UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

UNIVERSITE LIBRE DE BRUXELLES

BDMA

2017-19

COCKROACH DB

NewSQL Distributed, Cloud Native Database

Kashif Rabbani 000456582
Ivan Putera MASLI 000456536

December 20, 2017

Contents

1 Introduction

Nowadays world is moving towards cloud infrastructure. Research has shown that by 2019 the majority of IT infrastructure spend will be cloud based, as shown in the figure 1. Existing technologies are now shaping themselves to adopt the growing needs of Silicon Valley. Particularly 'data' which is considered as a backbone of every industry is now demanding quick and easy access with its huge geographically growing size. Existing databases particularly monolithic systems require a lot of engineering and configuration to run smoothly on clouds, especially when it comes to the distributed databases. Businesses are being evolved worldwide and so is the data. As businesses are evolving, their needs are also changing. In this air of clouds every business requires a scalable, survivable, consistent and highly available solution for its data and applications regardless of its location. Here comes a need of the database which contains these features in the native environment. Providing such solution is a trivial task. A NewSQL database 'CockroachDB' built on native SQL engine claims to provide all these features in the distributed environment.

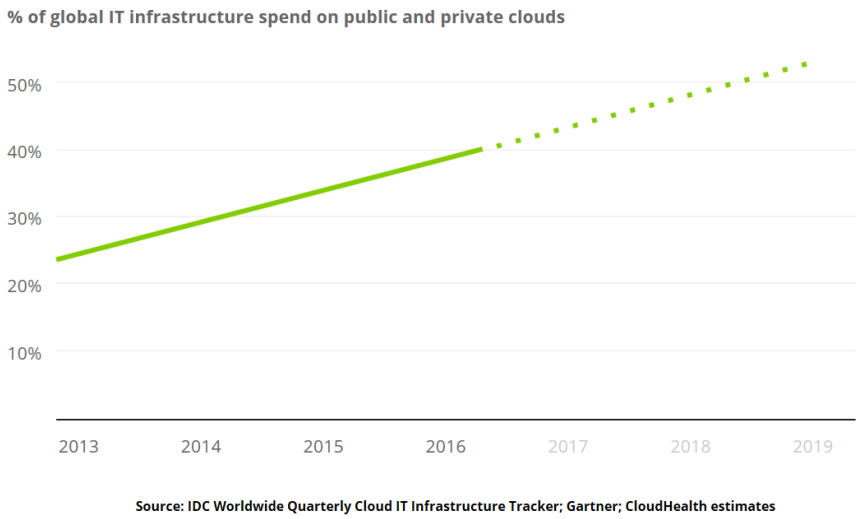

Figure 1: By 2019 the majority of IT infrastructure spend will be cloud spend.

This report contains all the detail required to understand CockroachDB to get started with your application from development to testing phase with a short explanation of NewSQL. Moreover, we have also benchmarked cockroachDB on different parameters against PostgreSQL for our application.

2 NewSQL

NewSQL is a class of modern relational database management systems that seek to provide the same scalable performance of NoSQL systems for online transaction processing (OLTP) read-write workloads while still maintaining the ACID guarantees of a traditional database system. (1)

2.1 From SQL to NewSQL

We should actually say it like: Journey from SQL to NoSQL, and NoSQL to NewSQL, because the term NewSQL was not known before NoSQL. NoSQL was introduced to fill the gaps present in SQL databases. With the team being, it get popularity in the database world. However, No database has been remained perfect at any time since the beginning of the databases. So it would not be wrong

to make an analogy that these terms are just wheels of data, carrying the data in a new form and a different method with the passage of time. These wheels get transformed according to the industrial and business needs plus the requirements and behaviour of the people controlling the world of data. We all know SQL, most of us know NoSQL too, but the main dilemma is when to choose which tool for our needs? Because not everything is perfect when you think about future and want to take your past along in the race of future. NoSQL did very well to fill that gaps but again here comes a choosing of choice. So tech guys moved back and started to fill the gaps by not loosing the traditional Standard Query Languages and ACID transactions. This resulted in NewSQL which we would say an update of SQL NoSQL having best aspects of both, merged as one.

2.2 What NewSQL is?

Other than the definition of NewSQL, here we would describe it as a rule of thumb. If you favor availability and require some specific data models, NoSQL would come in your mind. Similarly if you need transparent ACID transactions with primary, secondary indexes and industrial standards, SQL will come into your mind. But if you need both of the above set of requirements, NewSQL should click here! Precisely, get a speed of NoSQL at-scale with stronger consistency and standard powerful query language.

2.3 Characteristics of NewSQL

- NewSQL provide full ACID transactional support.
- NewSQL minimizes application complexity.
- NewSQL increases consistency.
- NewSQL doesn't require new tool or language to learn.
- NewSQL provides clustering like NoSQL but in a traditional environment.

3 CockroachDB

CockroachDB is a distributed SQL database build on transactional and strongly-consistent key-value store. It can scale horizontally, survive disk, rack, machine and data center failures without manual intervention with disruption of minimum latency. It supports strongly consistent ACID transactions and provides a familiar SQL API to perform any function on the data. (2) More precisely, CockroachDB is a SQL database for building global cloud services.

3.1 Brief History

Cockroach Labs was founded in 2015 by Spencer Kimball, Ben Darnell and Peter Mattis (ex-Google employees). Kimball and Mattis were key members of the Google File System team while Darner was a key member of Google Reader Team. They had used BigTable at Google and were acquainted with Spanner. After leaving Google, they wanted to design and build something similar for companies outside of Google. They started working on CockroachDB software on June 2015.(3)

Spencer Kimball wrote the first iteration of the design in January 2014 and to allow outside access and contribution, he decided to start the project as an open-source and released it on GitHub in February 2014. Luckily, it got attracted by community and may experienced developers contributed in its development. It also earned the honor of Open Source Rookie of the Year because of its collaborations on GitHub.(4)

Currently CockroachDB is a production-ready having 1.1.3 as its latest release version. Many companies of various sizes are using their global services with CockroachDB e.g. Baidu, kindred futures and heroic Labs.

3.2 Brief Descriptions

CockroachDB is a main software of CockroachLabs. Its main goal is to make data easy, that is to grow up to any scale, survive disasters, be available everywhere and build apps not workarounds. It is built using a Google whitepaper on Spanner as an open-source database. The database is scalable in such a way that while building an app, a single instance can be created from a laptop which can be scaled to thousands of commodity servers as business grows. It is designed to run in the cloud and be resilient to failures.(3)

CockroachDB follows a new standard of availability in which groups of symmetric nodes are used intelligently to agree on write requests. Once consensus is made, writes are available for read operation from any node in the cluster. Read/Write traffic is sent to any node because of their symmetric behaviour serving as client gateways. Load get balanced dynamically towards healthy nodes, no need to use any error-prone failover setups. That's how consistent replication with greater data redundancy and availability is achieved across the nodes.(5)

CockroachDB reduces the operational overhead with self-organizing nodes that support built-in scaling, failover, replication and repair. Enterprise-grade infrastructure is maintained with zero downtime rolling upgrades. Cluster's health and performance can be managed through the admin UI. Moreover, It can run on any cloud or on-premise(5). Figure 2 below demonstrate the difference between internal working of traditional RDBMS, NoSQL and CockroachDB.

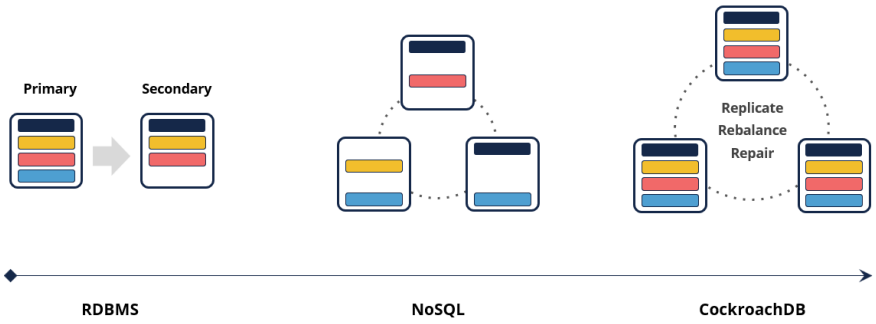

Figure 2: Difference between RDBMS, NoSQL and CockroachDB

CockroachDB is only suitable for global, scalable cloud services. It's not suitable for Low-latency use cases, unstructured data and Non-SQL analytics.

3.3 Architecture

CockroachDB has a layered architecture, it was designed to create an open-source database which is scalable and consistent at the same time. As developers often have questions about internal working of the process, this section will answer all the questions regarding architecture of CockroachDB and how we achieved it. Although not everything is covered here in details. If you want to read more in depth about internal working of every component, you should visit cockroachlabs documentation website.

Goals of CockroachDB

CockroachDB was designed to achieve the following goals:

- .
- Offer consistency at industrial level regardless of deployment's size. This means hassle free configuration along with distributed transactions.

- Provide a database which is always available, and permit reads/writes on every node without any conflicts.
- Remove dependency with deployment server. Should be able to deploy on any server.
- Support familiar tools for working with relational data (i.e., SQL). Keeping in view all the above goals, CockroachDB lets you build a global, scalable, and resilient cloud services.

Glossary
It is necessary to understand few terms before reading the architecture.

Term	Definition
Cluster	CockroachDB deployment, which acts as a single logical application that contains one or more databases.
Node	An individual machine running CockroachDB. Many nodes join to form a cluster.
Range	A set of sorted, contiguous data of cluster.
Replicas	Copies of ranges sorted on at least 3 nodes just to ensure and achieve survivable behaviour.

Concepts
In order to understand what CockroachDB architecture achieves, it is necessary to understand few concepts on which CockroachDB heavily relies.

Term	Definition
Consistency	CockroachDB uses "consistency" in the sense of ACID semantics and the CAP theorem, although less formally than either definition. In simple, your data will be anomaly-free.
Consensus	When a range receives a request to write, a quorum of nodes containing the replicas of the range acknowledge the write. In simple we can say that your data is safely stored on most of the nodes in consistent current state of the DB, even if some of the nodes are not online. When a write doesn't achieve consensus, forward progress stops to maintain consistency within the cluster.
Replication	Replication is creating and distributing copies of data, as well as ensuring the consistency of copies, there are two types of replication, synchronous and asynchronous. CockroachDB uses Synchronous replication. This type of replication requires all writes to propagate to a quorum of copies of the data before being committed. While asynchronous replication only requires a single node to receive the write and commit it, then propagate to each copy of the data after the fact. This type of replication was announced by NoSQL databases but this method often cause anomalies and data loss.
Transactions	A set of operations that satisfy the requirements of ACID semantics.

3.3.1 Overview

CockroachDB can start running on machines with just two commands:

- cockroach start command with a –join flag for all the initial nodes in the cluster (to let the process know about other nodes it can communicate)
- cockroach init command to initialize the cluster for one time.

As cockroach process starts running, developers can now interact with CockroachDB through SQL API, which is modelled after PostgreSQL. As all the nodes of cluster follow the symmetrical

behaviour, that's why user can send request to any of the node. This enables CockroachDB to integrate with load balancers easily.

CockroachDB nodes converts SQL RPCs into operations that work with distributed key-value store. It algorithmically starts data distribution across nodes by dividing the data into 64MiB chunks (these chunks are known as ranges). Each range get replicated synchronously to at least 3 nodes (this ensure survivability of DB). This way of handling the read/write request enables CockroachDB to make it highly available.

A node cannot serve any request directly, it finds the node that can handle it and communicates with it. So, user don't need to know about locality of data, CockroachDB is smart enough to track it for users and enable symmetric behaviour for each node.

How CockroachDB ensure isolation to provide consistent reads, regardless of which node user is communicating with, is handled by consensus algorithm. Finally, the data is written to or read from the disk using an efficient storage engine which keeps track of data's timestamp. User can find historical data for a specific period of time using SQL standard 'AS OF SYSTEM TIME' clause.

This high-level overview explains what CockroachDB does. In order to have much more understanding of its architecture, you need to look at how the cockroach process operates at each layer.

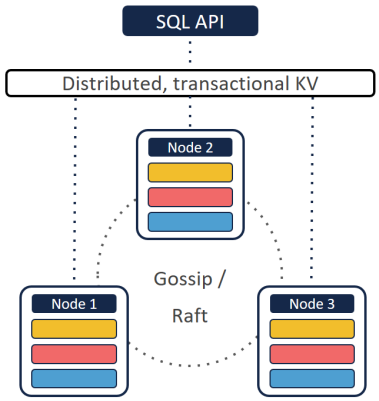

Figure 3: Architectural Overview (5)

Layers

At the highest level, CockroachDB accomplishes conversion of clients' SQL statements into key-value (KV) data, which get distributed among nodes and written to disk by its architectural process which is manifested in different layers communicating with each other as a service.
Following is the explanation of functions which are performed by each layer in the given order, independently by treating other layers as a black-box APIs.

Layer	Order	Purpose
SQL	1	Translate client SQL queries to KV operations.
Transactional	2	Allow atomic changes to multiple KV entries.
Distribution	3	Present replicated KV ranges as a single entity.
Replication	4	Consistently and synchronously replicate KV ranges across many nodes. This layer also enables consistent reads via leases.
Storage	5	Write and read KV data on disk.

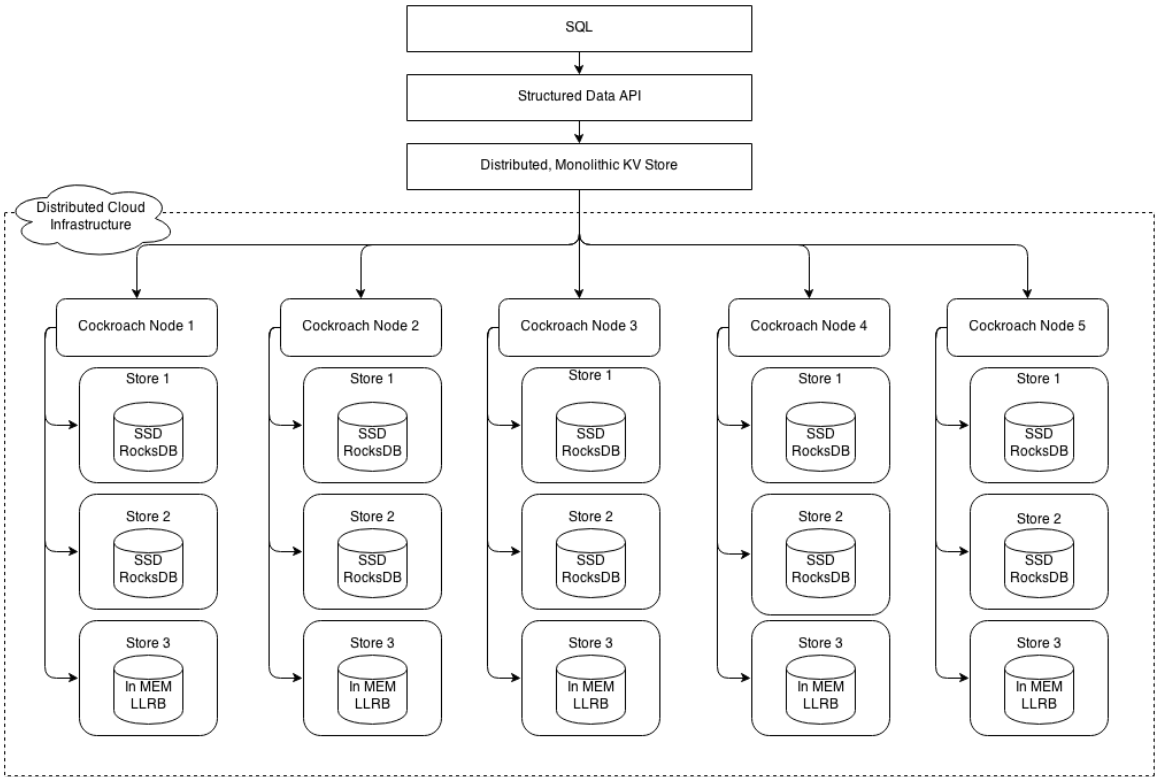

Figure 4: Layered Architecture Diagram (5)

3.3.2 SQL Layer

The SQL Layer of CockroachDB's architecture exposes its SQL API to developers. After deployment, just a connection string to the cluster is required to start querying the data using SQL statements. As all nodes of the CockroachDB are symmetric, developers can send request to any node. It's the job of CockroachDB to handle load balancers, which is being done in very efficient manner.

"Whichever node receives the request acts as the "gateway node," as other layers process the request".

Any request to the cluster arrives as SQL statement, but data is ultimately written to and read from the storage layer as key-value (KV) pairs. To handle this, the SQL layer converts SQL statements into a plan of KV operations, which it passes along to the Transaction Layer.

3.3.3 Transaction Layer

The Layer of CockroachDB's architecture implements support for ACID transactions by managing concurrent operations. Primarily, CockroachDB believes in consistency of the database as a most important feature. It is not possible to build reliable tools for businesses without providing consistency, and businesses can suffer from potentially subtle and hard time to detect anomalies otherwise.

CockroachDB implements full support for ACID transaction semantics in the Transaction Layer. However, it's important to know that all statements are handled as transactions, including single statements. For code samples of using transactions in CockroachDB, you can see documentation on transactions.

The Transaction Layer receives KV operations from planNodes executed in the SQL Layer. The TxnCoordSender sends its KV requests to DistSender in the Distribution Layer.

3.3.4 Distribution Layer

This layer provides a unified view of cluster's data. CockroachDB stores cluster's data in a monolithic sorted map of key-value pairs to make all data in a cluster to be accessible from any node. This keyspace is divided into 'ranges' i.e. contiguous chunks of the keyspace to make every key accessible from a single range. It also describes data and its location in the cluster. CockroachDB implements a sorted map of key-value paris to enable:

- **Simple lookups**: Because we identify which nodes are responsible for certain portions of the data, queries are able to quickly locate where to find the required data.
- **Efficient Scans**: By defining the order of data, it's easy to find data within a particular range during a scan.

The Distribution Layer's DistSender receives BatchRequests from its own node's TxnCoordSender, housed in the Transaction Layer.
The Distribution Layer routes BatchRequests to nodes containing ranges of data, which is ultimately routed to the Raft group leader or Leaseholder, which are handled in the Replication Layer.

3.3.5 Replication Layer

The Replication Layer of CockroachDB's architecture copies data between nodes and ensures consistency between these copies by implementing consensus algorithm.

As CockroachDB is highly available and distributed across nodes, it provides consistent service to your application even if some node goes offline. How CockroachDB achieves this is by replicating data between nodes to ensure that it remains accessible from any node. Ensuring such consistency when nodes go offline is a trivial task and many databases have failed in achieving this. CockroachDB solves this problem by using consensus algorithm to require that a quorum of replicas agrees on any changes to a range before COMMIT. CockroachDB high availability which is also known as Multi-Active-Availability requires 3 nodes in a cluster because 3 is the smallest number which can achieve quorum (i.e. 2 out of 3).

The number of failures that can be tolerated are calculated by following formula

$$\frac{Replication factor - 1}{2}$$

For example if the cluster contains three nodes, it can afford failure of one node and if it contains 5 nodes, it can afford failures of two nodes and so on.

CockroachDB automatically realizes nodes have stopped responding because of failure, and works to redistribute data to maintain high availability along with survivable behaviour. This process also works the other way around: when new nodes join your cluster, data automatically rebalances onto it, ensuring your load is evenly distributed.

CockroachDB automatically realizes/detect when failure happens and starts redistributing data to continue maximizing survivability. Similarly, when a new node joins the cluster, data get automatically rebalanced to ensure even distribution of data load.

In relationship to other layers in CockroachDB, the Replication Layer:

- Receives requests from and sends responses to the Distribution Layer.
- Writes accepted requests to the Storage Layer.

3.3.6 Storage Layer

Each CockroachDB node contains at least one store, specified when the node starts, which is where the cockroach process reads and writes its data on disk.

Data is stored as key-value pairs on disk using RocksDB, and it is treated primarily as a black-box API. Internally, each store contains three instances of RocksDB:

- One for the Raft log
- One for storing temporary Distributed SQL data
- One for all other data on the node

In addition, there is also a block cache shared amongst all stores in a node. These stores in turn have a collection of range replicas. More than one replica for a range will never be placed on the same store or even the same node.

In relationship to other layers in CockroachDB, the Storage Layer serves successful reads and writes from the Replication Layer.

3.4 Core Features

3.4.1 Simplified Deployment

An expensive prospect of deployment and maintenance of databases has always been a nightmare for developers, but this is not the case with CockroachDB because of its simplicity design goal. CockroachDB avoids external dependencies and considered as self-contained database. It also avoids traditional role structure like masters, primaries, slaves or secondaries, instead every node is symmetric with equal importance without any single point of failure.

- Without external dependencies.
- Self-organizable by the use of gossip network.
- Dead-simple configuration without "knobs".
- Well suited for container-based deployments because of symmetric nodes.
- Access to centralized admin console from every node.

As shown in the figure 5 below.

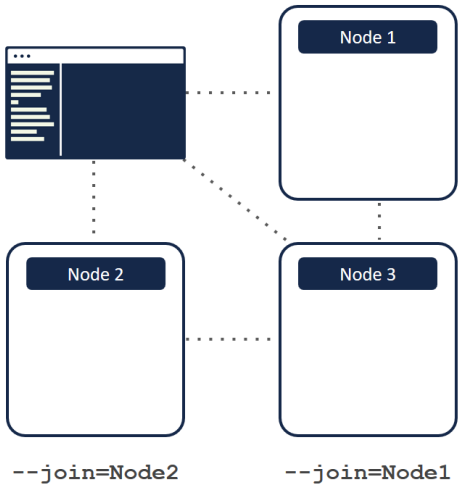

Figure 5: Simple Deployment with Symmetric Nodes (5)

3.4.2 Strong Consistency

CockroachDB guarantees consistency among replicas using replication layer of its architecture. Key properties in consistency are:

- Serializable SQL transaction.
- No downtime for server restarts, machine failures, or data center outages.
- Server restart, machine failures supported in zero downtime.
- Replication locally or in wide-area.
- No stale reads when failure occurs.
- Use of Raft Consensus algorithm.

How does this work?

- Multi-version concurrency control manages the versions of stored data, it allows for reads to limit their scope to the data visible at the start of transaction.
- Raft Consensus Algorithm is used to service the writes. This algorithm guarantees that any majority of replicas agree or disagree on COMMIT status of update/write at any node altogether.

CockroachDB uses timestamp cache to cache the last value of members read by ongoing transaction. It provides isolation in read and write transactions.

3.4.3 SQL

CockroachDB is a distributed, strongly-consistent, transactional key-value store at the lowest level, but it follows Standard SQL as external API with extensions. Therefore, it provides all the relational concepts of SQL such as schemas, tables, columns, indexes etc. No need to learn a new language, SQL developers can now structure, manipulate and query the data using well-established, time-proven tools and processes. It also supports PostgreSQL wire protocol which helps developers to connect their applications simply by plugging language specific drivers for PostgreSQL.

3.4.4 Distributed Transactions

CockroachDB supports traditional ACID semantics, i.e. All or nothing, defaults to the Serializable isolation level. Distributed transactions in CockroachDB need lower-level primitive to bootstrap atomic "commit" of transaction:

- Write to a range (i.e. a Raft consensus group)
- Transaction record keyed by the transaction ID
- Atomic commit or abort via Raft write to transaction record
- One phase commit fast path

This shows that if you have few servers at single location or many servers distributed across different datacenters, it doesn't make any difference for CockroachDB transactions. CockroachDb's transactions are distributed across the cluster without you knowing the precise location of your data. CockroachDB transactions have zero downtime and has no additional latency when rebalancing is going on. You can move the entire database or few relations from your data to other clouds or data centers even if the cluster is under load.Precisely, just talk to the one node and get whatever you want.

3.4.5 Automated Scaling and Repair

CockroachDB scales horizontally. It has minimum operator overhead. It can be run on a local computer, a single server, a corporate development cluster, or a private or public cloud. It's very easy to add/increase capacity now. Just add pointer to a new node and it belongs to your cluster now. CockroachDB starts off with an empty single range at key-value level. As data is added in this single range, it touches the threshold of 64MB (by default). Data get splits into two ranges after reaching the threshold. Each new range covers a contiguous segment of the entire key-value space. This shape into an indefinite process as data flows in. Each range always keep its size small and consistent by continue splitting. When new nodes are added in the Cockroach cluster, ranges get divided and automatically rebalanced across the nodes having more capacity. CockroachDB uses a peer-to-peer gossip protocol to communicate opportunities for rebalancing. This protocol provides exchange of information between nodes such as storage capacity, network address or other information.

- Add resources to scale horizontally, with zero hassle and no downtime
- Self-organizes, self-heals, and automatically re balances
- Migrate data seamlessly between clouds

Scaling out of 3 nodes to 5 nodes has been represented in the figures 6, 7 and 8 respectively.

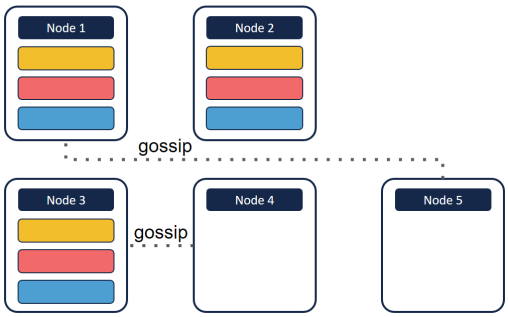

Figure 6: Adding additional nodes to scale, New nodes connect / communicate via gossip (5)

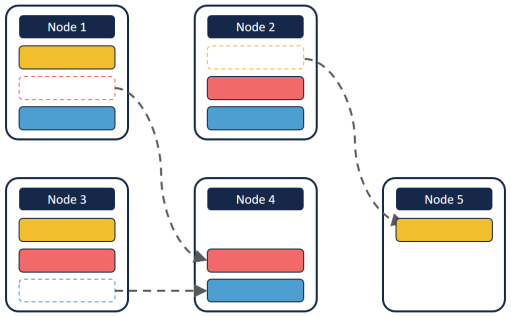

Figure 7: Nodes self-organize via rebalancing (5)

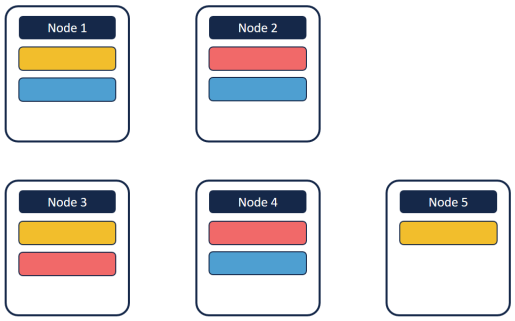

Figure 8: Old replicas are garbage collected

3.4.6 High Availability

CockroachDB survives software and hardware failures. This is accomplished by consistent replication and automated repair feature of CockroachDB.
We have already explained how CockroachDB replicates in the architecture section. Here we will consider a use case between three nodes to elaborate Consensus Replication clearly.

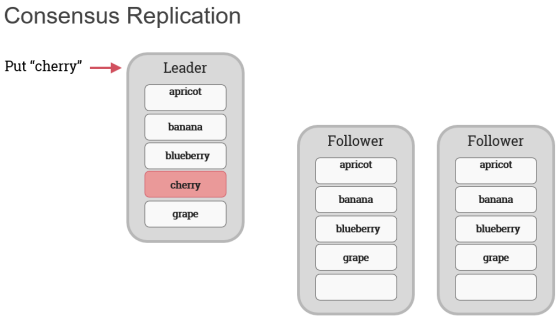

Figure 9: 3 Cockroach Nodes Cluster - leader node receives a write request

Raft is used for consensus replication. CockroachDB uses one raft group per range not for the whole database. It has lot of rafts. Let's say these are cockroach nodes having a copy of database shown in figure 9 . We want to put cherry in the database. Most of the request goes to the leader. Leader gets to put cherry command first (figure 9). Then it tells the follower to put cherry (figure 10). When there is a quorum, that 50 percent of the nodes have written the record (figure 11), command officially happens (figure 12), and acknowledgement is received (figure 13).

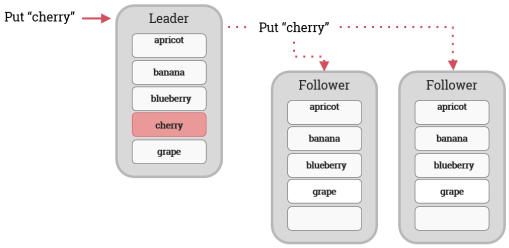

Figure 10: Leader replicate the request to the followers

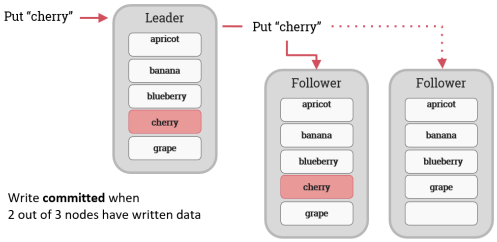

Figure 11: Write get committed.

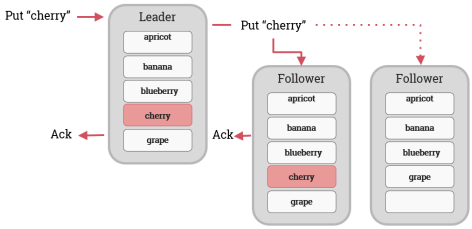

Figure 12: Follower Acknowledge the Write

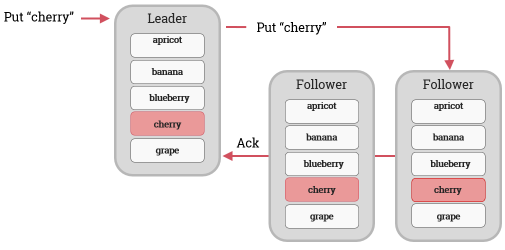

Figure 13: Follower Acknowledge the Write

What happens during failure? If one of the follower dies. Let's consider the same case, we want to put cherry again in the database as shown in the figure 14 . Cherry goes into the leader and then the leader dies somehow (figure 15). Rest of the followers hold an election and they elect a new leader (figure 16). So, the new leader doesn't have cherry, but old follower does have cherry. And for some reason it comes backup, your computer restarts, your disk is backup and the follower re-join the cluster (figure 16). So, the reader gets a request to read cherry, but the new leader does not have a cherry and it will return Key-not-found as shown in the figure 17 and eventually the follower with cherry gets cherry clean-up.
So these consensus provides us "atomic" writes (But only for each range).

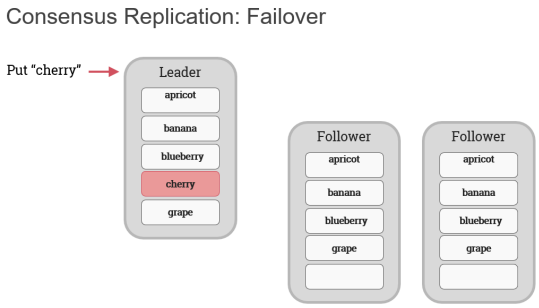

Figure 14: 3 Cockroach Nodes Cluster - leader node receives a write request

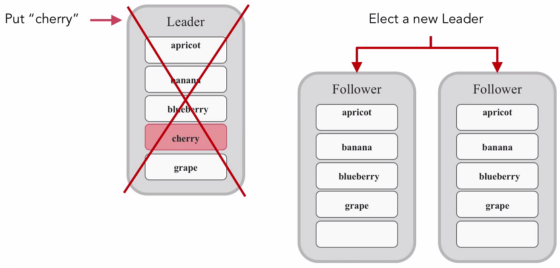

Figure 15: Leader goes down! Followers held an election

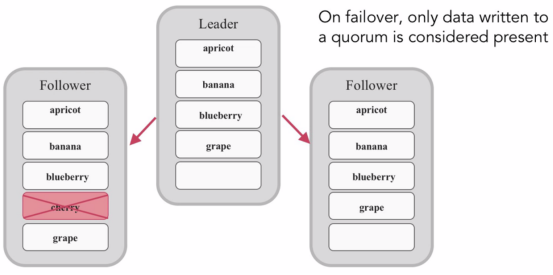

Figure 16: Failover State

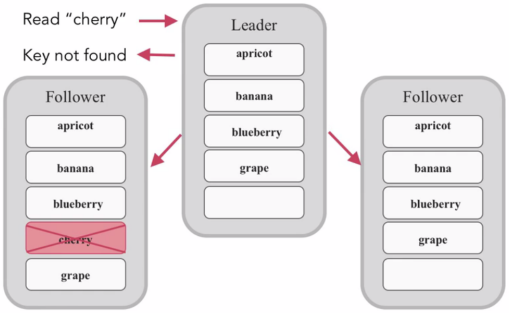

Figure 17: Leader node comes back as a follower

3.4.7 Open Source

CockroachDB doesn't require any complex licensing or agreements to start working on your local laptop, public or private cloud like other traditional DBMS require e.g. ORACLE etc. It is an open source project and will always remain. You can be a part of Cockroach Community as a developer or a user. You can also contribute to the design and implementation if you really love the databases.

- Keep your options open and avoid vendor lock-in
- Easy experimentation and enhancement
- Bigger and more active community for support and troubleshooting
- Debug problems through your entire stack

3.5 Installation

This tutorial covers creating 3 nodes on separate Virtual Machines. It is meant to run as a multi-node cluster. Already connected to 3 VM. One connection in each tab. First, we need to get CockroachDB on each machine by downloading the binary and then moving into the directory path, as shown in the figure 18 below.

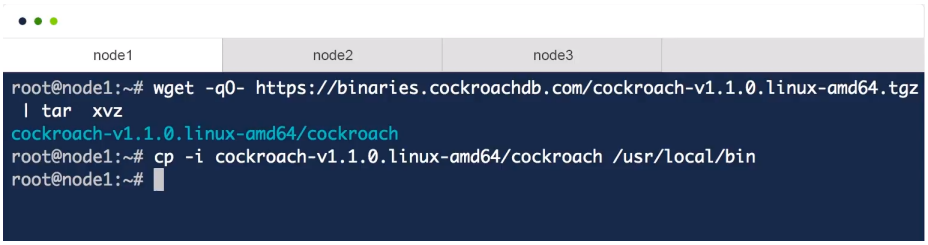

Figure 18: CockroachDB Installation

With the binary in place, we prepare nodes to start by issuing a Cockroach Start Command with a Join Flag which identifies each nodes host, as shown in the figure 19. For the rest of docs checkout the documentation. Similarly repeat this process for remaining two nodes of the cluster. Note that you can have as many as nodes you want in your cluster. We have chosen 3 nodes just for the tutorial purpose.

Once CockroachDB has been setup on all the nodes. We will now perform one-time cluster initialization from any node by running the cockroach init command, figure 20.

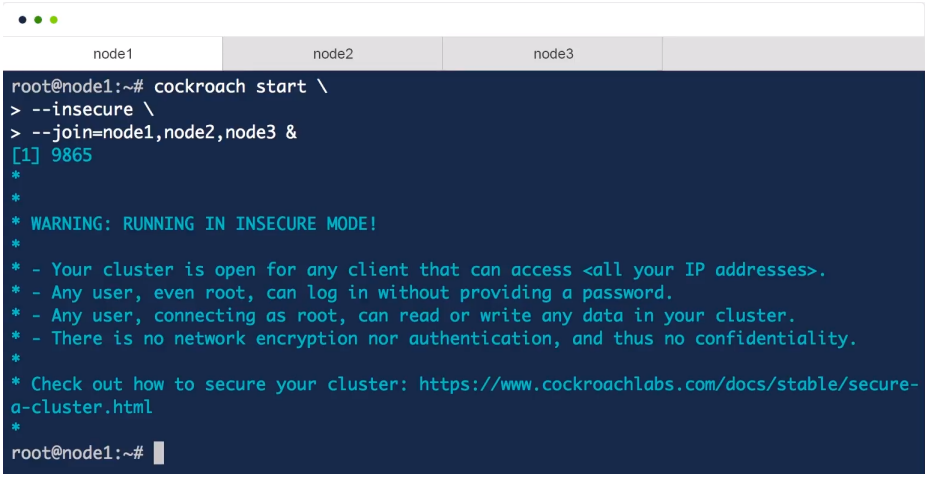

Figure 19: CockroachDB Start Command

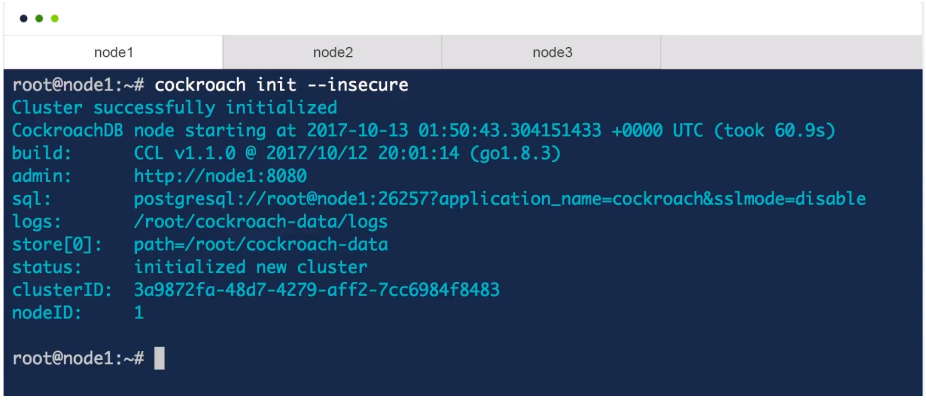

Figure 20: CockroachDB Initialization

With that we are up and running, and can now connect our application with the cluster. For this demo we are going to use a CockroachDB specific load generator to simulate high traffic we service. Because CDB supports PostgreSQL wire protocol for driver and ORM support. You connect app to it just like you would with any PostgreSQL database but with different port, as shown in the figure 21

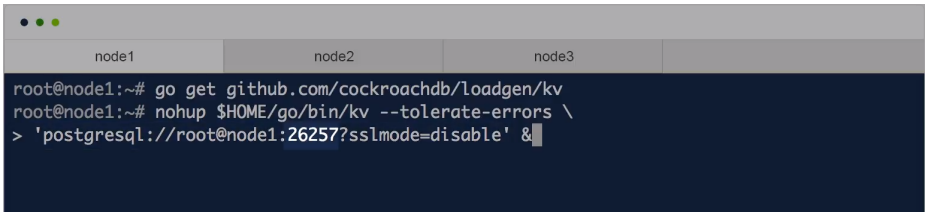

Figure 21: CockroachDB setting up load generator

Now that the load generator started we can view its activity in the CockroachDB Admin UI, figure 22. Which we can access on every node using browser. Here we can see the Cluster overall performance Query per second and the Status nodes in your cluster, as shown in the figures 23, 24, 25.

Figure 22: Launching Admin UI

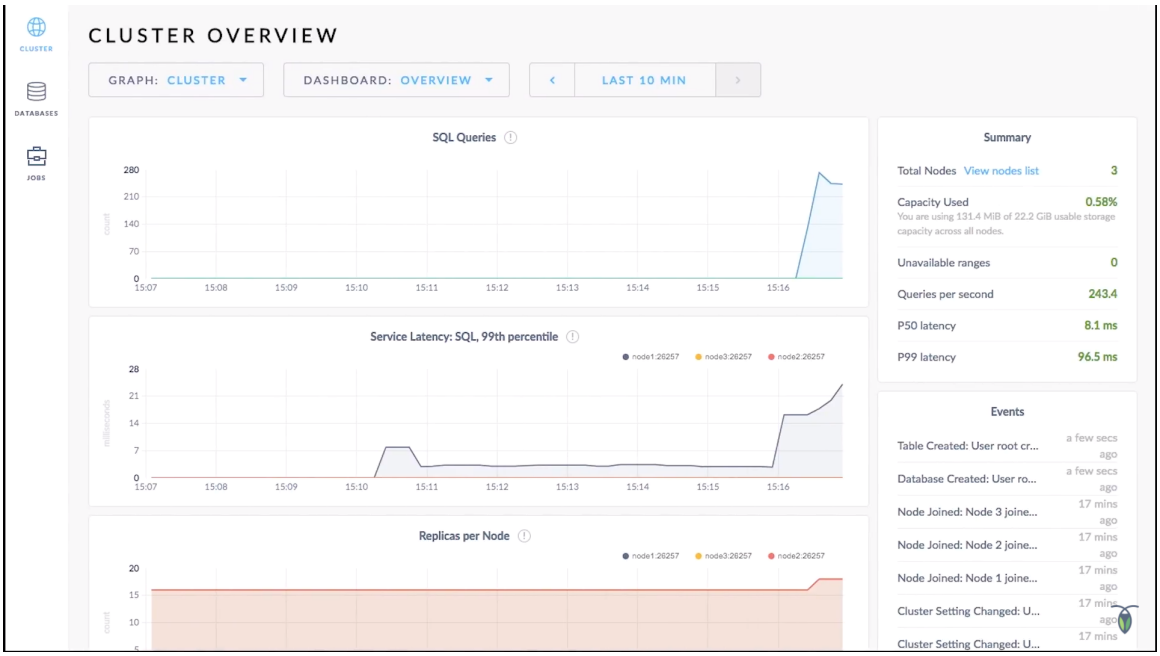

Figure 23: Cluster Overview - Admin UI

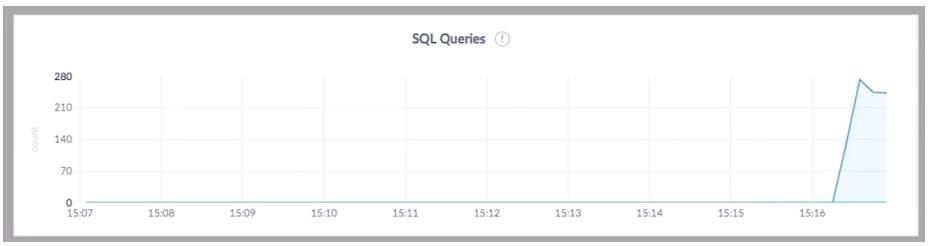

Figure 24: Query Performance

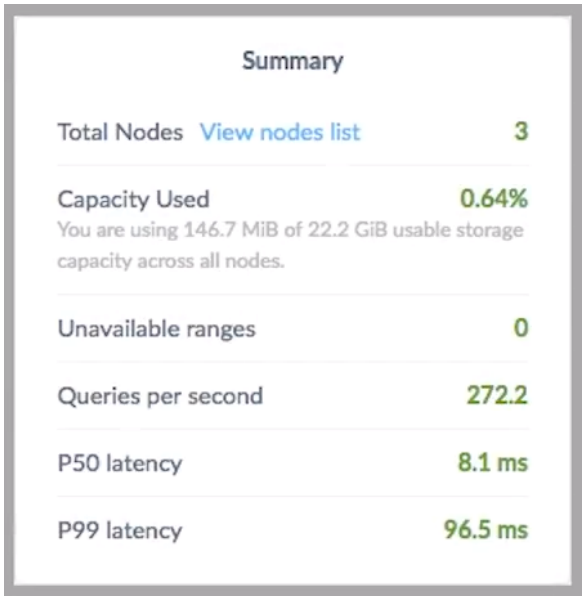

Figure 25: Summary Panel - Viewing nodes status!

What will happen in case if a node goes down?

- Traditional RDBMS goes down.
- NoSQL risks inconsistency
- CockroachDB offers high availability & consistency.

Going to the 3rd node and abruptly killing the cockroach process by command as shown in the figure 26. Back in the Admin UI we can see in the summary panel registers that one of the node is going down, figure 28. However, our application is continuously serving traffic, this means our cluster was able to survive an unexpected machine failure. Without expecting any downtime upon failure, as shown in the figure 27.

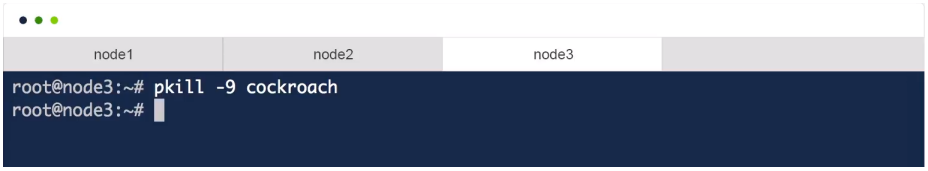

Figure 26: Killing a Cockroach Process at Node 3

Figure 27: Application is still serving traffic continuously.

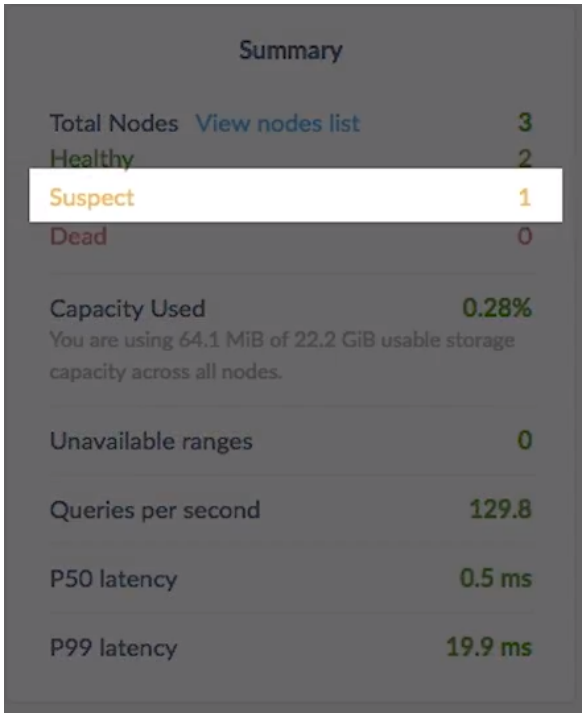

Figure 28: Inspecting Summary Panel - Node down Suspected!

4 Building an Application with CockroachDB

We have build a DVD-Rental Database application for our project. This application has quite rich schema which really suits well to the current cloud businesses. This DVD Rental database represents business process of a DVD rental store. It contains 15 tables, its ERD diagram is shown in the figure 29. The schemas are adapted from a sample PostgreSQL database designed by postgresqltutorial.com (6),

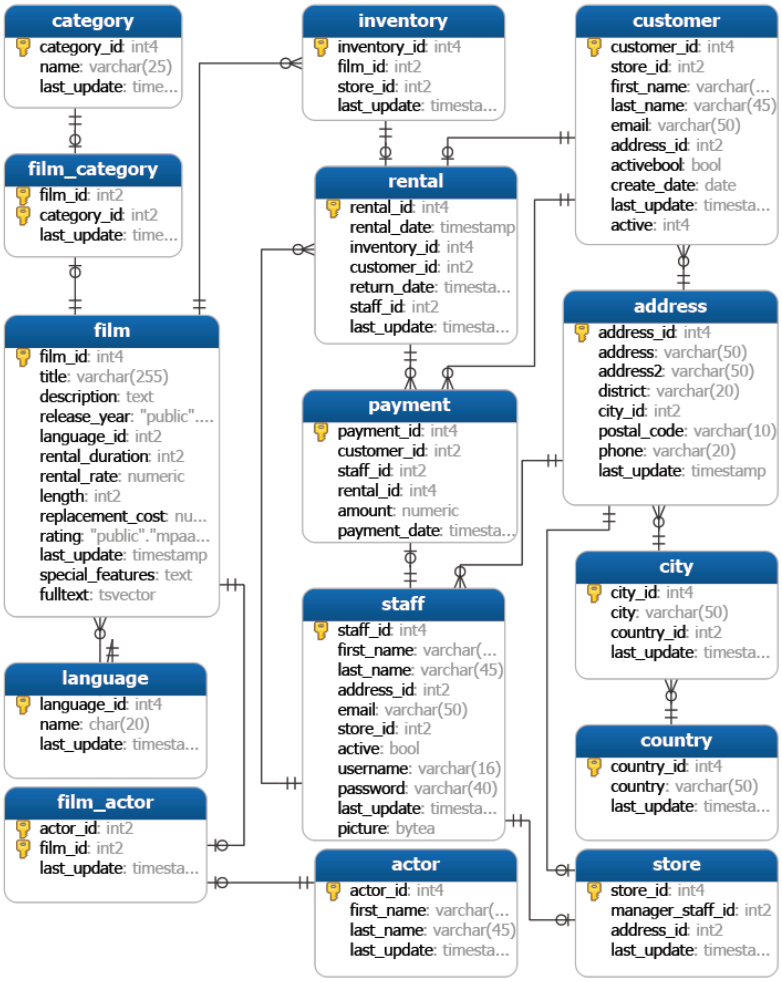

Figure 29: ERD of DVD Rental Database

4.1 Creating a Database

CREATE DATABASE DVDRental

```
CREATE TABLE actor (
        actor_id INT NOT NULL
        ,first_name STRING(150) NOT NULL
        ,last_name STRING(150) NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (actor_id ASC)
        ,FAMILY "primary" (
                actor_id
                ,first_name
                ,last_name
                ,last_update
                )
        );

CREATE TABLE address (
        address_id INT NOT NULL
```

```
        ,address STRING(150) NOT NULL
        ,address2 STRING(150) NULL
        ,district STRING(20) NOT NULL
        ,city_id INT NOT NULL
        ,postal_code STRING(10) NULL
        ,phone STRING(50) NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (address_id ASC)
        ,FAMILY "primary" (
                address_id
                ,address
                ,address2
                ,district
                ,city_id
                ,postal_code
                ,phone
                ,last_update
                )
        );

CREATE TABLE category (
        category_id INT NOT NULL
        ,"name" STRING(125) NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (category_id ASC)
        ,FAMILY "primary" (
                category_id
                ,"name"
                ,last_update
                )
        );

CREATE TABLE city (
        city_id INT NOT NULL
        ,city STRING(50) NOT NULL
        ,country_id INT NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (city_id ASC)
        ,FAMILY "primary" (
                city_id
                ,city
                ,country_id
                ,last_update
                )
        );

CREATE TABLE country (
        country_id INT NOT NULL
        ,country STRING(150) NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (country_id ASC)
        ,FAMILY "primary" (
                country_id
                ,country
```

```
                ,last_update
                )
        );

CREATE TABLE customer (
        customer_id INT NOT NULL
        ,store_id INT NOT NULL
        ,first_name STRING(45) NOT NULL
        ,last_name STRING(45) NOT NULL
        ,email STRING(100) NULL
        ,address_id INT NOT NULL
        ,create_date DATE NULL
        ,last_update TIMESTAMP NULL DEFAULT now()
        ,active INT NULL
        ,CONSTRAINT "primary" PRIMARY KEY (customer_id ASC)
        ,CONSTRAINT customer_fk_address FOREIGN KEY (address_id) REFERENCES address(ad
        ,INDEX customer_address_id_idx(address_id ASC)
        ,FAMILY "primary" (
                customer_id
                ,store_id
                ,first_name
                ,last_name
                ,email
                ,address_id
                ,create_date
                ,last_update
                ,active
                )
        );

CREATE TABLE LANGUAGE (
        language_id INT NOT NULL
        ,"name" STRING(80) NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (language_id ASC)
        ,FAMILY "primary" (
                language_id
                ,"name"
                ,last_update
                )
        );

CREATE TABLE film (
        film_id INT NOT NULL
        ,title STRING NOT NULL
        ,description STRING NULL
        ,release_year INT NULL
        ,language_id INT NOT NULL
        ,rental_duration INT
        ,rental_rate INT NULL
        ,length INT NULL
        ,replacement_cost INT NULL
        ,rating STRING NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
```

```
        ,FULLTEXT STRING NULL
        ,CONSTRAINT "primary" PRIMARY KEY (film_id ASC)
        ,CONSTRAINT film_fk_language FOREIGN KEY (language_id) REFERENCES LANGUAGE (la
        ,INDEX film_language_id_idx(language_id ASC)
        ,FAMILY "primary" (
                film_id
                ,title
                ,description
                ,release_year
                ,language_id
                ,rental_duration
                ,rental_rate
                ,length
                ,replacement_cost
                ,rating
                ,last_update
                ,FULLTEXT
                )
        );

CREATE TABLE film_actor (
        actor_id INT NOT NULL
        ,film_id INT NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (
                actor_id ASC
                ,film_id ASC
                )
        ,CONSTRAINT fk_film_id_ref_film FOREIGN KEY (film_id) REFERENCES film(film_id)
        ,INDEX film_actor_auto_index_fk_film_id_ref_film(film_id ASC)
        ,CONSTRAINT fk_actor_id_ref_actor FOREIGN KEY (actor_id) REFERENCES actor(acto
        ,FAMILY "primary" (
                actor_id
                ,film_id
                ,last_update
                )
        );

CREATE TABLE film_category (
        film_id INT NOT NULL
        ,category_id INT NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (
                film_id ASC
                ,category_id ASC
                )
        ,CONSTRAINT fk_category_id_ref_category FOREIGN KEY (category_id) REFERENCES c
        ,INDEX film_category_auto_index_fk_category_id_ref_category(category_id ASC)
        ,CONSTRAINT fk_film_id_ref_film FOREIGN KEY (film_id) REFERENCES film(film_id)
        ,FAMILY "primary" (
                film_id
                ,category_id
                ,last_update
                )
```

```
        );

CREATE TABLE store (
        store_id INT NOT NULL
        ,manager_staff_id INT NOT NULL
        ,address_id INT NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (store_id ASC)
        ,CONSTRAINT fk_address_id_ref_address FOREIGN KEY (address_id) REFERENCES addr
        ,INDEX store_auto_index_fk_address_id_ref_address(address_id ASC)
        ,FAMILY "primary" (
                store_id
                ,manager_staff_id
                ,address_id
                ,last_update
                )
        );

CREATE TABLE inventory (
        inventory_id INT NOT NULL
        ,film_id INT NOT NULL
        ,store_id INT NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (inventory_id ASC)
        ,CONSTRAINT fk_film_id_ref_film FOREIGN KEY (film_id) REFERENCES film(film_id)
        ,INDEX inventory_auto_index_fk_film_id_ref_film(film_id ASC)
        ,CONSTRAINT fk_store_id_ref_store FOREIGN KEY (store_id) REFERENCES store(stor
        ,INDEX inventory_auto_index_fk_store_id_ref_store(store_id ASC)
        ,FAMILY "primary" (
                inventory_id
                ,film_id
                ,store_id
                ,last_update
                )
        );

CREATE TABLE staff (
        staff_id INT NOT NULL
        ,first_name STRING(45) NOT NULL
        ,last_name STRING(45) NOT NULL
        ,address_id INT NOT NULL
        ,email STRING(100) NULL
        ,store_id INT NOT NULL
        ,username STRING(50) NOT NULL
        ,password STRING(40) NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (staff_id ASC)
        ,CONSTRAINT fk_address_id_ref_address FOREIGN KEY (address_id) REFERENCES addr
        ,INDEX staff_auto_index_fk_address_id_ref_address(address_id ASC)
        ,CONSTRAINT fk_store_id_ref_store FOREIGN KEY (store_id) REFERENCES store(stor
        ,INDEX staff_auto_index_fk_store_id_ref_store(store_id ASC)
        ,FAMILY "primary" (
                staff_id
                ,first_name
```

```
                ,last_name
                ,address_id
                ,email
                ,store_id
                ,username
                ,password
                ,last_update
                )
        );

CREATE TABLE rental (
        rental_id INT NOT NULL
        ,rental_date TIMESTAMP NOT NULL
        ,inventory_id INT NOT NULL
        ,customer_id INT NOT NULL
        ,return_date TIMESTAMP NULL
        ,staff_id INT NOT NULL
        ,last_update TIMESTAMP NOT NULL DEFAULT now()
        ,CONSTRAINT "primary" PRIMARY KEY (rental_id ASC)
        ,CONSTRAINT fk_inventory_id_ref_inventory FOREIGN KEY (inventory_id) REFERENCE
        ,INDEX rental_auto_index_fk_inventory_id_ref_inventory(inventory_id ASC)
        ,CONSTRAINT fk_customer_id_ref_customer FOREIGN KEY (customer_id) REFERENCES c
        ,INDEX rental_auto_index_fk_customer_id_ref_customer(customer_id ASC)
        ,CONSTRAINT fk_staff_id_ref_staff FOREIGN KEY (staff_id) REFERENCES staff(staf
        ,INDEX rental_auto_index_fk_staff_id_ref_staff(staff_id ASC)
        ,FAMILY "primary" (
                rental_id
                ,rental_date
                ,inventory_id
                ,customer_id
                ,return_date
                ,staff_id
                ,last_update
                )
        );

CREATE TABLE payment (
        payment_id INT NOT NULL
        ,customer_id INT NOT NULL
        ,staff_id INT NOT NULL
        ,rental_id INT NOT NULL
        ,amount INT NOT NULL
        ,payment_date TIMESTAMP NOT NULL
        ,CONSTRAINT "primary" PRIMARY KEY (payment_id ASC)
        ,CONSTRAINT fk_customer_id_ref_customer FOREIGN KEY (customer_id) REFERENCES c
        ,INDEX payment_auto_index_fk_customer_id_ref_customer(customer_id ASC)
        ,CONSTRAINT fk_staff_id_ref_staff FOREIGN KEY (staff_id) REFERENCES staff(staf
        ,INDEX payment_auto_index_fk_staff_id_ref_staff(staff_id ASC)
        ,CONSTRAINT fk_rental_id_ref_rental FOREIGN KEY (rental_id) REFERENCES rental(
        ,INDEX payment_auto_index_fk_rental_id_ref_rental(rental_id ASC)
        ,FAMILY "primary" (
                payment_id
                ,customer_id
                ,staff_id
```

```
                ,rental_id
                ,amount
                ,payment_date
                )
        );

-------------------- For foreign key from City to Country --------------------
CREATE INDEX ON city (country_id);

ALTER TABLE city ADD CONSTRAINT city_fk_country FOREIGN KEY (country_id) REFERENCES co

-------------------- For foreign key from Address to City --------------------
CREATE INDEX ON address (city_id);

ALTER TABLE address ADD CONSTRAINT city_fk_address FOREIGN KEY (city_id) REFERENCES ci

-------------------- For foreign key from Store to Staff --------------------
CREATE INDEX ON "store" (manager_staff_id);

ALTER TABLE store ADD CONSTRAINT store_fk_manStaffID FOREIGN KEY (manager_staff_id) RE
```

4.2 Querying the database

As CockroachDB uses Standard SQL query interface, therefore it has the same queries as like relational SQL databases have e.g. SELECT * FROM TABLE.

```
SELECT payment_id
FROM payment p
        ,staff s
WHERE s.staff_id = ${id}
        AND p.staff_id = s.staff_id
        AND p.staff_id = ${id};
```

4.3 Connecting with applications

As CockroachDB is build over PostgreSQL wire protocol, therefore you can connect your application with any language using compatible PostgreSQL driver.

CockroachDB supports JAVA, Python, Ruby, NodeJS, C++, PHP, C Sharp etc. For more information read the documentation.

4.4 Managing the application

As explained in the tutorial section, you can manage your cockroach Admin UI, Admin UI has many feature available to see the statistics of ongoing nodes health and CRUD operations on the databases.

5 Comparing CockroachDB with PostgreSQL

In this section we will describe our comparisons between CockroachDB and PostgreSQL. Specifically, we will do comparison by its features and performances, the latter is done via benchmarking. Two benchmarking will be used YCSB benchmark and a holistic benchmark (Insert and Select Join) done on DVDRental Database. We choose PostgreSQL as a comparison because both of them are using

same wire protocl, which implies that CockroachDB is more identical to PostgreSQL than any other RDBMS.

5.1 Setting up the environment

We will use Amazon Web Service EC2 cloud computing unit for this benchmarking purposes. Specifically, we use m4.xlarge. The detailed specification of m4.xlarge machine is shown in Table 2.

vCPU	4
Memory	16 GiB

Table 2: EC2 m4.xlarge specification

Each machine will run only one node of CockroachDB. Therefore, for testing three nodes of CocokroachDB as an example, three EC2 instances will be used with same type.

For PostgreSQL we will also use services provided by Amazon Web Service RDS. Specifically we will use db.m3.large. The specification is shown in Table 3. We will use PostgreSQL version 9.6.5.

vCPU	2
Memory	7.5 GiB

Table 3: RDS db.m3.large specification

Even though those two machines has different specifications. We decide to continue using them since those two machines has same cost (around 0,3 EUR per hour). Remark also that PostgreSQL RDS is tuned up and maintained by Amazon Engineer directly so that we expect that it will have better performance compared to self-installed PostgreSQL. For having multiple nodes of PostgreSQL, Master-Slave Read replica scheme will be used.

In order to benchmark the Database clusters properly load balancers will be used. Since CockroachDB is running on EC2 instances, Elastic Load Balancer provided also by Amazon Web Services can be used. However, Elastic Load Balancer can not be used for load balancing RDS databases, therefore we will use HAProxy on top of an EC2 instance. The configuration of the HAProxy is

```
1  global
2    maxconn 4096
3
4  defaults
5      mode tcp
6      timeout connect 10s
7      timeout client 1m
8      timeout server 1m
9
10 listen psql
11     bind :5432
12     mode tcp
13     balance roundrobin
14     server pgsql1 <node1 address>:5432
15     server pgsql2 <node2 address>:5432
16     server pgsql3 <node3 address>:5432
```

5.2 Setup CockroachDB

These are the steps to install CockroachDB to the EC2 machine.

1. Download and extract CockroachDB using this command

```
wget -qO- https://binaries.cockroachdb.com/cockroach-v1.1.3.linux-amd64.tgz | tar xvz
```

2. Copy the CockroachDB to the /usr/local/bin folder

```
cp -i cockroach-v1.1.3.linux-amd64/cockroach /usr/local/bin
```

5.3 Feature comparisons

While PostgreSQL and CockroachDB is identical we found key differences. These differences are significant when multiple database nodes are used

1. **Single Point of Failure** PostgreSQL multiple nodes approach is master-slave. That implies that while read can be done on all nodes, it is possible to only write to the master node. The availability of PostgreSQL clusters then can be jeopardized when the only master node is down. In contrast, all nodes in CockroachDB are identical, therefore both read and write can be done on any nodes. This means that CockroachDB clusters do not have a Single Point of Failure. Furthermore, since write operations can be distributed, this will avoid a cluster bottleneck resulted from massive write operations.

2. **Extra Middleware** While existing load balancer or HAProxy can be used to distributed the read and write queries to each nodes in CockroachDB. That is not enough for PostgreSQL clusters. Read queries can be distributed to all nodes, but write queries must exclusively directed to the master node only. Either developers must program their application to use different Database addresses (HAProxy/Load Balancer address when reading and master node when writing) or extra middleware that helped the routing must be used.

3. **Multiple Geography writing performance** Suppose that an Internet company which has bustling customers in Europe, Asia and America and they use PostgreSQL clusters. Their master node is located on Europe, while the slave nodes are located on Europe, Asia and America. In that case, a user from Asia need to transmit data to Europe Database server when using their products, which is very time consuming. However, when CockroachDB clusters are used instead, user from any zone just need to transmit data to their nearest Database node since writing operation can be done on all nodes.

5.4 YCSB

YCSB (Yahoo! Cloud Serving Benchmark) is a set of tools created by Brian Cooper for Database benchmarking. YCSB supports various types of Database from different paradigm, from SQL to NoSQL(7). However, it should be known that YCSB benchmarking focus is testing the strength of Key Value operations. Consequently, no operations relatied to relation database features such as join will be tested using this benchmark suite.

5.4.1 Preparing the Database

Initially, YCSB suite will load data to the selected database before the benchmarking began. The amount of data loaded can be tuned by the tester.

The schema for the YCSB SQL schema is given below.

```
CREATE TABLE usertable (
    YCSB_KEY VARCHAR(255) PRIMARY KEY,
    FIELD0 TEXT, FIELD1 TEXT,
    FIELD2 TEXT, FIELD3 TEXT,
    FIELD4 TEXT, FIELD5 TEXT,
    FIELD6 TEXT, FIELD7 TEXT,
```

```
    FIELD8 TEXT, FIELD9 TEXT
);
```

5.4.2 Workloads

YCSB offers six different benchmarking sceanrios, defined as workload. All the workloads is done on the same data. Hence, before doing a YCSB workload, make sure that the schema is already created on your database and the data is loaded. For current benchmarking purpose, we will use three kind of workloads out of six provided. The explanation is below.

1. **Workload A** Workload A characteristic is heavy update operation, in which the amount of read and write operations proportion is equal.
2. **Workload B** Workload B is characterized by stronger emphasize on read operations. Now read operations comprised about 95% operations
3. **Workload C** This is a read operation exclusive workload.

5.4.3 Installing YCSB

1. **Download the YCSB** Visit YCSB download page at its official Github repository here. Download the latest stable version (0.12.0). There is version 0.13.0 but it is still a Release Candidate.
2. **Extract the package** In your terminal, navigate it to the location in which the YCSB is located. Afterward, extract the compressed YCSB file with this command

```
tar xzf ycsb-0.12.0.tar.gz
```

3. **Download the Postgre JDBC driver** Download the latest Postgre JDBC driver from here. Subsequently, copy the downloaded driver to

```
~<ycsb_location>/jdbc-binding/lib
```

5.4.4 Parameters

We will describe parameters used on this test

1. **Operation**, There are two kind operations, **load** and **run**. Load operation is used to populate the data to the table for a given workload. While the run operation will do the operations defined on each workload
   Example

```
./bin/ycsb load
```

2. **Database** Specify the database type that will be used for this test. For testing Cockroachdb and PostgreSQL, the type used is **jdbc**
3. **Workload** Choose the workload that will be used in the test, flagged with **-P** flag.
   Example

```
./bin/ycsb load jdbc -P workload/workloada
```

   Will load data for workload A.

Jobs	Workload	PostgreSQL	CockroachDB
Load	A	14596	149885
Run	A	10097	159355
Load	B	13859	95449
Run	B	8679	105185
Load	C	14837	121323
Run	C	8198	124202

Table 4: Time for each operations in milliseconds

4. **Driver** Specify the driver of the database, filled under **db.driver** option and **-p** flag. We will use PostgreSQL JDBC driver that is downloaded and copied before, the value of this option should be **org.postgresql.Driver**
   Example

```
./bin/ycsb load jdbc -P workload/workloada -p db.driver=org.postgresql.Driver
```

5. **Database Host, Username and Password** Specify the host, username and password. Use JDBC standard synthax for the database host.
   Example

```
./bin/ycsb load jdbc -P workload/workloada -p db.driver=org.postgresql.Driver /
-p db.url=jdbc:postgresql://cockroach:26257/ycsb /
-p db.user=root /
-p db.passwd=root /
```

6. **Record Count** Specify the number of records that will be loaded to the table during a load process.
7. **Threads** Denotes the number of threads that will be used in a given operation.
8. **Operation Counts** Specify the amounts of operations used during a run operation.
9. **Batch Size** How many records that will be batched before a certain operations is committed.

For all of the test we will use 100000 records, 100000 operations, 50 threads and batch size of 100. Here is an example of complete command for a load test

```
./bin/ycsb load jdbc -P workload/workloada -p db.driver=org.postgresql.Driver /
-p db.url=jdbc:postgresql://cockroach:26257/ycsb /
-p db.user=root -p db.passwd=root /
-p recordcount=10000 -threads 50 -p operationcount=10000 -p db.batchsize=100
```

We will test only on one node for this benchmark.

5.4.5 YCSB Results

In all operations (load and run) we found that PostgreSQL perform much better compared to CockroachSQL Table 4 and Figure 30 illustrates running time (in milliseconds) for each Job and process.

We found that the performance of CockroachSQL is lagging on this kind of benchmark. For all operations we done, CockroachDB's running time is almost 10 times that of PostgreSQL.

Interestingly, during the test we found that the performance of CockroachDB is sensitive to the number of thread used. Here we shows the running time for loading workload C to CockroachDB with different number of threads. Figure 31 illustrates the running time for loading workload C data if we vary the number of threads.

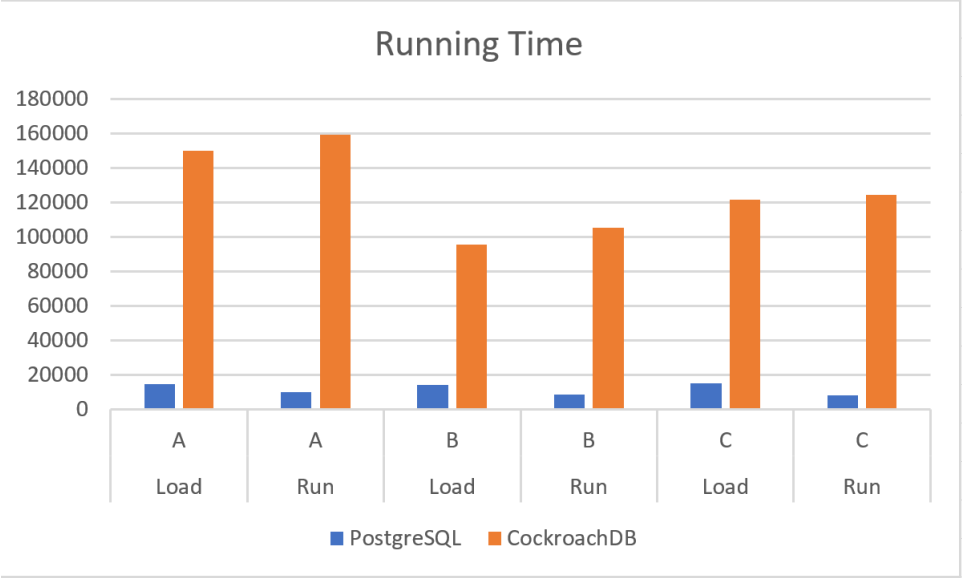

Figure 30: Running time for YCSB benchmark in milliseconds

5.5 Holistic Benchmark

In this benchmark we will test two operations, Insertion and Selection with Join. We will use DVD Rental Database mentioned above. Specifically we will focus on Payment table. This test will be done in both Single and Cluster modes of CockroachDB and PostgreSQl.

This benchmark will be done in four different Database configurations.

1. PostgreSQL 1 Node
2. PostgreSQL 3 Nodes
3. CockroachDB 1 Node
4. CockroachDB 3 Nodes

5.5.1 Insertion

The Payment table has two foreign keys, one to Staff table and one to Rental table. At maximum, we have one million payment records. In this test, we will vary the number of records inserted by 100000 records, 500000 records and 1000000 records.

As stated previously, the number of threads matters very much for the Database performance. As a result, we will do some preprocessing on our records before inserting.

1. **Divide the records** We will use *head* program to get the number of records required. For entire payment records, there are 1020000 lines. Hence to have a file with a hundred thousand records, we will take the first 102000 lines out of the entire records. Similarky to have five hundred thousand records, we take 510000 lines.

```
1        head -n 102000 payment.sql > payment_100k.sql
```

2. **Splitting the records** We will split the data to ten chunks. The *split* program from Linux will be used. The parameter of the split is the number of lines in each chunk. Using the same computation above, we just need to divide the number of lines by ten.

```
1        split -l 10200 payment_100k.sql
```

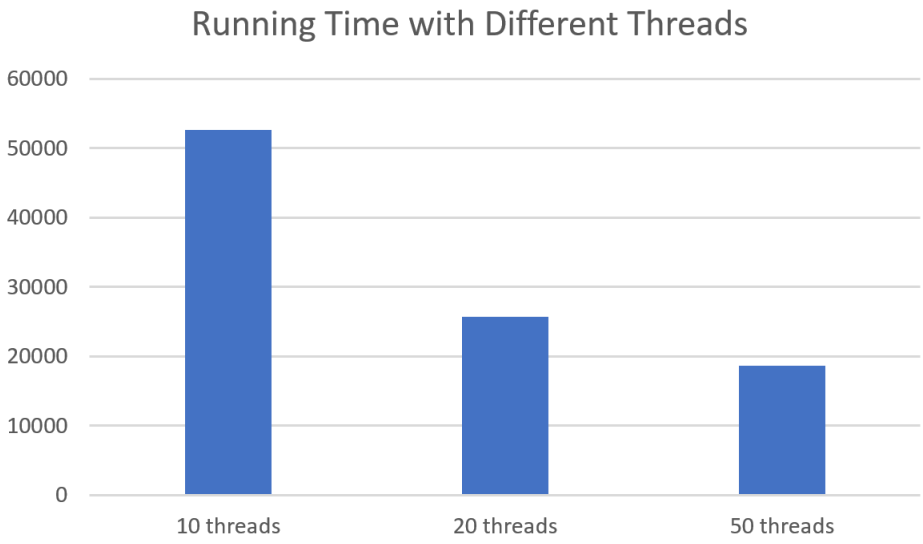

Figure 31: Running time with different thread numbers

The output of the split is ten files. Each file has a prefix of xa and the last letter varies from a-j.

The next step is to write a script for testing. We will use standard PostgreSQL client for both CockroachDB and PostgreSQL. The required parameters for psql are follows.

```
GPASSWORD=<password> psql -U <user> -h <Database Address> /
-p <Port Number> -d <Database Name>
```

Ultimately, this is the script that is used

```
#!/bin/bash
GPASSWORD=<password> psql -U <user> -h <Database Address> /
-p <Port Number> -d <Database Name> < xaa > /dev/null &
GPASSWORD=<password> psql -U <user> -h <Database Address> /
-p <Port Number> -d <Database Name> < xab > /dev/null &
.
.
.
GPASSWORD=<password> psql -U <user> -h <Database Address> /
-p <Port Number> -d <Database Name> < xaj > /dev/null &
wait
```

For CockroachDB multiple nodes benchmark, the database address should be the address of the load balancer. On the other hand, when doing PostgreSQL multiple nodes benchmark, the address should be the master node address.

We are not interested with the output of the insert so we directed them to /dev/null. Ampersand is used so that each insert job is put on the background and next job can be started without waiting the previous job to be finished. Finally, wait is used so when we calculate the time required, it will wait all the jobs to be finished first.

time program is used to calculate the running time of this operation. Before running the script make sure that the script is executable

```
chmod +x benchmark.sh
time ./benchmark.sh
```

Size	PostgreSQL 1 Node	PostgreSQL 3 Nodes	CockroachDB 3 Nodes	CockroachDB 1 Node
100000 records	5,771	6,5	31,23	25,8
500000 records	28	29,2	222,7	167
1000000 records	59	65	467	261,912

Table 5: Running time of Records Insertion in seconds

5.5.2 Insertion Results

In this test we also observe that the CockroachDB insertion running time is much higher compared to the PostgreSQL time. We found that CockroachDB is expected to take five times longer to insert same number records. Interestingly, we also found that running time for CockroachDB 3 Nodes is significantly higher than the CockroachDB single node.

We have two hypothesis which explain this phenomenon

1. **Divided Thread** As we mentioned before, CockroachDB performance is highly influenced by the number of threads used. In the benchmark, we use same number of threads for both single node and three nodes configuration. While on the single node configuration, the sole node will receive full 10 threads exclusively, the number of threads for each machine is divided in three nodes configuration. Each node is expected to only receive $\frac{10}{3} = 3$ threads.
2. **Replication overhead** CockroachDB will automatically replicated records received by a node to other nodes in a cluster. This operation will consume available computing resources.

Figure 32 and Table 5 shows the running time for each benchmark.

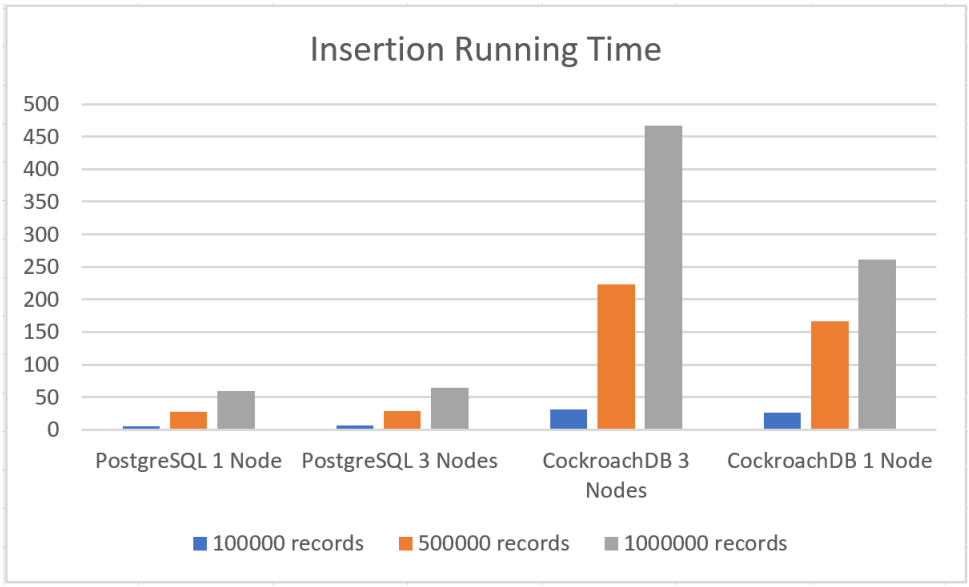

Figure 32: Running time of Records Insertion in seconds

5.5.3 Selection with Join

In this benchmark, we will use JMeter to do the benchmark. We use JMeter because of its features that we deemed essential for this benchmark namely multi-thread queries, built-in reporting and number generator are available to use.

This query will be used.

```
SELECT payment_id from payment p, staff s WHERE s.staff_id = \${id} AND
p.staff_id=s.staff_id AND p.staff_id=\${id};
```

${id} will be provided randomly by Apache JMeter.

These are JMeter configurations used

1. **Number of threads** 100 threads will be used
2. **Ramp-up period** 10 seconds
3. **Loop Count** Number of records to be benchmarked divided by number of threads. For example, if we are going to benchmark 100000 records, then $\frac{100000}{100} = 1000$ count of loops will be used.
4. **Max number of Connection** 100

Unlike the insertion benchmark, when doing multiple nodes benchmark for both CockroachDB and PostgreSQL, their Load Balancer or HAProxy address is used.

Looking at the query, we might wonder why p.staff_id equality is included if we have s.staff_id = p.staff_id already. The answer is that we found that CockroachDB query optimizer for join is still immature. For better explanation, consult these query plans provided by the DBMS. When we ommit the last p.staff_id equality, here is the Query plan. Notice that CockroachDB will do table scan spanning all records in Payment table, even though it is not necessary.

```
+-------+--------+----------+-------------------------+
| Level | Type | Field | Description |
+-------+--------+----------+-------------------------+
| 0 | render | | |
| 1 | join | | |
| 1 | | type | inner |
| 1 | | equality | (staff_id) = (staff_id) |
| 2 | scan | | |
| 2 | | table | payment@primary |
| 2 | | spans | ALL |
| 2 | scan | | |
| 2 | | table | staff@primary |
| 2 | | spans | /1-/2 |
+-------+--------+----------+-------------------------+
```

After the latter equality added, here is the query plan.

```
+-------+--------+---------------+-----------------------------------------------------+
| Level | Type | Field | Description |
+-------+--------+---------------+-----------------------------------------------------+
| 0 | render | | |
| 1 | join | | |
| 1 | | type | inner |
| 1 | | equality | (staff_id) = (staff_id) |
| 1 | | mergeJoinOrder | +"(staff_id=staff_id)" |
| 2 | scan | | |
| 2 | | table | payment@payment_auto_index_ |
| 2 | | | fk_staff_id_ref_staff |
| 2 | | spans | /1-/2 |
| 2 | scan | | |
| 2 | | table | staff@primary |
| 2 | | spans | /1-/2 |
+-------+--------+---------------+-----------------------------------------------------+
```

Notice that table scanning span is now limited. As a comparison, here is query plan for PostgreSQL with the latter equality omitted. Notice the scanning on the payment table.

Numbers	PostgreSQL 1 Node	PostgreSQL 3 Nodes	Cockroach 1 Node	Cockroach 3 Nodes
20000 times	927	311	10	16
50000 times	More than 1000	766	25	31
100000 times	More than 1000	More than 1000	52	68

Table 6: Time for Select Join operations in seconds

```
                                  QUERY PLAN
------------------------------------------------------------------------------------------
 Nested Loop (cost=0.29..9940.41 rows=10 width=4)
   -> Index Only Scan using staff_pkey on staff s
        (cost=0.29..8.31 rows=1 width=4)
         Index Cond: (staff_id = 1)
   -> Seq Scan on payment p (cost=0.00..9932.00 rows=10 width=8)
         Filter: (staff_id = 1)
(5 rows)
```

A possible shortfall for this immature Query Optimization is that extra tunings will be required for each queries with joins. In addition, a complex query with numerous and deep joins might not be feasible to be tuned manually. Furthermore, it might be impossible to manually tunes numerous amounts of query and tuning queries generated by Object Relational Mapping library might not be possible.

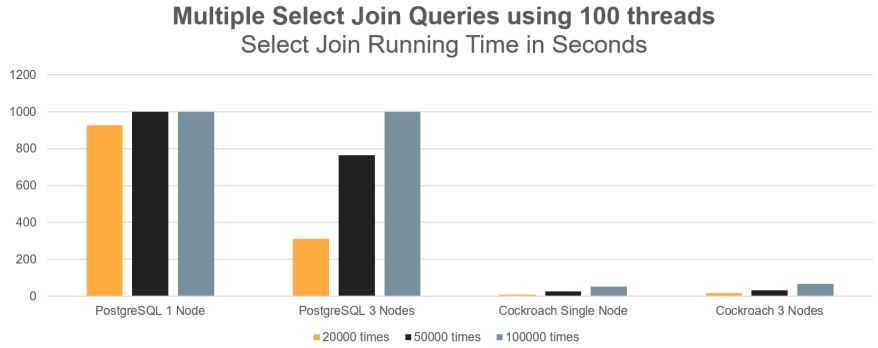

Figure 33: Running time Selection Join

5.5.4 Select Join Results

In this benchmark we see that CockroachDB performs significantly better compared to PostgreSQL by almost 100 times. In addition, PostgreSQL 3 Nodes running time is about 30% of the running time of single node PostgreSQL which shows that the read operations are equally distributed to three nodes available. For efficiency, benchmark process that is above 1000 seconds will be terminated and considered "more than 1000 seconds".

Similar to the insert benchmark we notice that running time for three nodes is arguably higher for three nodes compared to single node. In addition, to the divided thread hypothesis as a cause, we also believe that 100 threads we use as this benchmark is not enough to put a significant burden to the CockroachDB nodes. As a result, the benefit of distributed read query is not observed.

Figure 33 and Table 6 illustrated the running time for selection join benchmark.

6 Conclusion

CockroachDB database, while a new product offers many interesting advantages. From offering easy scalability while maintaining native SQL protocols to providing the advantages of RDBMS and NoSQL solutionns. In addition, it is very to install and build clusters without difficult configurations and extra middleware. We also showed a number of CockroachDB clustering solutions' advantages compared to traditional PostgreSQL master-slave replica.

However, we also notice that current implementation of CockroachDB still focussed on features and correctnes instead of performances. This reflected in CockroachDB's weak performance in YCSB benchmark , records insertion benchmark and Query Optimizer performance.

Having said that we believe that CockroachDB do have excellent prospect to become a major Database System player given enough time. Furthermore, its cloud centric deployment is aligned with current Business trends to move towards the cloud.

References

[1] A. Pavlo and M. Aslett, "What's Really New with NewSQL?" *ACM SIGMOD Record*, vol. 45, no. 2, pp. 45–55, 2016.

[2] CockroachLabs, "About cockroachdb." [Online]. Available: https://github.com/cockroachdb/cockroach

[3] J. Novet, "Peter fenton's latest investment is a database startup called cockroach," 2017. [Online]. Available: https://venturebeat.com/2015/06/04/peter-fentons-latest-investment-is-a-database-startup-called-cockroach/

[4] B. D. Software, "black duck software announces open source rookies of the year," 2015. [Online]. Available: https://www.blackducksoftware.com/about/news-events/releases/black-duck-software-announces-open-source-rookies-year

[5] CockroachLabs, "Cockroachlabs official website." [Online]. Available: https://www.cockroachlabs.com

[6] PostgreSQLTutorial.com, "Postgresql sample database." [Online]. Available: http://www.postgresqltutorial.com/postgresql-sample-database/

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," *SoCC*.

[8] S. Kimball, "Cockroachdb meetup nyc sf," 2017.