Tampere University

Timo Moilanen

SMALL-SCALE APACHE CASSANDRA CLUSTER AS A HIGH AVAILABILITY DATABASE

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Jukka Vanhala
Karri Palovuori
June 2023

ABSTRACT

Timo Moilanen: Small-scale Apache Cassandra cluster as a high availability database
Master of Science Thesis
Tampere University
Master's Degree Programme in Electrical Engineering
June 2023

This thesis investigates the usage of a small-scale Apache Cassandra cluster as an on-premises database solution, focusing on single node and small multinode configurations. The reason why small-scale clusters were investigated is because they are not that well researched compared to larger cluster. The primary objectives of this thesis is to investigate typical use cases that are encountered during long-term usage of Cassandra, and how to manage them in a way that availability of the database is not compromised.

A series of tests were conducted as a part of a real-world application that utilizes Cassandra as a time-series database. The tests went over aspects such as scalability, transition from single node to multinode cluster, short-term and long-term node disruptions, software and operating system upgrades, and establishing important diagnostics for monitoring cluster's health and performance. Using the cluster as a part of a real-world application provided a more accurate representation of the situations encountered in actual deployments, as opposed to relying on synthetic benchmarks or simulations.

The results demonstrated that it was reasonably straightforward to handle the different situations when following best practices. However, some issues were encountered which required manual work. Notably, the main challenges during node disruption tests were related to insufficient network capacity when using multinode configuration, which led to database repair failures and dropped connections under heavy read loads. After the network infrastructure was upgraded, problems in that regard stopped, but too large tables still caused repairs to end up in timeouts. In a single node cluster, these issues are not as relevant, as the network overhead is significantly lower and repairs do not need to be run.

On the client side, issues maintaining high availability were encountered a couple of times, which caused temporary failures in client requests during server disconnection tests. Although the exact cause for those problems was not investigated during the tests, it highlights the importance of proper configuration of the client-to-cluster communication to prevent such situations from happening in production deployments. In other cases, maintaining high availability was simple when the cluster was configured correctly.

The scalability tests demonstrated that scaling a single node cluster to multinode enhances performance while also providing fault tolerance benefits. On average, a three node cluster was able to handle 94% more requests than a single node cluster, and a four node cluster further improved the performance by 22% when compared to the three node cluster. The methods of performing software updates and single node to multinode transitions were also explored in this thesis. The tests were able to identify reliable techniques to ensure smooth system upgrades in general cases.

Keywords: Apache Cassandra, high availability, NoSQL-database, time-series, scalability

TIIVISTELMÄ

Timo Moilanen: Pienikokoinen Apache Cassandra klusteri korkean saatavuuden tietokantana
Diplomityö
Tampereen yliopisto
Sähkötekniikan DI-ohjelma
Kesäkuu 2023

Tässä diplomityössä tutkitaan pienikokoisen Apache Cassandra klusterin käyttämistä paikallisena (on-premises) ja korkean saatavuuden omaavana tietokantaratkaisuna. Kohteena oli yhden ja muutaman koneen kokoiset klusterit, joista aikaisempaa tutkimusta löytyy vähän verrattuna suurempiin klustereihin. Työn tavoitteena on tutkia tyypillisiä Cassandran pitkäaikaisen käytön aikana ilmeneviä käyttötapauksia ja niiden hallintaa siten, että tietokannan saatavuus ei vaarannu.

Työssä suoritettiin erilaisia testejä Cassandran ollessa osana sovellusta, joka käyttää sitä aikasarjatietokantana. Testit käsittelivät muun muassa skaalautuvuutta, yhden solmun klusterin päivittämistä monen solmun järjestelmäksi, lyhyt- ja pitkäaikaisia häiriötilanteita, ohjelmisto- ja käyttöjärjestelmäpäivityksiä sekä klusterin monitoroinnin kehittämistä.

Tulokset osoittivat, että eri käyttötapausten käsittely oli suoraviivaista, kun noudatti Apachen ilmoittamia parhaita käytäntöjä. Testeissä kuitenkin ilmeni ongelmia, jotka vaativat manuaalista työtä. Suurimmat haasteet palvelinten häiriötilannetesteissä liittyivät verkkokapasiitettiin, mikä johti tietokannan korjausoperaatioiden epäonnistumiseen ja yhteyksien katkeamiseen suurilla lukukuormilla. Verkkoinfrastruktuurin päivitettyä ongelmat siltä osin loppuivat, mutta liian suurikokoiset taulut aiheuttivat edelleen korjausten epäonnistumisia. Yhden solmun klusterissa nämä ongelmat eivät ole yhtä merkityksellisiä, sillä se vaatii huomattavasti vähemmän verkkokapasiteettia eikä kannan korjausoperaatioita tarvitse suorittaa.

Asiakasohjelmapuolella korkean saatavuuden ylläpitämisessä koettiin muutaman kerran ongelmia, joiden takia luku- ja kirjoituspyynnöt epäonnistuivat kahden solmun ollessa päällä. Tarkkaa syytä tälle ei testien aikana löydetty, mutta se korostaa asiakasohjelman ja klusterin välisen yhteyden oikeanmukaisen konfiguroinnin tärkeyttä, jotta tällaisilta tilanteilta vältytään tuotantokäytössä. Muissa tapauksissa korkean saatavuuden ylläpitäminen oli yksinkertaista, kun järjestelmä oli konfiguroitu oikein.

Skaalautuvuustestit osoittivat, että yhden solmun klusterin laajentaminen useamman solmun järjestelmäksi parantaa vikasietokykyä, mutta myös lisää suorituskykyä merkittävästi. Kolmen solmun klusteri kykeni keskimääräisesti suorittamaan 94% enemmän lukupyyntöjä kuin yhden solmun klusteri. Neljän solmun klusteri paransi suorituskykyä keskimääräisesti vielä 22% verrattuna kolmen solmun klusteriin. Työssä myös tutkittiin ohjelmistopäivitysten tekemistä, ja yhden solmun muuntamista useamman solmun klusteriksi. Kyseiset testit pystyivät tunnistamaan luotettavat ohjeet varmistamaan sujuvat päivitykset normaaliolosuhteissa.

Avainsanat: Apache Cassandra, korkea saatavuus, NoSQL-tietokanta, aikasarja, skaalautuvuus

PREFACE

First, I would like to thank my employer Insta and the client for providing me the opportunity to work on this thesis. I found the topic to be challenging, but also very interesting because I did not have experience with the operation of distributed database systems beforehand. Through this project, I was able to gain very valuable experience in this area, for which I am grateful.

I would like to especially thank Marko Kajaniemi, Sami Merilä, and Timo Harju for supporting me throughout the thesis, and providing me a strong basis for the topic right from the start. While I was given free hands to investigate and test the different use cases, they were always available to provide guidance and help me not getting sidetracked to unrelated issues. I would not have been able to complete the thesis within due time without their advice and feedback. Big thanks to Ville Hietanen and other colleagues for helping me get the testing environment running, troubleshooting issues, and helping me with numerous other things. Finally, I want to thank my family and friends who have supported me during these years of studies.

Tampere, 6th June 2023

Timo Moilanen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ACID-principle	Acronym standing for Atomicity, Consistency, Isolation and Durability. ACID is a set of properties that ensure database transaction reliability.
CAP-theorem	Theorem that states that a distributed system can simultaneously only have two out of three guarantees of consistency, availability, and partition tolerance.
CL	Consistency Level
Cluster	A collection of one or more nodes that form the Cassandra database.
CommitLog	A file that logs all mutations to the database. The logs are used to recreate Memtables in the case of unexpected shutdown.
CQL	Cassandra Query Language
DBMS	Database management system
DC	Datacenter
DMS	Database Middleware Service
GC	Garbage Collection
GiB	Gibibyte
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KiB	Kibibyte
Memtable	In-memory data structure containing the latest writes to the database. Memtables are periodically flushed to SSTables.
MiB	Mebibyte
Mutation	Insertion, deletion, or deletion operation
Node	A Cassandra instance.
NoSQL	A database that does not use the traditional relational model for its data.
Partition	Tables are divided into partitions according to the partition key. All data of a partition resides in the same node.

Primary key	A unique identifier for each row in a database table. A primary key consists of one or more partition keys and zero or more clustering keys.
RDBMS	Relational database management system
Relational Database	A database that stores data in tables of rows and columns, and allows data to be related between tables.
RF	Replication Factor
Snitch	Component that determines the cluster's topology, enabling efficient request routing and replica placement.
SQL	Structured Query Language
SSTable	An immutable file format used by Cassandra to store data on disk.
TTL	Time-To-Live
Vnode	Virtual node
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1. INTRODUCTION

Scalable, fast, and high availability database management systems have become increasingly important in today's digital age. As the amount of data generated and stored is growing, companies face new challenges in how to effectively store, retrieve, and manage this data. The rise of modern applications has increased demand on system reliability and recovery time, creating significant pressure on the need to improve the capabilities of database management systems. High availability is now a vital part of these systems, as businesses require 24/7 access to their data, and in some cases only allowing minutes of downtime per year. Downtime in this context could lead to significant loss of revenue as critical data can be lost.

Relational databases have been the most commonly used technology for data storage. However, they were found to struggle with the scalability and performance demands of big data. These problems have been tried to address in different ways, but each approach has had its own set of challenges. Vertical scaling, achieved by buying better hardware, has been expensive and has only offered temporary relief. Horizontal scaling, accomplished by adding additional servers in a cluster of computers, has brought up the complications of data replication and consistency, which has been difficult to handle in these systems. To make databases perform better, queries have been optimized, new caching methods have been implemented, and data denormalized. However, these approaches have created even more consistency issues and increased the complexity of managing relational data. These obstacles have made it difficult to achieve high availability in a reliable and efficient manner. All of this led to the need for new solutions in early and mid-2000s. [1, pp. 3–4]

To meet these new demands, NoSQL databases, like Apache Cassandra, have emerged as a popular solution. Their main points being performance, scalability, availability, and fault tolerance. These databases are designed to handle large amounts of data across multiple servers and provide high availability through data replication. The distribution of data makes it possible that even if some servers become unavailable, data loss does not occur.

2. NOSQL DATABASES

The main difference between NoSQL, standing for "Not Only SQL", and relational databases is in how the data is organized and stored, as well as the relationships between the data. This affects how queries are executed, what their performance is, and the ease of scalability. The most common features of NoSQL databases include a simpler and more flexible data model, which has the ability to speed up processing and allow handling of unstructured data, easier scaling to multiple machines in a distributed manner, granular availability control, and the ability to handle vast amounts of data. The trade-offs and challenges include the lack of support for RDBMS ACID transactions (atomicity, consistency, isolation, and durability). As a consequence, some reliability is lost, but there are often ways to mitigate this. Consistency of data requires some configurations, and strong consistency can be difficult to implement, which can be a problem for some applications. NoSQL databases lack a unified language (cf. SQL), resulting in a wide range of query languages and ways to access data. Support for complex queries is often limited due to the nature of the data. [2, pp. 216–220]

NoSQL databases have the shared feature of being non-relational, but beyond that there is a diverse range of different kinds. Some of the most popular categories can be classified into the following [2, p. 218]:

- Key-value database (Amazon's DynamoDB, Riak)
- Column-oriented database (Cassandra, BigTable)
- Document database (MongoDB)
- Graph database (Neo4j)

Key-value databases store key-value pairs where the key maps to the value using a hash table. Values can be anything from simple to complex compound objects. Column-oriented databases store data tables by column rather than by row. This make targeting specific columns more efficient and is effective option for complex datasets due to its scalability. Document based databases store data as documents which can contain more complex data structures and relationships. A document can contain for example multiple key-value pairs, arrays, or nested documents, often stored in JavaScript Object Notation (JSON), Extensible Markup Language (XML), or YAML Ain't Markup Language (YAML) format. Graph databases store and represent data as graphs, consisting of nodes

(entities) and edges (relationships). Graph databases are used to focus on relationships between data rather than the data itself. [2, p. 218]

3. APACHE CASSANDRA

Apache Cassandra is one of the databases that emerged in the NoSQL wave in the late 2000s. It was originally developed at Facebook in 2007 where it was used to improve the performance of its inbox search functionality. In 2008, Facebook published it as an open-source project, later in 2009 Cassandra was moved to an Apache Incubator project, and finally in 2010 it was voted into an Apache top-level project. [1, pp. 28–29] Based on information from 2021, some of the largest production deployments include Apple with more than 160 000 instances that store more than 100 petabytes of data across more than 1000 clusters. Netflix has more than 10 000 instances that managed millions of transactions a second. [3] In 2012, eBay handled more than 400 million writes and 100 million reads per day using Cassandra [4]. In 2022, Discord had a 177 node system that was used to store trillions of messages [5].

3.1 Cassandra as a distributed and decentralized database

Apache Cassandra is an open-source, distributed and decentralized NoSQL-database. Its aim is to be easily scalable to large amounts of data and to be highly available without compromising its read and write performance. The data model is based on Google's Bigtable [6] database and Architecture on Amazon's DynamoDB [7] database. Modern versions of Cassandra use CQL as its formal query language, which is similar to SQL syntax. [1, p. 17]

Cassandra is distributed, which means that it can be run on multiple computers physically located in separate places. There is no central master server, and the whole distributed system appears as a unified whole to the client applications. This collection of machines is also called a cluster. A single machine in the cluster is called a node, which is an instance of the database, and stores a portion of the overall data. [8] Although Cassandra can run on a single computer as a single node, it misses the advantages of running a multinode system, which Cassandra is specifically designed for.

Scalability is claimed to be linear in nature, which implies that as the number of nodes in the cluster increases, the system's performance and capacity increases proportionally, regardless of the size of the cluster. A client application can send a data write request to any of the nodes, and Cassandra is able to synchronize it between the whole cluster.

[8] In a production environment, this indicates that if at some point the data throughput requirements of the database get bigger, you only need to add more machines to the cluster, which can be done without downtime or interruption. This concept is called horizontal scaling while vertical scaling is adding better hardware to existing machines [1, p. 16]. Moreover, Cassandra is designed to take full advantage of multiple processing units to improve its performance and optimizing for high write throughput [1, p. 24].

There is no single point of failure in the database, as the cluster is decentralized, meaning that every node is similar and can do anything that any other node can. However, they do have some short-term specialized responsibilities, but they are not limited to any node. Internally, requests to the cluster are load balanced to generate better performance. This design improves general database availability, as the system does not rely on a "master" node like many distributed database solutions. The way how Cassandra manages this is also called peer-to-peer design, which will be discussed further in Section 3.3. In the case where a node drops out of the network, applications do not notice a difference if the data accessed has been replicated from the offline node to another node. The traffic intended for a failed node is forwarded to another node with the replicated data to handle. This network of independent processing nodes with access to a replicated database is of type active/active database [9]. In Cassandra, the occurrence of node and zone failures is an expected scenario, and the system is designed against these incidents to achieve high availability.

The mechanism of data replication is done based on the configurations set. For example, a data replication factor of three means that every piece of data resides as three copies in three different nodes, and it is safe to read either of those even if one of the nodes is down. In comparison to the master-slave architecture, where a single master node manages the whole cluster consisting of slave nodes, the functioning of the whole database can be hindered if the master node goes offline. [1, p. 15]

On the surface, it may seem that Cassandra is able to and should be used everywhere, but upon delving deeper, there are situations where its utilization may not be the best choice due to the trade-offs that are required to make Cassandra's features possible. If a project does not need the benefits of running a multinode system, which Cassandra is specifically engineered towards, it may be more suitable to investigate different databases depending on the use case. It might be better to go for a relational database system like MySQL or PostgreSQL as opposed to NoSQL options if the application uses small or medium data. Medium data can be described to fit on a single machine with capability to serve hundreds of concurrent users and where vertical scaling is enough for most cases. Relational databases guarantee that consistency is always met through ACID compliance [1, p. 6], but once horizontal scaling is taken into action, things get complicated quickly. In those situations, Cassandra might be a good option.

Cassandra does not provide full ACID compliance, which makes it not compatible with applications that require it, for example, a banking application [10]. ACID is an acronym used to refer to a set of properties that ensure database transaction reliability [1, pp. 6–7]. In short, one of the things that ACID operations ensure is that when there are multiple simultaneous requests, they are processed sequentially, with the first request being completed before the processing of the second one begins. [1, p. 7] This maintains data integrity and consistency. Consistency means that each server returns the right response to each request, for example a read always returns the most recently written value.

With Cassandra and NoSQL databases in general, these things are not so straightforward. Instead, trade-offs have been made to prioritize other factors such as scalability and fault tolerance. Cassandra's consistency can be adjusted according to the requirements of the application to various levels. Different levels of consistency determine how many nodes should acknowledge a database operation before it is considered successful. [10] With one of the lowest consistency settings of "ONE", the operation is successful after it is completed in one of the nodes that stores replicas of that data [11, p. 134]. The data is being replicated according to the replication factor in this case, but it takes some time to fully synchronize. During that short period of time, requests may return different responses in different nodes. On the other end, consistency level can be set to "ALL", which waits for all the replicas to complete the operation [11, p. 134]. With these settings, the balance between consistency, availability, and performance can be optimized based on the requirements of the application. This design is tied to Brewer's CAP theorem [11, p. 43].

Brewer's CAP theorem ties distributed systems like Cassandra to balance between:

- Consistency
- Availability
- Partition tolerance

The theorem, presented in Figure 3.1, indicates that a distributed database system can only have two out of three of those attributes. Consistency means that everyone has an identical view of the data. With availability, every request receives a response without downtime, although it does not guarantee if it is the most recent value or not. Partition tolerance means that the system maintains operations despite any network errors in which messages are dropped. Distributed databases are often either "AP"-type by prioritizing availability over consistency or "CP"-type by prioritizing consistency over availability. Cassandra is often described as an "AP"-system, but can be configured towards a "CP"-system by replication factor and read or write consistency levels. [11, pp. 43–46, 134] More on the topic is discussed in Section 3.3.3.

Cassandra's data model is optimized for large-scale data and high-velocity workloads, but

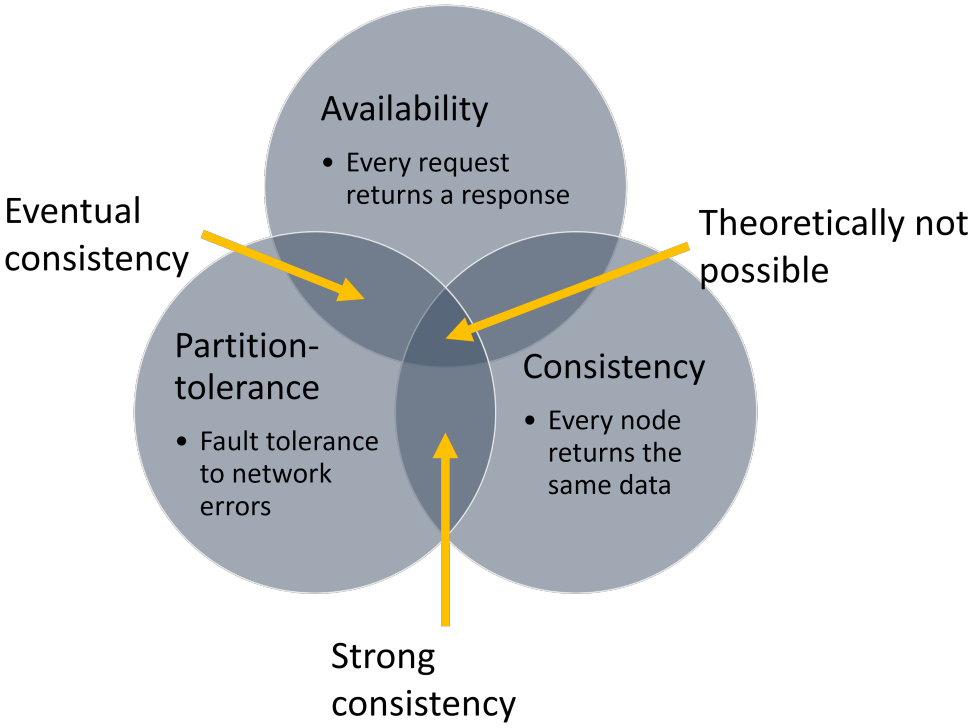

Figure 3.1. *CAP Theorem (adapted from [11, p. 46]).*

it has limitations when it comes to performing complex queries [1, pp. 83–85]. Consequently, there is a learning curve to data modeling, and it requires thoughtful consideration on how the data is stored to accommodate the used queries right from the beginning. Schema changes or query extensions can be difficult to implement and require planning and coordination.

If the system requires many writes, Cassandra might be a good fit. If it requires lots of different types of queries (i.e., many queries with different WHERE clauses) with complex multi-table relationships, further testing may be necessary to determine Cassandra's suitability. In Cassandra, there is no relational integrity between tables, which necessitates designing the table schema in such a way that all fields required for a specific query are present within a single table. This kind of query-centric model enhances query efficiency because they do not involve multiple tables. [12] As a consequence, this often leads to either handling table joins on the client side for queries that need data from multiple tables, or data denormalization by creating more tables with redundant data. Denormalization increases storage requirements and the need to handle data duplication and consistency. The good thing is that storage is cheap, and writes are fast, and in Cassandra's case denormalization is almost always the preferred way and normal technique to handle these cases. [13, p. 3]

3.2 Data model

The data model refers to the way the data is organized and stored in the database. This includes the structure of the data and the relationships between pieces of data. It also includes the way data is accessed, what types of queries can be run and what is their performance and limitations.

3.2.1 Basic Structural Elements

The data and storage engine model of Cassandra is based on Google's BigTable and uses terms that are inherited from it. The most relevant to understand the data model being [1, p. 58]:

- Row: A row represents a collection of related data. Each row is identified by a unique primary key.
- Column: Columns represent individual data points within a row.
- Table: Is the fundamental unit which defines the typed schema for a collection of partitions. Tables contain partitions that contain grouped rows.
- Keyspace: Groups related tables together and defines replication strategies for those tables.

The model can be described as a *partitioned row store*, where data is stored in multidimensional hash tables as key-value pairs. In basic terms, a column is a unit of data that consists of a key and a value, for example *user_email* as key and *johndoe@example.com* as value. This pair is then stored as a separate entry in the hash table. [1, p. 27] Additionally, each time data is written, a timestamp is generated for each column value that gets updated. The timestamp is used to resolve conflicts. [1, p. 63] There is also a fourth attribute called "time-to-live" or "TTL" that can be used to expire data on a column basis after a specified amount of time [1, p. 65].

Multiple columns are grouped together in a row. The rows are sparse, meaning that non-populated columns of the row do not take up space. Hence, null values can be given to columns, which will not be stored in the hash table. In relational storages, the tables are rigid, and each row has all the fields regardless of whether the cells have values or are empty. [1, p. 27] Figure 3.2 demonstrates the row container.

To associate different cells to a row, a common unique identifier needs to be associated to them. This is called a primary key or a row key. More specifically, Cassandra uses a composite key as a primary key, which itself consists of one or more partition keys and zero or more clustering keys. [1, p. 57] Primary keys are often called simple primary keys if they only use one column name as the key.

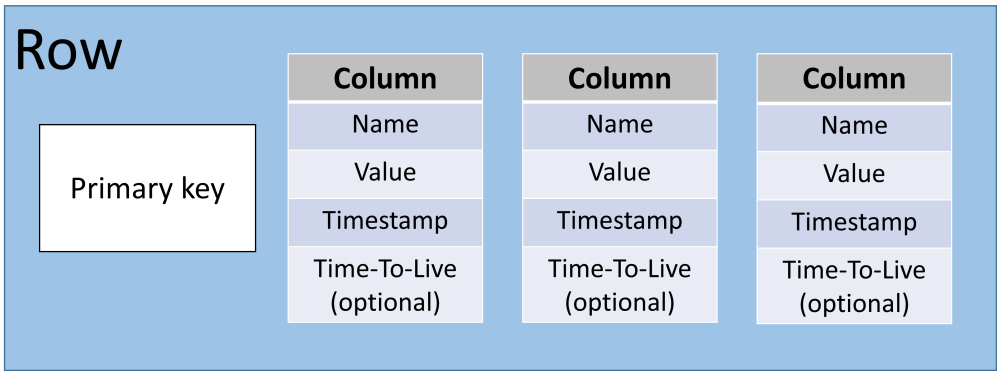

Figure 3.2. *The basic layout of a row (adapted from [14, p. 30, 1, p. 57]).*

The partition key of a composite key represents groups of related rows called partitions, which determine how data is distributed across nodes. Once a primary key has been chosen, every row needs to contain the columns defined in it. The clustering key on the other hand determines how the data is ordered in within a partition and it is based on the columns that are included in it. Sorting data using clustering keys improves the retrieval of adjacent data. [12]

On the second most outer layer, rows are grouped together in tables. A table defines the schema, which is the layout of the data inside the table. It contains the names and data types of the columns, as well as the primary key components. [12] Figure 3.3 represents how tables consist of rows. The row keys or primary keys must be unique to be able to identify the rows. In the Figure 3.3, columns 2 and 3 are not part of the primary key, so they do not need to be present in every row of the table.

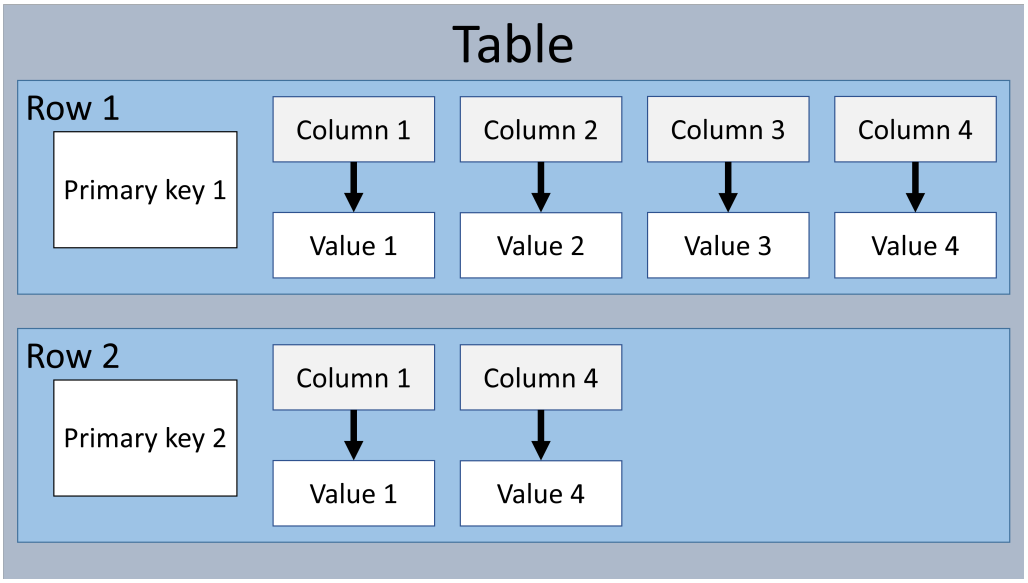

Figure 3.3. *The basic layout of a table (adapted from [12, 1, p. 58]).*

The outermost layer for the data is a keyspace. A keyspace is a grouping of related tables that allows easier data management. A keyspace defines some common configurations

to the tables inside it, for example data replication strategies [1, p. 59]. It can also be used to isolate different data sets from each other.

In Cassandra, the rows have flexible schema. This means that you can have different column names for different rows within the same table and not need to have the all the other columns. The flexibility allows adaptation of the data model as the application requirements change over time, but it also means that it is more important to have good design and modeling practices to ensure that the data is stored in a way that makes it easy to query and retrieve. Moreover, going schemaless means that the schema is nondirectly implemented by the application and not enforced in the database side. In newer versions, going schemaless is more restricted than in older versions of the storage engine where you could have no definitions about what a row contains. This was possible by utilizing the now removed database interface called Thrift API [1, pp. 27–28]. Thrift API was replaced with the Cassandra Query Language (CQL).

3.2.2 Cassandra Query Language

CQL is the primary formal language used to interact with the database and it's at the center of data modeling in Cassandra with the building blocks described in this chapter. CQL is similar to SQL syntax, which makes it easier to learn with prior experience. Regarding this, it is important to keep in mind that using CQL should not be mixed with relational database modeling ideology. CQL supports flexible schemas by offering collections such as maps, lists, and sets to give the option for different number of field values. The collections behave similarly to data structures found in programming languages like C++, but should not be used for big amount of data. In addition to that, CQL supports all typical native data types, but also user-defined types, tuples, and custom types. [15]

To begin using Cassandra with CQL, it is necessary to define keyspaces, tables, and columns to establish the structure of the data. First, by creating a keyspace that defines replication specific settings:

```
CREATE KEYSPACE my_keyspace
WITH replication = {'class': 'SimpleStrategy',
    'replication_factor': 3};
```

Shortly, *SimpleStrategy* defines how the replicas are stored within the cluster, and every data is in three copies. The CREATE TABLE command is used to create a table:

```
CREATE TABLE my_keyspace.my_table (
  id uuid,
  name text,
  phone_number text,
```

```
  age int,
  PRIMARY KEY ((id), age)
);
```

In the example, id is defined as the partition key as it is the first attribute, and age is used as a clustering key that orders the data. Finally, data can be inserted:

```
INSERT INTO my_keyspace.my_table (id, name, phone_number, age)
VALUES (uuid(), 'Alice', '555-12345678', 30);

INSERT INTO my_keyspace.my_table (id, name, phone_number, age)
VALUES (uuid(), 'John', '555-87654321', 22);
```

Unique id is generated using uuid() function. Data can be accessed using SELECT:

```
SELECT * FROM my_keyspace.my_table
WHERE id = 647b6810-6712-48ed-a07e-9372bf80c0dd;
```

3.2.3 Goals of data modeling

Designing the data model for collected data can be one of the most challenging parts of using Cassandra because it requires careful consideration and planning to ensure good performance. Basic goals of this process are [13, p. 5]:

1. Spreading data evenly across the cluster.
2. Minimizing the number of reads to different partitions per operation.
3. Designing the model with future scalability in mind.

Other things to consider are minimizing the number of writes and data duplication, although they are expected in the design. If these can be minimized, the better it will be in terms of the performance. Shortly put, partitions are groups of rows that have the same partition key, and these rows are located on the same node. [12] Regarding the first case, the goal is to distribute data as widely as possible in the cluster of multiple nodes by choosing a good partition key. The result is a more even load on the system when partitions are of similar size. The focus of minimizing the number of reads is to make the tables correspond to a query pattern. Relational modeling starts from domain models, which are represented in tables and their relationships, finally focusing on the queries that need to happen to combine the information. In Cassandra, instead of starting from the domain models, the starting point of data modeling is the actual use cases and queries on which the data organization will be based on. The goal being the minimum number of partitions read per request. In a bad designed model, a read can be unnecessarily divided to multiple partitions, increasing request overhead and latency. [12] To counteract this, data is commonly duplicated between tables, but as mentioned previously, denormaliza-

tion is expected in Cassandra. Other ways are by using indexes and advanced querying methods, but those areas are not the focus in this thesis.

For example, if our application is a bookstore inventory system, queries can be optimized to different use cases. To query books by author, one can use a CREATE TABLE query similar to the following:

```
CREATE TABLE my_keyspace.books_by_author(
    author text,
    title text,
    ISBN text,
    publication_date date,
    PRIMARY KEY ((author), title)
);
```

A second table optimized for querying books by publication date could be:

```
CREATE TABLE my_keyspace.books_by_date(
    publication_date date,
    title text,
    author text,
    ISBN text,
    PRIMARY KEY ((publication_date), title)
);
```

Since the partition key determines data locality between nodes, selecting the primary key affects the performance of data accesses. In the first example, author is the partition key, which results in that all entries with the same author's name are located on the same partition and the same physical node. Furthermore, because where are using title as a clustering key, unique rows are determined by the combination of an author and a title. By specifying the author's name in query's WHERE clause, resulting search happens only inside one node and one partition without the need to access other nodes or partitions:

```
SELECT *
FROM my_keyspace.books_by_author
WHERE author = 'William Shakespeare';
```

The clustering key in that example orders the books by title name. If one were trying to access all the books by one author from the second table where the partition key is *publication_date*, it would require a scan of all partitions on all nodes with inefficient access pattern. That is the importance of data denormalization for a query pattern. When attempting to use the previous query, Cassandra will even respond with an error message: "Cannot execute this query as it might involve data filtering and thus may have unpredictable performance." and requires the use of ALLOW FILTERING command in the request. Another way is to use secondary indexes to make accessing the data

possible by using other attributes than those in the primary key. Using secondary indexes is significantly more expensive than regular queries and should only be used in specific use cases or when adding support for new queries that were not in the original data model design [1, p. 147].

3.3 Architecture

The architecture of a database refers to the overall design and organization of the database system that handles management of the data. It includes the physical and logical components of the complete system, such as the hardware, software, and network components. It also includes how the data is replicated and distributed across multiple nodes, and the way the system handles failures and scalability.

3.3.1 Cluster Topology

Examining the underlying architecture is a good starting point to understanding how clusters are organized in Cassandra. The main terms going from high-level to low-level being:

- Cluster or ring
- Datacenter
- Rack
- Node

At the bottom of the physical topology are nodes, which are instances of Cassandra running in a physical server or in a virtual machine. It stores a portion of the data and handles read and write requests like any other node in the system. [16, p. 6] Every node is in terms of responsibilities and actions the same, but they can have some temporary responsibilities. For example, a node can have a special responsibility as a seed node. When a new node is added to the cluster, it learns the topology of the network from seed nodes [14, p. 17]. It is recommended that a cluster has at least two seed nodes in case where one of them goes down. To clarify, marking a seed node is as simple as adding that node's IP to the seed node list in the cassandra configuration file cassandra.yaml, and it can just as easily be removed from there.

Cassandra is often implemented as a system that is geographically dispersed to lessen latencies and to increase the level of resiliency and performance in globally wide applications. For this reason, nodes are grouped in racks and the racks are grouped in datacenters in the network topology [16, p. 6]. These terms may not correlate to the physical world but can be based on them. A rack is a grouping of nodes that are located close together, for example behind the same network switch [16, p. 6]. In a production system,

racks can be used to make sure that data is not replicated only in a single rack, but evenly between racks. Consequently, if one of the racks goes offline, data is not lost. Racks are grouped in datacenters which can consist of multiple racks with multiple nodes, generally in the same building. The cluster itself consists of these datacenters that can be located in different parts of the world. [16, p. 6, 17]

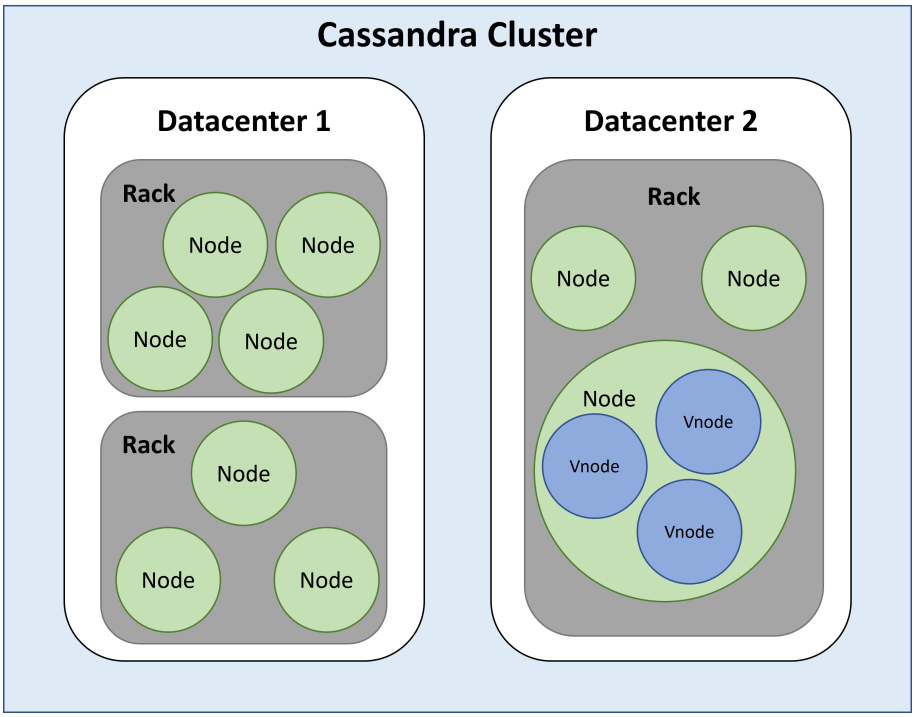

Figure 3.4. *Cluster Topology (adapted from [1, p. 108]).*

Figure 3.4 highlights the hierarchical arrangement of the topology from top to bottom, allowing organized management of the network. At each level, some of the configurations are inherited from the higher level to the lower one. In addition to the physical topology, there is also another element called *virtual node* or *Vnode* that is related to the data distribution. Section 3.3.4 delves deeper into the concept of Vnodes.

3.3.2 Ring structure

Nodes of a cluster are organized in a peer-to-peer architecture where they can directly communicate to each other through inter-node communications protocol called gossip. Gossip is used to share information about the nodes' state and, therefore, is used to keep track of the condition of nodes. Some of the purposes of gossip are failure detection to avoid routing client requests to offline nodes and acknowledging new additional nodes. [1, pp. 109, 111]

One key component that uses gossip is called *Snitch*. Snitch determines the topology of the Cassandra network by assigning datacenters and racks to nodes. The snitch is then used primarily for two functions: first, spreading replicas around the cluster in a way

that minimizes the risk of failures, and second, routing requests to nodes that will answer the fastest. Some ways this is done is by determining the client's relative proximity to nodes and monitoring the performance of them. There are multiple kinds of snitches e.g., SimpleSnitch for single datacenter deployments and GossipingPropertyFileSnitch which is recommended for multi-datacenter deployments. [1, pp. 110–111, 222]

Each node in a cluster manages a range of data based on a token calculated from rows' partition key, and the range is determined by the number of nodes. When reading or writing data, a partitioner function is called that derives a hash value with a consistent hashing algorithm from the partition key that will determine the node that owns that row. The nodes communicate their token ranges with each other through gossip so that they know where a client request should be forwarded. With the token mechanism, every node is assigned data partitions, and the partition key helps index the data in each node. For better visualization, the token range used in the Figure 3.5 is from 0 to 191, but in the real system the token is a 64-bit integer that ranges from -2^{63} to $+2^{63} - 1$. [1, pp. 111–113]

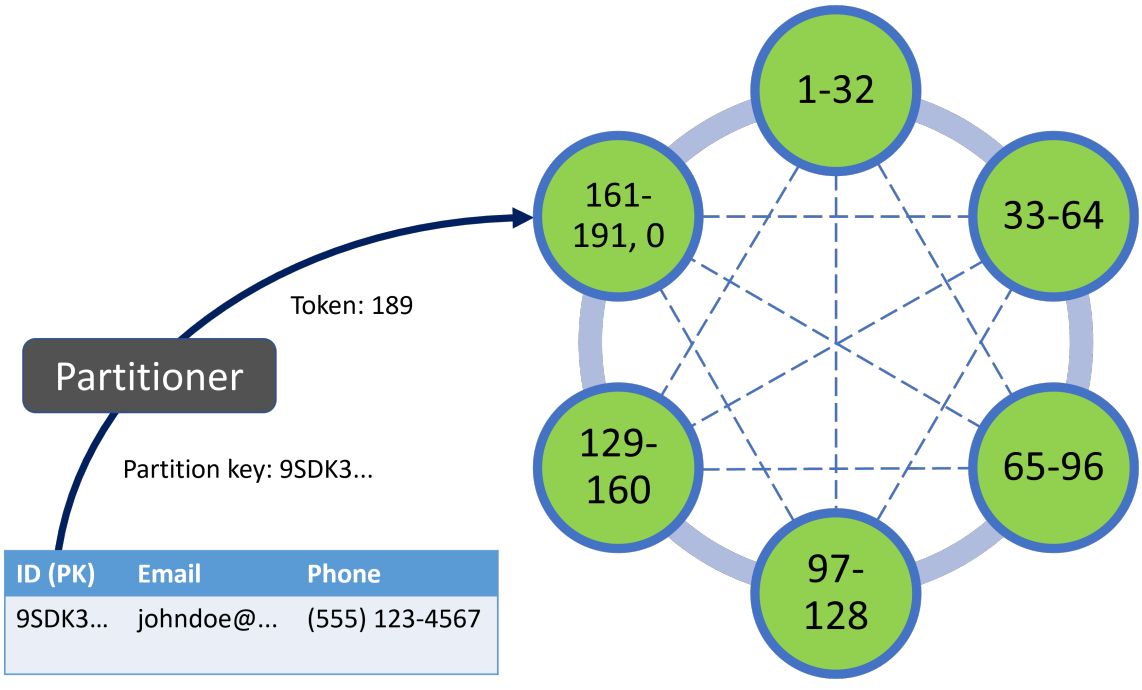

Figure 3.5. *Node ring with example token ranges (adapted from [1, p. 112]).*

The concept of partitioning is one of the most important things to understand about the data modeling and architecture of Cassandra. Like previously discussed, choosing how to partition data has a big effect on the performance of the queries. When designing a model for the data, the first objective is to partition it in a way that makes queries fast and minimizes the number of partitions read per query, but also so that partitions do not become too bloated and the cluster having hot spots of unevenly distributed data. For instance, using a boolean value for partition keys is not a good idea, as the data would be distributed at most only across two nodes. Uneven partition sizes affect for

example performance, memory usages and database repairs. Understanding how to balance the two requires understanding the query patterns and the specific requirements of the application.

3.3.3 Replication and tuneable consistency

Another important aspect is the replication of data. There are three main variables that can be configured for replication and consistency [1, pp. 114–116]:

- Replication factor: The number of copies of each row.
- Consistency level: The number of copies of that must be read/written before a request finishes.
- Replication strategy: Determines which nodes receive a copy of a token.

The first replica node will have its token in its primary token range and the rest will have the token in their secondary range. The number of replicas is determined by *replication factor*. A factor of three means that there are three copies of every row under a keyspace, as the replication factor is a property of a keyspace. [1, p. 114] Replication factor of three is commonly used in a typical setup as a starting point as it strikes a balance between how much data is copied and failure tolerance. The chance that three servers fail simultaneously is very small. The more the replicas, the better the failure tolerance, however also the more data storage required and the more work for writes. A single write to the cluster will in this case multiply to three writes that can affect the performance a bit. The first replica is placed on the node that handles its token, and the following are determined by an algorithm called *replication strategy* [1, p. 115]. Replication strategy uses the information provided by the snitch to place replicas on the ring.

Consistency level is tightly related to replication. Consistency level means the level of certainty that is wanted about the state of the data, ensuring that the replicas in the system have the same version of the data. Cassandra has tuneable consistency specifiable to individual requests, so that consistency can be modified to specific needs. With higher consistency level, the stronger the guarantee of data accuracy, but also increased latency, possible performance drop, and worse availability. [11, pp. 134–136] By using this mechanism, it is possible to shift the system's focus from "AP" towards "CP" in CAP-theorem terms (see Figure 3.1). The main write levels are [11, pp. 134–135]:

- ANY: Lowest consistency, write will succeed if any node acknowledges.
- ONE|TWO|THREE: One|two|three replica node(s) acknowledge the operation.
- QUORUM: The majority of replica nodes acknowledge the operation. Additionally, there is LOCAL_QUORUM for acknowledgment within the same datacenter and EACH_QUORUM for acknowledgment from multiple datacenters.

- ALL: Highest consistency, all replica nodes acknowledge the operation.

All of these except consistency level *ANY* are also available for read requests. *QUORUM*-levels ensure strong consistency and, as such, are often used where strong consistency is needed. There is an equation determined to represent strong consistency [1, p. 116]:

```
IF (R + W > RF)
THEN
    strong consistency
ELSE
    eventual consistency
```

Where *R* is read replica count, *W* is write replica count and *RF* is replication factor. Table 3.1 represents various combinations between replication factor and consistency

Table 3.1. *Interaction of replication factor and consistency level.*

RF	WRITE consistency level	READ consistency level	Consistency	Availability		Performance
				Data loss after (worst case)	Impact to application after (worst case)	
3	ONE	ONE	Eventual consistency	Loss of **1** node	Loss of **3** nodes	Read from 1 node Write to 1 node
	QUORUM	QUORUM	Strong consistency	Loss of **2** nodes	Loss of **2** nodes	Read from 2 nodes Write to 2 nodes
	THREE	ONE	Strong consistency	Loss of **3** nodes	Loss of **1** node	Read from 1 node Write to 3 nodes
5	ONE	ONE	Eventual consistency	Loss of **1** node	Loss of **5** nodes	Read from 1 node Write to 1 node
	QUORUM	QUORUM	Strong consistency	Loss of **3** nodes	Loss of **3** nodes	Read from 3 nodes Write to 3 nodes
	ALL	ALL	Strong consistency	Loss of **5** nodes	Loss of **1** node	Read from 5 nodes Write to 5 nodes

level and their consequences for consistency, availability, and performance. In the table it is assumed that RF is smaller or equal to the cluster size. The larger the cluster size, the better the availability, but the worst case is still limited to the replication factor. For example, when using a replication factor of three with write consistency *ONE*, if the node that receives a write goes down before having time to propagate the value to the replicas, the data is lost. Impact to application happens after reads or writes cannot execute anymore because consistency level cannot be achieved.

The impact of replication factor and consistency levels to performance have been studied by Gorbenko *et al.*, as presented in their paper [18]. The study was conducted on a three node cluster running Cassandra 2.1 versions, with a replication factor of three.

One of their goals was to examine the latency and throughput of Cassandra with varying consistency configurations and workloads. In terms of latency, *QUORUM* showed an average increase of 15% in read latency and 7% in write latency compared to *ONE*, while *ALL* showed an average increase of 26% in read latency and 13% in write latency compared to *ONE*. *ONE* had an average read throughput that was 15% better than *QUORUM* and 25% better than *ALL*. For write throughput, *ONE* was an average 5% more performant than *QUORUM* and 10% more performant than *ALL* in operations per second. [18] These results confirm that Cassandra has better write speeds than read speeds on larger loads.

3.3.4 Virtual nodes

Figure 3.5 demonstrates how a token is calculated from the primary key through the partitioner. To be more precise, this illustrates the older way in which nodes were assigned a single token but handled the range from the previous token up to its own token number. The problems with this kind of simplistic data distribution were that setting the token for a new node was a manual process, and rebalancing of tokens when adding or replacing a node was slow. This is true especially for large databases, as it requires streaming the data from a replica node to its new owner. With Cassandra version 1.2, *Vnodes* were added to optimize this data distribution. Instead of assigning a single token to a node, the token range is divided into multiple smaller ranges that are automatically assigned by Cassandra. [19, pp. 37–38] These smaller ranges are distributed randomly and noncontiguously in the cluster, as illustrated in Figure 3.6 [20]. For example, node 1 in the Figure 3.6 with single-token architecture owns a primary token range A similarly how in Figure 3.5 a node owns a range 1-32. The node 1 also contains replicas to ranges F and E, while in the virtual node architecture the node 1 is assigned multiple smaller ranges. Adding a node with Vnodes speeds up the rebalancing process because the load spreads to several smaller ranges located in different nodes, improving fault tolerance. Thirdly, Vnodes makes supporting a heterogeneous ring of nodes with varying computing power easier. [19, p. 38]

3.4 Data reading and writing mechanisms

3.4.1 Write path

Another temporary responsibility a node can have is when it is handling a client request. As a client application can see the cluster as a singular end point, it does not matter which node answers a request. The node that performs the operation becomes the so-called *coordinator*, and any node can become one. The coordinator determines which nodes the requested replica resides in, and forwards the operation to them. [1, p. 117] How the

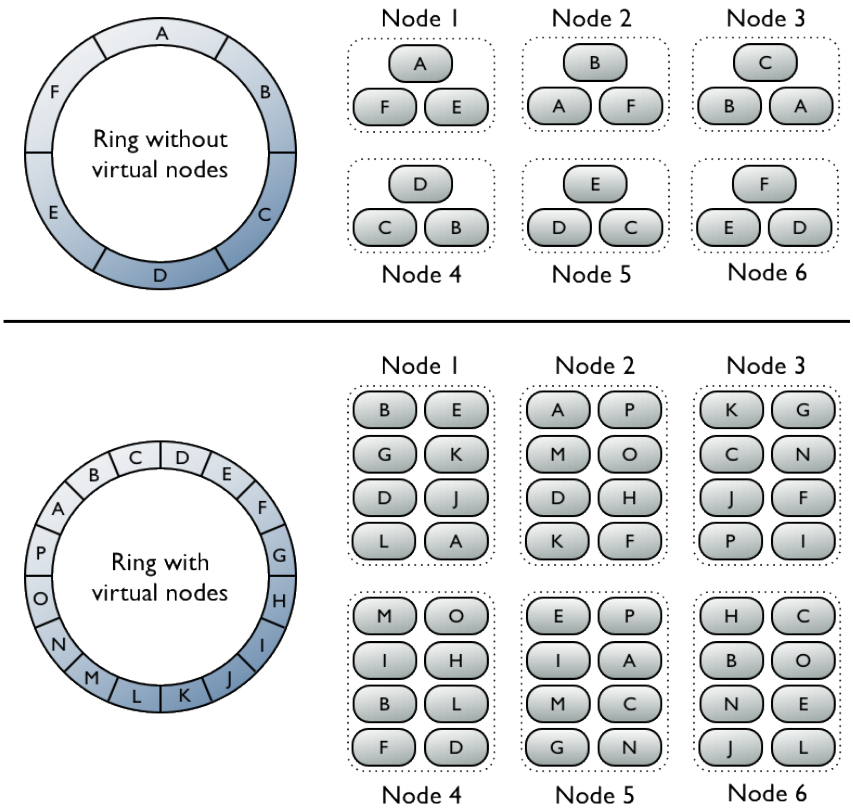

Figure 3.6. *Node ring with single-token architecture and Vnode architecture [20].*

reads and writes function is determined by the consistency level described earlier. The coordinator can be a replica for the requested data and in that case count as one of the acknowledgments for the consistency level.

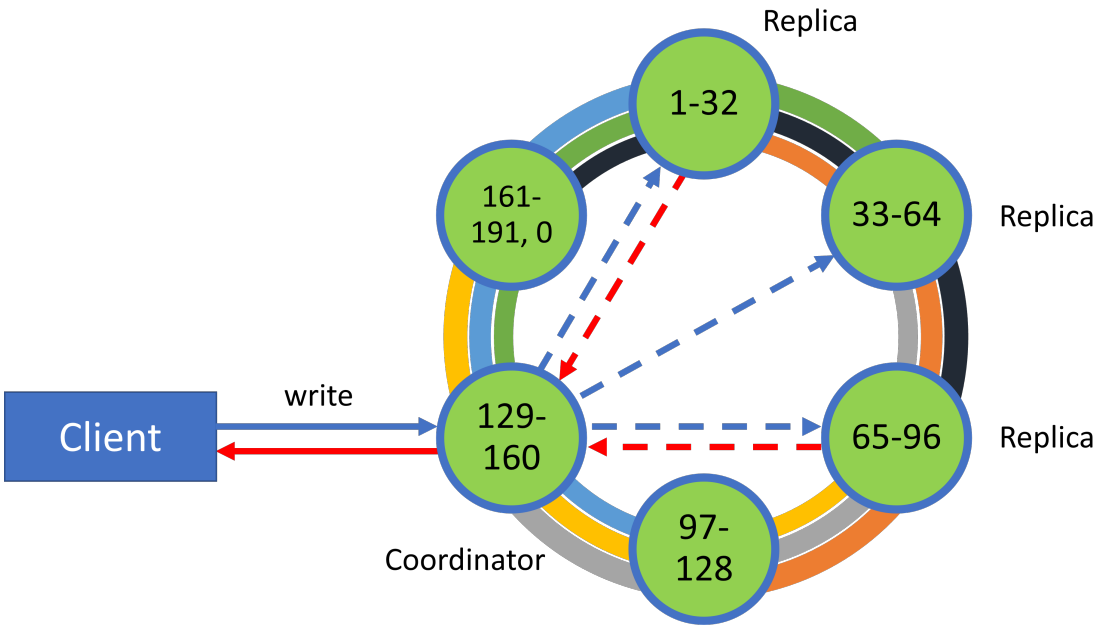

Figure 3.7. *Write path in multinode cluster within a datacenter (adapted from [1, p. 117]).*

For example, in Figure 3.7, a client sends a write to a system with a replication factor

of three. The assigned coordinator determines the replica nodes using the partitioner described earlier and sends a write operation to all of them. With consistency level *QUORUM*, the operation will return after two out of the three replica nodes return acknowledgment to the coordinator node. If the required number of replicas is up and in normal state, the query will complete successfully. Otherwise, an error is returned. Some of the replicas can be located in different datacenters, which requires the local coordinator to send the writes to remote coordinators as well [1, p. 189]. There, replicas acknowledge the write to the original coordinator in the first datacenter.

3.4.2 Storage engine

Understanding the physical storage structure of a node is important for comprehending node behavior in database operations. During writes, Cassandra uses on-disk files called *CommitLogs* to track every mutation that happens to the node. A mutation can be an INSERT, an UPDATE, or a DELETE statement. The primary purpose of the CommitLog files is to recover from service restarts or crashes, as the CommitLog data is not lost. [1, p. 122] Every node has its own CommitLogs that are divided into segments on disk [21]. The next step during the write is to write the value to a data structure in memory called *memtable* that acts as a buffer. After the value is written to the memtable, it is considered successful, and the node acknowledges it by responding to the coordinator node or client. When the memtable or CommitLog size limit is reached, the values of the memtable are flushed to an immutable data structure on disk called *SSTable* or sorted string table. SSTables are stored sequentially on disk and maintain data for each database table. Data for a table can be stored in multiple SSTables but there is only a single active memtable per table. The operations used for writing are designed to require only appends, resulting in high write speeds. For this reason, one of the limiting factors for write performance in Cassandra is the disk speed. [1, pp. 122–124]

With the combination of CommitLogs, a crashed node can restore the state of memtables by replaying the writes stored in the CommitLog. [1, p. 124] When CommitLog's segment size exceeds its limit, it is synced to the disk and CommitLog continues to the next available segment. Mutations maintained in a segment are marked as clean (i.e., flushed to SSTables) or dirty (i.e., persisted only in CommitLog), and Cassandra can only delete segments after all the mutations are persisted in SSTables. On a memtable flush, Cassandra marks the CommitLog positions clean, and once the entire segment is clean, it is deleted. [21]

As mentioned, SSTables are immutable and cannot be modified after they have been created on disk. Over time, this means that there will be obsolete data collected on older SSTables and large amounts of data files accumulated. Besides taking up disk space, read performance is affected as more and more SSTables are accessed during

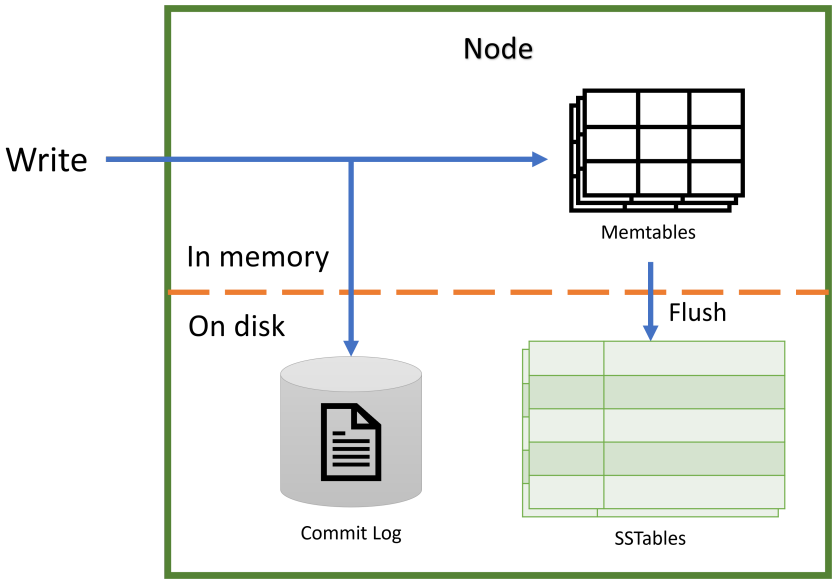

Figure 3.8. *Main data structures of a Cassandra node during write (adapted from [22]).*

data retrieval. The process of *compaction* takes up similar sized SSTables and merges them together, removing deleted data while persisting the other data. [1, pp. 125–126] The compaction algorithm is chosen per table and configured by the types of operations that happen. According to [1, p. 126], in general:

- SizeTieredCompactionStrategy for write-intensive workloads, it is also the default.
- LeveledCompactionStrategy for read-intensive workloads.
- TimeWindowCompactionStrategy for TTL'ed time series data.

Compaction is an automatic process that is done in the background. There is also a specific type of compaction, a major or full compaction, that is executed manually and combines all SSTables for a given table into a single SSTable. [1, p. 127] The usage of major compaction has been discouraged in production environments because the resulting large SSTable will likely not get compacted in the future by automatic compactions due to its size. However, since Cassandra 2.2, there is a way to split the created SSTable into smaller pieces [23].

3.4.3 Node repairs

Moving forward with the write operation and its implications when one of the replica nodes is unresponsive due to various factors such as it being down, having network errors, hardware problems, or being at an overloaded state. Over time, as these failed writes and inconsistencies between nodes accumulate, it forms *entropy* in the system [11, p. 136]. To combat this, Cassandra has anti-entropy mechanisms that are repair during the write path with *hinted handoffs*, repair during the read path with *read repairs*, and *anti-entropy repair* [1, p. 189]. Repairs are designed to synchronize the data between nodes to get rid

of these inconsistencies.

In the hinted handoff mechanism, the coordinator node saves the failed mutation as a *hint* in its storage [1, p. 118]. In other words, if a node is storing hints, it's an indication that it cannot properly communicate with some of the other nodes. When the target node is seen to be online through gossip, the hint is replayed and the write is completed in the background [1, p. 118]. Understanding this, the distinction between consistency levels *ANY* and *ONE* be differentiated more clearly. When using consistency level *ANY*, the operation is considered successful even if only a hint is stored, whereas with *ONE*, the write must complete. Hints are deleted after *max_hint_window* configuration, which is three hours by default. That makes *ANY* a very low reliability setting. In the case of Figure 3.7 and consistency level *QUORUM*, up to one of the nodes can be down for the coordinator to acknowledge the query as successful. It will also store the hint for the downed node.

Hinted handoffs improve making maintenance on the cluster because replaying hints is an efficient method to make the downed node consistent again. If the node remains down longer than the hinted handoff time, some of the hints from that period have already been discarded and other measures must be taken. The next option is *read repairs* that Cassandra uses to repair inconsistencies due to the node being down longer than hinted handoff time [11, p. 136]. When reading data, the values of the replicas that were read according to the consistency level are compared. If differences are found, a read repair is executed, and the outdated values are updated with the most current value. A read repair is a blocking mechanism that the client needs to wait before a response is returned. [24] While the blocking read repairs are currently in use, background read repairs were removed in Cassandra 4.0 because they caused extra internal load and were redundant with other repair methods [25].

Read repairs are not a full substitute for anti-entropy repairs (coming up next) to resolve data inconsistencies, but it has a place in fixing some corner cases during normal operation. The data in nodes could not be up-to-date even after a read repair has been performed as seen in Figure 3.9. If the read consistency is not set to *ALL*, there will be nodes that are not checked for inconsistencies during every read.

Anti-entropy repairs or manual repairs are designed to be run as a regular maintenance to a cluster and in situations where the other methods are not sufficient. Anti-entropy repairs are not executed automatically because they are disk- and memory-intensive and time-consuming operations, but should be run regularly to keep the state of the cluster healthy in the long term. Manual repairs consist of full repairs, that operate over the whole data, and incremental repairs, that operate over new data since the previous incremental repair. [1, pp. 275–276] Running these repairs ensures that the data stored on the node and its replicas are consistent.

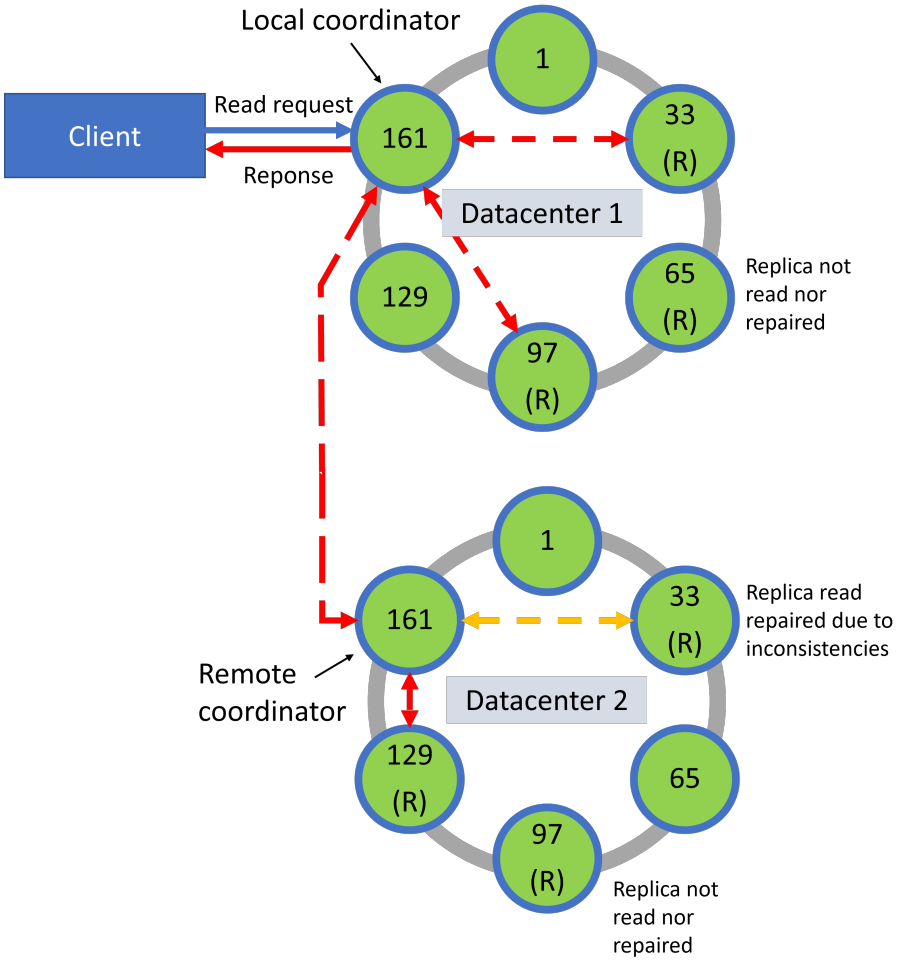

Figure 3.9. *Read repair performed during an EACH_QUORUM level read when data is inconsistent across two datacenters. The nodes marked with (R) are replicas (adapted from [24]).*

3.4.4 Deletions

Deletes are handled in a special way due to the distributed design of Cassandra. When a client sends a data deletion request, the data is not deleted immediately, but rather marked with a *tombstone*. A tombstone is a marker to a column that indicates of expiration date/time. Previously mentioned time-to-live attribute works similarly that after the defined delay, the column or row is marked with a tombstone. Once the tombstone reaches the time defined in the table's *gc_grace_seconds* configuration (default 10 days), the data is deleted as part of the compaction process. [1, pp. 210–211]

The design of the tombstone mechanism can be attributed to the scenario in which some replica nodes are down during a delete request and do not receive any indication of it. Without the tombstone mechanism, downed nodes would assume that the rest of the replicas have been missing writes and stream the now deleted data back to them upon getting back online. [1, p. 210] A row or column that reappears after deletion

is called a "zombie" [26]. Now with tombstones, if the downed nodes come up within the GC grace period and are repaired, they receive the tombstones and the previous situation is solved [1, p. 210]. Despite some amount of delay, this approach enables a more streamlined approach to distributed data deletion in a consistent matter across the network despite varying states. If the node comes up within the hinted handoff time, hints will deliver the tombstone mutation to it. If the node is down longer than the hinted handoff time, Cassandra does not automatically replay the mutation, and a repair must be run. If the node is down for more than the GC grace period, the node misses the tombstone mechanism completely, meaning that the recommended action is to replace or rebuild the node. [1, p. 285]

3.4.5 Read path

Cassandra read path is similar to the write path. A client sends a read request to a node that becomes the coordinator, and it determines the replicas using the partitioner. Similarly, in multi-datacenter reads (i.e., EACH_QUORUM), the request is handled by a remote coordinator in each datacenter.

If the read's coordinator is not a replica, it determines the fastest responding replica node using the dynamic snitch discussed in Section 3.3.2 and sends a full read to it. Other replica nodes determined by the consistency level receive a digest read in which only a hash value of the data is returned. The coordinator then compares the hash values from the digest reads and the hash value calculated from the full read for any discrepancies. If all match, a response is sent to the client, otherwise read repairs are executed before the response like in Figure 3.9. [1, p. 201]

The operations that occur inside a node during a read are more complex than those in writes. As such, this section goes over the high-level overview of these operations. First, a cache that stores a subset of data from SSTables in memory, a row cache, is checked. If the row cache contains the requested data, response can be immediately returned. The read does not require disk accesses and increases read performance significantly. If the data is not in row cache, memtables and SSTables are searched for the requested data. The complete response is created by merging the information gathered from both. Since there are multiple SSTables for a single Cassandra table, the search can potentially be expensive. [1, p. 202] For this reason there are multiple optimizations implemented such as: partition key caching, bloom filter, SSTable indexes, summary indexes and compression offset maps. For example, the bloom filter is used to determine if the partition to be read exists in a given SSTable [27].

3.5 Management of nodes and clusters

Doing regular maintenance to a cluster in Cassandra is an important task to do to ensure that the database is healthy and performing as expected. Node level tasks include monitoring the performance and health metrics, doing maintenance tasks, tuning settings, and creating backups. Cluster level tasks include planning and monitoring the overall capacity in terms of storage, performance, and network resources, managing the cluster topology by adding new nodes, removing old nodes, or replacing nodes, and upgrading Cassandra versions.

Adding new nodes to a cluster can improve the database's performance and availability by distributing data and processing power across more machines. It also releases disk space to be used on the other nodes if replication factor stays the same because the added node takes a part of the token ranges from other nodes. Removing nodes is a necessary action when scaling down the size of the cluster or replacing failed nodes. Repairing a cluster involves identifying and fixing inconsistencies between replicas of data which needs to be done on a regular basis to maintain database's consistency and to prevent data loss.

The simplest way to manage a cluster is by using Cassandra's *nodetool* utility provided in the cassandra-home directory. Nodetool is a command line tool for inspecting, configuring, and doing various maintenance tasks to a node. Common operations are listed in table 3.2, a complete list can be found in Cassandra's documentation, see [28].

From the manual maintenance tasks, repair is the most important in keeping the cluster consistent, and it should be run regularly and every time a major change happens, or a node becomes inconsistent due to hardware, software, or network errors. For example, in the following cases [29]:

1. Updating data on a node that is read rarely and hence not repaired by read repairs.
2. Recover missing data or corrupted SSTables, run a full repair in this case.
3. Node being offline longer than the hinted handoff time, but less than the smallest value for gc_grace_period.
4. After node releases from being in an overloaded state.
5. After topology changes, i.e., node removal, node addition, node replacement, or when changing replication factors.

When a node failure happens and it is down longer than the hinted handoff time, hints can no longer make the node consistent again and a repair must be run using *nodetool repair* on the node. Therefore, a repair should be run regularly, at least within gc_grace_period as discussed in Section 3.4.4. Another case where a repair should be run is when a node becomes overloaded. In those situations, the node might miss writes and make the data inconsistent.

Table 3.2. *Common nodetool operations.*

Group	Command	Action
Cluster and node information	status	Print cluster status.
	info	Print node information.
	tpstats	Print thread pool information.
	tablestats	Print overview of keyspaces and tables.
	bootstrap	See progress of node bootstrap (node joining the cluster).
	compactionstats	Monitor compaction status.
	gcstats	JVM garbage collector statistics.
Basic maintenance operations	repair (--full)	Start an incremental (or full with –full flag) repair.
	flush	Flushes memtables on the node to SSTables.
	drain	Flushes memtables on the node to SSTables and stops listening for connections.
	cleanup	Cleans up data from the disk that is no longer owned by the node.
	snapshot	Backing up the state of all keyspaces and tables in a node.
	compact	Force a major compaction
Topology changes	decommission	Start a graceful removal of a node.
	removenode	Removes the node from the cluster when it is offline.
	assassinate	Forces a removal of node, used when other methods do not work.

Given the increased complexity of managing larger clusters, big organizations typically have dedicated operators to handle repairs and utilize various tools for these tasks. On the other hand, with smaller cluster deployments, automated tools can be the primary way to schedule most maintenance tasks, including manual repairs. This approach minimizes the need for manual intervention, as it is not be desirable or needed for less complex systems. Running a full repair is a common operation as well. Running it monthly is generally adequate, but should be run more often if warranted [29].

Adding a new node to a cluster can be done in the following steps [30]:

1. Install Cassandra on the new node.
2. Configure cassandra.yaml settings and make sure that cluster name matches, and seed nodes are set. If the nodes are identical, all the nodes can use the same cassandra.yaml file as a basis, and then node specific settings like addresses modified for each.
3. Start the node and the bootstrap process of joining the cluster will begin.
4. Verify that the node is up and normal with `nodetool status`.
5. Run `nodetool cleanup` on the other nodes to delete keys that now have been

transferred to the new node. This frees up disk space.

If the new node is planned to be a seed node, it should be first be added to the cluster as a non-seed node, and after the bootstrap process the node can be added to the seed list in every cassandra.yaml file. Initialization of a new cluster is not that much different. It includes installing Cassandra on each node, choosing the settings of the cluster including the cluster name and determining seed nodes. Then do a rolling start first on the first seed node, and then the rest of the nodes, transforming some of the nodes to be seed nodes after they have bootstrapped. Non-seed nodes always require communication to a seed node to join a cluster. Starting the nodes one-by-one prevents problems induced by concurrent initialization, for example token collisions that prevent nodes from starting normally. One important change that needs to be done on the system tables that come automatically with Cassandra is to increase the system_auth table's replication factor from one to a higher number. Otherwise, it's a single point of failure for authentication requests.

Removal of a node is done depending on which state the node to be removed is in by checking with `nodetool status`.

1. If the node is up and its state is normal, executing `nodetool decommission` streams its data to other nodes and removes itself from the ring.
2. If the node is offline, executing `nodetool removenode <host_id>` removes the node, and its data is streamed from the replicas to a new node to maintain the replication factor. If there are not enough nodes to maintain the replication factor, adding `--force` flag to the command is required.
3. In other rare cases, `nodetool assasinate <host_id>` can be executed to forcefully remove a node. No data is streamed.

There are several situations where a node needs to be replaced, e.g., due to hardware failure. The recommended steps to replace a dead node are the following:

1. If the node is running, stop it by executing `nodetool drain`.
2. Install Cassandra on the new node and start it with a JVM flag indicating which node to replace.
   (a) `cassandra.replace_address_first_boot=<node_ip>`
3. Wait until data is streamed onto the new node, progress can be monitored using `nodetool netstats`.

The method can be used to replace a running node as instructed in the first step, but it will reduce the available nodes by one until the new node has finished the joining process. The implications of this, for example, in a small three node system would be that during the replacement period, no other nodes can go offline to maintain availability. Also, if consistency level *ONE* is used, there is a risk of losing data even if RF is three because

the replaced node can contain the only copy of a record at that specific moment [31]. The other safer way is to first add the new node and then decommission the old node, but the disadvantage is that the data is streamed twice. Once to the new node when it is added and again when decommissioning the old node.

3.5.1 Cassandra monitoring

Cassandra is written in Java and runs as a Java process. As a Java application, Cassandra uses Java Management Extensions (JMX) to allow remote management and monitoring of nodes. For example, the *nodetool* acts as a JMX command-line wrapper for managing a node. To make diagnostic data more easily understandable, Cassandra uses a library called Dropwizard Metrics to collect and report diagnostic metrics [32]. The metrics are organized into a hierarchical structure with JMX format:

```
[domain]:[key]=[property],[key2]=[property2],[key3]=...
```

Metric names are formed with a combination of a domain name, a category, and a specific metric name, which ultimately accesses an underlying MBean object. An MBean is a managed Java object that represents a manageable resource and follows JMX specification [33]. Cassandra specific domain names are prefixed with `org.apache.cassandra`. Some of the Cassandra categories include: db, net, request, service, and metrics. For example, compacted bytes are reported in

```
org.apache.cassandra.metrics:type=Compaction,
    name=BytesCompacted.
```

4. SYSTEM TESTING AND ANALYSIS

4.1 Testing objectives

This thesis examines Cassandra's suitability to be used in a small cluster, primarily consisting of three nodes within a single datacenter. More specifically the focus is a three node system, because it is the smallest recommended cluster size for a multinode system. The evaluation of the system's behavior under various use cases provides insights into the technology's reliability. The tests focus on common situations that occur in the long-term usage of Cassandra on the software and hardware side, and what steps need to be taken to handle them. Additionally, the benefits of scaling a single node cluster to multiple nodes is inspected in one of the use cases.

At the center of the tests is a small-scale cluster. One of the reasons why smaller multinode systems are inspected is because there is somewhat limited research available into the characteristics of such clusters. This is relevant as transitioning from a single node to a small multinode cluster is the first step when upgrading a simple database setup. For this transition, the goal is to recognize and test what kind of additional configuration and maintenance requirements arise, what the benefits and tradeoffs are, and assess how the requirements of the infrastructure surrounding the cluster changes.

Often multinode clusters are run in the cloud because it provides flexible and scalable resources that can be added or removed whenever demands change. In such cases, the size of the cluster can quickly grow to tens of nodes when more performance is needed, as horizontal scaling allows for a straightforward way to improve the database's capacity. However, in this thesis, Apache Cassandra is deployed as an on-premises database where scaling to a cluster with tens of nodes is more difficult to achieve. On-premises Cassandra is installed and runs on the computers located on the site rather than at a remote facility. The upfront costs of such systems are higher compared to cloud systems because they require purchasing and maintaining the hardware and other infrastructure. Adding, replacing, and removing hardware requires extra work in on-premises systems, but the long-term costs are potentially lower. Other benefits include fast access to data, better data security because it is stored locally without outside access, and greater ease of compliance with regulations.

Although using two nodes as a cluster is technically possible, it is not recommended due to the lack of benefits gained from it. The motivation behind upgrading from a single node to multinode in this thesis is to gain the scalability and availability benefits while maintaining strong consistency. In a two node cluster, high availability can only be achieved with eventual consistency, because the only viable consistency level is *ONE*. Using strong consistency would mean losing high availability as none of the nodes can go down without causing service interruptions. For these reasons, two node clusters are not inspected in this thesis. However, a three node cluster where only two nodes are online is tested during the performance tests of Chapter 4.10.

4.2 Application context

Within the scope of this thesis, Cassandra is being utilized for handling time-series data. The data model of the time series data includes each data point containing a timestamp, a value, and a quality indicator that indicates the reliability of the value. Additionally, each data point includes a tag id. A tag is a form of metadata that associates a data point to some context. In the design's data model, a tag corresponds to the source of the data.

The application uses data where samples are taken at a specified sampling rate, often in the range of 0.5 - 10.0 seconds. A value is only stored to the database if it changes from the previous cycle, hence the inputs are first buffered before being written to Cassandra. Most of the write load occurs in batches, with each write request containing multiple data points. Batch writes are good when atomicity is required, because either all succeed or none of them do. They can also save network traffic especially when writing to a single partition [34]. Data is accessed both on a per-row basis (i.e., querying the latest value) and through range queries with lengths varying from minutes to weeks of data, generating a significant read load on to the cluster.

4.3 System architecture

The architecture of the system under test, shown in Figure 4.1, consists of clients reading and writing data, as well as the Cassandra servers. Each server includes a Cassandra node and an associated database middleware service (DMS). The DMS provides an abstraction layer that simplifies how clients perform requests as they do not need to know CQL syntax or request internals. Instead, the clients can use application specific structures and messages to write and retrieve data. The DMS is implemented in C++ and communicates with the cluster using DataStax C/C++ Cassandra driver [35]. In this thesis, a client means the applications that send the read and write requests to the DMS, which then sends the appropriate CQL queries to Cassandra using the client driver. Additionally, the middleware layer can perform aggregation requests and publishes

diagnostics about node health and performance. The aggregation requests consist of, for example, calculating time weighted average data over a specified interval, or retrieving the maximum good quality value within a given interval. Connections from the clients to the middleware services are load balanced so that the requests are distributed evenly between them. The DMSs are connected to the cluster and can communicate to any node there with Cassandra's internal load balancing policy. Double load balancing should typically be avoided in Cassandra systems [1, p. 235], but it has been selected as the best option for this kind of usage in the system architecture.

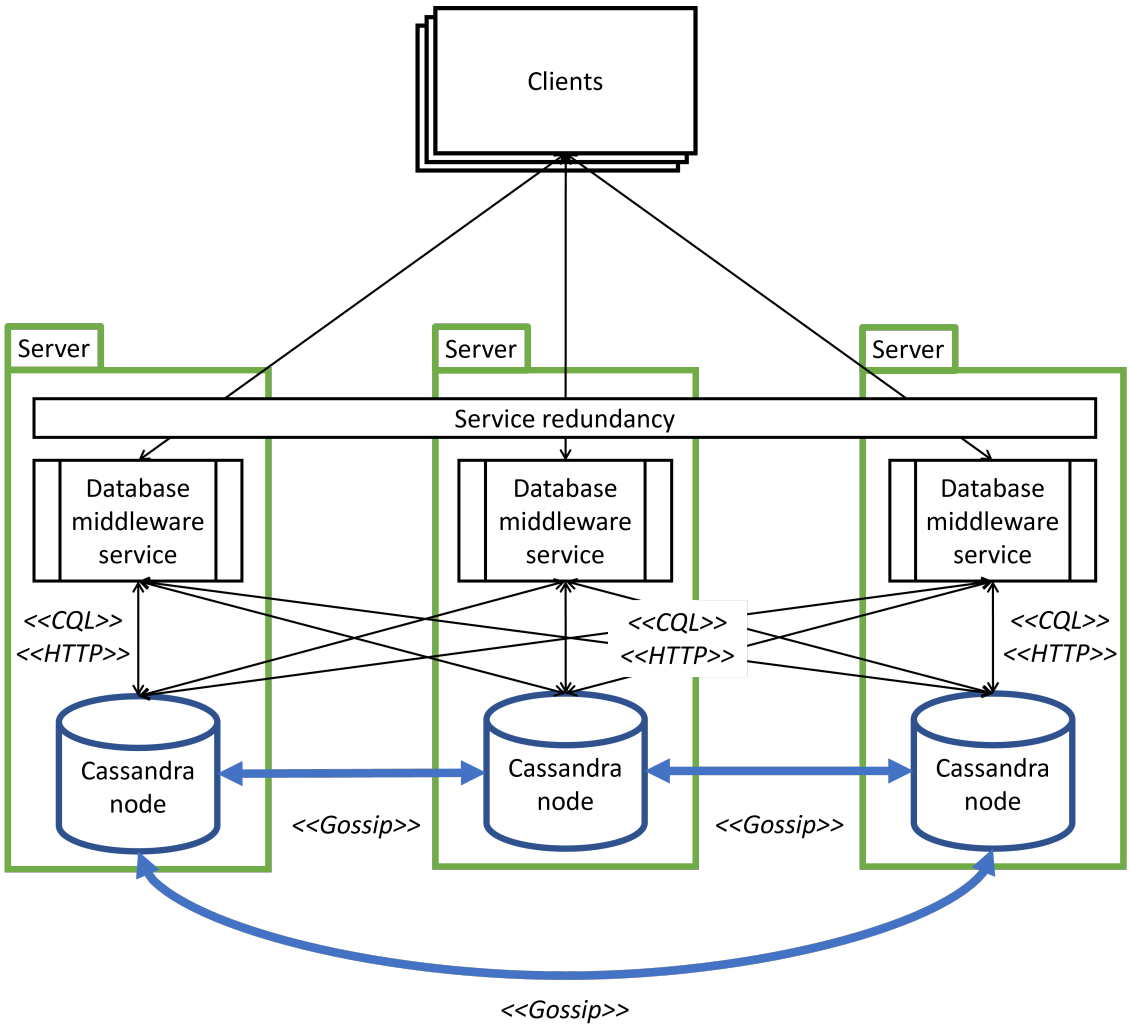

Figure 4.1. *Overall system architecture.*

The system under test uses *Jolokia* [36] version 1.7.1 to allow accessing JMX data over HTTP from the database middleware service, as seen in Figure 4.1. Jolokia introduces a new attack vector to the system if its endpoints are exposed to the network. For that reason, it is configured only for local access inside a server. During the upcoming use cases, most of the Cassandra diagnostics have been collected using Jolokia and stored in database to be retrieved after the tests.

4.4 Testing environment and Cassandra settings

The main test system was built to resemble a production system that is running on the premises. It includes three identical machines with Intel Xeon Silver 4110 processor @ 2.1 GHz with 8 cores and 16 threads, and 48 GB of DDR4 ram @ 3200 MHz. The servers are located within the same rack and connected to the same network switch. The servers are contained within their own subnetwork for easier network management and security. Between the clients and the database is a hardware firewall that manages traffic. At the beginning of the testing phase, a 100 Mbit/s network connection was tested, and later upgraded to 1 Gbit/s connection during Section 4.8, when it was deemed inadequate. The operating system was at first 64-bit Ubuntu 18.04 and later upgraded to Red Hat Enterprise Linux 9 during Section 4.9. The storage medias for each server consisted of a 1 TB hard disk drive, configured with hardware RAID 1 (mirrored disks) for extra redundancy. RAID (Redundant Array of Independent Disks) of type 1 is a configuration in which all data is maintained in two or more disks as exact copies. In this case, each server contained two 1 TB drives with a combined storage of 1 TB.

Cassandra is set to have the following settings:

- Version: Cassandra version 4.1.0 is used, as it is the most recent version at the time of writing.
- Replication: Replication factor of three with NetworkTopologyStrategy is used for all keyspaces to ensure data durability and resilience against node failures.
- Partitioner: Murmur3Partitioner
- Endpoint snitch: GossipingPropertyFileSnitch
- Table settings: LZ4 compressor and Size Tiered Compaction Strategy
- Consistency level: LOCAL_QUORUM for all write and read operations.
- JVM: Heap size set to 8GB and upgraded to 24GB during use case 4, G1 Garbage Collector
- Key cache size: 200 MiB
- Row cache size: 50 MiB
- Number of tokens: 256
- Request timeouts: 60000 ms

Rest of the main settings affecting performance were set to the default values.

4.5 Use case 1: Monitoring cluster health and performance

An administrator responsible for managing the Cassandra cluster, whether large or small, needs to actively monitor its health and performance to ensure that it is performing as expected. This scenario focuses on how it is known that the cluster is healthy, specifically what metrics need to be monitored. Additionally, it identifies what early warning signs start to manifest when the cluster's health is compromised. This allows for early detection of issues and gives time to perform maintenance and troubleshoot problems before they become critical. Additionally, regular checkups are beneficial to prevent and predict capacity, performance, and data modeling issues. The goal is to find anomalies and trends to proactively anticipate needs or changes.

To get a full picture of the system state, it is necessary to monitor not only how Cassandra is operating on each node, but also other aspects of the servers itself. These can be roughly categorized into system resources, JVM resources, and Cassandra diagnostics. Tables 4.1, 4.2, and 4.3 list some of the most important metrics that were identified to be monitored, based on a combination of sources [32] [37] [1, pp. 247–259, 263–268] and testing of the database. System resource metrics contain information about the underlying hardware and operating system, as listed in Table 4.1.

Table 4.1. *System resources to monitor.*

System resources	
State	Uptime
CPU usage	Overall usage Per core usage
Memory	Total system memory usage Swap usage
Network	Receive and transmit speeds
Disk	Disk space available Write and read I/O
System logs	OS level errors

It is important to track the system resources as there are multiple processes running on the server, each demanding a share of available resources. Metrics such as disk space usage and disk I/O can be used to indicate when additional storage space is needed, or disks need to be replaced. Other metrics such as CPU usage is a good indicator about server overload.

The second area to monitor is Java Virtual Machine metrics, as shown in Table 4.2. The JVM metrics mainly contain memory and garbage collection as the most crucial area for monitoring. Tracking the total, heap, and non-heap memory usages reveals if there are memory leaks, garbage collection issues, or other trends indicating slowly

Table 4.2. *Java Virtual Machine metrics to monitor.*

JVM metrics	
Memory	Heap memory usage Non-heap memory usage
Garbage collection	The frequency of collections Duration of collections

rising memory usages. Tracking the frequency and duration of garbage collections helps to identify potential GC-related performance issues and bottlenecks. During a garbage collection pause, all operations of the Cassandra process are suspended for the duration of the pause. Therefore, the cause of frequent pauses of more than a second should be investigated [38].

The third category encompasses Cassandra specific diagnostics, which cover areas such as request throughputs, latency measurements, error rates and their types, node states, and communication-related metrics. These are composed in Table 4.3. From the node

Table 4.3. *Cassandra diagnostics to monitor.*

Cassandra diagnostics	
Node state	Operation mode Live and unreachable nodes Cassandra versions Schema mismatches
Log files	debug.log, system.log and gc.log
Client requests	Timeouts Failures Latencies Throughputs Number of client connections Commit log size
Node communication and internode metrics	Dropped messages and their types Timeouts per host Cross-node latency Number of dropped mutations
Storage	Storage and repair exceptions Load Hints in progress and on disk Column family latencies Partition sizes
Repairs	Read repairs executed Percentage of SSTables repaired
Thread pools	Active, pending, completed and blocked tasks
Compactions	Pending and completed compactions Bytes compacted

metrics, the state of the node should first and foremost be monitored (i.e., normal, boot-

strapping, leaving, etc.) and its connection to the other nodes. By recording and aggregating these from all nodes, connection problems can be identified, and the affected nodes determined, allowing for targeted investigation. The client request information is used to identify bottlenecks in terms of performance, but also resource constraints, connection and hardware issues, and cluster overloading. If timeouts, failures, or exceptions do not show up in client request metrics, applications do not notice any difference even if nodes have slight communication problems or one of the three nodes goes offline.

Information about misbehaving nodes first starts to show up in the 95th and 99th percentiles of client latencies as abnormal spikes. Local latencies (keyspace and table/column family latencies) give a better indication of node's query latencies that describe the time it takes to read and write to disk and memory. Thus, they can be used to identify slower nodes and slow queries, as they exclude the time it takes to achieve consistency (including network waiting time), unlike client latencies. Node communication metrics, published by *MessagingService*, gives insight to network and performance problems between nodes. For example, mutations are dropped when a node cannot keep up with the requests sent to it, and cross-node latencies measure the latency of messages between nodes. Assessing storage-related metrics enables evaluating cluster's data management efficiency and to address potential storage issues, for example, if large-sized partitions start to appear, it is an indication that the data model should be refined. Cassandra divides its operations into different stages and each of them maintains thread pools to execute its tasks. Taking note of trends in the number of tasks per category is a good way to know when to add more capacity to the cluster.

In many cases, effective monitoring can be achieved by focusing on key indicators such as node states, connections between nodes, client and cross-node message information, request throughputs, and exceptions. More comprehensive tracking provides a better view of every aspect of the system, which becomes more and more important as the size and workload of the cluster(s) increases. When performing periodic assessments or debugging the source for problems, it is a good idea to first check the log files for any indications of errors. Gc.log files records information about garbage collection events, debug.log files record detailed information about Cassandra's internal operations, and system.log files contain information about the overall functioning and health of the node on a higher level than debug logs.

4.5.1 Moderate write and read load

The three node system was run with a constant load of ~105 writes/second per node, obtained using the *OneMinuteRate* attribute from:

```
org.apache.cassandra.metrics:type=ClientRequest,scope=Write,
    name=Latency
```

MBean. The writes were batch writes, each containing in the scope of 10-100 values of time-series data. In total, there were writes to 20 000 different data tags and 250 diagnostic collection tags in Cassandra, with varying sampling rates, resulting in around 5000 values generated a second. The read load averaged 370 reads/second per node, fluctuating between 100 and 1200. It was obtained using the *OneMinuteRate* attribute from:

```
org.apache.cassandra.metrics:type=ClientRequest,scope=Read,
    name=Latency
```

MBean. Most of the read load was created by 40 clients that made range queries to the accumulated time-series data. For these queries, most of the processing time is taken by Cassandra process with minimal DMS processing time as aggregation queries were not used. The queries' start timestamp was set to current time, while the end timestamp was randomly selected within a range between current time and up to 15 hours prior. The rest of the reads consisted of reading the latest value from a tag. As a result, the read load on the server increased slowly throughout the test.

Results of the test:

- The network transmit speeds started at ~9 Mbit/s and increased to ~21 Mbit/s on each server. The network receive speeds started from ~8 Mbit/s and increased to ~13.5 Mbit/s on each server. Network usage increased significantly as larger and larger reads were executed towards the end, as seen on Figure A.1.
- Average server CPU usages started at 12-16% and increased to 14-19%, occasionally jumping as high as 60-70%. Five-minute moving averages are depicted in Figure A.2.
- The heap memory usage shows a pattern of increasing to 7.5 GB, near to the configured maximum limit, and then decreasing to 1.5-4.0 GB before rising again, as shown in Figure A.3. This kind of pattern is also called "a saw" and is caused by the JVM garbage collection that deallocates unused objects automatically. This was indicated by examining the gc.log file where a GC pause was logged during the drop. The heap usage was monitored with *HeapMemoryUsage* attribute from MBean

  ```
  java.lang:type=Memory.
  ```

- The amount of data in each node (also called node load in Cassandra) increases linearly at the rate of 4.25GB per day. The Figure A.4 depicting disk usage shows a fluctuating pattern, indicated by increases with dips, which is attributed to the compaction process of Cassandra. Node loads were monitored using Mbean

  ```
  org.apache.cassandra.metrics:type=Storage,name=Load.
  ```

- No reported message timeouts.
- The mean client read latency started at ~2.5 ms and increased to ~6.0 ms. 99th percentile of read latencies increased from ~30 ms to ~80 ms. The highest spikes in 99th percentile latency were ~180 ms. Client latencies were monitored using MBean

  ```
  org.apache.cassandra.metrics:type=ClientRequest,
      scope=<RequestType>,name=Latency,
  ```

  where *RequestType* is *Write* or *Read*. As a sidenote, all latency numbers in the tests refer to the time it takes for the request to complete between the DMS and the cluster, and they do not include the time it takes between the clients and the DMS.
- Although the number of writes remained constant, the mean client write latency increased from ~3.25 ms to ~7.5 ms with increasing read load. 99th percentile of write latencies increased from ~25 ms to ~86 ms. At times there were quite large spikes, the largest being 455 ms on node 2 when monitoring the 99th percentiles. Average client request read and write latencies can be seen in Figure A.5.
- Column family latencies or local latencies were on average small, around 0.1-1.0 ms, as seen in Figure A.6. Interestingly, there were proportionally huge spikes only in column family read latencies every 30 minutes, the height being affected by the amount of read load. Towards the end of the test, the biggest was a 48 ms spike in average read latency. Column family latencies were monitored using MBean

  ```
  org.apache.cassandra.metrics:name=<LatencyType>,
      type=ColumnFamily,
  ```

  where *LatencyType* is *ReadLatency* or *WriteLatency*.
- Cross-node latencies remained relatively constant without any spiking, as seen in Figure A.7: the 50th percentile ranging between 3-8 ms with seemingly no effect from the read load, and average 99th percentiles increasing from ~29 ms to ~50 ms. Cross-node latencies were monitored using MBean

  ```
  org.apache.cassandra.metrics:name=CrossNodeLatency,
      type=Messaging.
  ```

These results provided a starting point for further analysis in subsequent testing.

4.5.2 Overloading one, two or three nodes by stressing the CPU

The primary objective of these tests was to introduce unevenness within the servers and to observe how the situation is seen in diagnostic measurements. Another point was to analyze the potential impact of such situations on the system's performance and behavior.

For example, determining whether the cluster would still function well if one server was temporarily utilized for other tasks. These tests were conducted by overloading the CPU of one, two, and three servers, resulting in a total of three scenarios. The CPU stressing was achieved by using a command called "`yes`", which was executed in parallel to match the number of CPU cores:

```
for i in $(seq $(getconf _NPROCESSORS_ONLN));
    do timeout 30m yes > /dev/null & done
```

The yes command repeatedly prints "y" to the output, in this case the "/dev/null" file, which is a special file in Unix that discards anything written to it. The yes-processes are terminated after 30 minutes. Cassandra was able to utilize some portion of the CPU during this time, but the competing "yes" processes affected the utilization enough so that it created disturbances.

The results of the tests are combined in Table 4.4. The averages were calculated from a 20-minute period during the 30-minute test window, beginning 5 minutes after the beginning of the test and ending 5 minutes before the end. The interval was chosen to exclude any potential outliers in at the beginning or end of the test. The averages before the tests were calculated from the preceding 2 hours to provide a baseline. Similarly like before, latency numbers refer to the time it takes for a request to complete between the DMS and the cluster. The same write load was used as in Section 4.5.1. The read load consisted of 40 clients doing requests, with the data time range for each request varying from 0 to 24 hours in duration. Each client would wait for a request to finish before sending a new one.

During the tests no timeouts, failures or exceptions happened in client request statistics, meaning that the load was moderate enough to not cause disruptions. These can all be obtained using the ClientRequest metrics. Timeouts indicate that the configured maximum time is exceeded and therefore the operation canceled, unavailable exceptions mean that there are not enough nodes to fulfill the consistency level. Failures can occur for several different reasons, which needs to be investigated to determine what went wrong. Node communication metrics (MessagingService) on the other hand showed some internode message timeouts, although very minimal amount, meaning that internal communication of nodes noticed some slowness in the configured setting during the three node overload.

The system returned to the original function quickly after the tests ended. As a conclusion, client latencies seem to be a good way to determine node imbalances and issues within the system. Both write latency and read latency increased, write latency slightly more. However, keeping in mind that they are affected by their respective loads. Once imbalances are detected, the problematic node can be identified by examining, for example, system resource usages such as CPU and memory utilization, local latencies, client and

Table 4.4. *Results of CPU stress tests*

Metric \ Test	Before tests	Node 1 overloaded	Node 1 and 2 overloaded	Node 1, 2 and 3 overloaded
Average client read latency (ms)	N1 5.17 N2 5.37 N3 5.79	N1 21.58 N2 6.96 N3 7.08	N1 25.01 N2 25.73 N3 46.41	N1 67.65 N2 67.22 N3 67.56
Client reads per second	N1 365.28 N2 364.82 N3 365.43	N1 365.55 N2 364.26 N3 364.71	N1 363.10 N2 363.93 N3 364.33	N1 363.72 N2 363.13 N3 379.88
Average client write latency (ms)	N1 7.00 N2 7.05 N3 7.18	N1 23.77 N2 58.10 N3 56.63	N1 80.62 N2 73.23 N3 79.65	N1 105.11 N2 105.75 N3 109.72
Client writes per second	N1 105.6 N2 105.6 N3 105.9	N1 105.8 N2 105.6 N3 105.5	N1 105.5 N2 106.0 N3 105.8	N1 105.6 N2 105.8 N3 105.8
Average column family read latencies (ms)	N1 0.510 N2 0.381 N3 0.466	N1 1.154 N2 0.191 N3 0.244	N1 0.254 N2 0.099 N3 0.847	N1 0.605 N2 0.770 N3 0.071
Average column family write latencies (ms)	N1 0.189 N2 0.171 N3 0.201	N1 0.110 N2 0.158 N3 0.192	N1 0.143 N2 0.110 N3 0.199	N1 0.049 N2 0.133 N3 0.537
Average cross-node latencies of 99th percentile (ms)	N1 48.6 N2 40.6 N3 57.0	N1 332.7 N2 428.1 N3 455.4	N1 472.5 N2 482.2 N3 505.4	N1 550.2 N2 539.7 N3 540.7
Internode message timeouts	N1 0 N2 0 N3 0	N1 0 N2 0 N3 0	N1 0 N2 0 N3 1	N1 0 N2 1 N3 11

internode timeouts and failures on each node. A direct way to notice an error situation is to examine error and exception messages on a per-node basis in the Cassandra logs and investigating the behavior of application drivers which usually raise programmatic exceptions, for example *UnavailableException* and *OperationTimedOutException*. They directly indicate of a failed operation. On the other hand, the default timeout values in Cassandra configuration file are strict when used with lengthy read or range queries and need to be set according to the types of requests used.

4.5.3 Overloading the system with read requests

In this test, the purpose was to overload the system with long reads to the point where it could no longer function normally, resulting in client requests failing. Overload due to long reads is more likely to happen in the production systems compared to overloading due to writes, because they generate significantly more load than writes in this use case. The 44-hour test included 20 clients sending read requests to data that had been stored with varying intervals, on average at two-second intervals. Read lengths increased from 0.1 to 10.0 day mark linearly by rising a day every 10 minutes, and then back to 0.1, repeating

over the test. The data tags were randomly chosen and similarly like in the previous test, the clients did not execute another read if the previous request from that client was still ongoing. The DMSs did not do any data processing tasks for these requests but rather just forwarded the answers to the clients.

Message statistics:

- Messaging service timeouts 16 000 - 20 500 per node.
- Client read failures 40 - 95 per node, write failures 40 - 95 per node, read timeouts 170 - 280 per node, and write timeouts 170 - 280 per node.
- Column family read latencies spiked from 0.1 to over 300 ms in regular intervals. There was also write latency increase before the spike.
- Node 3 dropped the connection to another node once during the test, as seen when monitoring *DownEndpointCount* and *SimpleStates* attributes from Mbean

  `org.apache.cassandra.net:type=FailureDetector.`

  The dropped connection persisted for a couple of seconds.
- Node 3 in general had more trouble functioning than the other nodes, rendering the diagnostics monitoring data from that node unusable. This issue arose because in the event of a complete connection drop, the DMS would cease transmitting data to the clients that collect the diagnostics information, even if the connection came back, which was identified to a software bug.
- Client write latency 99th percentiles rose as far as 2400 ms when 10 day range queries were sent. The average write latency remained constant between 3.7 and 60.0 ms. The average client read latencies were smaller, ranging between 1.5 and 18 ms.
- The average CPU usage increased to 25%, but peaked at 100% frequently.

Client read and write timeouts as well as failures were maintained at identical levels. First appearing during the first iteration of 10 day reads and then increasing linearly. At 26-hour runtime mark, there was a big increase in Node 1 timeouts, and then a couple hours later client request failures started to occur. Signs of overloading started to be seen immediately from messaging service timeouts, indicating that cross-node messages were being slow. The number of timeouts increased linearly over time. The average network usages were *35 Mbit/s* during the longest reads, as seen in Figure A.8 but it frequently reached the network's maximum capacity of *100 Mbit/s*, which was the main reason for connection problems and request failures.

4.6 Use case 2: Transition from a single node to a multinode cluster

Smaller installations of Cassandra often start with a single node system when its performance is adequate for the application and high availability is not required. A single node cluster is simpler to run than a multinode cluster for several reasons:

1. There are fewer configurations to think about. In contrast, multinode clusters require adjusting data distribution, replication, and network communication settings more carefully.
2. It is easier to maintain, as there is only one instance to monitor, update, and troubleshoot.
3. All data resides in a single node, making data consistency more straightforward, e.g., no need to think about performing repairs. Backups become more important with no extra nodes to contain redundant data. On the other hand, its simpler to manage backups with only one node.
4. The system architecture is less complicated.
5. Lower hardware costs, especially in on-premises installations. When transitioning to a multinode environment there is not only the cost of buying the extra servers but also often requiring upgrading various other components of the whole infrastructure. This includes, for example, enterprise- or industrial-grade networking hardware, which can be costly.

However, as the load grows and performance starts to degrade, it may become necessary to migrate to a multinode cluster, providing better performance and the availability benefits that the single node system lacks. The use case goes over the steps and configuration changes that needs be done when going from single node to multinode cluster. For future research, it would be valuable to perform a comparative analysis of similar costing systems consisting of different amounts of nodes. For example, three high-cost nodes versus five moderately priced nodes versus seven low-cost nodes.

4.6.1 Test runs

Changing from a single node to a multinode cluster requires several steps to ensure that the cluster will meet performance expectations, maintain fault tolerance with high availability, and accommodate the requirements of the application through correctly set replication factors and consistency levels. As a first step, the number of nodes to be added to the cluster should be determined. For a multinode cluster, a minimum of three nodes is recommended, but the cluster can be scaled much further if needed. Another point is to plan the topology of the nodes in the racks and datacenters. This test used a single datacenter with a single rack. The number of nodes impacts aspects such

as performance, fault tolerance, and operational complexity, which typically increase in proportion with the number of nodes added. On that basis, the replication factor for all keyspaces needs to be decided.

For a small cluster of three nodes, the replication factor of three is the most suitable choice in the context of this thesis to meet the requirements of the application being tested. It enables the use of *QUORUM* consistency levels, which simplifies achieving strong consistency. The replication factor of three ensures that the system can tolerate a failure of one node. Replication factor of four in a four node system does not provide any extra benefit with *QUORUM* level requests, as they therefore require

$$QUORUM = RF/2 + 1 = 4/2 + 1 = 3$$

responses, still allowing only one node to go down. A replication factor of five requires that *QUORUM* requests receive

$$QUORUM = RF/2 + 1 = 5/2 + 1 = 3.5$$

responses, which is rounded to three. This provides the benefit of two node failure tolerance in a five node system. Replication factor should not exceed the number of nodes in the cluster, as requests will require more responses than there are nodes with that data.

In this use case, two approaches were explored for transitioning from a single node system to a three node multinode system. First, most straightforward method is to add the two extra nodes one at a time without altering any existing configurations. Once the setup is complete, modifying the replication factors, consistency levels, and other settings as needed. Running a full repair is mandatory after the replication factors have been changed. The downside with this method, as seen with the upcoming test, is that read requests will not be consistent across nodes until the repair has been completed.

The second way tested was by first changing the settings on the first node to accommodate the multinode system before the new nodes were added. This method requires special attention, as some settings can be adjusted beforehand while others must be altered later. To make it simple, only replication factors should be modified from *RF = 1* to *RF = 3*, while keeping other settings same. The special case here is the *system_auth* keyspace that will use the parameters *auth_read_consistency_level* and *auth_write_consistency_level* defined in the cassandra.yaml file. By default they are set to *LOCAL_QUORUM* and *EACH_QUORUM*, respectively. This means that if *RF* is set to three for that keyspace, authentication requests will fail until more nodes have joined the cluster or until the settings are changed. Setting these to consistency level *ONE* should only be done temporarily. One of the advantages of the second method include not

needing to run full repairs right away, as replica streaming is done automatically during the bootstrap process of the added nodes. Secondly, the number of data streamings required is reduced.

Both approaches were tested using a virtual machine setup with Docker [39] containers running Cassandra instances. As a starting point, the single node system used the following settings:

- Replication: *RF = 1* for all keyspaces and SimpleStrategy
- Consistency level: *ONE* for all requests
- Snitch: SimpleSnitch
- Grace period: 0 for instant data deletion

The first approach was tested first. With one node running, two instances were installed with the same Cassandra version with the single node settings. Cassandra.yaml files were kept the same in all installations, but specifically properties *cluster_name* matched the first node, *seeds* list contained the first node's IP-address, and since *SimpleSnitch* was still used, datacenter and rack names were the default values. After the nodes bootstrapped and `nodetool cleanup` was executed on the first node, each node contained approximately 33% of the cluster load. Next, the following multinode settings were set:

- Replication: RF = 3 for all keyspaces and NetworkTopologyStrategy
- Consistency level: *LOCAL_QUORUM* for all requests
- Snitch: GossipingPropertyFileSnitch
- Grace period: 10 days

First, the multinode replication factor and replication strategy were set to each keyspace, and grace periods to each table using *ALTER* clauses. It was observed that *SELECT* clauses began to return different information on each node until a full repair was executed. This resulted in that client read requests did not receive consistent data. Similarly, when *system_auth* keyspace was set to have *RF = 3*, authentication requests started to fail until full repair was executed for it. Finally, the snitch was updated to *GossipingProperty FileSnitch* on each node as well as setting the datacenter and rack names to match the previous ones within the rackdc configuration file. Consistency levels for the Cassandra statements were changed to *LOCAL_QUORUM*. The cluster was now transitioned to the multinode system with appropriate settings.

Then second approach was tested, first by changing all keyspaces to contain the multinode replication settings, except for keeping the *system_auth* at *RF = 1*. Request consistency levels were kept at consistency level *ONE*. Similarly as before, the two nodes were added one by one. During the bootstrap process, no issues were encountered with the

client requests. After the bootstrap, *system_auth* keyspace was modified to contain *RF = 3* and a full repair was run for it. The snitch was changed to *GossipingPropertyFile Snitch*, and consistency level to *LOCAL_QUORUM*. Again, the datacenter and rack names were changed to match the previous. As a result, this method proved to be a better way to transform a single node cluster into a multinode cluster compared to the first approach because fewer problems were encountered.

4.6.2 Test analysis

The previous approaches did not take into account how this affects the physical setup. Upgrading to a multinode cluster is not only about increasing the number of nodes and configuring the database, but it also involves addressing other components such as the network hardware. The recommended network bandwidth is 1 Gbit/s with multiple networked nodes. [1, p. 235] This means means that if the on-premises system did not start with hardware that supports it, upgrading that component of the system would be necessary. Depending on the circumstances, this might not be possible later on and should be planned beforehand if an installation might require upgrading to a multinode cluster in the future.

Second, the clocks on all nodes and clients should be synchronized, for example, using Network Time Protocol (NTP). This is due to the fact that Cassandra resolves conflicts using the timestamps of columns [1, p. 235]. Third, the inspected architecture requires load balancing to happen when clients send requests to the database middleware ser vices, which should be confirmed to be working correctly. Diagnostic monitoring becomes more complex, as every node needs to be monitored and tools created that report when problems are encountered in any of the nodes. Backup and recovery mechanisms need to accommodate the multinode system. Finally, security configurations might need to be updated. This may involve using encryption for node-to-node communication [40], and adjusting firewalls to allow internode communication to pass.

4.7 Use case 3: Short term node disruption

Use case 3 focuses on performing maintenance work that causes a node to go offline for a short period of time. This can be caused by, for example, upgrading software or operating system versions, fixing software or network problems, or replacing hardware such as the power supply or fans. The operations cause the node to go down for less than the configured maximum hint window. Therefore, repairs do not have to be executed because hints will restore the node's state to be consistent with the other nodes. The system should be able to maintain the data collection, consistency, and availability despite a temporary node outage.

To test this scenario, a short node drop was executed by taking a node offline for two hours. For this test, the write load included the 5000 time series data points generated a second, as in Section 4.5.1, minimal read load was induced to the system.

The expectations for this test were that the nodes would have been able to service the clients without interruptions with high availability and consistency. However, during the first test where the network cables were unplugged, it was observed that some of the DMS' had difficulties communicating with the cluster, and requests started to raise an exception:

```
All hosts in current policy attempted and were either un-
    available or failed.
```

The error message indicates that the affected middleware services were unable to establish a connection to the cluster. In this case, restarting the services fixed the errors and requests started to flow again with two nodes out of three online. It is suspected that the issue was attributed to a bug in the DMS's code, specifically related to handling cluster topology changes correctly. However, a more thorough investigation is required to confirm the exact cause. Going over the diagnostics of the nodes, it was seen that the problem was identified to originate from a previous incident where nodes were disconnected from the network by unplugging them. The attribute *OneMinuteRate* obtained from MBean

```
org.apache.cassandra.metrics:type=ClientRequest,scope=Write,
    name=Latency
```

reported that the nodes started to receive different amounts of writes: node 2 receiving ~35 writes/s, node 3 ~90 writes/s, and node 1 ~195 writes/s. When the DMS's were restarted, the number of writes returned to ~105 writes/s per node, as seen in Figure A.9.

After restarting the middleware services, the test was executed again, now without encountering problems. After the node came back up online, the hints processing took a total of 15 minutes. The `hinted_handoff_throttle` was set to the default value of *1024 KiB/s* and `max_hints_delivery_threads` to the default value of 2. During the drop, the total number hints accumuled to *2.294 million* on node 2 and *2.287 million* on node 3. The network receiving speed on the downed node increased to an average *20.5 Mbit/s* during the hints processing and decreased to *5.9 Mbit/s* after returning to a consistent state. The transmitting nodes used transmit speeds around *13.4 - 13.5 Mbit/s* during the hints processing and returned to *5.5 - 5.6 Mbit/s* after all hints were processed. That is, both nodes sent hints at speed of *8 Mbit/s*, as configured by the `hinted_handoff_throttle` value and `max_hints_delivery_threads`. During the time the node was offline, requests divided evenly to the rest of the nodes, for example the write rate increased from 105 to 158 on the other two nodes, as seen in Figure A.10.

In conclusion, the cluster was able to handle a short-time node failure while maintaining

high availability without the need to run repairs. The results indicate that even if the Cassandra cluster is configured to operate with some of the nodes unavailable, it is crucial that client communication is configured properly, since it can become the single point of failure.

4.8 Use case 4: Extended node disruption

Long node drops can happen, for example, due to hardware failures, resulting in the server being offline for an extended period of time. It's important to ensure that the system can maintain data collection, consistency, and availability during downtime, as well as to know what steps must be taken when the server comes back online. Other issues that can cause long node drops include datacenter or infrastructure problems, such as network outages due to broken network switches, power outages, or even damage from fire or water. This use case differs from use case 3 in that the node is down for more time than the maximum hint window and requires repairs to be run to make it consistent again. Performance aspects are inspected in Section 4.10.

To test the scenario, a long node drop was executed to simulate a node failure. As a starting point, the nodes 1, 2, and 3 contained 494.16 GiB, 516.89 GiB, and 508.95 GiB of data, respectively. The same write load as in Section 4.5.1 of 5000 values per second was maintained throughout the process.

The node was offline for more than the hinted handoff time, meaning that running repairs was mandatory to keep it consistent. Specifically, the following steps were taken:

1. Read and write load was induced into the system, and the following script was executed on node 1:

   ```
   sleep 32400; sudo service cassandra stop;
   sleep 172800; sudo service cassandra start;
   ```

2. After 9 hours, Cassandra service was stopped and remained offline for 48 hours until it was started again.
3. After that, the node had been online for ~7.5 hours, during which read repairs where performing. Incremental repairs had never been run on the system, so the *PercentRepaired* attribute acquired from MBean

   ```
   org.apache.cassandra.metrics:type=ColumnFamily,
       name=PercentRepaired
   ```

   returned near 0 value.
4. The command `nodetool repair` was attempted to run unsuccessfully. The error encountered was the following:

```
org.apache.cassandra.db.repair.PendingAntiCompaction
    $SSTableAcquisitionException: Prepare phase failed
    because it encountered legacy sstables that don't
    support pending repair, run upgradesstables before
    starting incremental repairs
```

The test system had been used for a long time with different versions of Cassandra up to this point without running `nodetool upgradesstables`.

5. `nodetool upgradesstables` was run to upgrade the old versions of the SSTables to the current version. The only updated table was system_auth.roles.
6. Following several unsuccessful repair attempts, it was determined that the streaming part of the repairs consumed the entire 100 Mbit/s network bandwidth during this time. This resulted in dropped connections, client application request errors, and nodes viewing each other as offline. Because of these issues, repairs experienced failures and needed to be restarted from the beginning each time. The issue was resolved by significantly limiting the streaming speeds in *cassandra.yaml* file:

```
entire_sstable_stream_throughput_outbound: 3250KiB/s
entire_sstable_inter_dc_stream_throughput_outbound:
    3250KiB/s
stream_throughput_outbound: 3250KiB/s
inter_dc_stream_throughput_outbound: 3250KiB/s
```

One important note was that if the streaming bandwidth was set to for example 50 Mbit/s in each node, the node that is receiving data would effectively experience a combined incoming speed of 100 Mbit/s (50 Mbit/s + 50 Mbit/s), which would still lead to network overload.

7. Now repairs for tables started to complete successfully without failure midway. Additionally, instead of running a repair for all unrepaired data, a more granular approach was taken by executing repairs on a table basis by using command `nodetool repair <keyspace> <table>`. The system was also upgraded to have 1 Gbit/s network connections.
8. Another challenge encountered was when repairing large sized tables, approximately 330 GB in compressed size on the largest node, where the validation process of a repair would take an extended period of time and timeout after 24 hours with an error message:

```
WARN  [OptionalTasks:1] 2023-04-03 02:56:53,864
    LocalSessions.java:449 - Auto failing timed out
    repair session LocalSession{sessionID=b21f76f0
    -d0e8-11ed-8da6-19b8cf42ca33, state=REPAIRING
```

```
INFO  [OptionalTasks:1] 2023-04-03 02:56:53,865
    LocalSessions.java:739 - Failing local repair
    session b21f76f0-d0e8-11ed-8da6-19b8cf42ca33
```

And later logging:

```
ERROR [AntiEntropyStage:1] 2023-04-03 06:04:59,978
    LocalSessions.java:933 - Error handling
    FinalizePropose message for LocalSession{
    sessionID=b21f76f0-d0e8-11ed-8da6-19b8cf42ca33,
    state=FAILED, coordinator...

java.lang.IllegalArgumentException: Invalid state
    transition FAILED -> FINALIZE_PROMISED
```

In this case, the 330GB of compressed data on disk formed 760GB of uncompressed data when inspected with `nodetool compactionstats` command during a vali- dation. What slightly helped for the problem was modifying the heap size, compaction throughput, and concurrent compactors according to guidelines [41], but the problems persisted.

9. After deleting the overly large table, repairs could run without issues. Dropping the table required increasing timeout values and retrying the operation multiple times. The data on the table was successfully removed from the disk of two of the nodes, but the third node still contained the SSTables when checked a couple of days later. The files were then manually removed because it was known that these tables were no longer used. In production systems, manually removing SSTables should be avoided. Finally,

   ```
   org.apache.cassandra.metrics:type=ColumnFamily,
       name=PercentRepaired
   ```

   now returned near 100 value, indicating of a successful repair.

As a summary:

- When using a multinode Cassandra system, the 100 Mbit/s network speed is not enough for medium or large workloads. Cassandra streaming speeds need to be heavily limited, repairs can last days with large amounts of data, and cluster topology changes such as removing, adding, or replacing nodes are very time consuming. In this test, the read load was minimized when network problems began to occur. However, during the tests for use case 2, network overload occurred only due to high read load, causing message failures. When taking into consideration typical read and write loads, backups, and repairs, a 100 Mbit/s network speed seems to be inadequate for maintaining a three node cluster properly. In a smaller

system with small workload, a 100 Mbit/s network can be suitable, but its capacity would still be limited to a certain point. A single node configuration works better under these conditions, as it does not need to perform repairs and synchronize data with other nodes, thus avoiding additional network overhead.

- If any network problems occur and packets drop due to network overload, it is highly probable that repairs will fail and subsequently other requests during this time. When in limited environment, streaming speeds need to be restricted in Cassandra's configuration more strictly.
- It is advisable to perform repairs per-table or per-keyspace basis, so that if a repair fails at any stage, it does not require restarting from the beginning, but rather only for the affected table.
- Excessively large tables can cause capacity and performance problems. In several tests, repair validation encountered timeout errors when attempting to repair large sized tables. As a result, data storing must be managed in a way that prevents these tables from increasing to such size. In a cluster with more nodes, each node can have less of the table's data stored, and these problems become less apparent.

4.9 Use case 5: Upgrading software versions

This use case focuses on upgrading the application and Cassandra versions in an existing system to a newer release and examines a scenario in which different versions of Cassandra are running temporarily in the cluster. This phase requires planning to ensure minimal impact to clients. The other test case goes over the steps required to update the operating system without losing data. In this case the targeted system undergoes partial formatting by only preserving Cassandra's data on disk.

4.9.1 Updating Cassandra versions

The three node cluster was running Cassandra versions 4.0.2, and the goal was to upgrade them to version 4.1.0 without data loss. The upgrade process was performed in a rolling manner, one node at a time. This ensures that if something goes wrong during the update, the other two nodes can keep the cluster running normally. Generally, during a cluster version update [42]:

- If some operations use consistency level *ALL*, they need to be temporarily lowered to maintain functionality during the update process .
- Avoid running data definition or data control operations, including DROP TABLE, TRUNCATE TABLE, CREATE TABLE, or operations about roles.
- Avoid running repairs during an upgrade, and run it after all nodes have been

updated. There might be inconsistencies in the way different software versions perform repairs.

- Similarly, taking backups during an upgrade should be avoided. They should be taken before the cluster upgrade process is started.
- Do not bootstrap new nodes or decommission existing ones to avoid compatability issues with streaming operations.
- Run the upgrade within *gc_grace_seconds* to ensure that repairs are completed successfully.
- Client drivers might require updating for the communication to the cluster to work.
- It is a good idea to run `nodetool upgradesstables` before starting the upgrade process to ensure that all SSTables are of current version.

When upgrading minor versions, like in this use case, breaking changes are likely not going to be introduced. But when upgrading major versions, the guidelines should be followed more strictly.

During upgrades, repairs should not be run because there are different versions of Cassandra nodes in the cluster, which can cause compatibility issues. This approach was tested during the updates, and even though the update was only one minor version apart, some errors were logged:

```
ERROR [InternalResponseStage:29] RepairMessage.java:78 - [#
    69dd1400-b136-11ed-8240-2bcde88317f8] SYNC\_REQ failed on
    /192.168.15.31:7000: TIMEOUT
```

Followed by:

```
WARN  [InternalResponseStage:29] RepairMessage.java:95 - [#
    69dd1400-b136-11ed-8240-2bcde88317f8] Not failing repair
    due to remote host /192.168.15.31:7000 not supporting
    repair message timeouts (version = 4.0.2)
```

The outcome of the repair was unambiguous, as the logs showed errors about timeouts but also messages indicating successful completion. After several iterations of updates, the following steps were formulated for future upgrades:

1. Backup Cassandra data and configuration files.
2. Run `nodetool drain` on the node to flush the memtables in memory to SSTables on disk and to stop the node from receiving new requests.
3. Stop Cassandra service using `sudo service cassandra stop`.
4. Download and extract updated Cassandra either via a tar package or a package manager, compatible Java Development Kit (JDK), and other required packages.

5. Customize the Cassandra installation with the same settings as the backed up configuration files, including cassandra.yaml (data storage and other paths, networking, security), cassandra-env.sh (Java options and memory settings), jvm-files (garbage collection), and set the same datacenter and rack name in the cassandra-rackdc.properties file. The contents of the files can change from version to version, so the best way to merge the values is by comparing them manually.
6. Configure the service file (systemd) for Cassandra when it is run as a service and installed through the tar package.
7. Start the service.
8. Check the log files for errors or warnings.
9. Repeat until all nodes have been updated.
10. Run `nodetool upgradesstables` command to update SSTables.
11. The installation files of the previous version of Cassandra can now be removed from the system.

Running `nodetool upgradesstables` is important when upgrading Cassandra versions because old SSTables can result in repair failures and degraded performance when the storage engine gets updated. Overall, the update process of a cluster was straightforward, and no complications were encountered with the previous instructions.

4.9.2 Changing the operating system

Another update situation is when migrating from one operating system to another, and in this test from Ubuntu 18.04 to Red Hat Enterprise Linux 9. Cassandra versions remained at version 4.1.0 during the update. This process entails preserving the current Cassandra data on disk, while the other system undergoes formatting during the installation.

As a prerequisite, Cassandra's data needs to be on a separate partition on the disk, which makes the update process simple. In addition, backups should be taken of the data before initiating the migration to counter possible data loss because unexpected things can happen. To keep the cluster functional, the update was performed in a rolling manner, one node at a time.

The following steps were taken to update each server, noting that alternative approaches may be used, for example some configurations can be done during or after the installation wizard:

1. Drain the node running `nodetool drain`.
2. Start the installation process `sudo reboot now` and boot from the installation media.

3. Setup the correct RHEL configurations and mount the preserved Cassandra data partition. Other partitions on disk are formatted.
4. Configure rest of the system, including the firewall to allow CQL transport and internode communication, and access rights of the data partition.
5. The clocks between the systems have to be synchronized, because Cassandra resolves conflicts using timestamps.
6. Required packages (Python for cqlsh, Cassandra, JDK, etc.) can now be installed, and Cassandra started.
7. Check the Cassandra logs for any warnings or errors.

Two out of three nodes were upgraded while preserving Cassandra's data by keeping that partition unchanged during the installation. This means that the data for the node came from the disk when it booted. Specifically, this is achieved by mounting the Cassandra data partition during the RHEL installation process, and then specifying that path to option *data_file_directories* in cassandra.yaml. The writes that occurred during this time were received through hints from the other two nodes after the boot. In this kind of transition state, the cluster works at its minimum capacity with two nodes, and no other major tasks should be executed to ensure its stability.

One of the nodes was upgraded by formatting all of Cassandra's data. In this scenario, the other two nodes will stream the appropriate data to the node and rebuild it when it joins the cluster. During its setup, it was observed that Cassandra logged an exception message in the debug.log file:

```
A node with address <ip> already exists, cancelling join. Use
    cassandra.replace_address if you want to replace this node.
```

To resolve the error, starting Cassandra with `cassandra.replace_address=<ip>` JVM flag or removing the old node id from the ring using `nodetool removenode <id>` is required. The second option was executed in the test. Second, if the formatted node was previously set to be a seed node, it first needs to be started as a non-seed node by modifying the seed list in its cassandra.yaml file. Ideally, this change should be applied to all of the nodes in the cluster and restarting them in a rolling manner or running `nodetool reloadseeds`. As a reminder, the new node cannot bootstrap if it is a seed node. If an upgrade takes more than three hours (default `max_hint_window_in_ms`), meaning that the node has been offline for that duration, a repair must be run. Another problem encountered was that when the network settings were being modified, Cassandra could not connect to the other nodes until the whole server was rebooted.

Apart from those notes, the upgrades went smoothly and without any major complications. It was possible to upgrade the cluster without data loss and keep it functional by doing the upgrades one at a time.

4.10 Use case 6: Expanding a small cluster through node addition

In this use case, scaling a single node cluster up to four node cluster is explored. Scaling is a common scenario encountered when demands for the database increase, primarily due to increased workload or fault tolerance requirements.

The system configurations chosen for the tests were:

1. four node cluster with four live nodes
2. four node cluster with three live nodes
3. three node cluster with three live nodes
4. three node cluster with two live nodes
5. single node cluster

In addition to the effects of scaling a cluster, the tests explored the impact of node drops on performance in a small-scale environment. On multinode configurations, all data was replicated with a replication factor of three and requests were performed with consistency level *LOCAL_QUORUM*. All machines had the same specifications, which are listed in Section 4.4. In the single node cluster, the replication factor was changed to one and the consistency level to *ONE*.

An important point here is to note that since we are running the DMS on the same hardware as the Cassandra node, it requires extra processing power from the server. Doing data aggregation tasks is resource intensive and can become the limiting factor of server performance. A simple way to improve performance is to parallelize these requests to multiple servers and multiple nodes. This is supported by the load balancing of the system architecture, as explained in Section 4.3. To estimate the performance of a real-world production system, performance tests include testing the entire solution, not only Cassandra.

The tests focus mainly on read request performance with a constant write load, as the reads generate much bigger workload on to the cluster than the writes in this application. The data used for these tests included 1000 tags, each storing one-second interval data with float values ranging from 0.00 to 1.00 from a period of 30 days, totaling 2.592 million data points for each tag. The same write load of 5000 values a second was used as in Section 4.5.1, but these were not for the same tags that were read by the tests. The read requests used were:

(A) Read for 1 tag, retrieve data for a 1-day period, and the DMS calculates the hourly averages before sending a response to a client.

(B) Read for 10 tags, retrieve data for a 1-day period, and the DMS calculating the hourly averages before sending a response to a client.

(C) Read for 1 tag, retrieve data for a 1-week period, and the DMS calculates six hour averages before sending a response to a client.

(D) Read for 1 tag, retrieve data for a 1-month period, and the DMS calculates the daily averages before sending a response to a client.

Initially, the cluster consisted of the three previously defined nodes, each containing 126 - 130 GiB of load. After adding the fourth node, the loads for nodes 1, 2, 3, and 4 were 130.40 GiB, 128.38 GiB, 126.83 GiB, and 95.94 GiB, respectively. As explained in Section 3.5, running `nodetool cleanup` command was necessary to free up disk space on the other three nodes. The command took *8h 15min*, *4h 33min*, and *7h 35min* to complete on nodes 1, 2, and 3, respectively. During it, disk usages temporarily increased. As in other kinds of compactions, a single cleanup task can increase disk usage by the size of the largest SSTable on a node. After the cleanup, each of the nodes settled to around 96-100 GiB of load or around 75% from before, as expected. The load increase rate dropped from ~4.25 GB per day to ~3.35 GB per day, or around 78-79%, including the fourth server diagnostic measurements.

The results of the tests are presented in Tables 4.5, 4.6, 4.7, and 4.8. Figure 4.2 combines the results of the performance tests. The values in Figure 4.2 represent the amount of requests completed per second compared to the single node system for every test. For example, the four node cluster was able to complete on average 2.50x amount of requests compared to the single node during the tests for type A requests.

Network traffic caused by the read loads was surprisingly high in multinode environments. The single node cluster utilized network speeds of a few megabits a second, whereas the multinode clusters operated with network speeds in the range of a hundred megabits per second during the tests. The write load generated on average 3.4 - 4.2 Mbit/s of network traffic per node on a four node cluster. In a three node cluster, the same writes generated on average 3.6 - 4.7 Mbit/s of network traffic per live node.

During each test, CPU utilizations were quite high on each node, generally decreasing slightly as the number of live nodes increased. The single node cluster was an exception in this regard, as its CPU usage was either lower or around the same as that of a three node cluster machines. Additionally, node 4 appeared to use more CPU than the other ones, suggesting a slight imbalance in hardware performance. In tests for read requests of type C and D, CPU utilization was less of a limiting factor. Having three live nodes in a four node cluster generated about the same performance as a three node cluster with three live nodes. The performance of two live nodes was found to be almost exactly halfway between that of the single node and three node configurations, as seen in Figure 4.2. The result suggests that if better performance is needed, investing in two nodes instead of three could be a viable option. However, that makes fault tolerance worse when strong consistency is used, because then there are two machines that can break

down instead of one.

In the four tests, a three node cluster was able to process 95%, 95%, 61%, and 119% more requests than the single node. A great performance boost was achieved, but it was not linear to the number of servers added. The four node cluster was able to process 150%, 165%, 72%, ad 157% more read requests than the single node cluster, or 7 - 36% more than the three node cluster. An outlier, where the performance improvement was not that substantial, was the tests for 1-week read requests. The specific reason why Cassandra had more trouble with those requests is unclear, but the results in Table 4.7 indicate that the servers utilized less CPU than in the other tests.

In conclusion, the use case tests demonstrated that upgrading a single node cluster to three nodes greatly improves its performance, on average about 94% with the read requests presented. Further upgrading to a four node cluster not only gives even better performance, but also reduces disk usage as each node only contains 75% of overall data with a replication factor of three. The performance improvement achieved with four nodes was on average 22% better than with three nodes.

Table 4.5. *Test results of type A read requests.*

Configuration	Requests/sec	Average CPU usages (Run 1)	Average network speeds	Errors	T = test length in seconds R = amount of requests
4 node cluster **4 live nodes**	Run 1: 22,44 Run 2: 22,19 Avg: 22,32 **(+149,5%)**	Node 1: 75% Node 2: 78% Node 3: 77% Node 4: 89%	Receive speeds: 163 - 203 Mbit/s Transmit speeds: 163-190 Mbit/s	No errors	Run 1: R=5633, T=251 Run 2: R=5060, T=228
4 node cluster **3 live nodes**	Run 1: 17,17 Run 2: 17,50 Avg: 17,34 **(+93,9%)**	Node 1: 88% Node 2: 86% Node 3: 87% Node 4: offline	Receive speeds: 123 - 134 Mbit/s Transmit speeds: 128 - 143 Mbit/s	No errors	Run 1: R=4792, T=279 Run 2: R=6090, T=348
3 node cluster **3 live nodes**	Run 1: 17,25 Run 2: 17,59 Avg: 17,42 **(+94,8%)**	Node 1: 88% Node 2: 83% Node 3: 87%	Receive speeds: 135 - 176 Mbit/s Transmit speeds: 132 - 169 Mbit/s	No errors	Run 1: R=4553, T=264 Run 2: R=5242, T=298
3 node cluster **2 live nodes**	Run 1: 12,38 Run 2: 12,27 Avg: 12,32 **(+37,8%)**	Node 1: 89% Node 2: offline Node 3: 91%	Receive speeds: 116 - 122 Mbit/s Transmit speeds: 109 - 122 Mbit/s	No errors	Run 1: R=11584, T=936 Run 2: R=2356, T=192
single node	Run 1: 9,11 Run 2: 8,78 Avg: 8,94	Node 3: 69%	Receive speed: 2,6 Mbit/s Transmit speed: 0,5 Mbit/s	No errors	Run 1: R=4818, T=529 Run 2: R=4328, T=493

Table 4.6. *Test results of type B read requests.*

Configuration	Requests/sec	Average CPU usages (Run 1)	Average network speeds	Errors	T = test length in seconds R = amount of requests)
4 node cluster **4 live nodes**	Run 1: 2,32 Run 2: 2,30 Avg: 2,32 **(+164,5%)**	Node 1: 84% Node 2: 81% Node 3: 84% Node 4: 90%	Receive speeds: 125 – 148 Mbit/s Transmit speeds: 121 – 146 Mbit/s	No errors	Run 1: R=1342, T=576 Run 2: R=702, T=305
4 node cluster **3 live nodes**	Run 1: 1,83 Run 2: 1,86 Avg: 1,85 **(+111,0%)**	Node 1: 88% Node 2: 90% Node 3: 88% Node 4: offline	Receive speeds: 137 – 166 Mbit/s Transmit speeds: 130 – 167 Mbit/s	No errors	Run 1: R=943, T=514 Run 2: R=506, T=272
3 node cluster **3 live nodes**	Run 1: 1,73 Run 2: 1,69 Avg: 1,71 **(+95,5%)**	Node 1: 90% Node 2: 88% Node 3: 90%	Receive speeds: 140 – 176 Mbit/s Transmit speeds: 146 – 181 Mbit/s	No errors	Run 1: R=876, T=506 Run 2: R=433, T=256
3 node cluster **2 live nodes**	Run 1: 1,25 Run 2: 1,23 Avg; 1,24 **(+41,4%)**	Node 1: 92% Node 2: offline Node 3: 91%	Receive speeds: 127 – 147 Mbit/S Transmit speeds: 132 – 147 Mbit/s	No errors	Run 1: R=668, T=535 Run 2: R=399, T=325
single node	Run 1: 0,89 Run 2: 0,86 Avg: 0,88	Node 3: 68%	Receive speed: 2,5 Mbit/s Transmit speed: 0,5 Mbit/s	No errors	Run 1: R=596, T=671 Run 2: R=534, T=619

Table 4.7. *Test results for type C read requests.*

Configuration	Requests/sec	Average CPU usages (Run 1)	Average network speeds	Errors	T = test length in seconds R = amount of requests
4 node cluster **4 live nodes**	Run 1: 2,08 Run 2: 1,96 Avg: 2,02 **(+72,2%)**	Node 1: 38% Node 2: 40% Node 3: 40% Node 4: 60%	Receive speeds: 80 – 96 Mbit/s Transmit speeds: 82 – 89 Mbit/s	No errors	Run 1: R=1081, T=519 Run 2: R=834, T= 425
4 node cluster **3 live nodes**	Run 1: 1,65 Run 2: 1,78 Avg: 1,72 **(+46,2%)**	Node 1: 47% Node 2: 48% Node 3: offline Node 4: 70%	Receive speeds: 70 – 83 Mbit/s Transmit speeds: 71 – 84 Mbit/s	No errors	Run 1: R=839, T=508 Run 2: R=533, T=299
3 node cluster **3 live nodes**	Run 1: 1,88 Run 2: 1,91 Avg: 1,90 **(+61,4%)**	Node 1: 59% Node 2: 68% Node 3: 56%	Receive speeds: 81 – 100 Mbit/s Transmit speeds: 85 – 94 Mbit/s	No errors	Run 1: R=1105, T=587 Run 2: R=1000, T=524
3 node cluster **2 live nodes**	Run 1: 1,52 Run 2: 1,51 Avg: 1,52 **(+29,1%)**	Node 1: 77% Node 2: offline Node 3: 80%	Receive speeds: 81 – 87 Mbit/s Transmit speeds: 96 – 94 Mbit/s	No errors	Run 1: R=711, T=468 Run 2: R=384, T=254
single node	Run 1: 1,20 Run 2: 1,15 Avg: 1,17	Node 3: 76%	Receive speed: 2,6 Mbit/s Transmit speed: 0,4 Mbit/s	No errors	Run 1: R=731, T=611 Run 2: R=751, T=652

Table 4.8. *Test results for type D read requests.*

Configuration	Requests/sec	Average CPU usages (Run 1)	Average network speeds	Errors	T = test length in seconds R = amount of requests
4 node cluster **4 live nodes**	Run 1: 0,52 Run 2: 0,55 Avg: 0,54 **(+157,6%)**	Node 1: 62% Node 2: 67% Node 3: 58% Node 4: 76%	Receive speeds: 73 – 106 Mbit/s Transmit speeds: 80 – 95 Mbit/s	Node 1: 0 timeouts Node 2: 0 timeouts Node 3: 0 timeouts Node 4: 19 timeouts	R=309 T=594 R=173 T=313
4 node cluster **3 live nodes**	Run 1: 0,48 Run 2: 0,45 Avg: 0,47 **(+124,0%)**	Node 1: 83% Node 2: 76% Node 3: 81% Node 4: offline	Receive speeds: 82 – 117 Mbit/s Transmit speeds: 85 – 109 Mbit/s	Node 1: 0 timeouts Node 2: 0 timeouts Node 3: 0 timeouts	R=256 T=535 R=231 T=508
3 node cluster **3 live nodes**	Run 1: 0,45 Run 2: 0,46 Avg: 0,46 **(+119,2%)**	Node 1: 75% Node 2: 85% Node 3: 79%	Receive speeds: 87 – 105 Mbit/s Transmit speeds: 78 – 96 Mbit/s	Node 1: 4 timeouts Node 2: 6 timeouts Node 3: 3 timeouts	R=266 T=586 R=136 T=296
3 node cluster **2 live nodes**	Run 1: 0,33 Run 2: 0,30 Avg; 0,32 **(+52,0%)**	Node 1: 87% Node 2: offline Node 3: 88%	Receive speeds: 67 – 79 Mbit/s Transmit speeds: 70 – 73 Mbit/s	Node 1: 46 timeouts Node 3: 45 timeouts	R=187 T=564 R=147 T=486
single node	Run 1: 0,22 Run 2: 0,19 Avg: 0,21	Node 3: 76%	Receive speed: 2,6 Mbit/s Transmit speed: 0,4 Mbit/s	Node 3: 160 timeouts	R=160 T=712 R=80 T=417

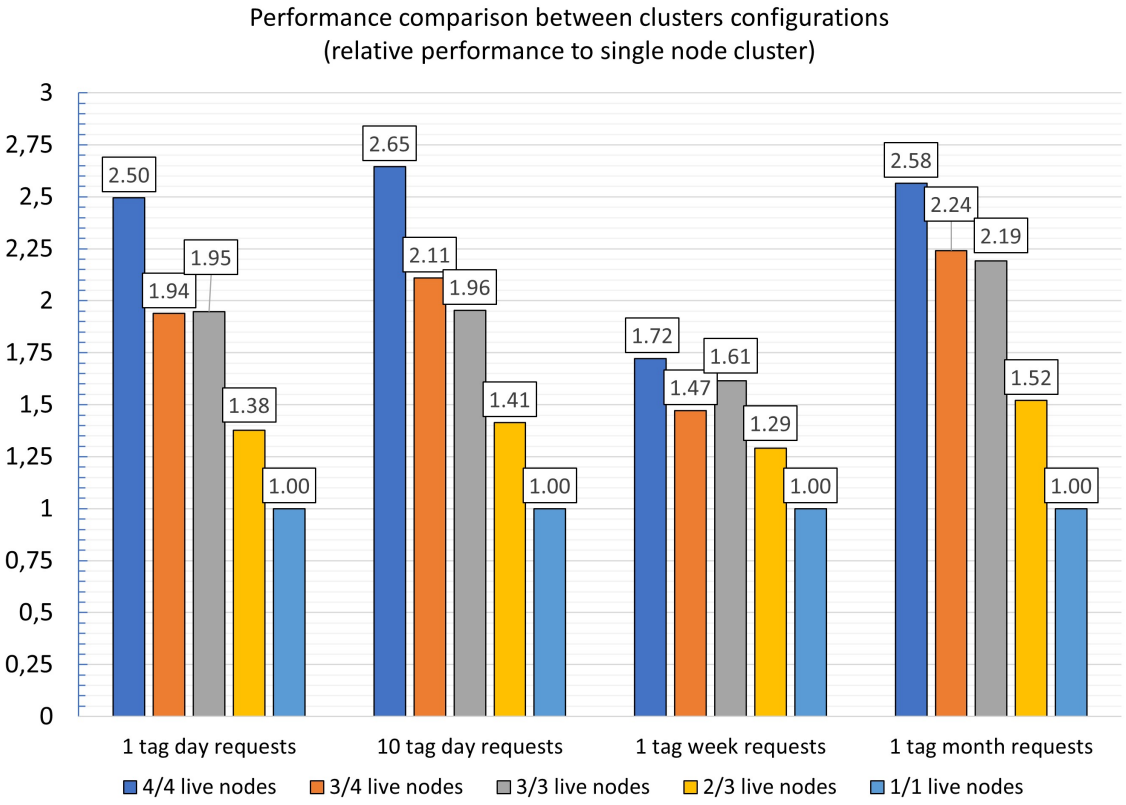

Figure 4.2. *Summary of performance test results. The values represent the relative amount of read requests completed per second compared to a single node cluster.*

5. CONCLUSIONS

This thesis aimed to investigate the behavior of a small-scale Apache Cassandra cluster as a part of a real-world application under various conditions. These included scaling a cluster, performing node disruption tests, and executing maintenance operations. The results obtained provide insights and offers a foundation for further development and optimization of the database in the system under testing.

The tests conducted in the thesis highlight the importance of monitoring the cluster's diagnostics and logs, as they serve as the primary indicators for performance, communication, and node health issues. Problems may not manifest immediately, but need to be identified and addressed before they accumulate. If this area is lacking, issues may not be detected in time, potentially leading to data loss and hindering the functioning of the system.

Examining cluster scaling revealed that horizontal scaling a single node cluster to three or four nodes was an efficient method to improve the databases performance and capacity. Based on the results, the single node to three node upgrade did not provide linear improvement to the number of servers added, but it was nearly double in terms of read requests completed. The fourth node improved the performance further 22% from the three node cluster, indicating that the method can be used to further scale a cluster. In on-premises systems, horizontal scaling requires buying additional hardware and configuring the present system to allow secure communications to it. A crucial point that emerged in the single node to multinode transition was that it might not be possible to scale to that kind of configuration with the existing single node network infrastructure. This highlights the importance of planning for potential multinode upgrades from the very beginning of doing on-premises installations, as it is expensive and might not be feasible to upgrade all of the networking components later on.

The tests demonstrated that a small-scale cluster can continue to operate and recover from node drops and unplanned network disconnections if proper actions are taken. The main problems in this area were handling network overloading that caused repairs and requests to fail in a limited environment, big tables causing repair timeouts, and improper preparation of the cluster for repairs. This included ensuring that all nodes are of same version and executing maintenance commands, such as updating SSTables to

most recent version. It was observed that running repairs on a per-table basis worked better when it was known that network issues were present. Furthermore, some issues were encountered not only on the database level but also on the client side, which handles the client-cluster communication, indicating that this component also requires attention.

Operating the database in these tests required quite a lot of manual work, but its important to note that such situations are not persistent and occur relatively infrequently, most often when something goes wrong or the overall topology is changed. The most common operations can be automated somewhat easily by establishing basic guidelines for the periodic maintenance tasks, and executing them through a task scheduler.

Various aspects of the system requires a good understanding of Cassandras internal details, and the hands-on approach to troubleshoot runtime issues and fine-tune the cluster played a big role in ensuring its overall health and stability. Various tools and enterprise solutions have been developed to help overcome these issues and to automate cluster management, which reduces the manual work required. However, exploration of these tools was beyond the scope of this thesis. Overall, when the cluster was configured correctly and the infrastructure was not a limiting factor, Cassandra was able to maintain high availability while delivering robust performance.

REFERENCES

[1] Carpenter, J. and Hewitt, E. *Cassandra: The definitive guide - 3rd Edition: Distributed data at web scale*. O'Reilly, 2022, pp. 3–7, 15–17, 24–29, 56–59, 63–65, 83–85, 108–118, 122–127, 147, 201–202, 210–211, 235, 247–268.

[2] Sahatqija, K., Ajdari, J., Zenuni, X., Raufi, B. and Ismaili, F. Comparison between relational and NOSQL databases. *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2018, pp. 216–220. DOI: 10.23919/MIPRO.2018.8400041.

[3] Foundation, T. A. S. *The Apache Cassandra Project Releases apache® Cassandra™ v4.0, the fastest, most scalable and secure Cassandra yet*. June 2021. URL: https://www.globenewswire.com/news-release/2021/07/27/2269647/17401/en/The-Apache-Cassandra-Project-Releases-Apache-Cassandra-v4-0-the-Fastest-Most-Scalable-and-Secure-Cassandra-Yet.html (visited on 01/26/2023).

[4] *Cassandra at eBay - Cassandra Summit 2012*. URL: https://www.slideshare.net/jaykumarpatel/cassandra-at-ebay-13920376 (visited on 05/06/2023).

[5] Ingram, B. *How discord stores trillions of messages*. 2023. URL: https://discord.com/blog/how-discord-stores-trillions-of-messages (visited on 05/06/2023).

[6] *Cloud Bigtable: Hbase-compatible, NoSql database*. URL: https://cloud.google.com/bigtable (visited on 05/05/2023).

[7] *DynamoDB: Everything you need to know about Amazon Web Service's NoSQL database*. URL: https://aws.amazon.com/dynamodb/ (visited on 05/05/2023).

[8] *Cassandra Basics*. URL: https://cassandra.apache.org/_/cassandra-basics.html (visited on 01/15/2023).

[9] *"Achieving Century Uptimes" An Informational eries on Enterprise Computing*. URL: https://www.shadowbasesoftware.com/wp-content/uploads/2016/08/What-Is-The-Availability-Barrier-Anyway.pdf (visited on 03/02/2023).

[10] *How are Cassandra transactions different from RDBMS transactions?* URL: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlTransactionsDiffer.html (visited on 01/19/2023).

[11] Harrison, G. *Next generation databases: Nosql, NewSQL, and Big Data*. Apress, 2015, pp. 43–46, 134–136.

[12] *Introduction*. URL: https://cassandra.apache.org/doc/latest/cassandra/data_modeling/intro.html (visited on 01/19/2023).

[13] *Data Modeling in Apache Cassandra™*. URL: https://www.datastax.com/resources/whitepaper/data-modeling-apache-cassandra.

[14] Kan, C. Y. *Cassandra Data Modeling and Analysis: Design, build, and analyze your data intricately using Cassandra*. Packt Publishing, 2014, pp. 17, 30.

[15] *Data types*. URL: https://cassandra.apache.org/doc/latest/cassandra/cql/types.html (visited on 01/28/2023).

[16] *Apache Cassandra™ architecture*. URL: https://www.datastax.com/resources/whitepaper/apache-cassandratm-architecture.

[17] Hall, A. *Distributed database things to know*. Mar. 2019. URL: https://www.datastax.com/blog/distributed-database-things-know-cassandra-datacenter-racks (visited on 02/02/2023).

[18] Gorbenko, A., Romanovsky, A. and Tarasyuk, O. Interplaying Cassandra NoSQL Consistency and Performance: A Benchmarking Approach. *Dependable Computing - EDCC 2020 Workshops*. Ed. by S. Bernardi, V. Vittorini, F. Flammini, R. Nardone, S. Marrone, R. Adler, D. Schneider, P. Schleiß, N. Nostro, R. Løvenstein Olsen, A. Di Salle and P. Masci. Cham: Springer International Publishing, 2020, pp. 168–184. ISBN: 978-3-030-58462-7.

[19] Neeraj, N. *Mastering Apache Cassandra - Second Edition*. Packt Publishing, 2015, p. 38.

[20] *How data is distributed across a cluster (using virtual nodes)*. URL: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeDistribute.html (visited on 02/09/2023).

[21] *Learn how commitlog works in Apache Cassandra*. URL: https://cassandra.apache.org/_/blog/Learn-How-CommitLog-Works-in-Apache-Cassandra.html (visited on 02/26/2023).

[22] *How is data written?* URL: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlHowDataWritten.html (visited on 02/23/2023).

[23] *FAQ - Why is forcing major compaction on a table not ideal?* URL: https://support.datastax.com/s/article/FAQ-Why-is-forcing-major-compaction-on-a-table-not-ideal (visited on 02/18/2023).

[24] *Read repair*. URL: https://cassandra.apache.org/doc/latest/cassandra/operating/read_repair.html (visited on 02/23/2023).

[25] *Remove read_repair_chance/dclocal_read_repair_chance*. URL: https://issues.apache.org/jira/browse/CASSANDRA-13910 (visited on 02/23/2022).

[26] *How is data deleted?* URL: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlAboutDeletes.html (visited on 02/25/2023).

[27] *How is data read?* URL: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlAboutReads.html (visited on 02/26/2023).

[28] *Cassandra Documentation: Nodetool Usage*. URL: https://cassandra.apache.org/doc/latest/cassandra/tools/nodetool/nodetool.html (visited on 03/02/2023).

[29] *When to run anti-entropy repair*. URL: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/operations/opsRepairNodesWhen.html (visited on 03/04/2023).

[30] *Adding or removing nodes, datacenters, or clusters*. URL: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/operations/opsAddingRemovingNodeTOC.html (visited on 03/04/2023).

[31] *Replacing a running node*. URL: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/operations/opsReplaceLiveNode.html (visited on 04/01/2023).

[32] *Monitoring*. URL: https://cassandra.apache.org/doc/latest/cassandra/operating/metrics.html (visited on 03/11/2023).

[33] *Essentials of the JMX API*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/tutorial/essential.html (visited on 05/06/2023).

[34] *Batch*. URL: https://docs.datastax.com/en/dse/6.0/cql/cql/cql_reference/cql_commands/cqlBatch.html (visited on 03/11/2023).

[35] Datastax. *Datastax/CPP-driver: Datastax C/C++ driver for Apache Cassandra*. URL: https://github.com/datastax/cpp-driver (visited on 04/25/2023).

[36] Huss, R. *Jolokia: Overview*. URL: https://jolokia.org/.

[37] *Important metrics and alerts*. URL: https://docs.datastax.com/en/dseplanning/docs/metricsandalerts.html (visited on 05/06/2023).

[38] URL: https://support.datastax.com/s/article/Common-Causes-of-GC-pauses (visited on 03/19/2023).

[39] *Cassandra Docker image*. URL: https://hub.docker.com/_/cassandra.

[40] *Security*. URL: https://cassandra.apache.org/doc/latest/cassandra/operating/security.html (visited on 05/07/2023).

[41] *Cassandra.yaml file configuration*. URL: https://cassandra.apache.org/doc/latest/cassandra/configuration/cass_yaml_file.html (visited on 04/27/2023).

[42] *Migrate from Cassandra 3.x to 4.X*. URL: https://www.datastax.com/learn/whats-new-for-cassandra-4/migrating-cassandra-4x (visited on 05/07/2023).

APPENDIX A: TEST FIGURES

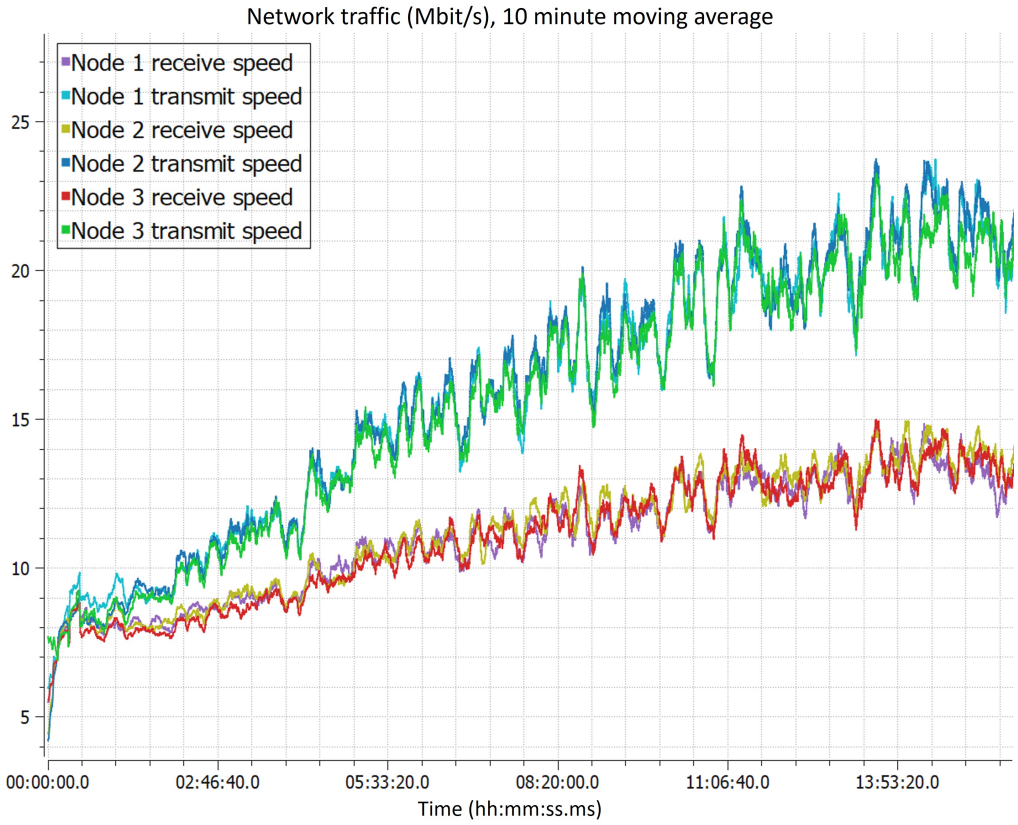

Figure A.1. *Network traffic during moderate write and read load*

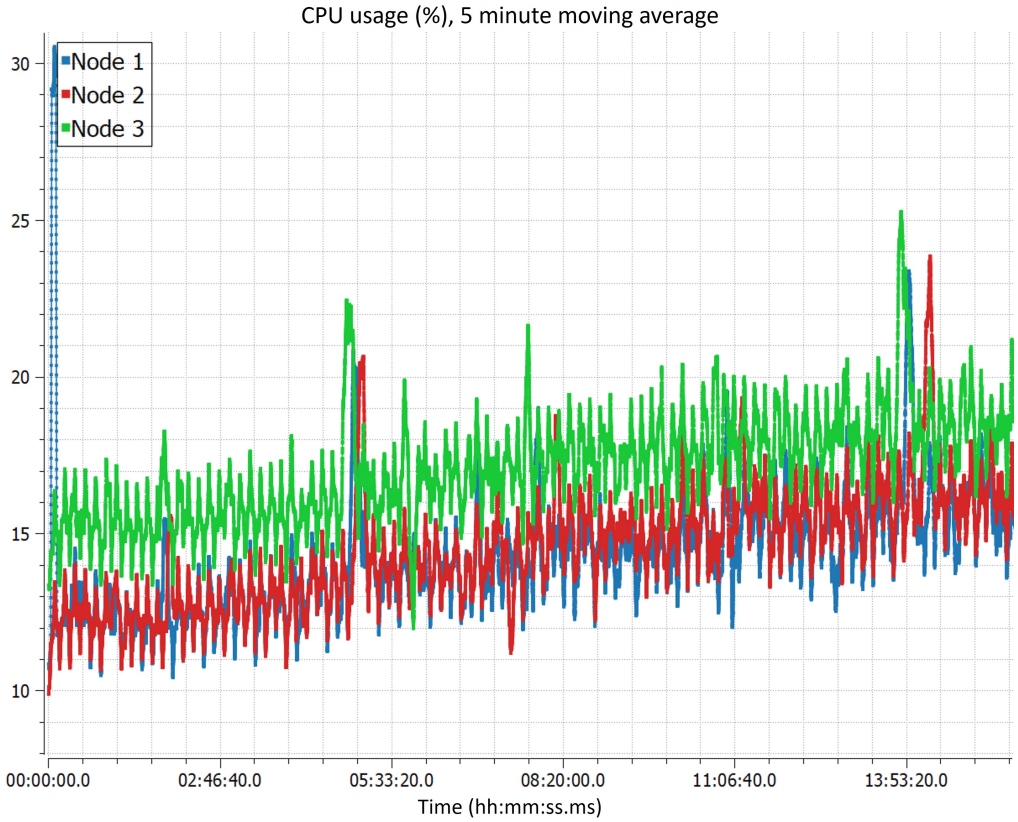

Figure A.2. *CPU usages during moderate write and read load*

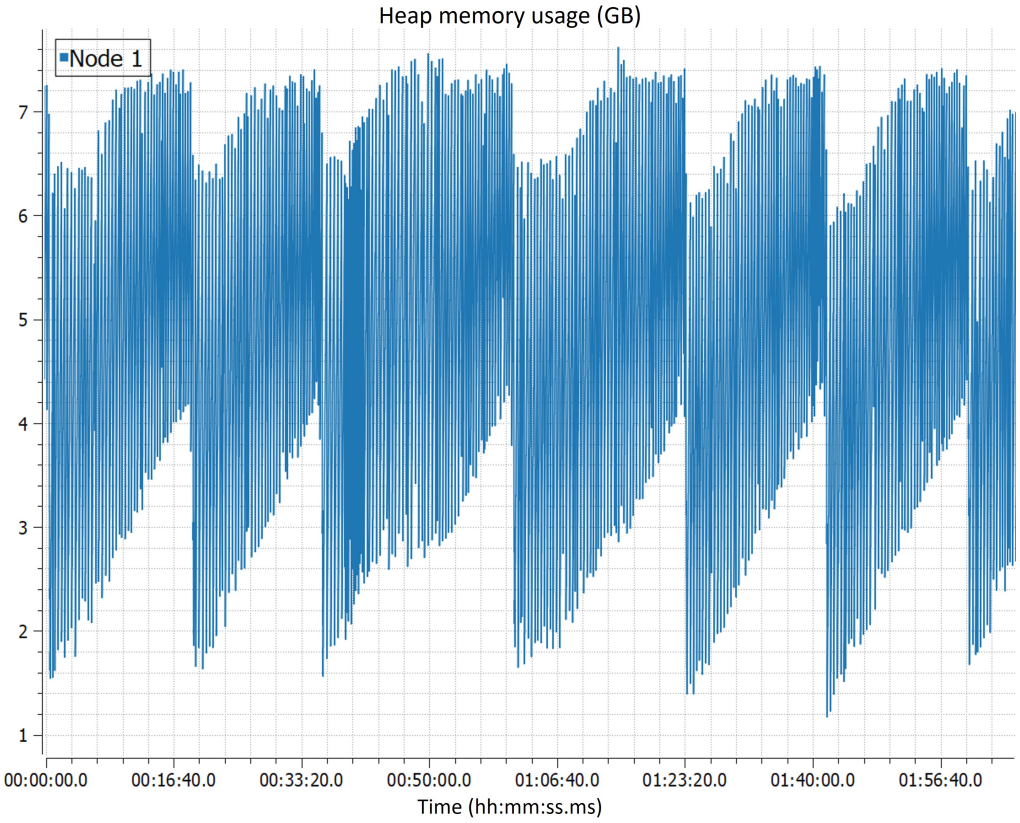

Figure A.3. *Heap memory usage during moderate write and read load*

Figure A.4. *Disk usage during moderate write and read load*

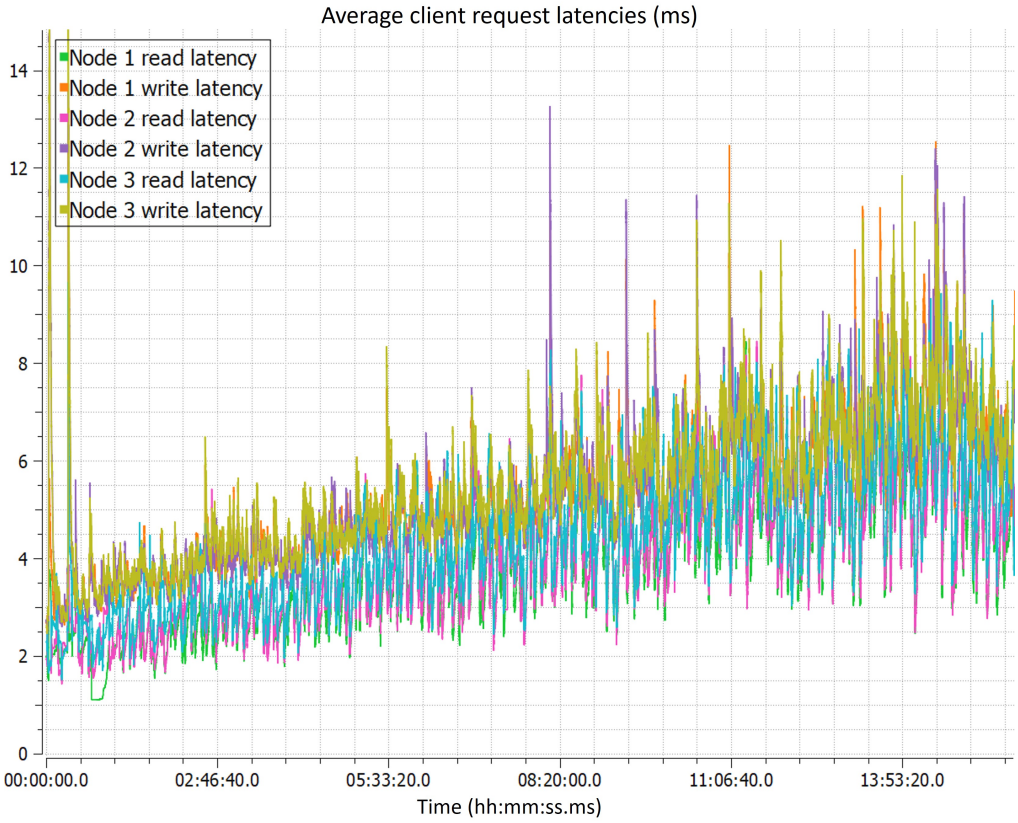

Figure A.5. *Client request latencies during moderate write and read load*

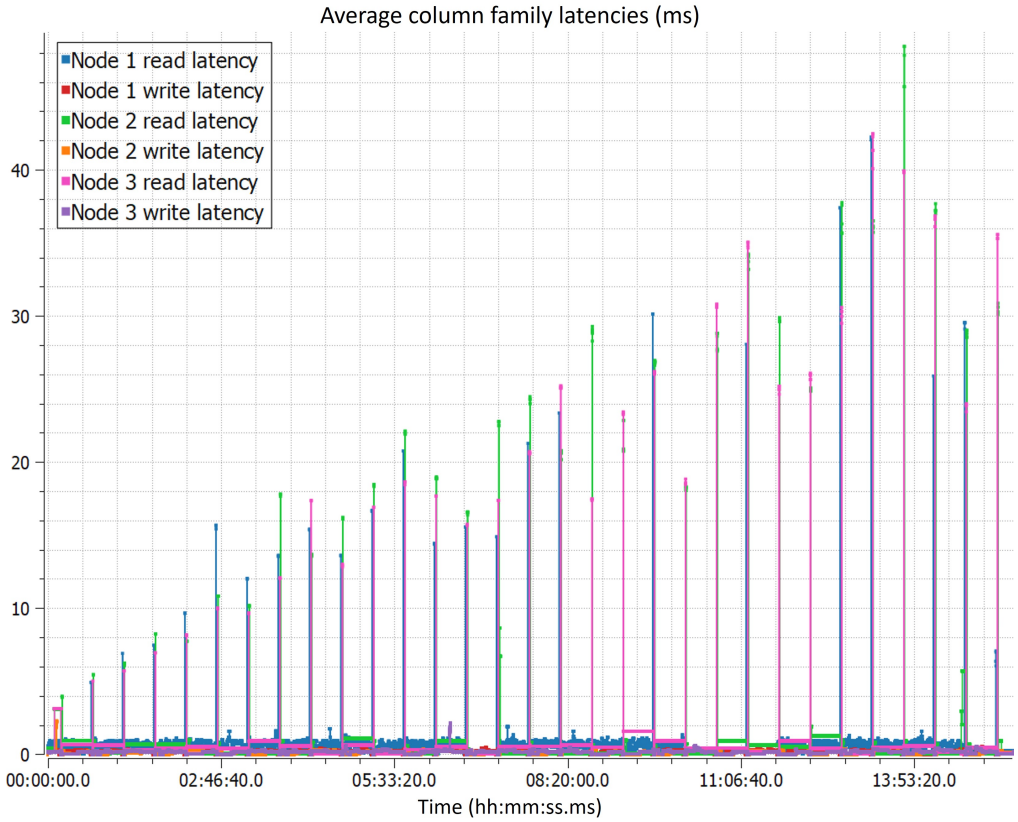

Figure A.6. *Column family latencies during moderate write and read load*

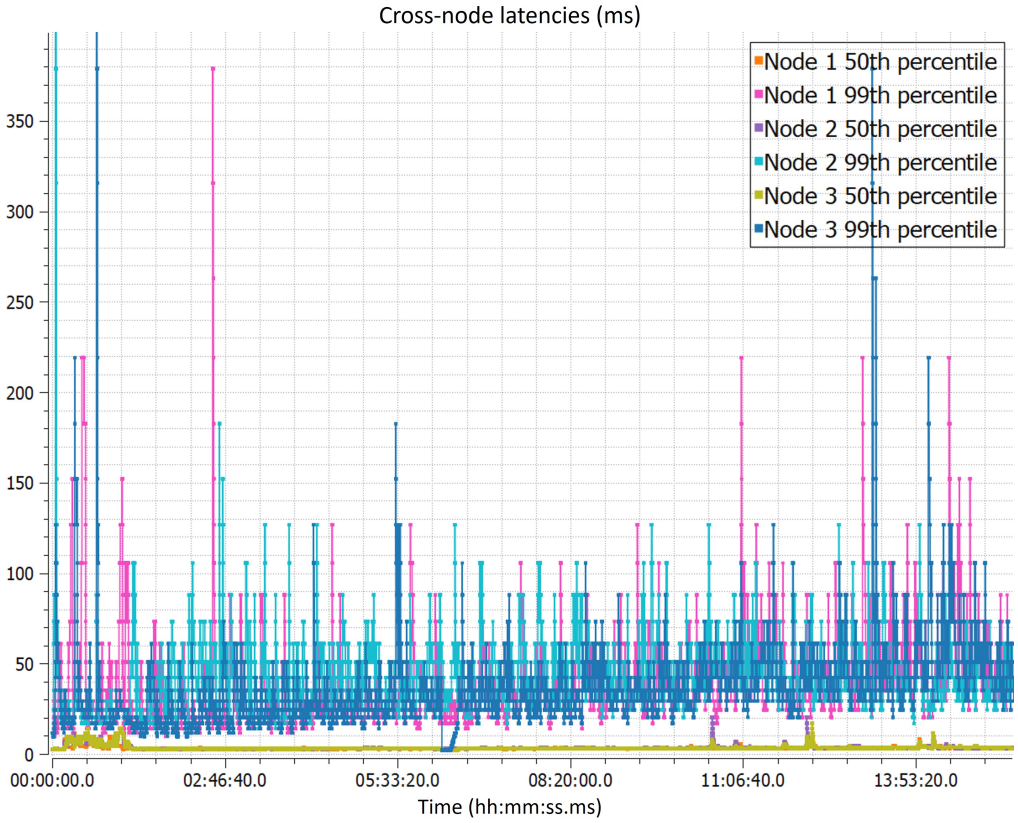

Figure A.7. *Cross-node latencies during moderate write and read load*

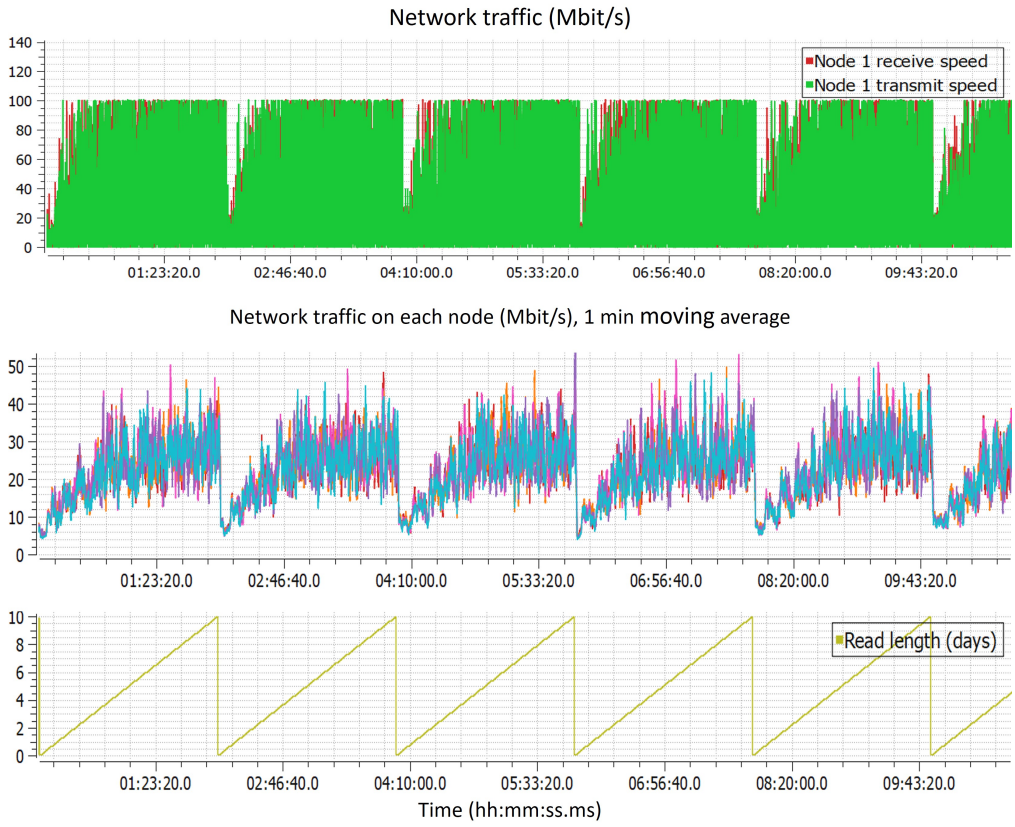

Figure A.8. *Network traffic during 4.5.3 tests*

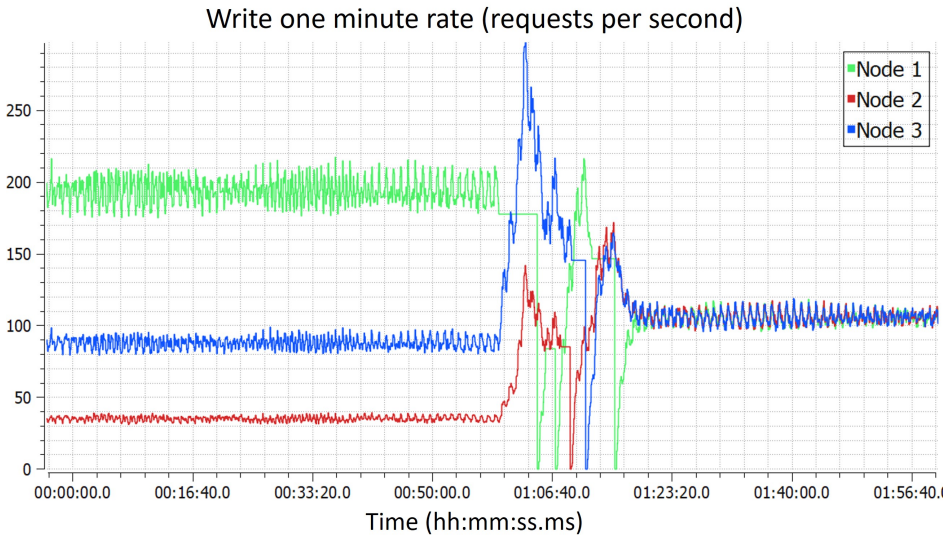

Figure A.9. *Write request rates before and after middleware service restarts during 4.9 tests*

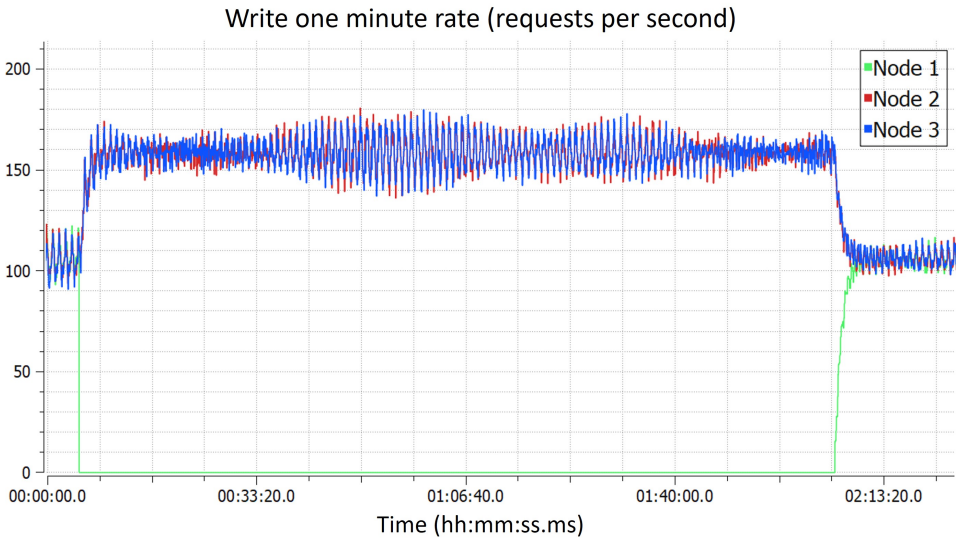

Figure A.10. *Write request rates with one node down*