyteDB**: Demonstrates good scalability up to a point but may face performance bottlenecks or overheads that prevent further increases in throughput at the highest operation counts, especially at larger scales.
2. **CockroachDB**: Appears to handle the oc_10000 operation count efficiently at medium record counts but shows some limitations at the highest record count.

3. **MySQLDB**: Exhibits a flat throughput performance, indicating it may have hit its maximum capacity for this workload, or there might be a performance cap due to the testing environment or the database's internal management.

Considering throughput is essential when selecting a database for a given workload. YugabyteDB might be preferred for moderate scales where the highest throughput is achieved, while MySQLDB's consistent performance could be beneficial in scenarios that require predictable performance. CockroachDB may need tuning or additional resources to improve scalability at higher workloads.

6.4.2.2. Throughput Distribution for Workload-e

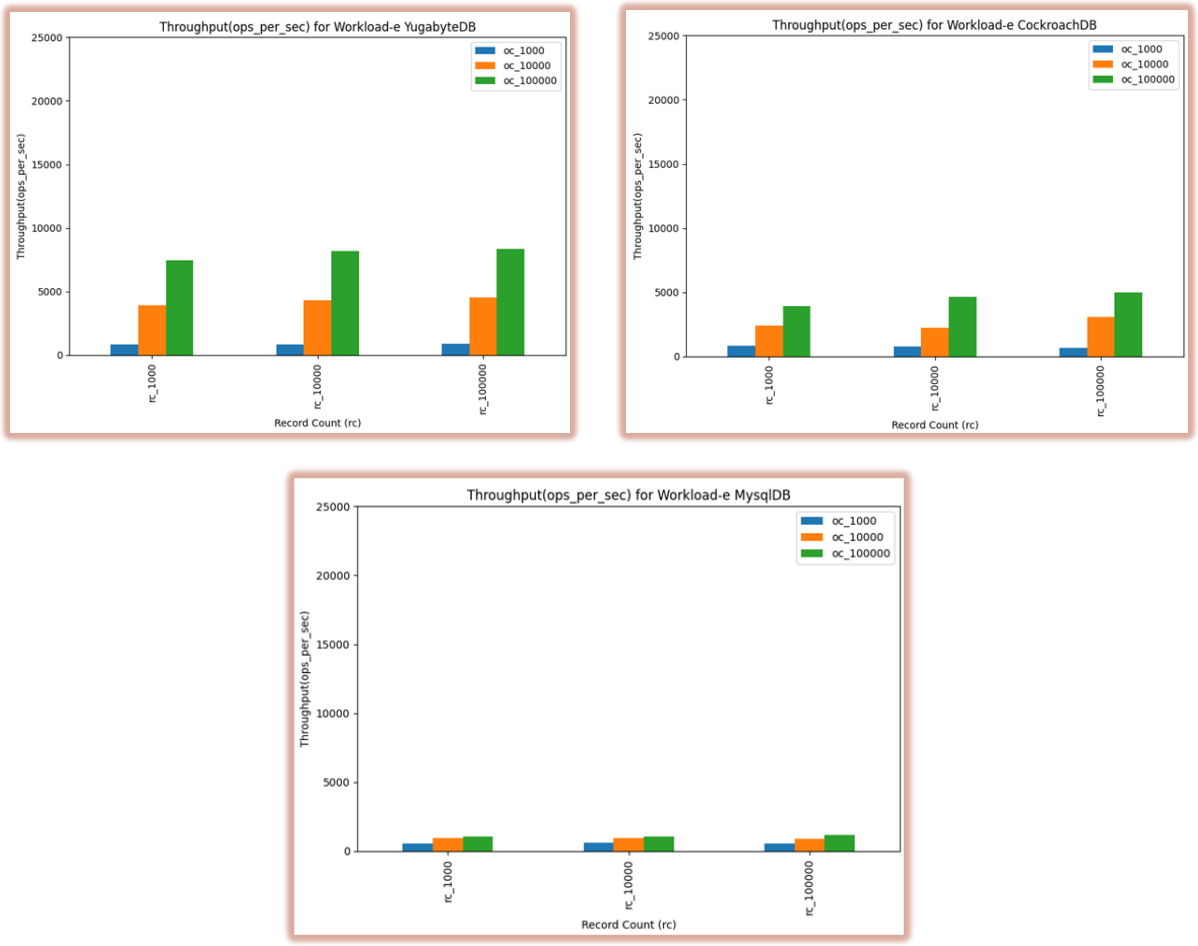


Fig 6.7: Throughput Distribution for workload-e

The three bar charts represent the throughput, measured in operations per second, for workload-e across three different databases: CockroachDB, MySQLDB, and YugabyteDB. The throughput is displayed across three different operation counts (oc_1000, oc_10000, oc_100000) and three record counts (rc_1000, rc_10000, rc_100000). Here's a comparative analysis:

YugabyteDB

1. Exhibits an increase in throughput with the operation count at rc_1000 and rc_10000, but the throughput slightly declines at the highest operation count for rc_100000.
2. The throughput for oc_10000 is the highest across all record counts, which indicates YugabyteDB handles moderate operation counts more efficiently than the smaller or larger counts tested.

CockroachDB

1. Shows a scaling throughput with increasing operation counts at rc_1000 and rc_10000, but a decrease in throughput at the highest operation count (oc_100000) for rc_100000, suggesting a bottleneck at very high workloads.
2. The highest throughput is observed at the intermediate operation count (oc_10000) for rc_10000.

MySQLDB

1. Demonstrates a relatively flat throughput across all operation counts and record counts, indicating stable performance but possibly hitting a performance ceiling that prevents further throughput gains.
2. The consistently low variation suggests that MySQLDB may either be efficiently handling the workload or that it has reached its maximum throughput capacity under the test conditions.

Comparative Observations

1. **At rc_1000**: The throughput increases as the operation count rises for CockroachDB and YugabyteDB, with the highest throughput achieved at oc_10000. MySQLDB's throughput remains consistent regardless of the operation count.

2. **At rc_10000**: A similar pattern is observed; however, CockroachDB's and YugabyteDB's throughput begins to decline at oc_100000, whereas MySQLDB's throughput remains consistent.
3. **At rc_100000**: CockroachDB's and YugabyteDB's throughput dips at oc_100000, suggesting scalability challenges with the highest operation counts. MySQLDB maintains consistent throughput across operation counts.

Overall Insights

1. **YugabyteDB**: Delivers the best throughput at moderate operation counts, but scalability might be an issue at very high operation counts, especially at the highest record count.
2. **CockroachDB**: Potentially faces performance challenges at high operation counts, particularly when handling the largest record counts.
3. **MySQLDB**: Offers consistent throughput, suggesting reliability and predictability in performance. However, it does not show significant throughput improvements with higher operation counts, which might be a limitation or an indication of maximum throughput being reached.

In selecting a database for workload-e, considerations should include not only the raw throughput numbers but also how each database's throughput scales with increased operation and record counts. While MySQLDB's performance is consistent, it might not scale as well as the others. YugabyteDB seems optimized for moderate workloads but may require tuning for very high workloads. CockroachDB shows good scalability up to a certain point but may need attention to handle the highest workloads efficiently.

6.4.3. Update Latency

Update Latency measures the time delay between the initiation of an update operation and its completion. The better the database performs, the lower its update latency is expected to be.

6.4.3.1. Update Latency for Workload-a

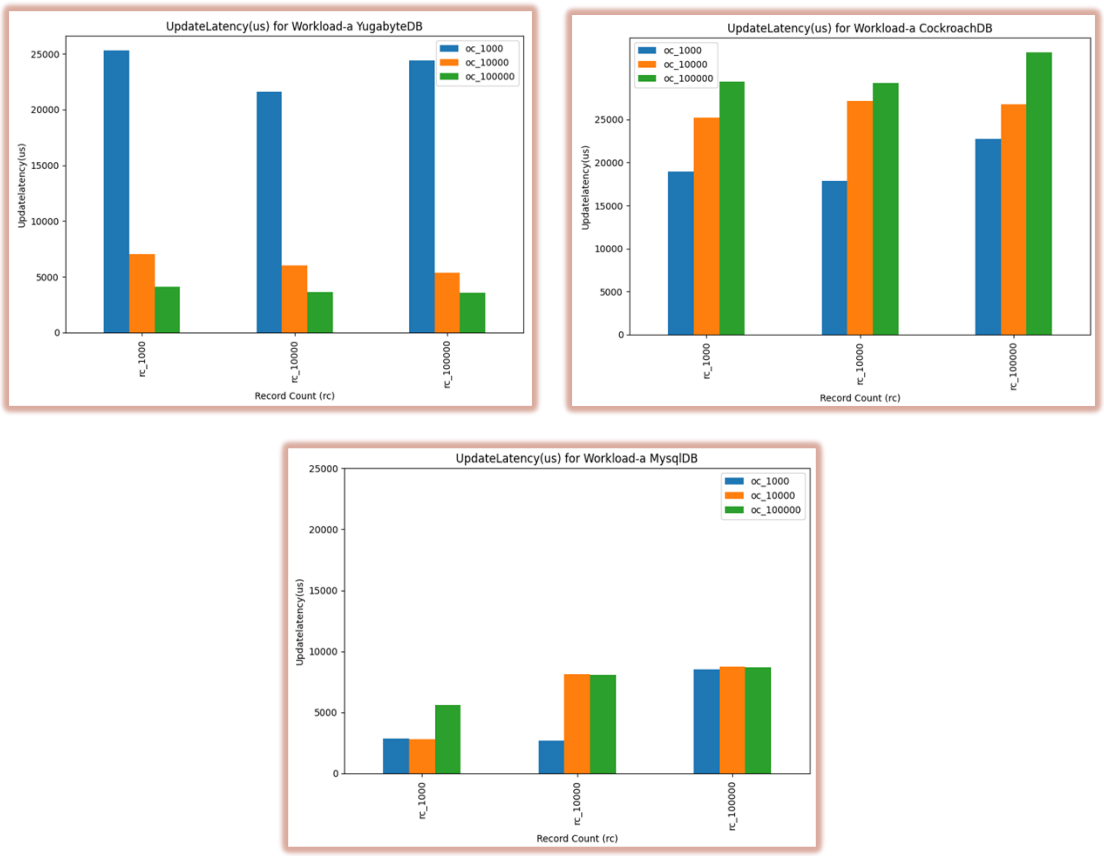


Fig 6.8: Update Latency for workload-a

The three bar charts represent the update latency (in microseconds) for workload-a across three different databases: YugabyteDB, MySQLDB, and CockroachDB. The latency is displayed across three different operation counts (oc_1000, oc_10000, oc_100000) and record counts (rc_1000, rc_10000, rc_100000). Let's analyze the update latency performance for each database:

YugabyteDB

1. Exhibits an increase in update latency with the increase in operation count for rc_10000 and rc_100000, indicating higher latency under heavier workloads.
2. The update latency at rc_1000 remains fairly consistent across different operation counts, suggesting better performance for smaller datasets.

CockroachDB

1. The update latency increases as the record count and operation count increase, with the highest latency observed at rc_100000 for oc_10000.
2. Interestingly, the latency at rc_100000 for oc_100000 is lower than for oc_10000, suggesting more efficient processing of larger operation counts.

MySQLDB

1. Shows a relatively consistent update latency across different operation counts for rc_1000 and rc_10000.
2. There is an increase in latency for rc_100000, especially at oc_10000, but it decreases at oc_100000, suggesting some optimization for handling larger batches of operations.

Comparative Observations

1. **At rc_1000**: All databases handle the smallest record count with relatively low latency. YugabyteDB and MySQLDB show minimal change in latency across operation counts, while CockroachDB has slightly higher latency at oc_10000.
2. **At rc_10000**: Update latency increases across all databases, with YugabyteDB showing a significant spike at oc_10000. MySQLDB and CockroachDB also show increased latency but less pronounced than YugabyteDB.
3. **At rc_100000**: This is where the differences are most notable. YugabyteDB's latency at oc_10000 is much higher than at oc_1000 and oc_100000. MySQLDB has its highest latency at oc_10000 but improves at oc_100000. CockroachDB shows a similar pattern, with a decrease in latency at oc_100000 compared to oc_10000.

Overall Insights

1. **YugabyteDB**: Appears to struggle with update operations as the workload increases, particularly at mid-level operation counts, but manages to handle very high operation counts slightly better.
2. **CockroachDB**: Exhibits increased latency with higher workloads but also seems to have mechanisms that improve performance at the highest operation count tested.

3. **MySQLDB**: Shows relatively stable performance but with some increase in latency under the heaviest workloads, which may suggest that it has optimizations that kick in at very high operation counts.

In scenarios where update latency is critical, MySQLDB appears to offer the most consistent performance across the board, while CockroachDB and YugabyteDB may require careful consideration and potentially additional optimizations to manage larger workloads effectively.

6.4.3.2. Update Latency for Workload-e

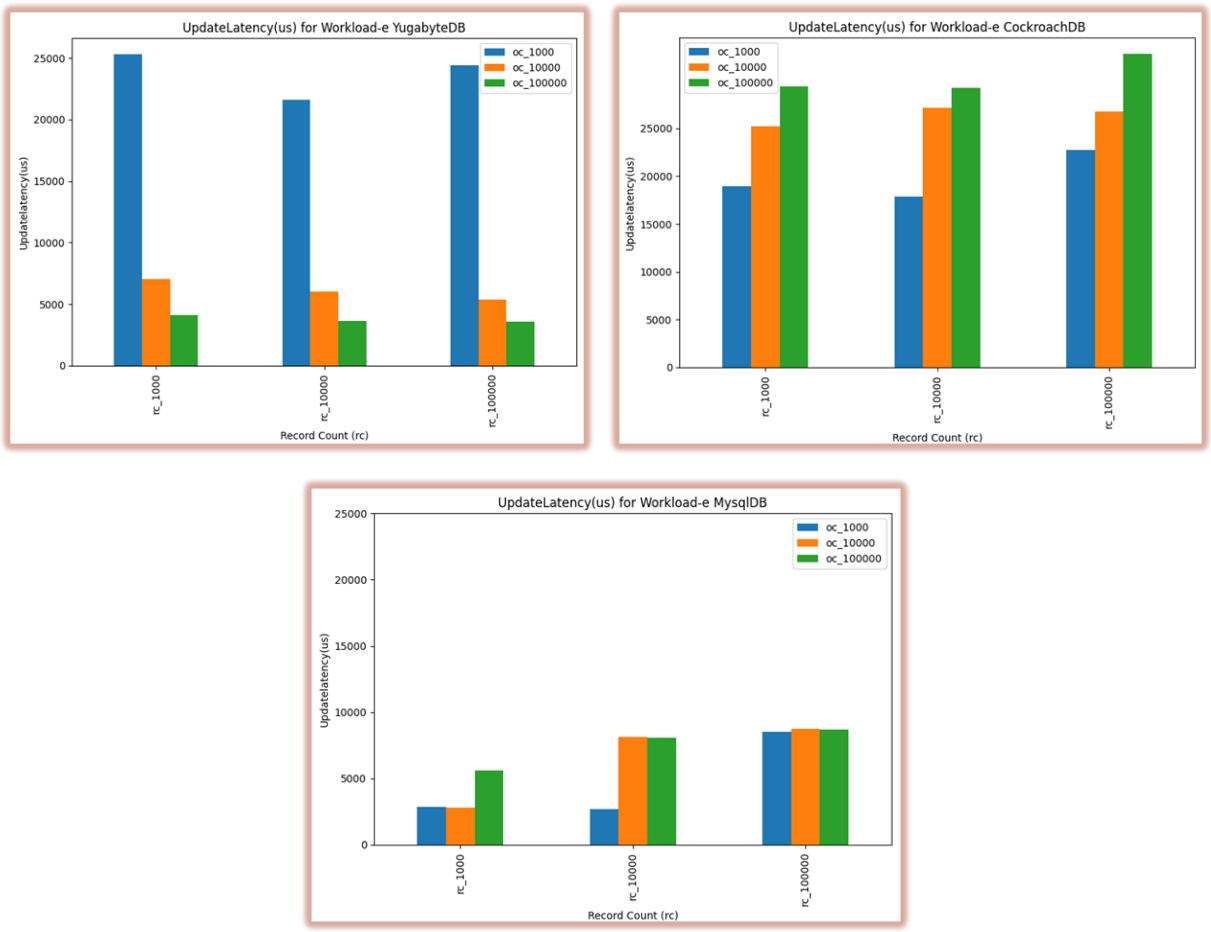


Fig 6.9: Update Latency for workload-e

The three bar charts depict the update latency for workload-a on three different databases: CockroachDB, MySQLDB, and YugabyteDB. The latency is measured across three different

operation counts (oc_1000, oc_10000, oc_100000) and record counts (rc_1000, rc_10000, rc_100000). Here's a summary of each:

YugabyteDB

1. Exhibits a significant increase in update latency from oc_1000 to oc_10000 at rc_10000, which then plateaus or slightly decreases at oc_100000.
2. At rc_100000, a similar trend is observed, with the highest latency at oc_10000.
3. Update latency at rc_1000 remains fairly consistent, indicating stable performance for smaller datasets.

CockroachDB

1. At rc_1000 and rc_10000, the update latency increases with the number of operations, suggesting that the system becomes slower with more concurrent updates.
2. However, at rc_100000, the update latency significantly decreases with oc_100000, which may indicate some form of optimization or batching that becomes effective at large scales.

MySQLDB

1. Shows a consistent increase in update latency with the number of operations at rc_1000.
2. At rc_10000, the latency peaks at oc_10000 and then decreases slightly at oc_100000, indicating better handling of larger operation batches.
3. The update latency at rc_100000 remains relatively flat across different operation counts, suggesting a limit to how much operation count impacts latency for large record sizes.

Comparative Observations

1. **At rc_1000**: All databases show increasing latency with increasing operation counts, with CockroachDB and YugabyteDB showing a more pronounced increase.
2. **At rc_10000**: The latency patterns are similar, with a notable spike for YugabyteDB at oc_10000, suggesting it struggles with update operations at this scale.
3. **At rc_100000**: CockroachDB and MySQLDB handle the largest scale of operations more efficiently, as indicated by the reduced latency at oc_100000, while YugabyteDB's latency remains high.

Overall Insights

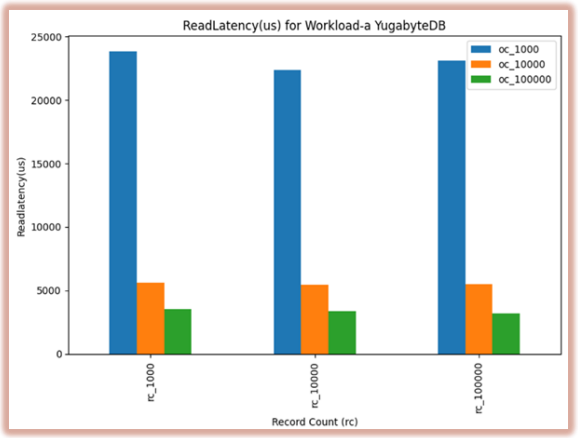
1. **YugabyteDB**: While it handles smaller workloads well, its performance may degrade under heavier update loads, particularly at an intermediate number of operations.
2. **CockroachDB**: May struggle with update operations as the workload increases, but shows an improvement at the highest operation count for the largest dataset, which could be beneficial for batch processing scenarios.
3. **MySQLDB**: Appears to offer a more stable update latency, which increases with workload size but does not show as dramatic a spike as YugabyteDB.

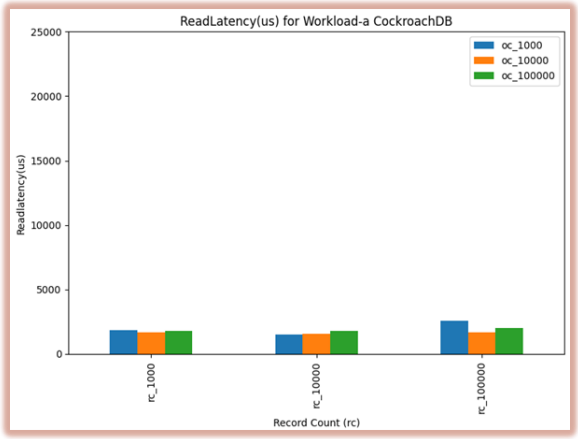
For applications where update performance is critical, MySQLDB seems to offer the most predictable behavior, while YugabyteDB may require tuning for high concurrency workloads. CockroachDB's performance may be variable depending on the scale and needs careful consideration when designing systems for update-intensive applications.

6.4.4. Read Latency

Read Latency measures the time delay between the initiation of a read operation and its completion. The better the database performs, the lower its read latency is expected to be.

6.4.4.1. Read Latency for Workload-a





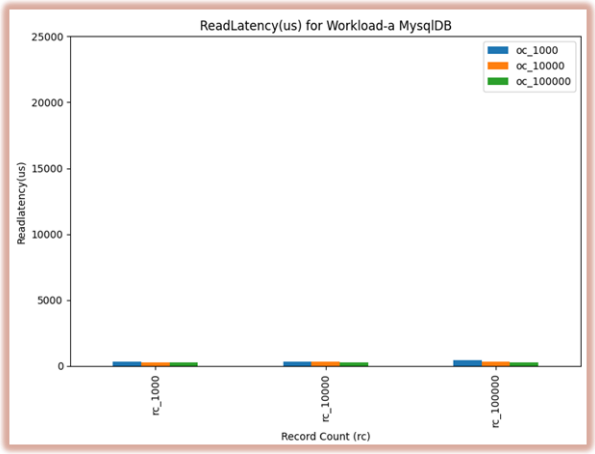


Fig 6.10: Read Latency for workload-a

The three bar charts present the read latency (in microseconds) for workload-a across three databases: CockroachDB, MySQLDB, and YugabyteDB, under different operation counts (oc_1000, oc_10000, oc_100000) and record counts (rc_1000, rc_10000, rc_100000). Here's a comparative analysis of the read latency performance:

YugabyteDB

1. Read latency increases significantly with the number of records; it is lowest for rc_1000 and highest for rc_100000.
2. The latency for oc_10000 and oc_100000 is considerably higher than for oc_1000, especially noticeable at rc_10000 and rc_100000.

CockroachDB

1. Shows an increase in read latency with the increase in record count. The latency is highest for rc_100000.
2. The operation count doesn't seem to significantly affect the read latency, as the bars are very close in height for each record count.

MySQLDB

1. Exhibits extremely low read latency across all operation and record counts, suggesting efficient read operations.

2. The consistency across different operation counts suggests that the operation count has little to no effect on read latency in MySQLDB.

Comparative Observations

1. **At rc_1000:** All three databases exhibit relatively low latency, with MySQLDB standing out as having the lowest.
2. **At rc_10000:** Read latency begins to differentiate, with MySQLDB maintaining low latency, CockroachDB showing moderate latency, and YugabyteDB displaying a significant increase, especially for higher operation counts.
3. **At rc_100000:** CockroachDB and YugabyteDB both exhibit increased latency, but it's particularly pronounced for YugabyteDB. MySQLDB maintains its low latency, indicating a better read performance scalability.

Overall Insights

1. **YugabyteDB**: Suggests potential scalability issues with read operations as the record count increases, particularly for higher operation counts.
2. **CockroachDB**: Demonstrates a moderate increase in latency with scale, but maintains consistent performance across different operation counts.
3. **MySQLDB**: Shows exceptional read performance with consistently low latency across all tested scales, indicating it is highly optimized for read operations.

In scenarios where read performance is critical, MySQLDB appears to be the best performer, indicating it could be the most suitable for read-heavy workloads. CockroachDB offers a balanced read performance but may need consideration for high-scale scenarios. YugabyteDB, while capable at lower scales, may require additional performance tuning or architectural considerations to maintain low read latency at larger scales.

6.4.4.2. Read Latency for Workload-e

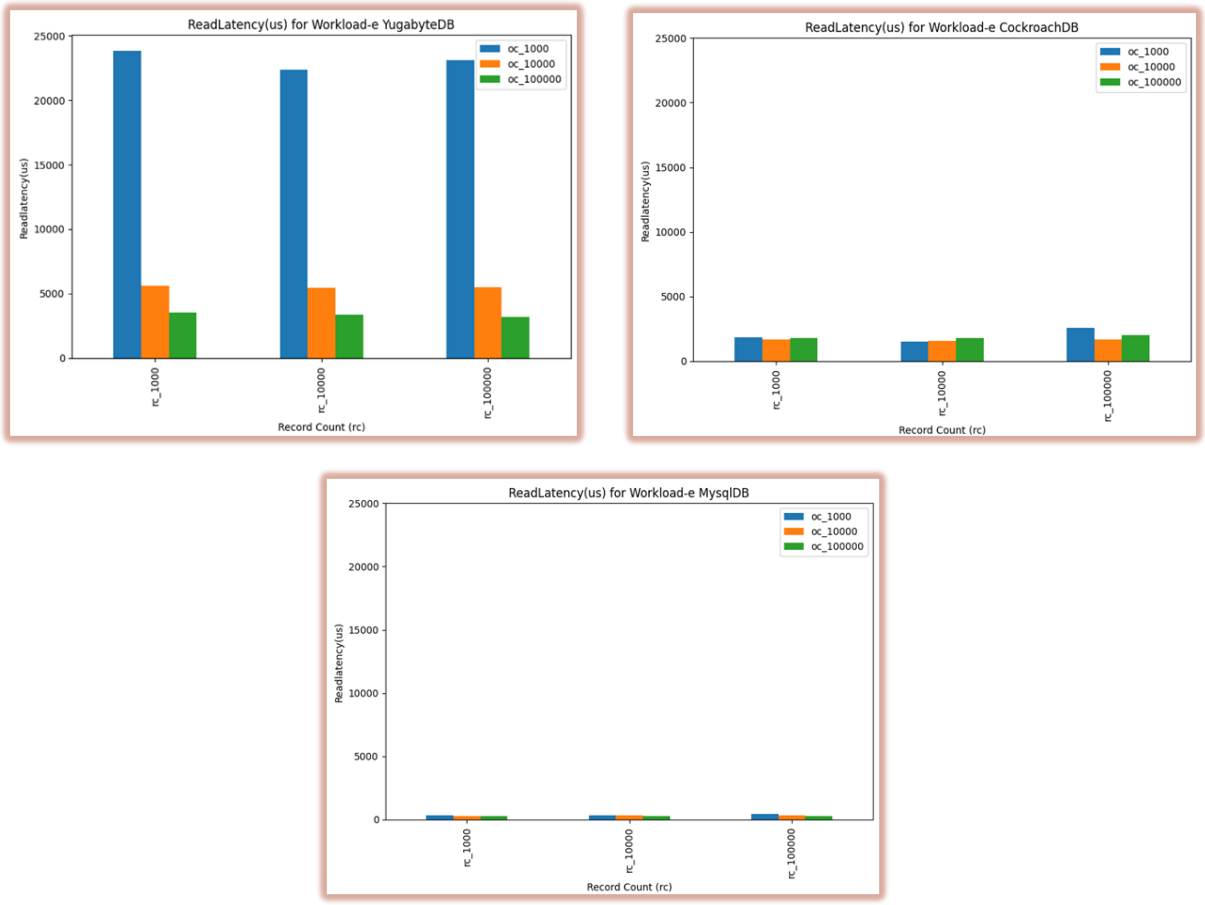


Fig 6.11: Read Latency for workload-e

The three bar charts represent the read latency for workload-e measured in microseconds across three databases: YugabyteDB, MySQLDB, and CockroachDB, at different operation counts (oc_1000, oc_10000, oc_100000) and record counts (rc_1000, rc_10000, rc_100000).

YugabyteDB

1. Exhibits a substantial increase in read latency as the record count increases, with the highest latency observed at rc_100000.
2. The operation count has a noticeable impact on latency, with oc_10000 and oc_100000 resulting in significantly higher latencies than oc_1000, especially at rc_10000 and rc_100000.

CockroachDB

1. Shows an increase in read latency with an increase in record count, but the latency remains lower than YugabyteDB at higher record counts.
2. Similar to MySQLDB, the operation count does not significantly affect read latency, with minimal variation between different operation counts at each record count level.

MySQLDB

1. Demonstrates extremely low read latency across all operation counts and record counts, suggesting that read operations are highly optimized and consistent.
2. The performance is not noticeably affected by either the record count or the operation count, indicating stable scalability in read operations for this workload.

Comparative Analysis

1. **At rc_1000:** All databases exhibit relatively low latencies, but MySQLDB stands out with exceptionally low read latency.
2. **At rc_10000:** While MySQLDB maintains low latency, YugabyteDB's latency increases markedly with higher operation counts, and CockroachDB shows a moderate increase.
3. **At rc_100000:** MySQLDB continues to display low latency, YugabyteDB shows very high latency for higher operation counts, and CockroachDB presents moderate latency.

Overall Insights

1. **YugabyteDB**: While it handles lower record counts well, it exhibits potential scalability concerns at higher record counts and operation counts, as indicated by the substantial increase in read latency.
2. **CockroachDB**: Offers a good balance between performance and scalability, with moderate increases in latency as the workload scales up.
3. **MySQLDB**: It is the standout database in terms of read latency, with near-flat performance across all conditions, suggesting that it is well-suited for read-heavy workloads.

For applications where read performance is critical, MySQLDB appears to be the optimal choice among the three databases based on the presented data. CockroachDB might serve as a reliable

alternative with reasonable scalability, while YugabyteDB may require performance tuning to handle larger workloads effectively.

6.5. Conclusion

To provide an overall conclusion for the performance measures of the distributed databases using the Yahoo Cloud Serving Benchmark (YCSB), we must consider the various aspects of database performance that have been analyzed:

1. **Insert Latency**
    - Insert operations are critical for initial data population and ongoing data ingestion. The latency here will affect how quickly new data can be made available for use in the database.
2. **Update Latency**
    - Update operations are common in most applications, and their latency directly impacts the user experience, especially when dealing with interactive applications where data changes frequently.
3. **Read Latency**
    - The speed at which data can be retrieved is crucial for read-heavy applications. Lower read latency is essential for analytics and reporting queries that must be performed quickly.
4. **Throughput**
    - This measures the number of operations that a database can handle per unit time. Higher throughput indicates a better ability to handle more transactions or operations, which is critical in high-load scenarios.
5. **Runtime**
    - This refers to the time taken to complete a certain number of operations. Shorter runtimes are generally preferable, as they indicate a more responsive and efficient system.

Based on the data for CockroachDB, MySQLDB, and YugabyteDB, here's the conclusion:

- **YugabyteDB** shows good performance at lower operation counts but tends to have higher latencies with mid to high operation counts, especially in update operations. While it can handle higher throughputs effectively in some cases, it may struggle under workloads with a mix of different types of operations.
- **CockroachDB** generally shows a scalability pattern where performance is maintained or improved with increased operation counts. However, it exhibits increased latencies in updates and reads with higher record counts, suggesting potential challenges in workloads with large numbers of records.
- **MySQLDB** demonstrates consistent performance across most metrics, without significant spikes or drops in latency or throughput. This could be beneficial for applications that require predictable performance but may also indicate a ceiling of performance that does not substantially improve with additional resources.

When selecting a distributed database for a particular application, the following considerations based on the YCSB benchmarking would be crucial:

- For applications that require high throughput with a reasonable update latency and can tolerate some variation in performance as the workload increases, YugabyteDB could be considered, though it might need optimization for update-heavy workloads.
- For workloads that are heavy on insertions and have moderate to large data sets, CockroachDB might require careful tuning to balance insert latency and overall runtime.
- MySQLDB could be a strong candidate for applications that need consistent and reliable performance without the need for scaling to very high operation counts.

In conclusion, there is no one-size-fits-all answer, and the choice of the database must be aligned with the specific performance requirements of the application, the nature of the workload, the scalability needs, and the operational complexity the team is prepared to manage. Each database's performance profile suggests different strengths and weaknesses, and the right choice will depend on which aspects are most critical for the application's success.

7. A Custom Use Case

7.1. Distributed Database in e-commerce platform

A distributed database scenario for insert, update, delete, read, and scan operations can be illustrated through a real-world use case like an e-commerce platform. Such platforms often require a distributed database to manage large volumes of data, ensure high availability, and support scalability. We can break down each operation—insert, update, delete, read, and scan in this context as:

1. **Insert Operation**
    a. **Use Case:** When a new product is added to the catalog.
    b. **Example:** A vendor adds a new smartphone model to the online store. The insert operation is used to add this new product, including details like name, price, specifications, and stock quantity, into the product catalog database.
2. **Update Operation**
    a. **Use Case:** Updating product information or inventory.
    b. **Example:** The price of the smartphone changes, or its stock level needs updating due to new shipments arriving or sales occurring. The update operation modifies these specific details in the database without affecting other attributes of the product.
3. **Delete Operation**
    a. **Use Case:** Removing discontinued products.
    b. **Example:** If the smartphone is discontinued or no longer offered by the vendor, the delete operation is used to remove this product from the online catalog, ensuring customers do not see or order it.
4. **Read Operation**
    a. **Use Case:** Viewing product details.
    b. **Example:** A customer browses the e-commerce site and clicks on the smartphone to view its details. The read operation fetches the information from the database and displays it to the user, including price, specifications, and availability.

5. **Scan Operation**
    a. **Use Case:** Searching or filtering through products.
    b. **Example:** A customer wants to see all smartphones within a certain price range or with specific features. The scan operation goes through the database to retrieve all products that match these search criteria.

7.2. Distributed Database Considerations

The distributed database considerations in the above ecommerce platform can be summarized as follows.

1. **Scalability**: As the number of products, vendors, and customers grows, the database needs to scale horizontally to manage increased load and data volume.
2. **Availability and Reliability:** The database must ensure high availability, as any downtime directly impacts sales and customer experience.
3. **Consistency:** Ensuring that all database nodes have up-to-date and synchronized information, especially after insert, update, or delete operations.
4. **Partition Tolerance:** The system should continue to function efficiently even if parts of the database are temporarily inaccessible.
5. **Performance:** Optimizing read and scan operations for performance, as these are frequent actions by users browsing the site.

This scenario demonstrates how a distributed database is essential for handling the diverse data management needs of a dynamic and large-scale e-commerce platform, ensuring efficient and reliable operations.

7.3. Distribution of operations

A general distribution of different operations in this scenario would look like follows.

1. **Read Operation: 50%**
    a. Read operations are typically the most frequent in e-commerce platforms. Every time a user browses products, views details, or checks inventory, a read operation occurs.
2. **Scan Operation: 25%**
    a. Scan operations, which involve searching and filtering products, are also quite common but may be less frequent than simple reads. Users often filter products based on categories, prices, or other attributes.
3. **Insert Operation: 10%**
    a. Insert operations occur when new products are added to the catalog. While essential, this happens less frequently than reads or scans.
4. **Update Operation: 10%**
    a. Update operations are necessary for changing product details or inventory levels. They are as frequent as inserts but are crucial for maintaining accurate and current data.
5. **Delete Operation: 5%**
    a. Delete operations are the least frequent. Products are removed from listings less often than they are added, viewed, or updated.

7.4. Benchmarking using YCSB

To illustrate the above use case, we prepared a custom workload in YCSB and measured performance metrics during load and run phase for YugabyteDB and CockroachDB. The test was performed for 100,000 records and 100,000 operations in total. The configuration parameters set is shown in figure below.

```
recordcount=100000
operationcount=100000
workload=site.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.5
updateproportion=0.25
scanproportion=0.1
insertproportion=0.1
deleteproportion=0.05
```

Fig 7.1: Configuration Parameter for custom use case

During load phase, YugabyteDB has better performance than CockroachDB with lower run time, high throughput and low insert latency.

On the other hand, when run time, throughput, insert latency and update latency are considered during run phase, performance of YugabyteDB seems to be better than CockroachDB. However, the read latency for YugabyteDB seems to be higher than that of CockroachDB.

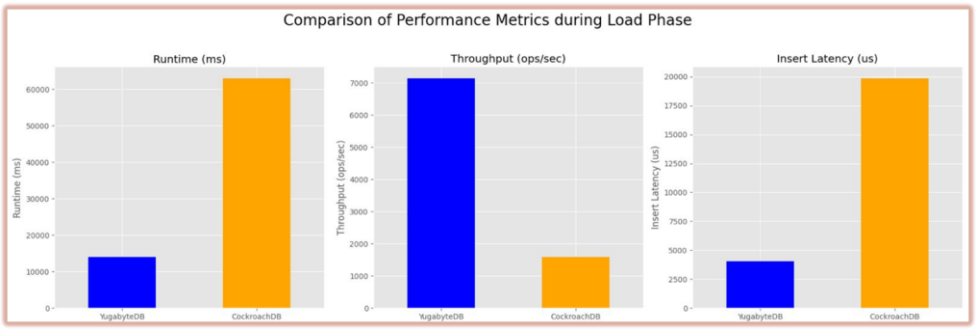


Fig 7.2: Comparison of Metrics during Load Phase

Fig 7.3: Comparison of Metrics during Run Phase

7.5. Performance Test of YugabyteDB and CockroachDB in OLAP Queries

Considering the E-commerce platform, we performed load test and performance test for Yugabyte and CockroachDB in OLAP Queries. For this, we generated data for products, customers and sales tables, each containing 10,000, 30,000 and 100,000 records respectively.

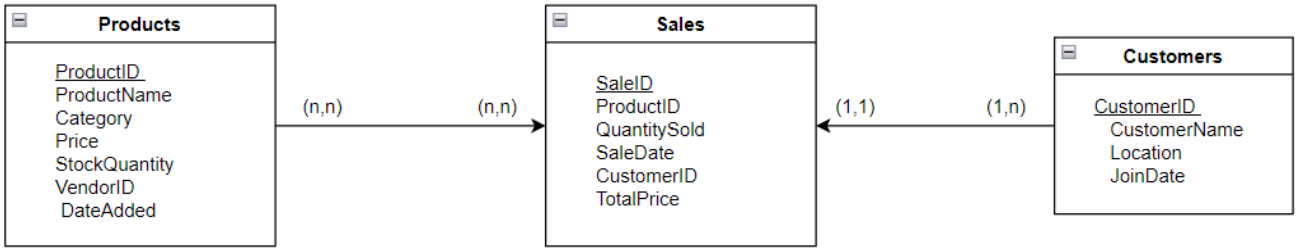


Fig 7.4: Conceptual Schema for e-commerce platform

The bar graph below shows the time taken by two databases to load above tables.

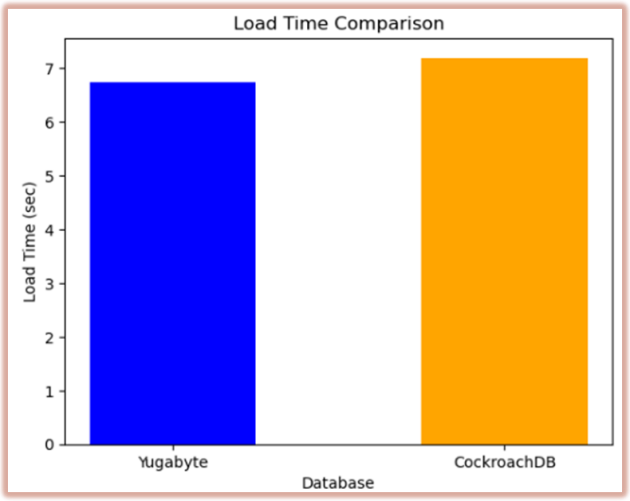


Fig 7.5: Bar Graph showing Load Time

Once the load test was completed, we prepared a few OLAP queries, which answered business questions such as

1. Monthly Sales Trend for a Specific Product
2. Total Sales by Category for a Given Year

3. Top 10 Customers by Sales Volume
4. Stock Analysis - Products Close to Stock Out
5. Sales Performance by Region

The bar graph below shows the time taken by two databases to execute these OLAP queries.

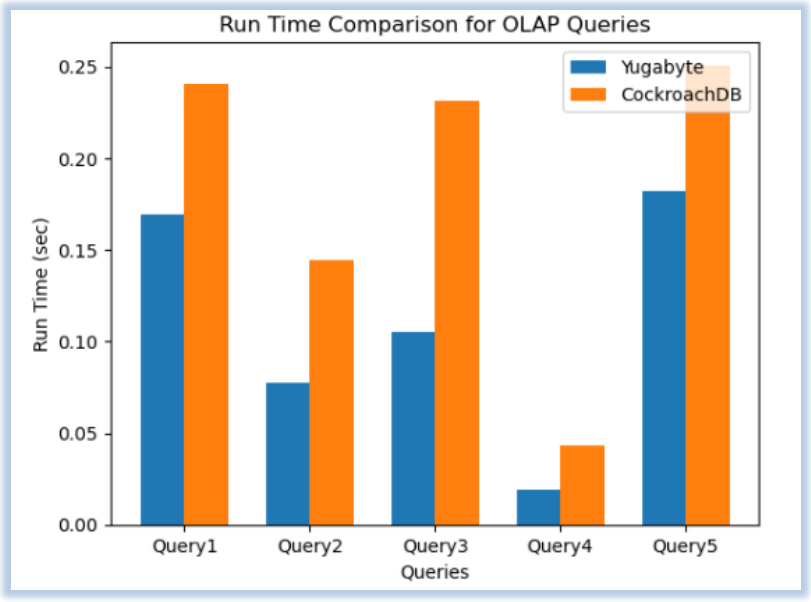


Fig 7.6: Bar Graph showing OLAP Queries execution time

Overall, in the above e-commerce scenario, Yugabyte took the least time to load data into the database and execute different OLAP Queries.

8. Conclusion

After conducting a thorough benchmarking analysis using a laptop with specification Apple M1, 16 GB, macOS Sonoma and YCSB (Yahoo! Cloud Serving Benchmark) benchmarking tool, our findings indicate that YugabyteDB demonstrates superior performance across various metrics. Specifically, YugabyteDB outperformed CockroachDB in terms of load time, overall throughput, as well as insert and update latency.

However, it is noteworthy that CockroachDB exhibited commendable performance in the realm of read latency, surpassing the other databases in this specific aspect. This nuanced observation highlights the nuanced strengths of each database platform under different workload scenarios.

In summary, our benchmarking results showcase YugabyteDB's robust capabilities in handling workloads with efficiency, emphasizing its strengths in load handling, throughput, and write-related latency. Meanwhile, CockroachDB excelled in providing low read latencies, showcasing its proficiency in read-intensive scenarios. These insights can be valuable for organizations tailoring their database choices to specific performance priorities and usage patterns.

9. References

Chant, B. (n.d.). *The Ultimate YCSB Benchmark Guide (2021)*. From https://benchant.com/blog/ycsb

Choudhury, S. (2020, 08 13). *Introducing yugabyted, the Simplest Way to Get Started with YugabyteDB.* From https://www.yugabyte.com/blog/introducing-yugabyted-the-simplest-way-to-get-started-with-yugabytedb/

Guy Harrison, J. S. (n.d.). *CockroachDB: The Definitive Guide.* OReilly.

Labs, C. (n.d.). *Deploy a Local Cluster from Binary (Secure)*. From https://www.cockroachlabs.com/docs/stable/secure-a-cluster

M. Tamer Özsu, P. V. (n.d.). *Principles of Distributed Database Systems.* Springer.

Mihalcea, V. (2023, 3 24). *YugabyteDB Architecture*. From https://vladmihalcea.com/yugabytedb-architecture/

Wikipedia. (n.d.). *CockroachDB*. From https://en.wikipedia.org/wiki/CockroachDB

Wikipedia. (n.d.). *YCSB*. From https://en.wikipedia.org/wiki/YCSB

Wikipedia. (n.d.). *YugabyteDB*. From https://en.wikipedia.org/wiki/YugabyteDB

Yugabyte. (n.d.). *YCSB*. From https://docs.yugabyte.com/preview/benchmark/ycsb-jdbc/