Bachelor Thesis
Blekinge Institute of Technology

Evaluation of CockroachDB in a cloud-native environment

Kristina Håkansson
Andreas Rosenqvist

Faculty of Computing
Blekinge institute of Technology
Sweden

Bachelor Thesis
Blekinge Institute of Technology

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the bachelor's degree in software engineering. The thesis is equivalent to 10 weeks of full-time studies.

Contact information:

Authors:

Kristina Håkansson	hakansson78@hotmail.com
Andreas Rosenqvist	andreas.rosenqvist83@gmail.com

Ericsson Advisors:
Ruwan Lakmal Silva
Pär Karlsson

University Advisor:

Davide Fucci	davide.fucci@bth.se

Faculty of Computing	Internet: www.bth.se
Blekinge Institute of Technology	Phone: +46 45538 5000
SE-371 79 Karlskrona, Sweden	Fax: +46 455 38 50 57

Abstract

The increased demand for using large databases that scale easily and stay consistent requires service providers to find new solutions for storing data in databases. One solution that has emerged is cloud-native databases.

Service providers who effectively can transit to cloud-native databases will benefit from new enterprise applications, industrial automation, Internet of Things (IoT) as well as consumer services, such as gaming and AR/VR. This consequently changes the requirements on a database's architecture and infrastructure in terms of being compatible with the services deployed in a cloud-native environment - this is where CockroachDB comes into the picture. CockroachDB is relatively new and is built from the ground up to run in a cloud-native environment. It is built up with nodes that work as individual machines, and these nodes form a cluster.

The authors of this report aim to evaluate the characteristics of the Cockroach database to get an understanding of what it offers to companies that are in a cloud-infrastructure transition phase. For the scope of characteristics, this report is focusing on performance, throughput, stress-test, version hot-swapping, horizontal-/vertical scaling, and node disruptions. To do this, a CockroachDB database was deployed on a Kubernetes cluster, in which simulated traffic was conducted. For the throughput measurement, the TPC-C transaction processing benchmark was used. For scaling, version hot-swapping, and node disruptions, an experimental method was performed.

The result of the study confirms the expected outcome. CockroachDB does in fact scale easily, both horizontally and vertically, with minimal effort. It also shows that the throughput remains the same when the cluster is scaled up and out since CockroachDB does not have a master write-node, which is the case with some other databases. CockroachDB also has built-in functionality to handle configuration changes like version hot-swapping and node disruptions. This study concluded that CockroachDB lives up to its promises regarding the subjects handled in the report, and can be seen as a robust, easily scalable database that can be deployed in a cloud-native environment.

Key Words: Cloud-native database, CockroachDB, Performance, Scalability

1 Introduction

CockroachDB is a relatively new database, released in 2017, that aims to meet the growing need of handling large data, stay robust, scale easily and provide consistent data. In this thesis, CockroachDB will be studied and examined how well it handles these key features.

1.1 Background

Databases, and Database Management Systems (DBMS) that allow ways to store, organize and collect data in a database, have been around for a long time. In the 1970s Edgar Codd published the academic paper *"A Relational Model of Data for Large Shared Data Banks"* that described the relational way of storing data. The data was stored in linked tables, and only stored once in the database. Storing data like this led to efficient use of storage:

1. Querying the database could be done efficiently with indexes and table joins.
2. Transactions could be used to keep combinations of reads and writes atomic.

Because of these advantages, relational databases grew popular in the following decades and became the primary way of storing data. Structured Query Language (SQL) became the language to use to query the database. [1] [2]

When the first relational databases were introduced, the amount of data and workloads that the databases had to handle were noticeably smaller than today. An increased internet usage over the years, consequently meant that the amount of workload on the database grew heavier. The relational database, however, was designed to run on a single machine and therefore had limitations in horizontal scaling and how much workload it could handle. To be able to handle the growing amount of workload, the databases needed to be able to grow horizontally, from a single machine to a cluster of nodes. To handle this scaling issue with relational databases, the *NoSQL* databases were introduced and grew in popularity in the early 2000s. These databases focus on the ease of scaling and ability to tolerate node failures. However, the NoSQL databases lacked some of the functionality of the relational databases, like joins, transactions, and efficient indexing. [1]

As the SQL and NoSQL databases evolved and made efforts to improve their respective shortcomings a newer breed of databases, *Distributed SQL,* was introduced. These databases are built from the ground up to take advantage of the cloud to scale easily, stay robust against failure of nodes and still handle functionalities that the NoSQL databases lack, and support SQL. CockroachDB is new on the market of distributed databases; it claims to be scalable, survivable, and consistent - hence the name. [3]

1.2 CockroachDB Architecture

Table 1.1 highlights important concepts for the CockroachDB architecture and the terminology used in this report.

Term	Definition
Cluster	A CockroachDB deployment, which acts as a single logical application.
Node	An individual machine running CockroachDB. Many nodes are joined together to create a cluster.
Range	CockroachDB stores all user data (tables, indexes, etc) and almost all system data in a large sorted map of key-value pairs. This keyspace is divided into ranges, which are contiguous chunks of the keyspace so that every key can always be found in a single range. From a SQL perspective, a table and its secondary indexes initially map to a single range, where each key-value pair in the range represents a single row in the table (also called the primary index because the table is sorted by the primary key) or a single row in a secondary index. As soon as that range reaches 512 MiB in size, it splits into two ranges. This process continues for these new ranges as the table and its indexes continue growing.
Replica	CockroachDB replicates each range (3 times by default) and stores each replica on a different node
Leaseholder	For each range, one of the replicas holds the “range lease”. This replica, referred to as the “leaseholder”, is the one that receives and coordinates all read and write requests for the range.

Table 1.1. Important CockroachDB architectural concepts and terminology [33]

CockroachDB is a distributed SQL database built on a key-value store. The stores are broken up into parts, called ranges. Each range has replications, which are multiple copies of the same pieces of data, that are kept in sync across machines. By default, CockroachDB ranges have three replicas. When for example a write comes in, it is distributed to all replicas, and when two of the three replicas have written to disk, it is acknowledged to the client. This means that even if a node fails the write is still present in the system, which enhances the resilience and ability to stay alive. [32]

Regarding reads, where no state is changing, going through all nodes and waiting to get a consensus answer from a majority of the nodes would be expensive; whereas reading from any node would make it uncertain if that node was up to date. CockroachDB solves this by having a leaseholder replica that is always aware of what writes have been committed and can service reads efficiently and with accuracy. Figure 1.1 shows an example of how data is divided up into three ranges with three respective replications, that in turn are distributed on a cluster of four nodes. [32]

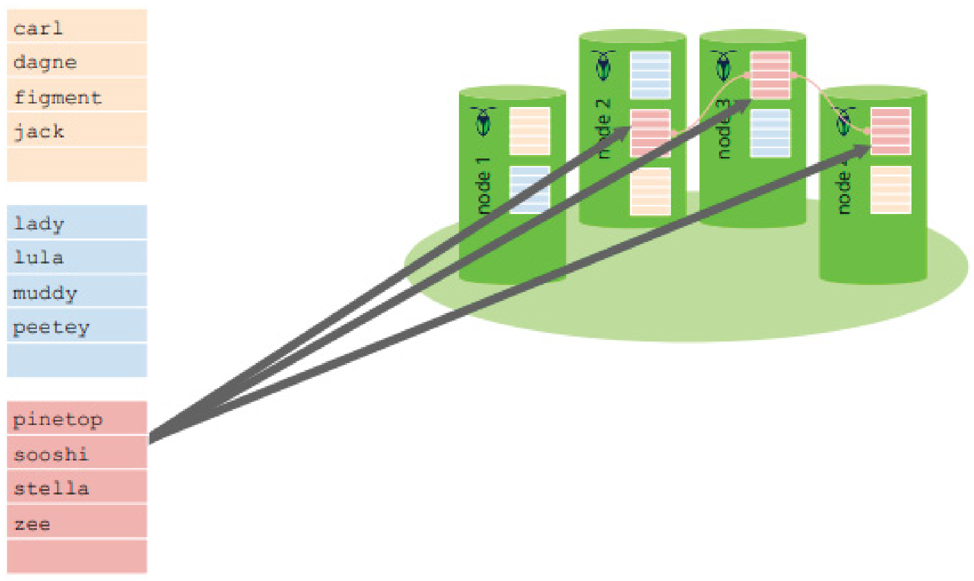


Figure 1.2. CockroachDB: The architecture of a Distributed SQL Database [32]

CAP theorem

The CAP theorem, also known as Brewer's theorem after computer scientist Eric Brewer, states that networked shared-data system can only fulfill two out of the three following properties, simultaneously:

- **C**onsistency - All users of a system see the same data at the same time, regardless of what node they are connected to.
- **A**vailability - Every query to the system receives a response, even if a node is down.
- **P**artition tolerance - The cluster continues to function, even if there is a communication break between nodes, although the nodes are still up and running. [34]

CockroachDB is a CP database system that delivers consistency and partition tolerance. Having strong consistency is part of what makes CockroachDB being able to offer guarantees that are usually expected from traditional non-distributed SQL databases. [35]

1.3 Purpose

The main purpose of this thesis is to give an understanding of the characteristics of CockroachDB, and if it lives up to its claims regarding scaling, throughput, version hot-swapping, consistency, and robustness.

In the era of Big Data, the ability to scale is of utmost importance when deciding which database to settle with. Some databases handle scaling better than others, but on the other hand, have some other disadvantages. Key factors in today's databases are scalability, robustness, and performance, and by evaluating CockroachDB in terms of strengths and weaknesses, this thesis will bring more clarity and insight into whether CockroachDB is a viable option as a modern database.

There is an extensive amount of comparisons between various databases. However, since CockroachDB is relatively new on the market, it stands reasonably untested in the industry, compared to databases that have been around longer. This thesis will help to fill that gap for companies that seek further knowledge.

1.4 Scope

This thesis will evaluate CockroachDB in the scope of its scalability, performance, robustness, and version hot-swapping. The time and effort of scaling the database, by allocating more CPU- or RAM, and/or adding more nodes will be measured and evaluated. The performance of the database will be evaluated in terms of throughput. The ability to upgrade, respectively downgrade the version, while still having it running will be examined. Furthermore, the robustness of the database will be studied by examining if and how well the cluster handles node failure while there is active traffic towards the cluster.

This study is being conducted in collaboration with an industrial partner - Ericsson. Therefore, the scope has been set after negotiations with Ericsson, and their requirements and priorities.

Outside scope
For the scaling procedure, this thesis will include the scaling processes of *scaling out* and *scaling up* an existing CockroachDB cluster. Scaling out a database system is done by adding more node(s) to the cluster. Scaling up a system, on the other hand, is done by keeping the number of nodes the same, but making them more powerful by adding resources. Scaling by changing the number of nodes is also referred to as *horizontal scaling* and could be done both by scaling *out* and *in* (if the number of nodes is decreased). In the same way, scaling by changing the resources of existing nodes can be done by scaling *up* as well as *down*, this can be referred to as *vertical scaling*. This study will include scaling out and up the nodes, i.e. making it more powerful either with increased resources or with more nodes. Scaling in/down a cluster will not be part of this study.

When examining version hot-swapping, the scope involves the procedure of updating the version and how the cluster handles this while exposed to traffic. If the data or efficiency in terms of throughput or latency is affected during or after the configuration changes are not examined and outside the scope of the study.

The robustness and how well the cluster handles node failure will be examined by taking down a node while traffic keeps running on the cluster. The traffic in this study will include only writes and the consistency of the data will be measured by checking that this data got inserted, after node failure and reparation of the cluster. Any other type of operations during node failure will not be part of the scope. It could also be interesting to see if and how the efficiency in terms of throughput and latency is affected while a node is down and during the repair phase, this is however not part of this study.

2 Research Questions

Each question is presented with goals, motivation, and expected outcome.

2.1 RQ1: What are the procedures to scale out/up CockroachDB?

The goal of this question is to get to familiarize with the procedures of scaling out/up CockroachDB. How many steps are needed, if replicas distributed on the nodes balance themselves on the new nodes, and if so, how long it takes before it is fully balanced.

Since the commonly used traditional relational databases are not designed for horizontal scaling, the procedures of doing so in these databases are complicated, and a reason to search for another database solution [4]. Finding out the procedures of scaling out/up CockroachDB will give an understanding of the complexity of the scaling process of this database.

Being a cloud-native distributed database built from the ground up, CockroachDB makes the following claim:

> *"Scale your database by simply adding new nodes and avoid any manual manipulation of data. CockroachDB automatically rebalances and replicates data throughout the cluster."* [25]

The database is therefore expected to scale easily and the newly added nodes in the scaling-out procedure are expected to automatically get replicas from the existing nodes to balance the cluster without any further effort by the user.

2.2 RQ2: How well does CockroachDB scale out/-up in terms of throughput?

The goal of this question is to get measurements of throughput at different scaling levels and different workloads and to analyze these results.

In contrast to other multi-node databases, such as PostgreSQL, in which the throughput would decrease proportionally to the nodes present in the cluster, with CockroachDB a different result is expected. The reason for this is that CockroachDB does not have a master node that is in charge of performing all the write operations, as is the default case for example PostgreSQL. PostgreSQL does allow for multi-master write nodes, but this comes with severe performance impact due to the need for synchronization between the nodes. [31]

In CockroachDB all of the nodes in the cluster have write-privileges [5] and, for this reason, the hypothesis is that instead of the throughput decreasing when the database is scaled out, the throughput will remain the same. The same result is expected when the system is scaled up. When the workload is increased the throughput is expected to increase as well and eventually flatten out when the system under test (SUT) is receiving more traffic than it is capable of processing.

2.3 RQ3: What are the procedures for version hot-swapping?

The goal of this research question is to verify if the version that the database is configured to, can be changed while there is ongoing traffic towards the cluster.

Since new versions of CockroachdB are expected to be presented, there is a need for the database to easily upgrade to a newer version. It is also important to be able to go back to a previous version and roll back the system if necessary. Understanding if the database has to be taken down for version change or if the version change can be done on the fly while traffic is still running, is a valuable characteristic for any database.

As CockroachDB should be able to balance between nodes, the expected outcome of this research question is that it should be able to adjust to version changes without user interference, by taking down one node at a time and up-/downgrade the version on that node, while the workload could keep running on the other nodes.

2.4 RQ4: How resilient is CockroachDB in the presence of node disruptions?

The goal for this question is to evaluate the recovery functionality of a node failure and if the cluster still is available for operations while a node is temporarily down.

Recovery procedures for database failures can be both time-consuming, complicated to undertake, and an availability threat. The ability to stay robust, recover, and keep running through a node failure is of great importance to any database.

CockroachDB claims to be robust and self-repair without interruptions of traffic in case of node failure [6]. The expected outcome of this research question is therefore that the database should repair itself after a node is taken down, and that it will be available for traffic during disruptions.

3 Research Method

3.1 RQ1: What are the procedures to scale out/up CockroachDB?

To learn the procedures of scaling out and up an existing CockroachDB cluster, the documentation of CockroachDB will be studied.

To test these approaches, a CockroachDB cluster will be initialized on an existing Kubernetes cluster. The scaling instructions from CockroachDB will be used to scale the cluster up as well as out.

3.1.1 Scaling out

The CockroachDB cluster on Kubernetes will be initialized with three nodes, then scaled out to six and nine nodes. The resources used for the nodes on the cluster are shown in Table 3.1.

CPUs	RAM (GB)
8	32

Table 3.1, Allocated resources when scaling out from three to six to nine nodes.

For each scaling step (three to six nodes, respectively six to nine nodes), time will be measured for the system to adjust to the newly added nodes. The starting time will be when the scaling starts, and the stop time when all nodes are balanced up, i.e. when the replicas have finished being copied over to the new nodes, and the number of replicas is balanced across the nodes. The balancing process and the number of replicas on each node will be monitored and read from the CockroachDB Console (see section 3.5.3).

3.1.2 Scaling up

The system will be initialized with three nodes. The allocated resources of the cluster will be scaled up from small to medium to large configuration, according to table 3.2.

	CPUs	RAM (GB)
Small	4	16
Medium	8	32
Large	12	48

Table 3.2, Allocated resources at different scaling up levels.

Throughout the scaling-up process, the nodes will be monitored using the CockroachDB console to check how the configuration changes are updated on the nodes of the cluster.

3.2 RQ2: How well does CockroachDB scale out/-up in terms of throughput?

Stress test

A stress test will be performed to determine the amount of workload that the SUT can handle before it becomes saturated.

1. The SUT will be initialized with size and resources according to Table 3.3.
2. It will then be exposed to a small workload (Table 3.4) with requests to the database for 10 minutes.
3. The throughput and efficiency will be measured.
4. The workload will then be increased.
5. Repeat the procedure.

Generating the workload and measuring the throughput as well as the efficiency (EFC), will be done through the TPC-C tool. The throughput will be measured in new order transactions per minute (tpmC), as defined by TPC-C [7]. The efficiency will be measured in EFC, i.e. how close the measured throughput is to the theoretical maximum throughput for the cluster. See section 3.5.2 for a description of the TPC-C tool and its units.

Nodes	CPUs	RAM (GB)
3	12	48

Table 3.3: Cluster size and resources for stress testing.

Test	Workload (warehouses)	Time
1	50	10 minutes
2	100	10 minutes
3	150	10 minutes
…	…	…

Table 3.4 Tests run to measure throughput at different workloads (as defined by TPC-C [7], see section 3.5.2).

Performance test

Scale-out the system
The SUT will be scaled out from three to six and then to nine nodes. After each scaling, a workload will be run on the cluster for 10 minutes and the throughput will be measured. The workload will be incremented and run on the cluster consequently. After this procedure, the SUT will be scaled out and the same procedure will be repeated.

Scale-up the system
The cluster will also be scaled up from small to medium, and then to large amounts of allocated resources, according to table 3.2. The same procedures as when scaling out the system will be conducted to measure the throughput after each scale.

The SUT will start with a size of three nodes and resources according to table 3.5 before scaling.

	Nodes	CPU	RAM (GB)
Scale-out start	3	8	32
Scale-up start	3	4	16

Table 3.5: Cluster sizes and resources before scale.

3.3 RQ3: What are the procedures for version hot-swapping?

A CockroachDB cluster of three nodes will be initialized on the Kubernetes cluster, and a workload of traffic will be run towards the database. The version change will be done by modifying the Statefulset file for the Kubernetes cluster. This will be done while the cluster is still receiving traffic.

A CockroachDB cluster of three nodes will be initialized on the Kubernetes cluster, and a TPC-C workload of traffic will be run towards the database (see section 3.5.2 for a description of the TPC-C and its workload). The version change will be done by modifying the Statefulset file, which is the workload API object to manage stateful applications. This will be done while the cluster is exposed to the TPC-C workload.

Changes of version will be done from an older to a newer version as well as the other way around.

Throughout the version change, the nodes will be observed through the CockroachDB console with respect to how the configuration changes are updated on the nodes of the cluster.

3.4 RQ4: How resilient is CockroachDB in the presence of node disruptions?

To test how CockroachDB handles disruptions in the cluster, a database consisting of three nodes will be exposed to intentional disruptions by killing one node in the cluster.

The test will be performed on a running cluster of three nodes. A bash script will create a table and perform inserts of rows. In the script, one out of the three nodes will be killed in the cluster while write-operations will proceed. To check that all insert operations are executed correctly, reads will be performed using an SQL query to count the number of inserted rows, as well as a randomly selected row. An acceptance sampling, in which rows will be verified to have been written to the database, will also be conducted to verify that the database is still available for write and read operations before and after the cluster has recovered from the node disruption.

The script used in this method can be found in the Github repository [Appendix A].

3.5 Tools used

3.5.1 Kubernetes

To deploy the database an existing Kubernetes cluster was used. Kubernetes is a portable, extensive, open-source platform for managing containerized workloads and services. [29]

3.5.2 Transaction Processing Benchmark, TPC-C

TPC-C was used in this thesis to measure throughput. It is a tool that generates five types of concurrent transaction-based workloads towards a database. The most frequent of these transactions is the new order transaction, which is also used to measure the throughput. This measurement of the throughput in transactions per minute, defined by TPC-C, is known as tpmC. [7]

The benchmark portrays a wholesale supplier. This fictional company consists of a given number of warehouses and their sales districts. Each sales district has 10 warehouses and each warehouse has 3000 customers. The company scales as the number of warehouses increases. [8]

3.5.3 CockroachDB Console

The cockroachDB Console is a tool provided by Cockroachlabs and provides a detailed overview of the cluster and database configuration. It can be used to monitor the traffic and the state of the cluster through various dashboards, such as Runtime-, Storage-, Replication-, and Hardware Dashboards. It is also possible to inspect transactions and network latency for the cluster. [18].

3.5.4 Lens

Lens is an open-source IDE to manage and control a Kubernetes cluster. It helps users to monitor and get a visual representation of pods and containers that are running on the Kubernetes. [30]

3.5.5 Kubectl

Kubectl is a Kubernetes command-line tool. It allows users to run commands towards the Kubernetes cluster. It is used for operations such as deployment, inspection, management of resources, and view logs.

4 Literature Review

In this section, we review the main studies from literature and practitioners' venues, such as blog posts.

4.1 SQL versus NoSQL

SQL databases are the traditional and mainly used databases and have been around since the early 1970:th. SQL databases are considered easy-to-use with the SQL language and support ACID (atomicity, consistency, isolation, and durability) transactions. With a growing demand for handling larger amounts of data, a new type of database, NoSQL, which scales better, was introduced. [1]

Anderson and Nicholson [24] mention that a SQL database is a natural fit when using relational data when data integrity is important, and scale-out capabilities are not necessary. According to them NoSQL, on the other hand, is preferred to use if the data does not fit in the relational model, scalability is important, and/or data integrity guarantees could be overcome.

When comparing NoSQL databases with SQL databases, the NoSQL databases have some advantages, such as scalability. On the other hand, their main drawback is a lack of support for ACID transactions [1]. NoSQL databases also lack compatibility with SQL, which according to Leavitt [14]:

> *"...require manual query programming, which can be fast for simple tasks but time-consuming for others."* [14]

Leavitt mentions that NoSQL databases can be implemented as distributed databases and that they could be scaled horizontally. Leavitt mentions some advantages with this, for example, that there is no limit in how much a cluster can scale, as is the case with vertical scaling. Also, when the cluster is scaled horizontally instead of vertically, cheaper machines could be used in the cluster.

Yishan and Sathiamoorthy [12] compare performance between NoSQL and SQL databases. They find that not all NoSQL databases have better performance than the SQL databases tested. There are differences in performance not only between NoSQL and SQL databases but also depending on which vendor of SQL or NoSQL database is used. They conclude that different databases are fitted for certain characteristics. For example, while some are better for read operations, others are better fitted for write operations.

Anderson and Nicholson [23] describe differences between SQL and NoSQL databases and also mention new types of databases, *NewSQL* and *Distributed SQL*. They describe them both as solutions to combining the advantages of SQL and NoSQL respectively, but with the difference that NewSQL is built by adding layers of distributed logic on top of a traditional relational database, while the distributed SQL databases are built from the ground up. They mention CockroachDB as an example of a distributed SQL database. Like other authors, Anderson and Nicholson, also mention scalability issues with relational databases in contrast to the ability to perform ACID transactions in NoSQL databases.

Radoev [38] mentions that traditional databases (SQL) aim to deliver data consistency, rather than high availability which NoSQL databases are prioritizing. This is a trade-off between the traditional databases and NoSQL databases, and results in creating systems known as BASE (Basically Available, Soft-state, Eventually consistent). They mention that trying to solve one issue has caused other issues to emerge.

4.2 NewSQL databases

Karambir and Sachdeva [36] have conducted a performance evaluation, and also describe benefits and characteristics of four NewSQL databases, among which CockroachDB is one of them and the others are NuoDB, VoltDB and MemSQL.

The authors have compared these databases regarding for example scalability and consistency. They show that the four compared databases have different scalability characteristics, where CockroachDB has horizontal scalability with automated scaling, and the others have similar characteristics, with horizontal and bi-directional scaling capabilities. The consistency is noted as strong by the authors for three out of the four the tested databases, CockroachDB included. NuoDB was listed to have eventual consistency. [36]

Strong consistency is that the data is consistent between nodes, and the users can always get an updated response. However the response time may be affected because of the user having to wait for the transferring of data between nodes. Eventual consistency means that all of the nodes will eventually get updated, but during this period of time for the nodes to get updated, there could be inconsistent data between the nodes. The trade-off for this is that the data is to a higher degree more available compared to a database that has strong consistency. [37]

4.3 CockroachDB distributed transaction model

CockroachDB was not the first system to deliver a scalable and distributed database. An internal product of Google - Spanner, introduced the new concept of delivering data on a global scale while still offering consistently distributed transactions. CockroachDB is based on Google Spanner data storage system which uses atomic clocks and GPS clocks for time synchronization [9, 10, 11]. To guarantee external consistency while still being cloud-independent is problematic for CockroachDB, Kimball [10] explains why.

> *"As a spanner-derived system, our challenges lie in providing similar guarantees of external consistency without having magical clocks at hand. CockroachDB was intended to be run on off-the-shelf commodity hardware, on any arbitrary collection of nodes. It's 'cloud-neutral' in that it can very well span multiple public and/or private clouds using your flavor-of-the-month virtualization layer"* [10]

What CockroachDB does differently from Spanner is that it sometimes retries reads, while Spanner on the other hand always waits after writes. This means that CockroachDB may read repeatedly, instead of always waiting until update, after a write. This enables CockroachDB to perform atomic transactions without utilizing locks [15, 16, 17]. A thorough comparison of Spanner and CockroachDB is out of the scope of this thesis.

4.4 Scalability

Although NoSQL databases are developed to scale better than SQL databases, there are also differences between NoSQL databases in how well they scale. Swaminathan and Elmasri [23] evaluate three of the most common NoSQL databases, MongoDB, Cassandra, and HBase, focusing on their scalability under different workloads and dataset sizes. The read, write, blind write, read-modify-write, and scan operations are used to run the tests. They conclude that the databases have different designs and are advantageous in the above-mentioned operations, due to this it is important to select the right database depending on what needs the user has. For example, the study showed that for large datasets Cassandra was the database that had the best throughput performance. Cassandra also performed well overall at different operations. For blind write operations, HBase performed better. MongoDB performed well in primarily read operations.

The ability to scale easily is one of the CockroachDB primary advantages compared to traditional, relational, SQL databases [1]. Many databases need to be stopped and started up again to be able to scale. A CockroachDB database, running in an orchestration platform like Kubernetes, is easy to scale with a simple command to the orchestration tool while the database keeps running. Heller [17] mentions this advantage of CockroachDB and tests the approach of scaling out the CockroachDB cluster from the command line of a Kubernetes cluster. With a single Kubectl command he demonstrates how a CockroachDB cluster scales out from three to ten nodes and then starts balancing itself by distributing the ranges across the nodes in the cluster.

4.5 Benchmarking distributed systems

Benchmarking is a well-known concept when trying to compare a system to another system - databases are no exception. According to Narayan [19], it is impossible to make fair comparisons across databases because of the lack of reporting complete results. Narayan notes:

> *"..not every database vendor reports complete results with hardware specifications, tuning parameters, and any other tradeoffs. For instances, while databases often report high throughput numbers, benchmarks don't always come with standardized latency metrics, nor is it standard to report on long-running loads"* [p1, 19]

With this in consideration, CockroachDB decided to use the TPC-C benchmarking tool for their performance analysis, because TPC-C is far more rigorous in its specification than other benchmarking tools.

Narayan also mentions a newer Online Transactional Processing benchmark (OLTP), TPC-E, which does not replace the TPC-C test but instead is used as a complementary tool for testing different aspects of the database. The key difference between TPC-E and TPC-C is that the former tests read in a higher ratio than writes, and it also stresses more random input/output operations compared to TPC-C. Narayan notes that the TPC-E has not seen much benchmarking due to being new on the market. [19]

Hobden [22] notes: "*Computers aren't the same, and distributed systems are even less so.*" and follows up with:

> *"Different RAM configuration, CPU sockets, motherboards, virtualization tools, and network interfaces can all impact the performance of a single node. Across a set of nodes, things as routers, firewalls, and other services can also interact with metrics" [22]*

According to Hobden, instead of focusing on specific workloads for a specific machine, it is more common in the industry to compare different versions of the same product, or different products, with the same hardware configuration. Hobden explains:

> *"Knowing relative numbers gives us the ability to understand how a new change might impact our product, or to learn our weaknesses based on the results of another product" [22]*

Benchmarking distributed databases appears to be undisputed, but what makes a benchmark good then? Hobden mentions two characteristics that the best benchmarks have in common - reproducible and convincing. To be reproducible the details about topology, hardware, and configuration for the deployment are vital for others. Reporting and displaying the weaknesses in the benchmark result is helpful to be convincing even if the result is not in favor of the product you are trying to promote. Being honest about the findings will make the report more convincing and credible. [22]

According to Kuhlenkamp et al. [27] performance benchmarks of databases are important in selecting the right technology. Their study focuses on two databases, Cassandra and HBase. By looking at a previous study [28], they reproduce the experiments conducted to measure performance and scalability benchmarking, and finally, they compare the results. They use three different workloads to see where each of the databases performs better or worse. The three workloads used are R (read-intense, with 95% read operations and 5% insert operations), W (write-intense, with 99% insert operations and 1% read operations), and RS (scan-intense, with 47% read operations, 47% scan operations and 6% insert operations).

The result of the read-intense workload is coherent with the result from the previous study using the same benchmarking tool and technique regarding the conclusion that HBase has higher latency than Cassandra. However, the latency measurements of the Cassandra database are worse than the ones measured in the previous study. As for the write-intense experiment, the authors experience a linear increase in throughput with an increasing number of nodes and low latency values for both databases. [27]

The authors conclude that Cassandra and HBase both scale linearly, and Cassandra read performance is better than HBase, while the latter outperforms Cassandra, in write operations. They further conclude that there were significant performance differences compared to the previous study conducting the same experiment, which is partly due to the experiments being reproduced on a virtualized infrastructure compared to a physical infrastructure used in the original experiment. [27]

5 Result and Analysis

This section reports the result of the analysis of each research question.

5.1 RQ1: What are the procedures to scale out/up CockroachDB?

In this section, the results and analysis of the first research question are presented.

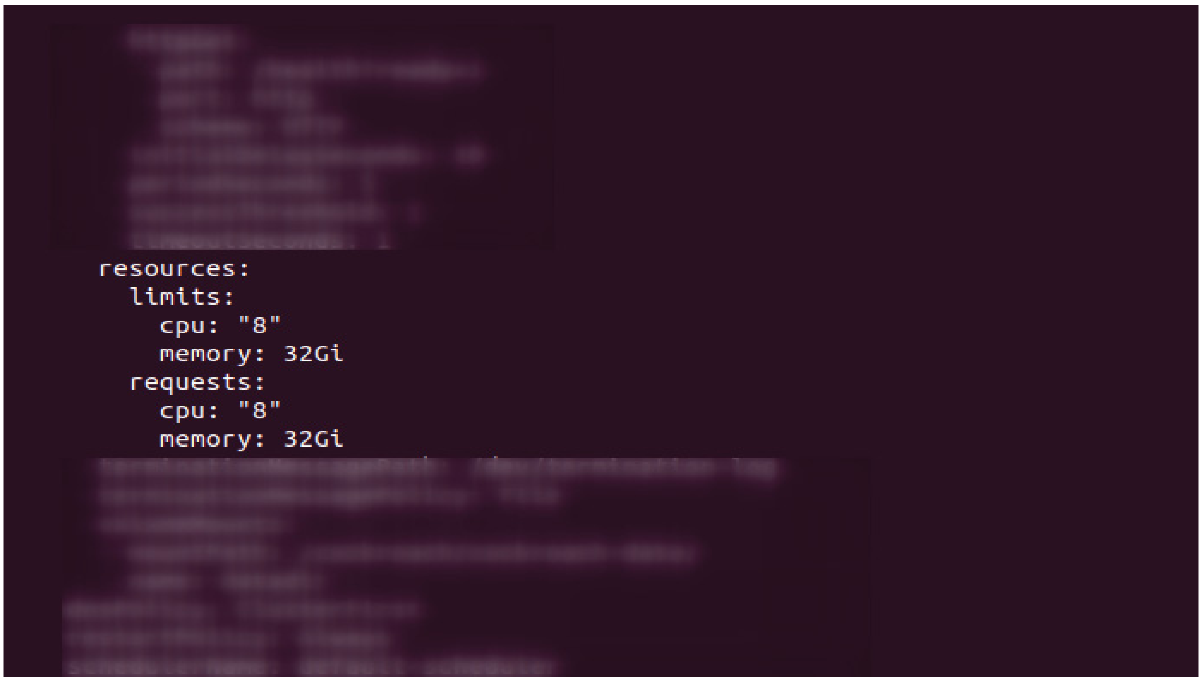
5.1.1 Procedures of scaling up the cluster

The procedures for scaling up the CockroachDB cluster, hosted on a Kubernetes cluster, involve editing the Statefulset configuration. The configuration of the Statefulset can easily be modified with a simple command line, such as Kubectl (see section 3.5.5) shown in Figure 5.1. This will bring up the Statefulset configuration (Figure 5.2).

After the Statefulset configuration has been edited and saved, with both CPU and memory changed according to Table 3.2, the process of scaling up begins. This will subsequently bring down one node, allocate the specified resources for that node, bring the node up again, and the same pattern repeats until all of the nodes in the cluster have the new specified configuration.

```
$ kubectl edit statefulset [deployment-name]
--namespace=[namespace-name]
```

Figure 5.1, Kubectl command for scaling the cluster by editing the Statefulset.



```
[redacted]
      resources:
        limits:
          cpu: "8"
          memory: 32Gi
        requests:
          cpu: "8"
          memory: 32Gi
[redacted]
```

Figure 5.2, Statefulset configuration for the resource allocation.

5.1.2 Procedures of Scaling out the cluster

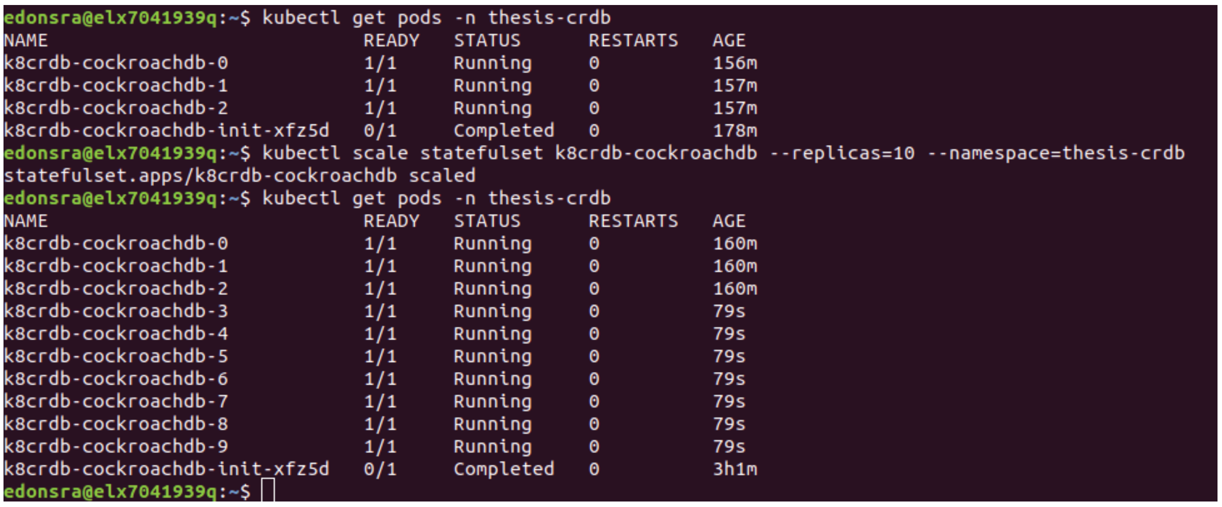
The scaling out-process of a CockroachDB cluster, running on Kubernetes, is similar to the procedures to scale up. It can also be done through a command line (Figure 5.3)

```
$ kubectl scale statefulset [deployment-name] --replicas=[number of replicas]
--namespace=[namespace-name]
```

Figure 5.3. Kubectl command for scaling the cluster by editing the Statefulset.

Executing this command with a larger number of nodes than currently in the system will scale out the system (Figure 5.4).

The cluster begins setting up new nodes and the replicas will immediately start to automatically balance between the original and the newly added nodes.



```
edonsra@elx7041939q:~$ kubectl get pods -n thesis-crdb
NAME                         READY   STATUS      RESTARTS   AGE
k8crdb-cockroachdb-0         1/1     Running     0          156m
k8crdb-cockroachdb-1         1/1     Running     0          157m
k8crdb-cockroachdb-2         1/1     Running     0          157m
k8crdb-cockroachdb-init-xfz5d   0/1     Completed   0          178m
edonsra@elx7041939q:~$ kubectl scale statefulset k8crdb-cockroachdb --replicas=10 --namespace=thesis-crdb
statefulset.apps/k8crdb-cockroachdb scaled
edonsra@elx7041939q:~$ kubectl get pods -n thesis-crdb
NAME                         READY   STATUS      RESTARTS   AGE
k8crdb-cockroachdb-0         1/1     Running     0          160m
k8crdb-cockroachdb-1         1/1     Running     0          160m
k8crdb-cockroachdb-2         1/1     Running     0          160m
k8crdb-cockroachdb-3         1/1     Running     0          79s
k8crdb-cockroachdb-4         1/1     Running     0          79s
k8crdb-cockroachdb-5         1/1     Running     0          79s
k8crdb-cockroachdb-6         1/1     Running     0          79s
k8crdb-cockroachdb-7         1/1     Running     0          79s
k8crdb-cockroachdb-8         1/1     Running     0          79s
k8crdb-cockroachdb-9         1/1     Running     0          79s
k8crdb-cockroachdb-init-xfz5d   0/1     Completed   0          3h1m
edonsra@elx7041939q:~$ 
```

Figure 5.4. The state of the cluster before and after scaling out.

Figure 5.5 shows a screenshot of the CockroachDB console (see Section 3.5.3) showing how the replicas get distributed among the nodes after scaling out and how the capacity (increased storage) of the cluster changes.

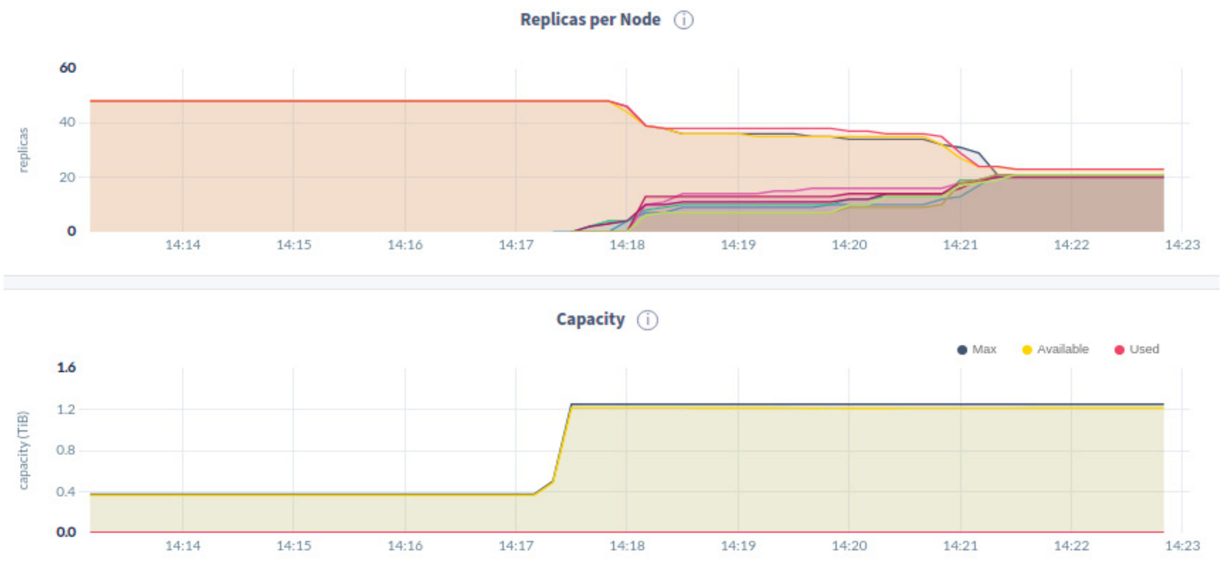


Figure 5.5. The balancing of nodes and capacity of the cluster during the scaling out process.

5.1.3 Balancing measurement

Table 5.1 shows the result of the time measurement for the cluster to self-balance itself. The result is plotted in Figure 5.6, which shows how many ranges each node has at a given time.

In this experiment, the cluster configuration was scaled from 3 to 6 nodes. The original nodes in the cluster (1-3) had 795 ranges each, and the three new nodes (4-6) are starting with no ranges since they just got deployed.

Time (min)	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
0	795	795	795	0	0	0
20	766	766	765	50	50	53
40	745	743	743	74	75	73
60	734	734	734	83	84	84
80	718	719	717	100	99	100
100	705	707	702	113	113	113
120	687	689	691	129	129	129
140	665	663	662	155	155	155
160	635	635	634	184	183	182
180	611	610	611	207	207	207
200	585	587	583	234	233	234
220	561	562	562	256	256	256
240	535	533	533	284	284	285
260	509	509	510	309	308	308
280	452	454	452	364	365	366
300	423	422	421	396	395	396

Table 5.1, Result of the time measurement for the cluster to get rebalanced.

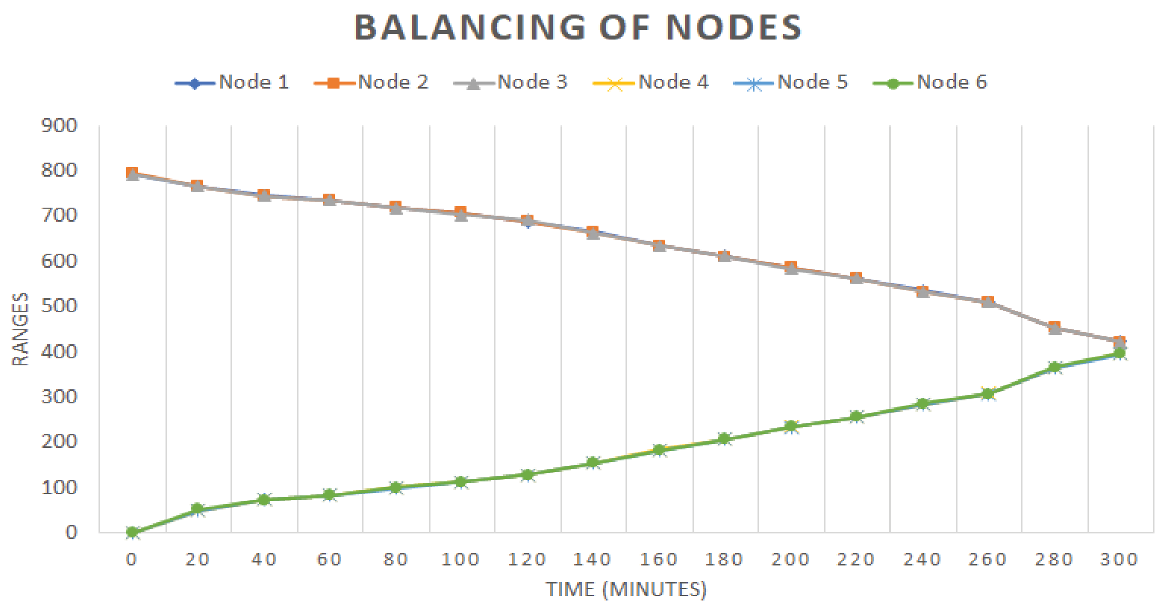


Figure 5.6. The timeline for the cluster to get balanced. Nodes 1, 2, and 3 are overlapping, as well as nodes 4, 5, and 6.

The result is coherent with the expected outcome, in which the new nodes will relieve the other nodes by transferring ranges to themselves. This automatic feature appears to be systematic, based on the timeline graph illustrated above. It is hard to distinguish the lines from each other since the balance procedure distributes the ranges among the new nodes.

For this cluster configuration with a 795 range per node and a scale out-factor of 1:2, it took approximately 5 hours for the cluster to get balanced again.

The same measurement was taken when the cluster configuration was scaled from 6 to 9 nodes. This measurement was taken after the experiment conducted above (3 to 6 nodes). In this setup, the original nodes (1-6) started with roughly 400 ranges each, and the new nodes (7-9) started with no ranges. Table 5.2 and Figure 5.7 shows the result of this experiment.

Time (min)	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8	Node 9
0	413	422	420	402	401	401	0	0	0
20	404	402	404	390	389	391	29	28	27
40	383	382	382	381	381	382	57	58	58
60	370	368	369	367	365	367	86	85	86
80	349	350	349	351	354	353	119	119	120
100	336	333	333	335	336	337	151	150	151
120	321	322	322	320	320	321	178	181	180
140	305	307	305	306	306	307	209	208	210
160	290	292	288	293	293	294	237	238	237
180	279	279	278	277	278	280	265	263	263

Table 5.2, Result of the time measurement for the cluster to get balanced.

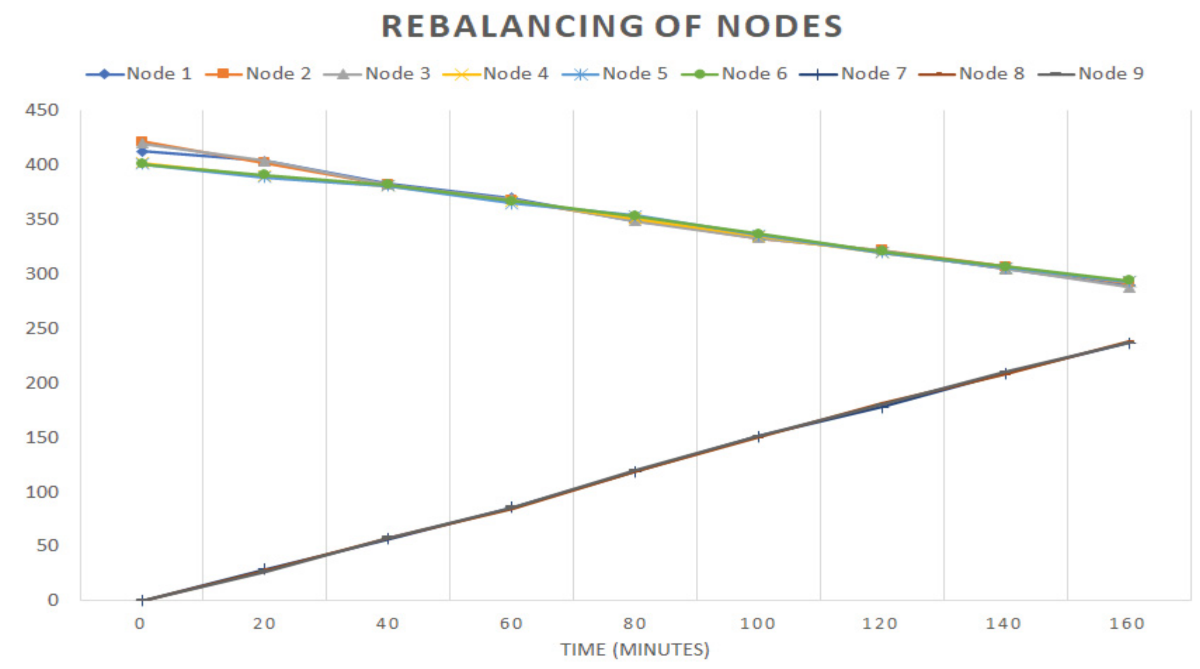


Figure 5.7 Time for the cluster to get balanced. Nodes 1, 2, 3, 4, 5, and 6 are overlapping, as well as nodes 7, 8, and 9.

Also, this result is aligned with the expected outcome. The self-balancing feature is relieving the original nodes by transferring ranges to the newly deployed nodes.

For this cluster configuration with roughly 400 ranges per node and a scale-out factor of 2:3, it took approximately 3 hours. The shorter time measured in this experiment can be attributed to the smaller scale-out factor, and that there are fewer ranges to move.

5.2 RQ2: How well does CockroachDB scale out/-up in terms of throughput?

In this section, the results and analysis of the second research question are presented.

5.2.1 Stress test

The result of the stress test is shown in Table 5.3 and Figure 5.8 below.

WH	tpmC	EFC (%)	avg (ms)	p50 (ms)	p90 (ms)	p96 (ms)	p99 (ms)	pMax (ms)
10	123.4	96.0	26.9	22.0	46.1	60.8	96.5	142.6
20	243.6	94.7	34.4	30.4	50.3	60.8	96.5	285.2
40	487.6	94.8	28.7	27.3	37.7	41.9	56.6	218.1
80	978.1	95.1	29.0	28.3	39.8	44.0	58.7	268.4
160	1963.2	95.4	35.0	30.4	52.4	62.9	96.5	243.3
320	3907.8	95.0	41.7	35.7	65.0	83.9	134.2	419.4
640	7834.7	95.2	63.0	56.6	104.9	125.8	176.2	453.0
800	9768.3	94.9	78.1	67.1	134.2	167.8	234.9	872.4
900	11000.6	95.0	87.8	75.5	159.4	192.9	285.2	671.1
1000	12189.7	94.8	107.8	92.3	201.3	243.3	335.5	671.1
1100	13422.6	94.9	148.0	125.8	285.2	335.5	453.0	1208.0
1200	14597.2	94.6	216.2	192.9	402.7	469.8	637.5	2952.8
1300	15728.9	94.1	314.9	285.2	570.4	671.1	939.5	5100.3
1400	16862.0	93.7	486.0	453.0	805.3	939.5	1275.1	3758.1
1500	17581.1	91.1	1057.9	906.0	1744.4	2281.7	4160.7	26843.5
1600	16088.2	78.2	4299.1	1275.1	12348.0	20401.1	47244.6	103079.2
1700	10754.0	49.2	18735.3	13421.8	42949.7	60129.5	103079.2	103079.2
1800	7611.0	32.9	33117.9	24696.1	85899.3	103079.2	103079.2	103079.2
1900	8230.7	33.7	28937.9	22548.6	77309.4	98784.2	103079.2	103079.2
2000	9741.8	37.9	26694.8	22548.6	55834.6	81604.4	103079.2	103079.2
2100	9566.4	35.4	30162.4	26843.5	62277.0	77309.4	103079.2	103079.2
2200	7667.4	27.1	37422.4	33286.0	85899.3	103079.2	103079.2	103079.2
2300	9151.5	30.9	30899.0	24696.1	77309.4	103079.2	103079.2	103079.2
2400	7268.5	23.6	44589.8	42949.7	103079.2	103079.2	103079.2	103079.2
2500	8193.7	25.5	44081.3	38654.7	98784.2	103079.2	103079.2	103079.2

Table 5.3. The result of the stress test conducted at a given workload.

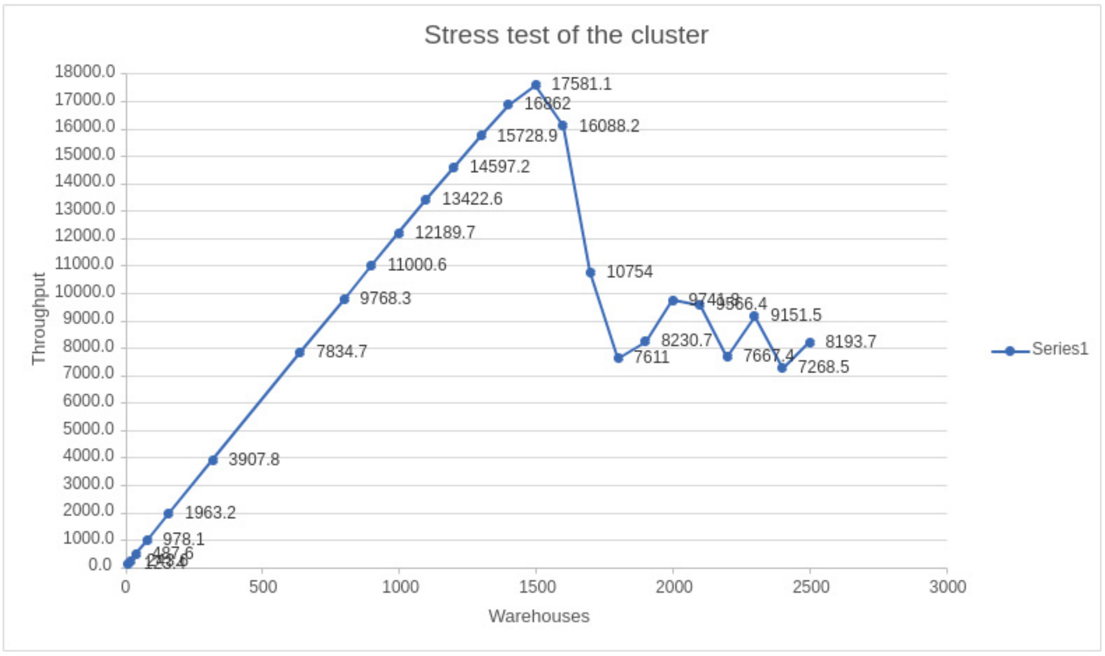


Figure 5.8. The result of the stress test conducted at a given workload.

The result of the stress test with the cluster configuration Table 3.3 follows the expected result. With a workload of 10-1300 warehouses, the cluster configuration is capable of dealing with the transactions as expected. With higher workload (warehouses) the throughput (tpmC) is increasing linearly, as shown in Figure 5.8.

However, something interesting happens when the SUT is pushed too high that it is getting saturated. When the workload is increased over 1500 warehouses the EFC (the theoretical maximum) decreases rapidly. This is a possible outcome when the workload is too heavy and the system becomes saturated. According to CockroachLabs [13], this could be because "*...the overload could allow transaction contention to compound*", which would make sense in our case with a heavy workload and a saturated system. Any further analysis of this outcome was not conducted, due to it being outside the scope of this thesis.

A performance test was conducted to measure the throughput given a different cluster configuration in terms of allocated resources for each node. For this test, the cluster was configured to have three static nodes and then scaled up from a Small, Medium, and Large configuration, shown in Table 3.2.

5.2.2 Scaling up

Warehouses	Resources	Throughput (tpmC)	EFC (%)	avg (ms)
100	Small	1220	94.9	49
	Medium	1218	94.7	64
	Large	1225	95.3	37
300	Small	3660	95.0	127
	Medium	3669	95.1	61
	Large	3662	94.9	40
500	Small	5974	92.9	571
	Medium	6098	94.8	68
	Large	6102	94.9	45
700	Small	3772	41.9	25128
	Medium	8558	95.1	82
	Large	8568	95.2	52
900	Small	2465	21.3	58792
	Medium	10966	94.7	153
	Large	11025	95.3	67
1100	Small	2896	20.5	54343
	Medium	11345	80.2	3957
	Large	13455	95.1	97
1300	Small	2575	15.4	74406
	Medium	7241	43.3	24780
	Large	15796	94.5	207
1500	Small	2901	15.0	74410
	Medium	7327	38.0	30360
	Large	18184	94.3	261

Table 5.4. The performance with various workload and with a different cluster configuration

By analyzing the result it is possible to determine that the Small configuration is capable of dealing with the workload up and until 500 warehouses, shown in Table 5.4. At the 500 warehouses mark, it is detectable that the SUT with a Small configuration is about to get saturated, based on the rapidly increased latency.

At 700 warehouses, the Small configuration has become saturated, which is detectable in the decreased amount of throughput, low efficiency (the theoretical maximum), and high latency. The Medium and Large configuration is working unhindered at this mark. The same pattern repeats at 1100 warehouses, but now with the Medium configuration becoming saturated. See Appendix B for a graph of the measured throughput at the different workloads and resources.

5.2.3 Scaling out

Warehouses	# Nodes	Throughput (tpmC)	EFC (%)	avg (ms)
100	3	1218	94.7	64
	6	1223	95.1	56
	9	1219	94.8	55
300	3	3669	95.1	61
	6	3658	94.8	61
	9	3673	95.2	49
500	3	6098	94.8	68
	6	6133	95.4	46
	9	6109	95.0	47
700	3	8558	95.1	82
	6	8565	95.1	45
	9	8560	95.1	48
900	3	10966	94.7	153
	6	11011	95.1	49
	9	11012	95.1	46
1100	3	11345	80.2	3957
	6	13465	95.2	54
	9	13457	95.1	54
1300	3	7241	43.3	24780
	6	15899	95.1	68
	9	15934	95.3	50
1500	3	7327	38.0	30360
	6	18301	94.9	94
	9	18339	95.1	55

Table 5.5. The performance with a given workload and with three, six, and nine nodes.

When scaling out the system from three nodes to six nodes, and again to nine nodes, it keeps the same throughput throughout the different scaling settings, shown in Table 5.5. The tests were run with different workloads and as long as the system was not saturated and could handle the workload, the throughput was not affected when scaled out.

This result was expected when scaling out the system. All the nodes in the CockroachDB cluster are configured with the same read and write privileges and can help balance the workload without extra overhead on copying data between nodes. In CockroachDB there is no single node being the only one with write privileges, which would lead to data being copied across nodes after a write, as could be the case in other databases. This overhead could lead to a throughput decrease when scaling out the system. Since CockroachDB does not have this extra overhead there is little chance of throughput decrease when scaling out the system, which is also what is shown in Table 5.5. [5]

See Appendix C for a graph of the measured throughput at the different workloads and scaling levels.

5.3 RQ3: What are the procedures for version hot-swapping?

In this section, the results and analysis of the third research question are presented.

In this test, the cluster was running a TPC-C workload while it was changing versions, known as hot-swapping. The workload was initialized with the command line, shown in Figure 5.9. The various settings [26] simulate traffic while conducting the version hot-swapping.

```
$ kubectl run workload-run it --image=cockroachdb/cockroach:latest
--rm --namespace=[namespace-name] -- workload run tpcc --warehouses=20
--ramp=3m --duration=10m
‘postgresql://root@[deployment-name]-cockroachdb-public:26257?sslmode=
disable’
```

Figure 5.9, Command-line for running the TPC-C workload towards the cluster for 10 minutes, using Kubectl.

While there was a constant workload running on the cluster, the configurations regarding the image version were changed.

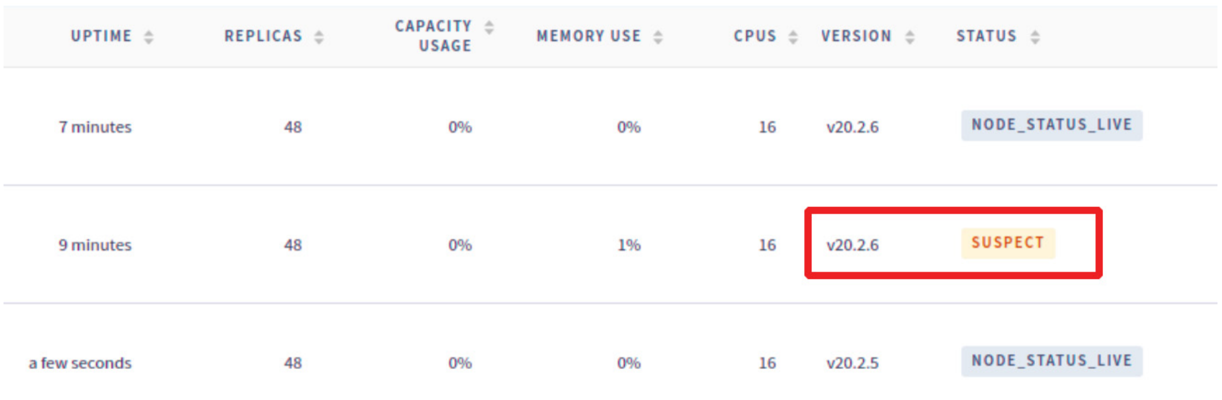
The cluster managed the version change by taking down one node at a time and updating the version for that same node. When one node was updated with the new version, another node was taken down. During this process, the workload kept running on the cluster.

Changes in version to a newer version as well as back to an older version were made with the same result of the cluster updating its version one node at a time.

The changes in the version were easily made by editing the configuration file of the cluster. And as described above this could be easily done without manually having to take down the nodes, or affecting the workload on the cluster.

Figure 5.10 shows an example of how one node at a time is taken down and changed to another version. In this example, the cluster has version 20.2.6 to start with and is changed to 20.2.5 by editing the configuration file. The cluster has finished upgrading the bottom node that has the desired version of 20.2.5, and a second node is now down and under the progress of updating its version.

In the event of a failure to update a node, the remaining nodes will wait before going down themselves. This happened one time during the testing, in which the limits to download an image container were reached. This was unrelated to how CockroachDB operates and was instead an external factor. The cluster kept the node down and operated on the remaining nodes.



UPTIME	REPLICAS	CAPACITY USAGE	MEMORY USE	CPUS	VERSION	STATUS
7 minutes	48	0%	0%	16	v20.2.6	NODE_STATUS_LIVE
9 minutes	48	0%	1%	16	v20.2.6	SUSPECT
a few seconds	48	0%	0%	16	v20.2.5	NODE_STATUS_LIVE

Figure 5.10 State of the cluster when downgrading from version 20.2.6 to version 20.2.5.

5.4 RQ4: How resilient is CockroachDB in the presence of node disruptions?

In this section, the results and analysis of the fourth research question are presented.

During this test, one node was intentionally killed to verify that the inserted rows were written to the cluster, despite that the cluster was disrupted and in a recovery state. When the insert operations had been completed, a randomly selected row was queried to see if it had been written or not. This was done by comparing row number to id number, which should be identical in this test configuration. The killed node was back online only after a couple of seconds after it had self-repaired.

Inserted Data	Checks		
# of inserts	Dead node?	# of rows after inserts	Row # = id #
2000	Yes	2000	Yes (test# 245)
4000	Yes	4000	Yes (test# 3333)
6000	Yes	6000	Yes (test# 5420)
8000	Yes	8000	Yes (test# 7001)
10000	Yes	10000	Yes (test# 10000)

Table 5.6 Insertion of rows and consistency check in a three-node cluster with one node fail.

Table 5.6 shows that the intentional node failure of one node in the cluster was successful in all runs, and the node was down during the insertion of rows in the database table. It also shows that after each test run all of the inserted rows got inserted into the database, no matter the number of rows inserted. The sample testing of each run shows that the tested row has the same id number as the row count which indicates consistency in the data inserted.

5.5 Discussion

An interesting discussion came up regarding the consistency of CockroachDB. This is not an aspect that we explicitly addressed in RQ2, but a topic that we indirectly tested and confirmed, due to the fact that the TPC-C workload was used. The TPC-C workload consists of transactions, therefore, CockroachDB would have crashed while trying to run the traffic, if it had not supported transactions. This was something that we had not considered when we decided to go for TPC-C workload, but that came apparent after discussion with Ericsson and when we gained further knowledge about both distributed databases, and the implications of TPC-C workload.

6 Conclusion

In this thesis, we examined CockroachDB as a distributed database in a cloud-native environment - Kubernetes. The result shows that CockroachDB scales easily both vertically and horizontally, can handle version hot-swapping, and is resilient against node failures.

The procedures of scaling the cluster vertically and horizontally are easily done and can be executed without taking down the cluster. Scaling vertically is carried out by modifying the configuration file to allocate more resources to a node. Scaling horizontally is accomplished through the Kubectl command-line tool by specifying the number of nodes that should form the cluster.

In terms of throughput, CockroachDB scales effectively. When the workload is within what the cluster can cope with, the throughput stays consistent as the cluster is scaled vertically as well as horizontally, with no noticeable indication of overhead for read-/write-operations. With a larger workload, the throughput increases with an increased number of nodes or more resources allocated.

CockroachDB also handles version hot-swapping. The version of CockroachDB can be changed while there is still traffic running on the cluster. CockroachDB automatically brings down one node at a time, updates the version, and is re-deployed. Simultaneously as the node is brought down for maintenance, the traffic is still handled by the remaining nodes. The tests performed included updating the version to a newer version as well as rolling back to a previous one.

One of CockroachDB's characteristics is its robustness, hence the name. In this study, the robustness of the system was tested by taking down one node in a three-node cluster while the cluster was still receiving traffic in the form of writes to the database. The purpose was to see if CockroachDB managed to recover itself and if the writes conducted, while a node was down, were executed. CockroachDB automatically repaired itself each time a node was brought down. The write-operations that were run against the database during the node failure were also confirmed to have been written to the disk.

7 Validity threats

During the execution of this study, the aspects listed below can threaten the result.

Shared resources
The Kubernetes hosted by Ericsson is vast and has multiple clusters within it, and at the same time, many teams and projects are using it daily. To organize this in a structured way, the use of namespaces comes in handy. Namespaces are a way to organize clusters into virtual sub-clusters, in which each is logically separated from others. However, since it is a sub-cluster, to run the tests reported in this thesis we had to share the same resources with other namespaces working in the same cluster.

For the stress test and performance test, we would consume a lot of these shared resources which would impact other projects. To mitigate this, a forewarning was recommended for other teams and projects to either approve or disapprove scheduled tests.

Accessibility
This is connected to the same issue as above. Since Kubernetes is hosted by Ericsson, access is very limited. During the research, it was problematic to access and control the pods on which the CockroachDB cluster was running. The pods that were used did not have any text editor, nor did we have the user privileges to install one, so creating and running scripts had to be done by piping with Kubectl remotely from localhost. This was time-consuming and led to extra latency since it had to be controlled remotely.

Kubernetes
Our research has been conducted by running CockroachDB in a provided Kubernetes namespace. Scaling the cluster and changing versions performed in research question one and three, has therefore been done on a Kubernetes level. This means that the procedures of how CockroachDB handles scaling and version changes are controlled by Kubernetes.

Knowledge and experience
Neither of us had any in-depth knowledge of databases, and the differences in their architecture compared to other databases, prior to this study, and neither of us had worked with Kubernetes before. It took us quite a long time to grasp the basic concepts of Kubernetes and how to deploy a functional CockroachDB-cluster that we could use for our experiments. We tried several different approaches in deploying the database, but since we only borrowed a portion (namespace) of Ericssons Kubernetes, we had limited deployment options. This was something we found out the hard way, with trial and error.

When the database was finally up and running we also struggled in how to actually address our research questions in order to answer them accordingly. We cannot guarantee that the setup that we used for the experiment was deployed and running as optimized as possible, there could be undetected overhead that had an impact on the performance.

Bash script for testing node disruption resilience
For testing the resilience of the cluster we deviated from the former used TPC-C tool to generate the workload. This was done because we wanted to know exactly which write-operations that we performed, so that we could validate the data during, and after the node failure.

The bash script that we created had only write-operations, and was not as extensive as the TPC-C tool. The bash script checks that a node had gone down and that the right amount of writes had been inserted to the table. A more sophisticated tool may have been able to check more thoroughly that the data had been inserted correctly.

8 Future Work

Scaling in and down

This study did not include scaling in and down due to time constraints and prioritizing. This could be complemented in research question 1, in which only scaling out and up was conducted.

Throughput when saturated system

During the stress test in research question 2, the system became saturated when the workload was too high for the system to cope with. This was an unexpected outcome that was outside of the scope and was not further investigated. The study could be complemented with further analysis and give an understanding of why this occurred.

Performance during rebalancing

The rebalancing of ranges between nodes when a cluster is scaled was measured in time. In future work, it could also be measured what happens to the performance of the cluster during this period, and if it gets affected by the scaling process.

Effect on data and/or performance during version hot-swapping

Regarding configuration changes, like version hot-swapping, it could be interesting to see if the data stays consistent through the changes. It could also be of value to see if the performance; for example, the throughput or latency is affected while updating the version.

Time until complete version change

During the version hot-swapping procedure, there could be interest in knowing how long it takes for a system to change from one version to another.

Effect on efficiency during node recovery

The same effect could be of interest regarding the robustness of CockroachDB, where one could examine if the efficiency is affected while the system is recovering itself after a node failure.

9 References

[1] D. Kelly, “A Brief History of Databases”, cockroachLabs.com, October 27, 2020. [Online]. Available: A Brief History of Databases (cockroachlabs.com). [Accessed February 22, 2021].

[2] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks”, IBM Research Laboratory, San Jose, CA, USA, 1970.

[3] J Walker, “What is Distributed SQL? An Evolution of the Database”, cockroachLabs.com, August 18, 2020. [Online]. Available: https://www.cockroachlabs.com/blog/what-is-distributed-sql/ [Accessed February 22, 2021]

[4] Sean Loiselle, Alex Robinson, “Database Scaling Strategies: A Practical Approach”, Feb 8, 2018, [Online], Available: https://www.cockroachlabs.com/blog/scaling-distributed-database/. [Accessed March 1, 2021]

[5] CockroachDB, “Reads and writes in CockroachDB”, cockroachLabs.com, [Online], Available: https://www.cockroachlabs.com/docs/stable/architecture/reads-and-writes-overview.html. [Accessed February 22, 2021]

[6] Bram Gruneir, “Distributed SQL (NewSQL) Made Easy: How CockroachDB Automates Operations”, October 5, 2017, cockroachLabs.com, [Online], Available: https://www.cockroachlabs.com/blog/automated-rebalance-and-repair/. [Accessed March 1, 2021]

[7] “TPC-C”, tpc.org [Online]. Available: http://www.tpc.org/tpcc/. [Accessed February 22, 2021]

[8] Transaction Processing Performance Council (TPC), TPC Benchmark C Standard Specification, Revision 5.11, February 2010.

[9] James C et al. “Spanner: Google’s Globally Distributed Database”, googleapis.com, August 2013. [Online], Available: : https://storage.googleapis.com/pub-tools-public-publication-data/pdf/65b514eda12d025585183a641b5a9e096a3c4be5.pdf [Accessed March 16, 2021]

[10] Spender Kimball and Irfan Sharif, “Living without Atomic Clocks", cockroachlabs.com, April 21, 2020. [Online], Available: https://www.cockroachlabs.com/blog/living-without-atomic-clocks/ [Accessed March 16, 2021]

[11] Matt Tracy. “How CockroachDB does Distributed, Atomic Transactions” cockroachlabs.com. September 2, 2015. [Online], Available: https://www.cockroachlabs.com/blog/how-cockroachdb-distributes-atomic-transactions/ [Accessed March 16, 2021]

[12] Yishan Li, Sathiamoorthy Manoharan, “A performance comparison of SQL and NoSQL databases”, Department of Computer Science University of Auckland New Zealand, August 27, 2013

[13] CockroachLabs, “Help interpreting TPC-C performance with CockroachDB”, February 10, 2021, forum.cockroachLabs.com [Discussion post], Available: https://forum.cockroachlabs.com/t/help-interpretating-tpc-c-performance-with-cockroachdb/4273

[14] N. Leavitt, "Will NoSQL databases live up to their promise?" Computer, Vol. 43, no. 2, pages: 12-14, February 8, 2010.

[15] CockroachDB, “Life of a distributed Transaction”, cockroachLabs.com, [Online], Available: https://www.cockroachlabs.com/docs/stable/architecture/life-of-a-distributed-transaction.html, [Accessed March 16, 2021]

[16] Nathan VanBenschoten, “Parallel commits: An Atomic Commit Protocol For Globally Distributed Transactions”, cockroachLabs.com, November 7, 2019. [Online], Available: https://www.cockroachlabs.com/blog/parallel-commits/, [Accessed March 16, 2021]

[17] Martin Heller, “CockroachDB review: A scale-out SQL database built for survival”, infoworld.com, January 14th, 2018. [Online], Available: https://www.infoworld.com/article/3244138/cockroachdb-review-a-scale-out-sql-database-built-for-survival.html, [Accessed March 22, 2021]

[18] CockroachLabs, “DB console overview”, cockroachLabs.com, [Online], Available: https://www.cockroachlabs.com/docs/v20.2/ui-overview.html, [Accessed March 23, 2021]

[19] CockroachLabs, “Performance Report: Benchmarking CockroachDB’s TPC-C Performance”, cockroachLabs.com, [Online], Available: https://resources.cockroachlabs.com/guides-private/performance-report-benchmarking-cockroachdb-2-0, [Accessed March 23, 2021]

[20] CockroachLabs, “2021 Cloud Report”, cockroachLabs.com, [Online], Available: https://www.cockroachlabs.com/blog/2021-cloud-report/, [Accessed March 23, 2021]

[21] CockroachLabs, “2020 Cloud Report”, cockroachLabs.com, [Online], Available: https://resources.cockroachlabs.com/guides/2020-cloud-report, [Accessed March 23, 2021]

[22] Ana Hobden, “Why benchmarking Distributed Databases is So Hard”, pingcap.com, July 8, 2021 [Online], Available: https://pingcap.com/blog/why-benchmarking-distributed-databases-is-so-hard, [Accessed March 23, 2021]

[23] Surya Narayanan Swaminathan, Ramez Elmasri, “Quantitative Analysis of Scalable NoSQL Databases”, University of Texas at Arlington, 2016 IEEE International Congress on Big Data

[24] Benjamin Anderson, Brad Nicholson, “SQL vs. NoSQL Databases: what’s the Difference?”, ibm.com, June 18, 2020 [Online], Available: https://ibm.com/cloud/blog/sql-vs-nosql, [Accessed March 24, 2021]

[25] Advertisement of CockroachDB, A scalable database for global deployments, *CockroachLabs*, https://cockroachlabs.com/product/scale

[26] CockroachLabs, “Performance Benchmarking With TPC-C”, cockroachLabs.com, [Online], Available: https://www.cockroachlabs.com/docs/v20.2/performance-benchmarking-with-tpcc-large.html, [Accessed February 2021]

[27] Jörn Kuhlenkamp, Markus Klems, Oliver Röss, “Benchmarking scalability and elasticity of distributed database systems”, Proceedings of the VLDB Endowment, Vol 1, no. 2, August 2008.

[28] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. G´omez-Villamor, V. Munt´es-Mulero, and S. Mankowskii. Solving Big Data Challenges for Enterprise Application Performance Management. Proceedings of the VLDB Endowment, 5(12):1724–1735, 2012.

[29] Kubernetes, [Online], Available: https://kubernetes.io/, [Accessed 2021]

[30] Lens, [Online], Available: https://k8slens.dev/, [Accessed 2021]

[31] PostgreSQL, “FAQ”, Postgresql.org, [Online], Available: https://wiki.postgresql.org/wiki/FAQ, [Accessed 2021]

[32] Nathan VanBenschoten, “CockroachDB: Architecture of a Geo-Distributed SQL Database”, Cockroach Labs, [Online], Available: https://www.datacouncil.ai/talks/cockroachdb-architecture-of-a-geo-distributed-sql-database, [Accessed 2021]

[33] Cockroach Labs, “Important concepts”, Cockroach Labs, [Online], Available: https://www.cockroachlabs.com/docs/v20.2/architecture/reads-and-writes-overview.html, [Accessed 2021]

[34] IBM, “What is the CAP theorem?”, IBM, [Online], Available: https://www.ibm.com/cloud/learn/cap-theorem, [Accessed 2021]

[35] Ben Darnell, Cockroach Labs, “The Limits of the CAP theorem”, Cockroach Labs, [Online], Available: https://www.cockroachlabs.com/blog/limits-of-the-cap-theorem/, [Accessed 2021]

[36] Karambut Kaur and Dr. Monika Sachdeva, “Performance Evaluation of NewSQL Databases”, [Online], Available: https://ieeexplore-ieee-org.miman.bib.bth.se/stamp/stamp.jsp?tp=&arnumber=8068585, [Accessed May 2021]

[37] GeeksforGeeks, “Eventual vs Strong Consistency in Distributed Databases”, [Online], Available: https://www.geeksforgeeks.org/eventual-vs-strong-consistency-in-distributed-databases/, [Accessed May 2021]

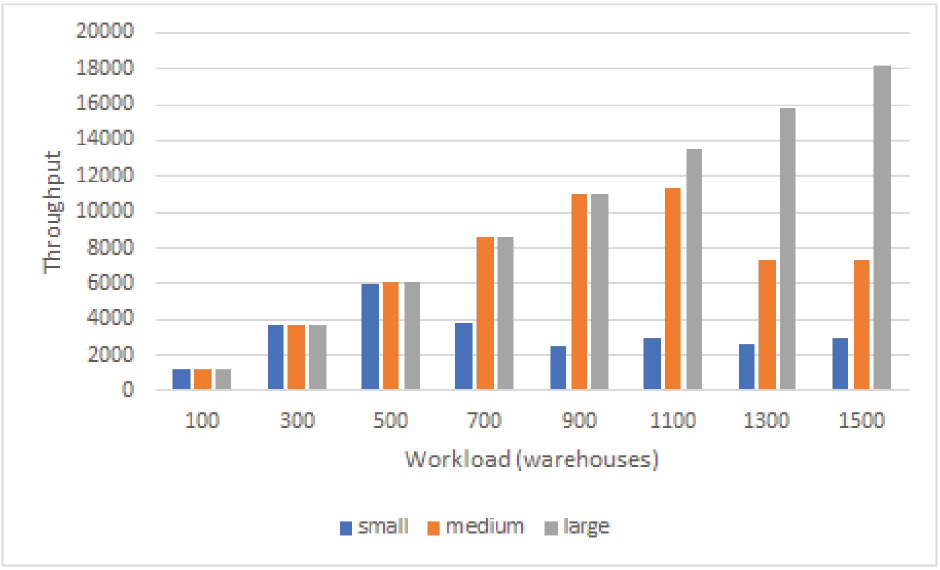
[38] Mitko Radoev, “A comparison between Characteristics of NoSQL databases and Traditional Databases”, [Online], Available https://www.hrpub.org/download/20171130/CSIT1-13510351.pdf, [Accessed May 2021]

10 Appendix

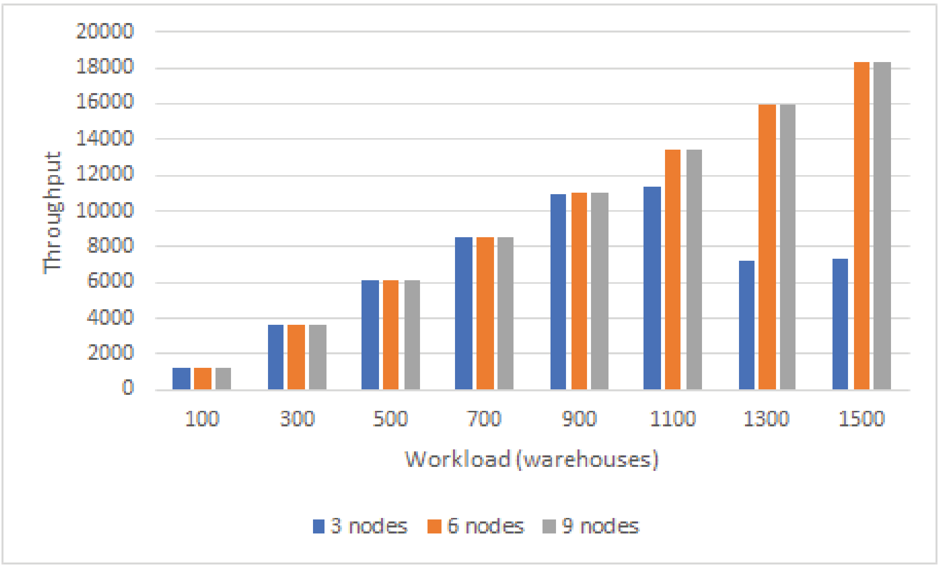
Appendix A

Link to the chaos-test:
https://github.com/krhk18/thesis-crdb

Appendix B



Appendix C


Throughput
Workload (warehouses)
3 nodes
6 nodes
9 nodes