CockroachDB

Emanuel Kranjec & Felix Tröbinger

May 5, 2020

Contents

1 What is CockroachDB? 2
- 1.1 History . . . 2
- 1.2 License and Pricing . . . 2

2 Layered Architecture 4
- 2.1 SQL Layer . . . 5
  - 2.1.1 Relational Structure . . . 5
  - 2.1.2 SQL API . . . 5
- 2.2 Transaction Layer . . . 9
  - 2.2.1 Writing and reading . . . 9
  - 2.2.2 Parallel commits . . . 9
  - 2.2.3 Isolation levels . . . 10
  - 2.2.4 Transaction conflicts . . . 10
  - 2.2.5 TxnWaitQueue . . . 10
  - 2.2.6 Transaction pipelining . . . 11
- 2.3 Distribution Layer . . . 12
- 2.4 Replication Layer . . . 13
- 2.5 Storage Layer . . . 14

3 Setting up a CockroachDB Cluster with Docker 15
- 3.1 Creating a bridge network . . . 15
- 3.2 Creating Nodes . . . 15
- 3.3 SQL Command Line . . . 16
- 3.4 Web-Interface . . . 16
- 3.5 Securing the Cluster . . . 20

References 22

1 What is CockroachDB?

author: Felix Tröbinger

CockroachDB is advertised as a "cloud-native SQL database for building global, scalable cloud services that survive disasters"[1].

CockroachDB is a relational and transactional database built for consistent key-value store and horizontal scalability and geo-replication as a big feature. It is built to ensure data survives failures of any kind, be that disk, machine rack or even entire datacenters. Furthermore the database system is "strongly-consistent" and guarantees the ACID properites.[1]

CockroachDB is considered a distributed database. It used geo-replication to ensure fast data access regardless of physical location. The distribution and geo-replication is presumably also what got the software its name.

Cloud native, often heard alongside *microservices*, is a term that describes a system or a network of applications where single application, services or nodes are run inside Docker containers. Another important component in cloud-native computing is a load balancer, for example Kubernetes. CockroachDB is a good example of cloud-native computing since the Database is intended to be used in multiple locations and these multiple nodes are usually run in Docker containers.

1.1 History

CockroachDB started as an open-source project in 2014. Cockroach Labs was founded by ex-Google employees in the year 2015.[2] Spencer Kimball who is now CEO is also the original creator of the GNU Image Manipulation Program (GIMP).

1.2 License and Pricing

As stated above, CockroachDB was original an entirely open-source project and licensed under the second version of the Apache License (APL 2.0). In 2019 however CockroachDB changed its license to a version of the Business Source License (BSL). In summary, CockroachDB can still be used with as many nodes as desired except for offering a commercial cloud database system. CockroachDB Core is no longer open-source, however the source code can still be viewed on the companies GitHub page. [3]

Another interesting change in their license change is that versions of CockroachDB that are older than three years are converted to the APL. Although

older versions (version 19.1 and downwards) are unaffected by this change and still use the Apache License as before. This is illustrated in figure 1.

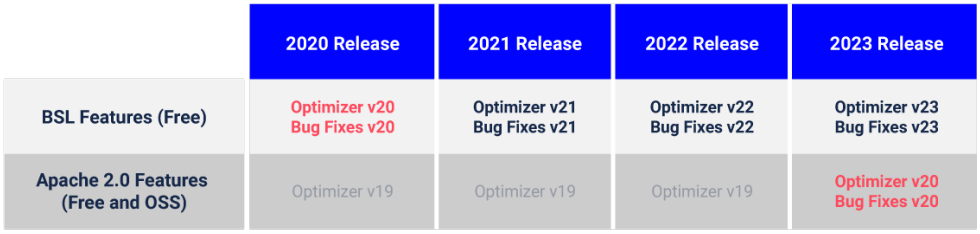


Figure 1: The new CockroachDB license model[3]

In terms of pricing, there are three available "Tiers" for CockroachDB:

- *CockroachCloud*
- *CockroachEnterprise*
- *CockroachDB Core*

CockroachCloud gives customers a fully hosted and managed database platform. The price for this service is calculated per node and hour and differs depending on whether the customer chooses AWS (Amazon Web Services) or Google Cloud as a hosting platform. The price also changes depending on the CPU power and hard drive size of the node. [4]

The mayor differences to the **CockroachDB Enterprise** tier is that this option gives the customer superior support and that it is self hosted, meaning that the customer does not rely on Cockroach Labs to store their data, but will rather host it themselves. This means that the customer has to pay another company to do that for them or take it on their own to host CockroachDB nodes in a distributed manner. As for price, Cockroach Labs invite customers to contact them. [4]

On the CockroachDB page for pricing the company states that **CockroachDB Core** is open-source,[4] yet an article covering the licensing of the product explains that this version of the database is not actually open-source according to OSI's Open Source Definition (though the source code is still viewable to anyone).[3] A more truthful description would be *source-available*. This version however is free to download and use.

2 Layered Architecture

author: Emanuel Kranjec

What CockroachDB offers is a highly modular architecture organized in layers. Each layer communicates with one another independently. In fact the layers treat each other as functional blackbox 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 2: Table of all layers in CockroachDB[7]

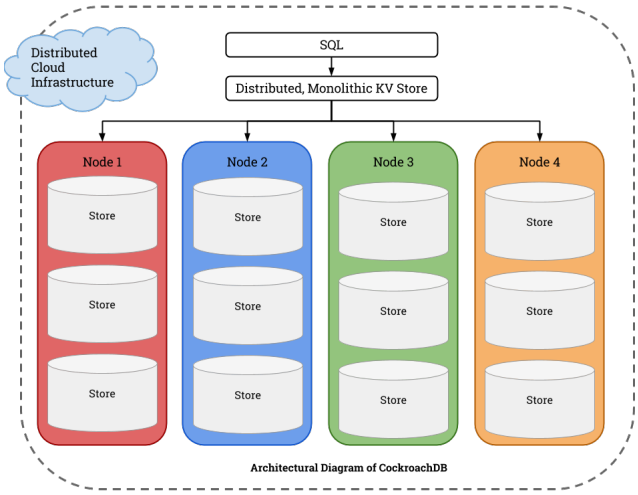


Figure 3: Diagram of CockroachDBs layered architecture[5]

Nodes are a collection of many stores. In reality they can be either physical or virtual machines.

Stores are the smallest possible physical unit. It's managed with RocksDB, an open source embedded key-value database.

2.1 SQL Layer

author: Emanuel Kranjec

To support working with relational data CockroachDB provides a SQL layer.

With the support of the SQL API developers are able to run familiar SQL queries. When running those statements the cluster gets an request, which initiates to write or read the particular data ultimately as key-value pairs into the storage layer. Therefore the layer converts beneath the surface these SQL statement into a load of key-value operations, which get processed by the next layer called transaction layer.

Developers are even capable of choosing to which node a request should be sent. The receiving node can either process the request or it acts just as so-called gateway node. This is possible because CockroachDB's nodes behave symmetrically. Thus the use of load balancers has ideal prerequisites.

2.1.1 Relational Structure

One of the headline features of CockroachDB is the relational structured data storage. The data is organized in rows, columns, tables and even multiple databases. This ensures that relational key features like constraints are supported. Application developers don't need to built data validation into the application logic because the database already guarantees consistent data.

2.1.2 SQL API

To keep CockroachDB simple to use it implements a SQL API. The API makes use of a large portion of the ANSI SQL standard, but as usual in other databases too, some SQL features are differently implemented and not all features are supported.
Besides the basic CRUD operations CockroackDB offers constraints like 'primary key', 'foreign key', 'unique', 'not null' and many more.

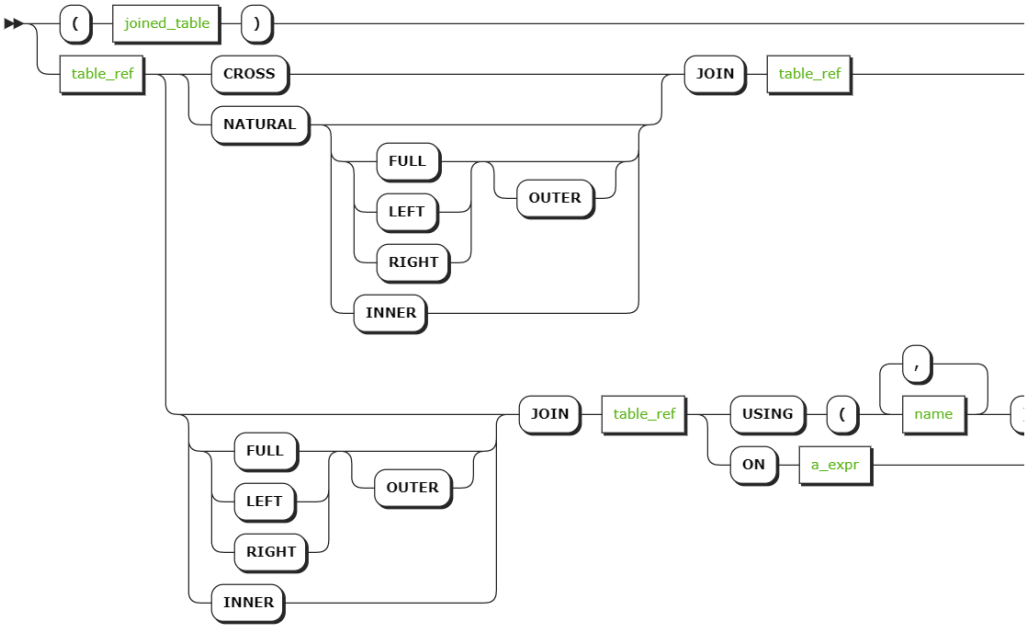
Transaction
Even transactions are available to keep the database consistent. Until now

CockroachDB has tried many approaches to guarantee ACID transactions. The previous concept was a serializable, lockless and distributed strategy. The problem with the lockless design was that it couldn't sufficiently guarantee consistency. Therefore this design was discarded. To provide serializable isolation it's now a more locking and lock-like structure. More information to this topic is in the transaction layer chapter.

Joins

As implementing efficient join algorithms is a big challenge in distributed SQL CockroachDB released since 2016 constantly new and improved implementations. CockroachDB currently supports 'merge joins', 'hash joins' and 'lookup joins' algorithms in production and additionally 'lateral joins' (a type of correlated subquery) since version 20.1 alpha. The launch of the hash join algorithm in 2017 reduced the asymptotic time complexity from previously O(N*M) to O(N+M). Later on the merge join algorithm got vectorized because the traditional sort-merge-join

> 'fails to take into account the overhead of integrating this algorithm into a system. Due to the nature of how data is stored and the interfaces used to process the data within the database, there are a few key factors that make this algorithm behave slower in practice than in theory.
>
> In a database, each row of data is encoded in a format that is optimized for a specific purpose, which may not necessarily be the same purpose across all cases. To be specific, the current state of the row by row engine stores each column of every row in something called a datum. To be able to compare this datum to another datum, there is extra overhead in each call to retrieve the value, as first the type of that value has to be determined after which the value has to be decoded. Now if this process happens for every single column of every single row, it is clear that isn't the most efficient use of processor time.
>
> Instead, it would make sense if the type check and conversion could happen once per column, since every value in the column has the same type. This is called vectorized execution, since the idea is to operate on values a column at a time.'[8]



Parameter	Description
joined_table	Another join expression.
table_ref	A table expression.
a_expr	A scalar expression to use as ON join condition.
name	A column name to use as USING join condition

Figure 4: Synopsis of joins[9]

Inner joins

'Only the rows from the left and right operand that match the condition are returned.'[9]

```
 <table expr> [ INNER ] JOIN <table expr> ON <val expr>
<table expr> [ INNER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL [ INNER ] JOIN <table expr>
<table expr> CROSS JOIN <table expr>
```

Left outer joins

'For every left row where there is no match on the right, NULL values are returned for the columns on the right.'[9]

```
 <table expr> LEFT [ OUTER ] JOIN <table expr> ON <val expr>
<table expr> LEFT [ OUTER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL LEFT [ OUTER ] JOIN <table expr>
```

Right outer joins

> 'For every right row where there is no match on the left, NULL values are returned for the columns on the left.'[9]

```
 <table expr> RIGHT [ OUTER ] JOIN <table expr> ON <val expr>
<table expr> RIGHT [ OUTER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL RIGHT [ OUTER ] JOIN <table expr>
```

Full outer joins

> 'For every row on one side of the join where there is no match on the other side, NULL values are returned for the columns on the non-matching side.'[9]

```
 <table expr> FULL [ OUTER ] JOIN <table expr> ON <val expr>
<table expr> FULL [ OUTER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL FULL [ OUTER ] JOIN <table expr>
```

The whole table of supported SQL features is here[10].

2.2 Transaction Layer

author: Emanuel Kranjec

In CockroachDB ACID transactions are fully supported in order to provide consistency. To ensure this every statement in CockroachDB is basically a transaction because by default 'autocommit mode' is enabled. Furthermore there is an own protocol to handle those transactions cross-range and cross-table. 'Parallel commit' is the name of the atomic commit protocol and in combination with transaction pipelining it reduces significantly the latency of such operations.

2.2.1 Writing and reading

To provide consistency in distributed transactions CockroachDB creates write intents and a transaction record.

Write intents provide the uncommited transaction writes to other tables/ranges like the multi-version concurrency control (MVCC) protocol and additionally a pointer to the transaction record. If there is a write intent which handles the same key there's a transaction conflict. Those write intents can be encountered by future operations while looking for MVCC values. The operation knows how to handle the write intent value by checking its transaction record. If there is no transaction record it decides based on the timestamp the validity of the write intent.

Transaction record stores the state of the transaction (PENDING, STAGING, COMMITED, ABORTED) and an array of writes from this transaction. Thus other transactions can check if this one is already commited and therefore in an consistent state.

2.2.2 Parallel commits

Depending on the state of the transaction record CockroachDB commits or restart the transaction. If the transaction record says it is in 'STAGING' state the transaction gets checked on whether it is successfully replicated across the cluster. If that is true the cleanup phase begins by changing the transaction record state to 'COMMITED', resolving the write intents to MVCC values and deleting afterwards these writes.

2.2.3 Isolation levels

Even though there are many ANSI transaction levels, CockroachDB decided to only support the 'SERIALIZABLE' level. This decision is based on a research paper from 2017, named 'ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications'[11] which shows the vulnerability of concurrent ACID transactions.
Previously the 'SNAPSHOT' level was also supported but the security aspect was valued higher to transaction throughput.

2.2.4 Transaction conflicts

CockroachDB handles a transaction conflict by the metrics priority and transaction record.

When there are for example two write intents which use the same key firstly the priority of there transaction gets compared. The one with the lower one gets in this case aborted. When the priorities don't differ the transaction which has either no transaction record or wasn't heartbeated within the transaction liveness threshold gets aborted. If there is still no decision the write intent which encountered the other one enters the TxnWaitQueue.

Heartbeat timestamps are there to evaluate if a transaction is abandoned. Active transactions are responsible for periodically updating it on its central transaction record.

2.2.5 TxnWaitQueue

As already mentioned in chapter 'Transaction Conflicts' if there is a write/write conflict the last option is to push one transaction into the TxnWaitQueue. It is a simple map which keeps track of blocking transaction IDs and their blocked companions.
txnA -¿ txn1, txn2
txnB -¿ txn3, txn4, txn5
The TxnWaitQueue gets manipulated only by the leaseholder which holds the transaction records. The queue receives a signal when a transaction got comitted or aborted. Based on this signal it let's particular transactions execute again.
Deadlocks are handled by choosing and aborting a random affected transaction.

2.2.6 Transaction pipelining

To reduce latency when making transaction with multiple writes CockroachDB is pipelining those transactions. Pipelining in this context means all of the affected ranges receive the write intents and write it to the disc in parallel when the transaction got commited.
More specifically the gateway node sends the statement to the leaseholders at first and then the leaseholders convert those statements into write intents. After spreading the write intents within the particular range and receiving confirmations from all nodes the transaction is considered as commited.

In the example below the resulting write intents are because of their UUIDs probably in different ranges. This would be an ideal case because it would be fully parallelized.

```
BEGIN;
INSERT into kv (key, value) VALUES ('apple', 'red');
INSERT into kv (key, value) VALUES ('banana', 'yellow');
INSERT into kv (key, value) VALUES ('orange', 'orange');
COMMIT;
```

2.3 Distribution Layer

author: Emanuel Kranjec

To ensure simple and efficient data lookups CockroachDB provides a monolithic sorted map of key-value pairs. Those key-value pairs include two fundamental elements. One stores the meta range and the other the table data.

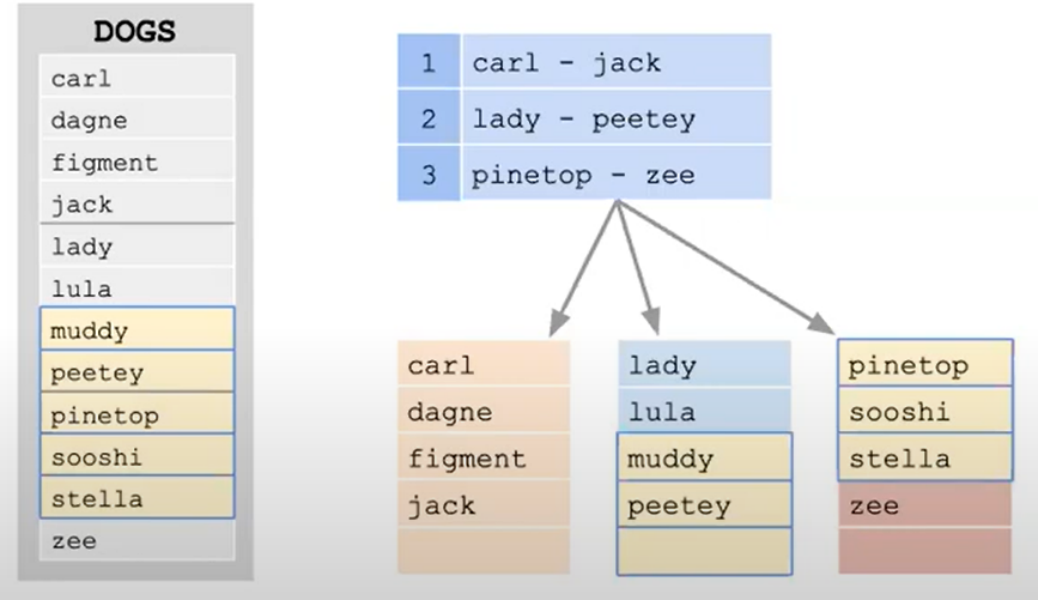


Figure 5: Simplified example of key-values for particular ranges[12]

Meta ranges describe the location of data including all of its replicas in the cluster.

Table data describes the rows of a table included in this particular range.

2.4 Replication Layer

author: Felix Tröbinger

CockroachDB consists of a cluster of nodes where each node stores data independently. CockroachDB stores its data in so called *ranges*. When a range exceeds 64 mebibyte (MiB = 1024^2 bytes) the it is split up into two ranges. Each range is then replicated (the default amount of replication is three) and each one of these replicas is stored on a different node.

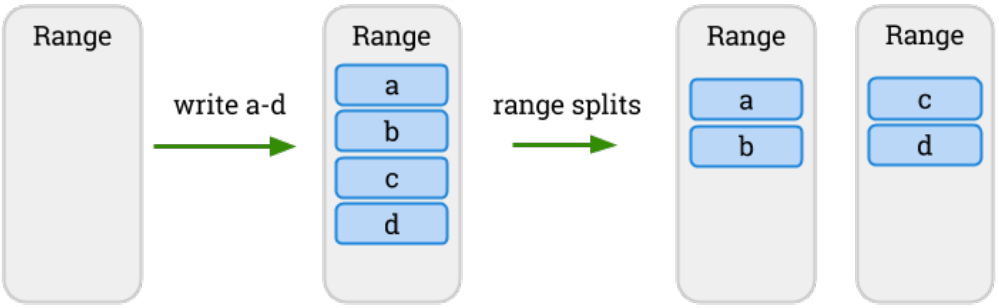


Figure 6: Range Distribution Rebalancing in CockroachDB[5]

Figure 6 visualizes the range splitting happening in CockroachDB. This happens indefinitely and nodes strive to have a relatively balanced and consistent size.

For each range there exists a *leader* who's purpose it is to coordinate read and write access. A leader along with enough followers has to agree before a write action is committed. Read access don't have to be coordinated with all followers, in this case the leader sends the result to the client without consulting it's followers. This greatly increases performance and speed. When interacting with CockroachDBs SQL API developers or users can choose to communicate with any given node of the cluster. If performing a write request on a node that cannot handle the request because it is not the leader, it finds the node who is and is able to fulfill the request. [6]

2.5 Storage Layer

author: Emanuel Kranjec

Not only the lookup table of the cluster is organized in key-value pairs also the actual data is stored as such. CockroachDB itself doesn't manage the embedded database storing process.
For this purpose they are using the high performance embedded database RocksDB. RocksDB also stores data in the form of key-values like CockroachDB does it in the lookup table. Two instaces of RocksDB are on each store. One for holding the write intents described in the transaction layer chapter and the other one to store the persistent data.

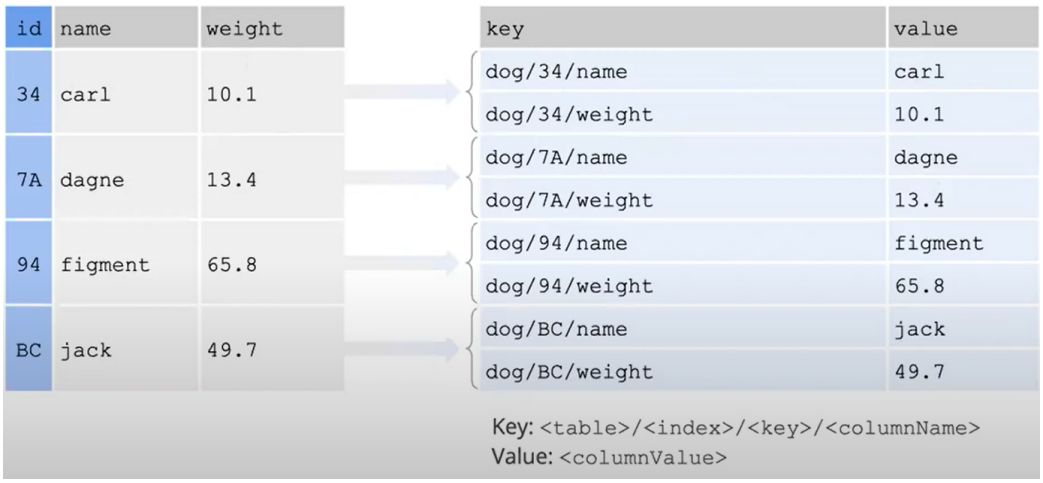


Figure 7: Conversion of SQL to a key-value store[12]

Furthermore CockroachDB uses MVCC for concurrency purposes. Therefore it is also able to time-travel as described in the SQL:2011 standard. You can get back as far as the garbage collection got.
Garbage collection occurs when a MVCC value exceeds a particular garbage collection period to free memory. The period can be set from cluster to table level individually.

3 Setting up a CockroachDB Cluster with Docker

author: Felix Tröbinger

3.1 Creating a bridge network

Even though all created nodes – or rather containers – in this example will run on the same host, they still need to be connected via a network of some sorts. As a solution this example will use a Docker *bridge network*.

A bridge network is a software bridge, which allows containers on the same host to communicate with each other. Features of docker bridge networks are that user-defined bridges provide automatic DNS resolution between containers and that containers are isolated in a better way.[13]

The following command will create a bridge network called `roachnet`. This name will be passed to each node on startup/creation.

```
docker network create -d bridge roachnet
```

3.2 Creating Nodes

This example will create a cluster of three connected nodes. The first one is created with the following command.

```
docker run -d \
    --name=roach1 \
    --hostname=roach1 \
    --net=roachnet \
    -p 26257:26257 -p 8080:8080  \
    -v "${PWD}/roach1:/cockroach/cockroach-data"  \
    cockroachdb/cockroach:latest start \
    --insecure \
    --join=roach1,roach2,roach3
```

The node is connected to the previously created bridge network and given a name, `roach1` in this case.

Port 8080 of the host machine is mapped to port 8080 of the container which is later used for accessing the web-interface (See chapter 3.4). Port 26257 of the container is mapped to the same port on the host. This port is used for communication.

Adding a volume to the container will give the user the possibility to see what CockroachDB stores on disk and save data on container restart. To completely reset the node, the user has to delete the volume. If no volume is mapped on container start, all data will be lost when the container stops.

The Docker image in use is called `cockroachdb/cockroach` and this example uses the latest version. The command `start` is passed, which will start the CockroachDB inside the container.

For the time being this example will use the insecure mode of CockroachDB which does not require certificates. More on that in chapter 3.5.

Finally the command will use the `join` argument to connect the three nodes in the cluster by specifying their `hostname`.

For this example two more nodes will be created (nodes `roach2` and `roach3`) using a very similar command as the one above, only changing the hostname and the volume directory on the host machine. The last thing that differs for the last two nodes is that the ports will not be mapped as witch `roach1`. This is because when the web-interface in this example will be accessed, it will access the first created node. For more information about the web-interface see chapter 3.4.

As the next step the cluster has to be initialized. This is done with the following command:

```
docker exec -it roach1 ./cockroach init --insecure
```

[14]

3.3 SQL Command Line

Using the CockroachDBs SQL client can be done via the following command. In this example the SQL client is used by accessing the first node that was created, `roach1`.

```
docker exec -it roach1 ./cockroach sql --insecure
```

CockroachDB uses the *PostgreSQL* dialect.

3.4 Web-Interface

Accessing the admin web-interface user interface is as easy as navigating to `localhost:8080` in a browser. Port 8080 is used because the host port 8080 is mapped to port 8080 on `roach1`. If this is already in use by another application running on the host machine, port 8080 within the container can

be remapped to a different port on the host machine. This would be done by changing `-p 8080:8080` to `-p <YOUR-PORT>:8080` in the command that started the first node.

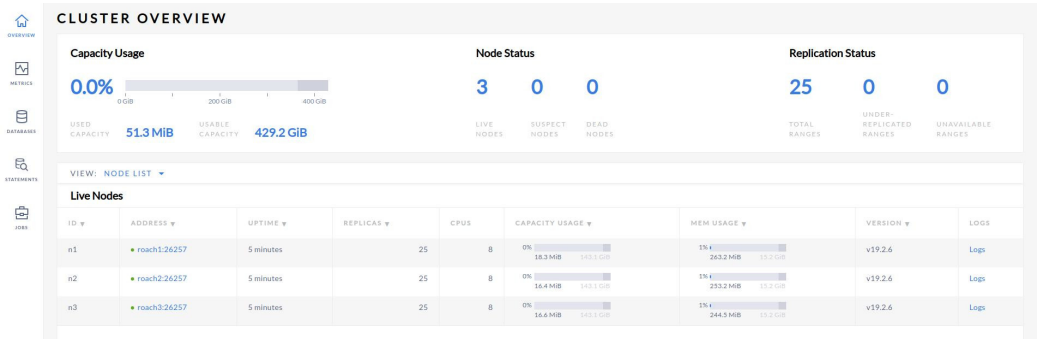


Figure 8: Cluster Overview

The first view the user is greeted with is the overview of the cluster which can be seen in Figure 8. This view shows all nodes connected to the cluster and the cluster capacity as well as the capacities for every single node. Also visible is the amount of failing and failed nodes. In the *Replication Status* one can see how many ranges are stored and replicated across all CockroachDB nodes.

In the above figure, the three previously created nodes are visible and all running. Note that every node thinks its full capacity regarding disk and memory usage are that of the main host. So in this case all three nodes assume that they can use the entirety of remaining disk space and RAM even though they really have to share these resources with the rest of the CockroachDB nodes (or really any other container or service) that also happens to run on the host machine.

Clicking on the *Logs* button next to the node in the *Live nodes* view would allow the user to see Logs specific for each node. In this example this is unavailable because the debugging mode is set to `local` by default which disables this feature.

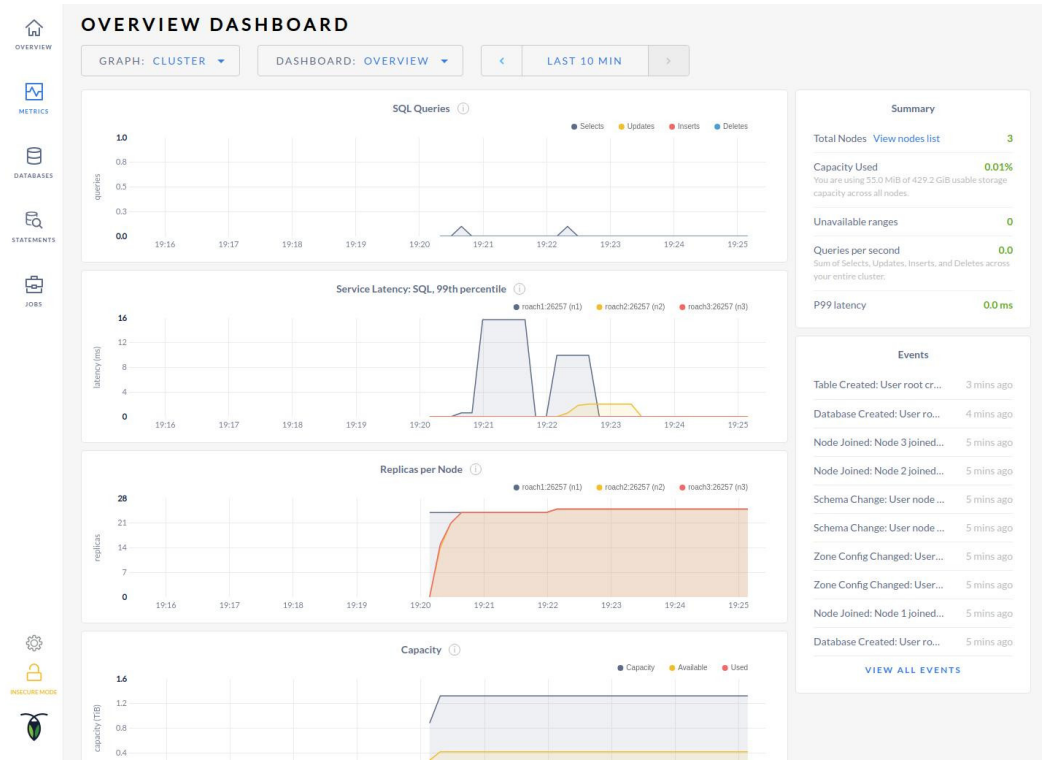


Figure 9: The CockroachDB Web-Interface Dashboard

On the Dashboard view (also labeled *Metrics* in the web UI) users can see graphs of SQL Queries ran on different nodes as well as latency and replication information. In this example we created tables and inserted data on the first node. This is visible in the *Replicas per node* graph where the line representing `roach1` immediately jumps to the value whereas the other two nodes take a little bit of time replicating the data on their system.

On the right-hand side a list of events that occurred in the database system is visible. In the example you can see that the latest events were the creation of the database and the tables therein.

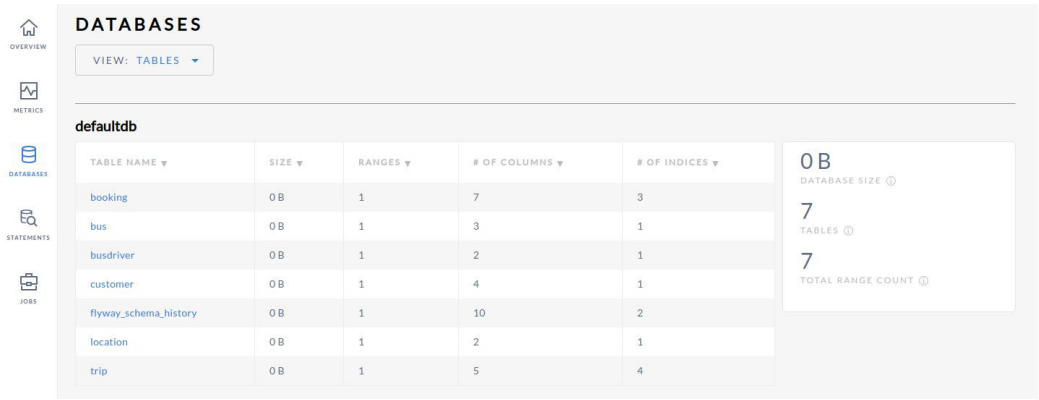


Figure 10: View of databases

The *Databases* view show a list of all databases and lists their tables. Note that it does not display contents of the database.

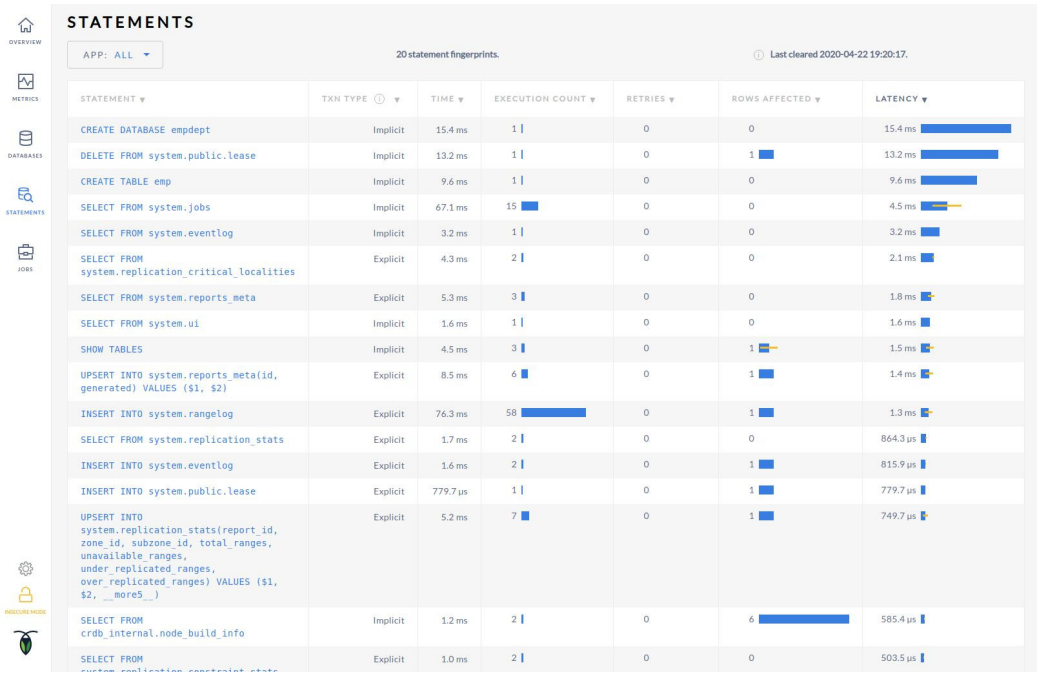


Figure 11: View of statements

The *Statements* view list all executed SQL commands/statements which can be filtered and sorted.

3.5 Securing the Cluster

author: Felix Tröbinger

In the previous example all nodes have been started with the command line option `--insecure`. In the bottom of the web-interface the user is also reminded that CockroachDB is running in insecure mode by the golden lock icon. This is obviously a bad idea to be used in production or for any database, really.

Running CockroachDB in secure mode requires using certificates. These can be created by running either the `cockroach cert` command or using `opensll`. Alternatively a custom CA can be used.[15] For this example the Cockroach command will be used.

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

The above command generates

1. a CA (= **C**ertificate **A**uthority) certificate and clients/node certificates
2. a CA key

When creating secure nodes or client certificates the CA key is required and referenced during the creation process. In a scenario where the CA key is lost, adding nodes to a cluster created from nodes that were signed with keys is not possible anymore.

Creating certificates and key pairs for a node can be done with the following command. (Replace *hostname* with your hostname/container name.)

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

When using a Docker container the certificates have to be in a specific directory of the container. When starting the container the `--certs-dir` is used to tell the container the specified location of the certificates. Similarly the `--ca-key` option is used to tell the container the location of the CA key when creating it. One possibility to add the certificates to the container is to use the volume command-line-parameter (similar to the one that was used

to create nodes in the beginning of chapter 3.2. A perhaps more elegant solution would be a Dockerfile that specifies options for either a single node or an entire cluster of nodes that run on a single machine.

The command will use the CA key that was created in the first command and stored in the *my-safe-directory* directory to create certificates for the new node.

In the next step a certified client is created, in this case for a user called *fred*.

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

After this step a SQL user by the same name is created and granted some rights in the database. [16]

```
cockroach sql \
    --certs-dir=certs \
    --host=localhost:26257 \
    --execute="CREATE USER fred;
    GRANT SELECT ON TABLE db.example_table TO fred;"
```

A certified client can then be used in an application.

References

[1] cockroachdb. Cockroachdb on github, https://github.com/cockroachdb/cockroach.

[2] Klint Finley. Ex-googlers get millions to help you build the next google, https://www.wired.com/2015/06/cockroach-labs/.

[3] Spencer Kimball Peter Mattis, Ben Darnell. Why we're relicensing cockroachdb, https://www.cockroachlabs.com/blog/oss-relicensing-cockroachdb/.

[4] Cockroach Labs. Cockroachdb pricing, https://www.cockroachlabs.com/pricing.

[5] Cockroach Labs. Architecture overview, https://www.cockroachlabs.com/docs/stable/architecture/overview.html.

[6] Jessica Edwards. Hello world: Meet cockroachdb, the resilient sql database, https://thenewstack.io/cockroachdb-unkillable-distributed-sql-database/.

[7] George Utsin. Vectorizing the merge joiner in cockroachdb, https://www.cockroachlabs.com/blog/vectorizing-the-merge-joiner-in-cockroachdb/.

[8] Cockroach Labs. Join expressions, https://www.cockroachlabs.com/docs/stable/joins.html.

[9] Cockroach Labs. Sql feature support in cockroachdb v19.2, https://www.cockroachlabs.com/docs/stable/sql-feature-support.html.

[10] Peter Bailis Todd Warszawski. Acidrain: Concurrency-related attacks on database-backed web applications, http://www.bailis.org/papers/acidrain-sigmod2017.pdf.

[11] CockroachDB. Cockroach labs live: The architecture of a distributed sql database - 2020 update, https://www.youtube.com/watch?v=cnqiif0rdvyfeature=emb$_t itle$.

[12] Cockroach Labs. Architecture overview, https://www.cockroachlabs.com/docs/stable/architecture/overview.html.

[13] docker docs. Use bridge networks, https://docs.docker.com/network/bridge/.

[14] Cockroach Labs. Start a cluster in docker (insecure), https://www.cockroachlabs.com/docs/stable/start-a-local-cluster-in-docker-linux.html.

[15] Cockroach Labs. cockroach cert, https://www.cockroachlabs.com/docs/v19.2/cockroach-cert.html.

[16] Cockroach Labs. Start a local cluster (secure), https://www.cockroachlabs.com/docs/stable/secure-a-cluster.html.