Table of Contents

List of Figures

Solutions

1 Cassandra Cluster Management

1.1 Question 1

Q: What happens if one misconfigures the CASSANDRA_CLUSTER_NAME variable in the docker-compose.yml?

A cluster name mismatch issue may occur if the CASSANDRA_CLUSTER_NAME is incorrectly set in the docker-compose.yml file. Cassandra will raise an exception such as "saved name cluster A!= configured name cluster B" when a node attempts to join a cluster with a different name. This misconfiguration would make it impossible for the Cassandra nodes to correctly form a cluster in a Docker environment. If the problem is not identified early, each node would effectively think it is part of a distinct cluster, which might result in communication failures, the inability to replicate data across nodes, and possible data discrepancies.

Making ensuring that every node in the docker-compose.yml file has the same CASSANDRA_CLUSTER_NAME value is going to rectify this. It might be necessary to delete the data and begin over if the cluster has already been formed with names that don't match. It was also advised to wipe out the /var/lib/cassandra directory.

1.2 Question 2

Q: What is the purpose of the CASSANDRA_SEEDS variable in the docker-compose.yml? Is it necessary to list all the servers?

The creation and upkeep of Cassandra clusters depend heavily on the CASSANDRA_SEEDS setting in docker-compose.yml. It identifies the Cassandra cluster's seed nodes. Special nodes known as "seed nodes" are employed during the bootstrap procedure when a node first boots up and needs to learn the cluster structure.
The following are the main functions of seed nodes:

1. To assist new nodes in learning the topology of the cluster
2. To help with the process of cluster bootstrapping
3. To make the gossip protocol easier to use to preserve cluster state data

It is crucial to remember that listing every server as a seed is neither required nor advised. A relatively modest number of seed nodes is advised by best practices.

Regardless of the cluster's overall size, one would normally designate two to three seed nodes.

Because too many seed nodes might slow down the gossip process and make cluster formation more complex, the number is limited. Initial contact and discovery are the main functions of seed nodes. Regardless of whether they were identified as seeds or not, a node will use the gossip protocol to learn about every other node after joining the cluster.

Typically, the first two to three nodes in the cluster could be designated as seeds. For instance,

```
CASSANDRA_SEEDS: "cass1,cass2"
```

This setup instructs every node to join the cluster by using cass1 and cass2 as initial contact points. These two seed nodes will be enough to build and sustain the cluster, even if it includes many more nodes.

1.3 Question 3

Q: What does the UN mean for the first column of the output from **nodetool status**?

The state of every node in the Cassandra cluster is represented by a two-letter code that appears in the first column of the output of the nodetool status command. One of the most popular and sought-after statuses one will come across is "UN". "Up" and "Normal" are what it stands for.

To break it down even more:

- U: Up indicates that the node is online and answering inquiries.
- N: Normal, signifying that the node is operating as intended and actively taking part in cluster activities.

A node is in an ideal state when it displays "UN" in the nodetool status report. It is completely functional, taking part in all cluster operations, including data replication and repair procedures, and responding to read and write requests.

Other status codes you may come upon are:

- DN: Down and Normal (the node was in a normal condition before to going down but is now offline).
- UJ: Up and Joining (the cluster is being joined by the node, which is online).
- UL: Up and Leaving (the node is online but is leaving the cluster in a gentle manner).

- UM: Up and Moving (the node is traveling to a new location within the token ring while still being online).
- DL: Going down and departing (the node was going out when it went down and is now offline).

All nodes in a healthy Cassandra cluster should normally have the "UN" status. It shows that the cluster is completely functional and that every node is enhancing the overall dependability and performance of the system.

1.4 Question 4

Q: In our cluster consisting of containers, where does Cassandra store data?

The data storage method in a containerized Cassandra cluster is made to strike a compromise between the necessity for dependable, long-lasting data storage and the transient nature of containers. Cassandra saves its data in the file system of the container by default. However, when containers are stopped or removed, this method alone would cause data loss, which is inappropriate for a database system.

To overcome this, Cassandra deployments in containers usually employ volume mappings to preserve data for longer than the lifespan of a single container. These volume mappings link Docker-managed named volumes or directories on the host system to directories within the container.

The data storage technique is different from standard settings in the Cassandra cluster configuration that is given, as specified by the docker-compose.yml file. Volume mappings for the Cassandra containers (db1, db2, and db3) are not specified in the setup. This implies that Cassandra would not have any permanent storage on the host computer; instead, all its data would be stored in the container's file system.

Figure 1 shows the given configuration of the docker-compose.yml file used for project 3. Under this setup, each container's /var/lib/cassandra directory would house Cassandra's data files, such as SSTables, commit logs, and other structures. Each container's /etc/cassandra directory would include configuration files. These folders would only be present in the container's file system because there are no volume mappings.

This arrangement has important ramifications. All data contained in a container would be lost if it were stopped or removed. This covers any configuration modifications and database content made after the container was launched. In essence, a new Cassandra instance would be launched each time the containers are restarted

using docker compose up. There would be no access to any data that was written during the prior cycle.

```
services:
  client:
    build: .
  db1:
    image: cassandra:4.1.3
    environment:
      - HEAP_NEWSIZE=128M
      - MAX_HEAP_SIZE=512M
      - CASSANDRA_CLUSTER_NAME=ece573-prj03
      - CASSANDRA_SEEDS=db1,db2,db3
  db2:
    extends: db1
  db3:
    extends: db1
```

Figure 1: docker-compose.yml file used for project 3.

While the containers are operating, the Cassandra cluster would continue form and function correctly even without persistent storage. According to Cassandra's settings, data would be duplicated between nodes, but only inside the containers that are now operating.

Since there would be no mapping to external volumes and data would be written directly to the container's file system, this configuration could provide somewhat improved I/O performance. It's crucial to remember that this setup would work better in testing or development settings when it's preferable to start from scratch every time, but it wouldn't work in any circumstance where data persistence is necessary.

It would be advised to include volume mappings to the docker-compose.yml file for production environments or any situation where data permanence is required. By doing this, data would be permanently saved on the host system or a networked storage solution, enabling it to withstand container removals and restarts.

1.5 Screenshots of task completion

The following set of figures portrays the completion of tasks necessary for project 3 until the end of the main heading Cassandra Cluster Management.

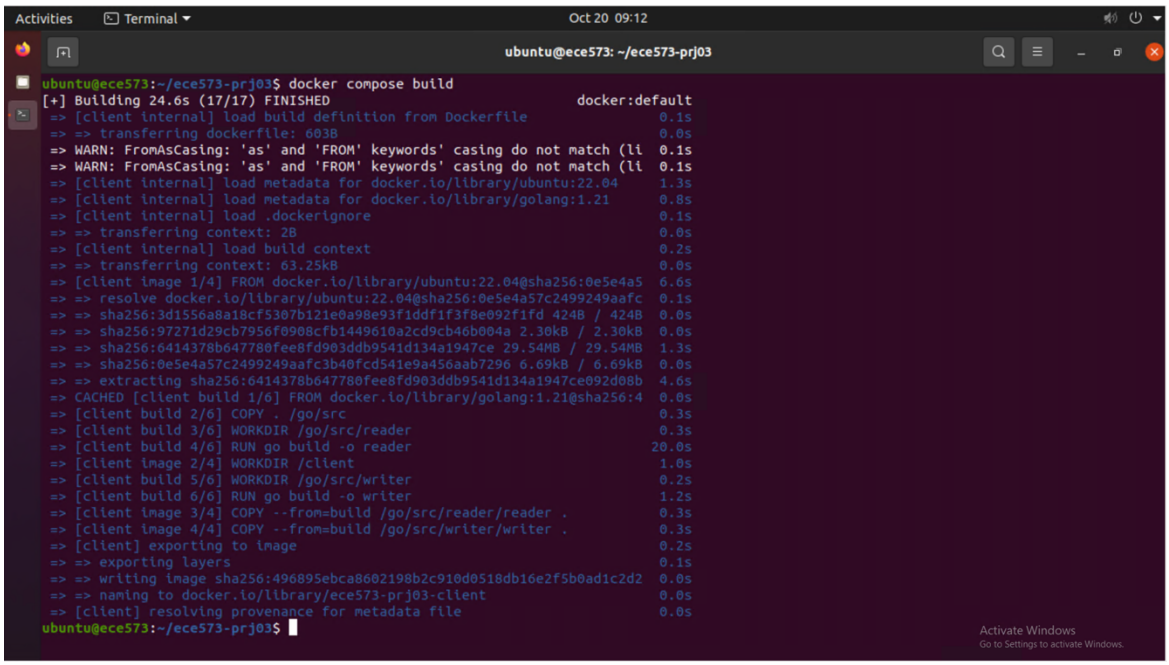

Figure 2: Running docker compose build command.

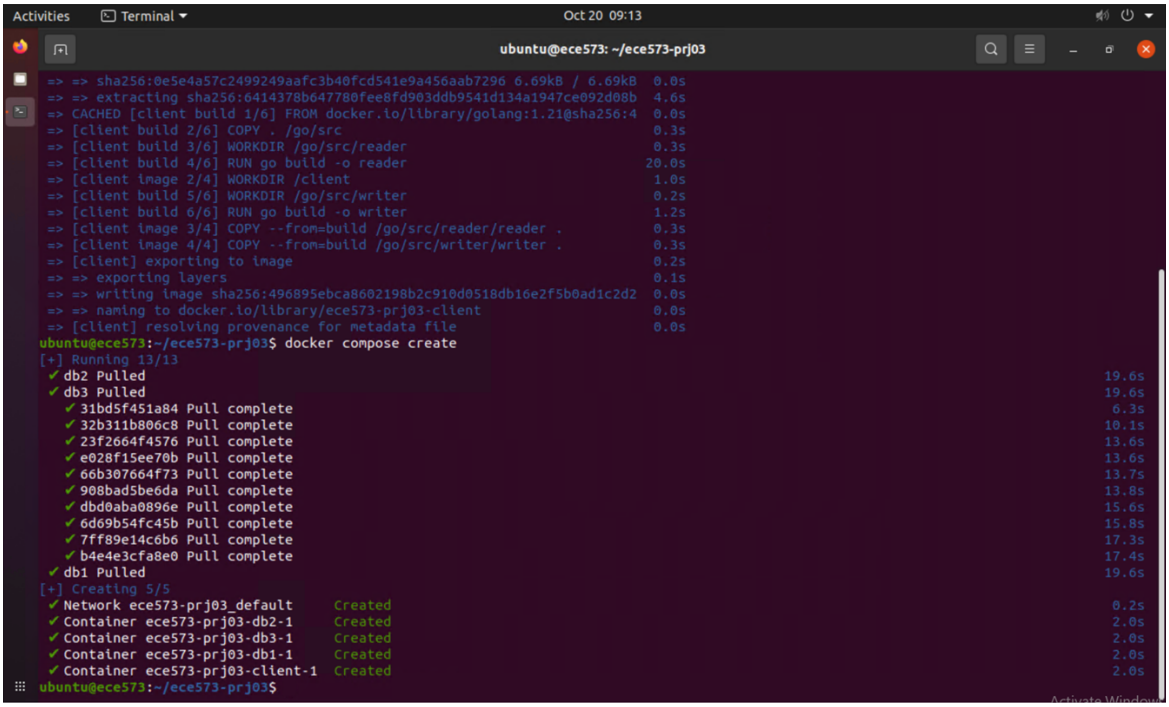

Figure 3: Running docker compose create command.

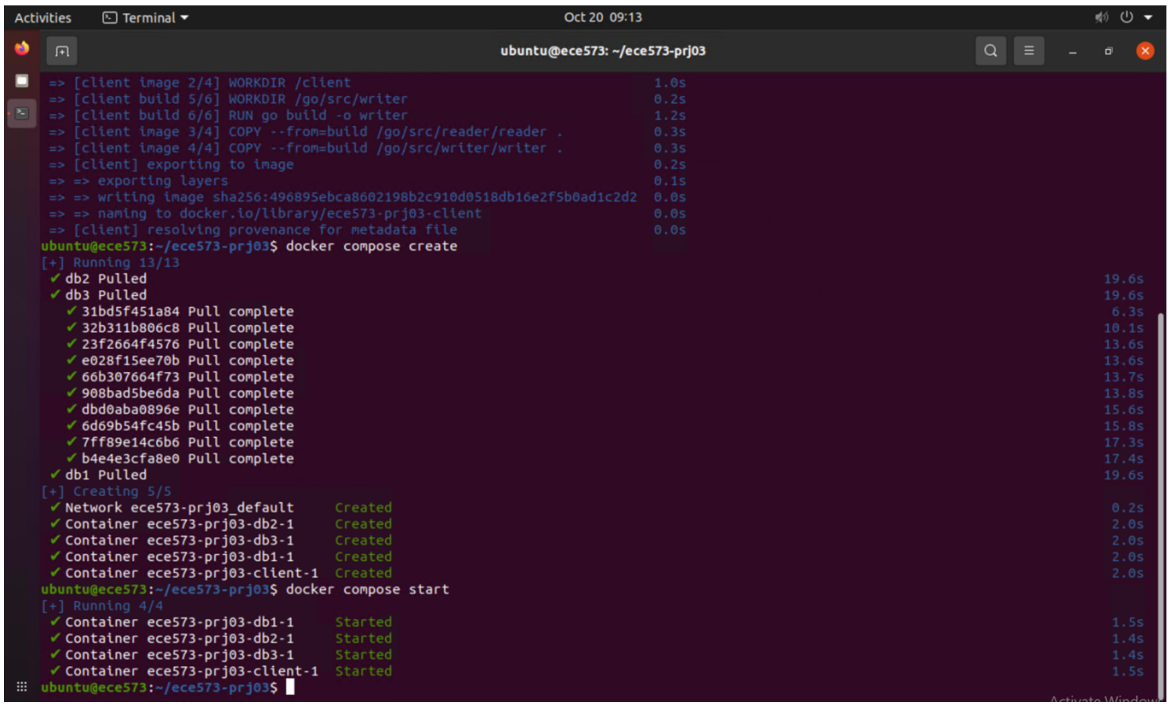

Figure 4: Running docker compose start command.

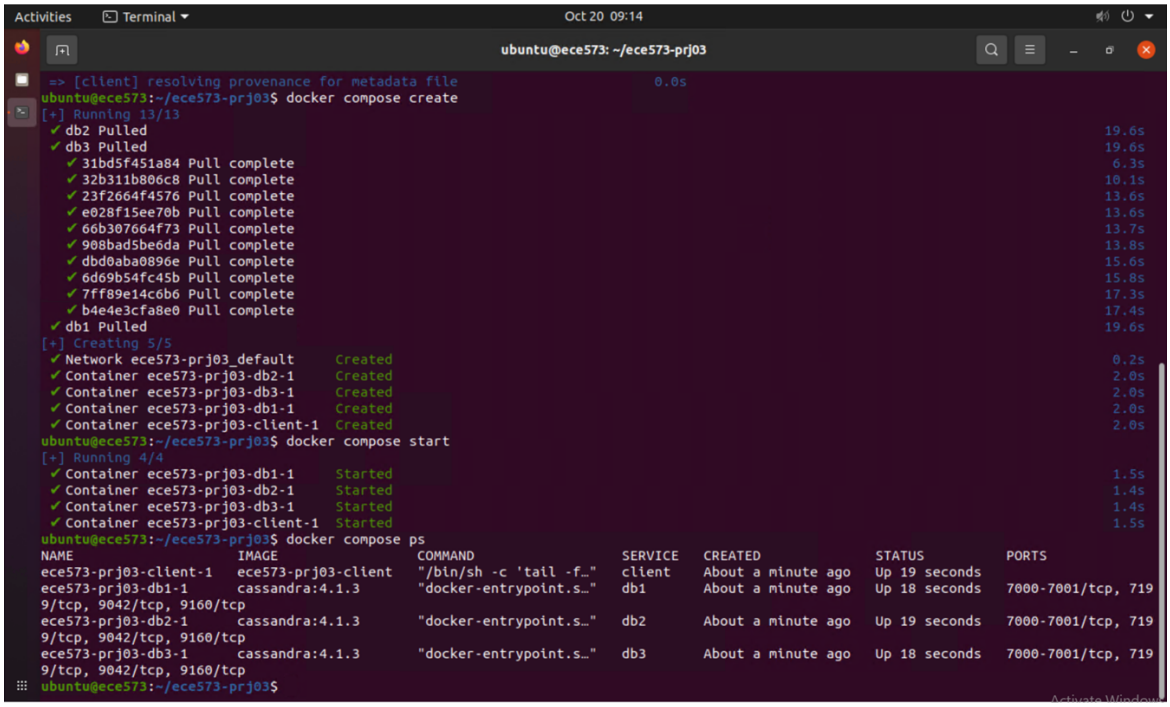

Figure 5: Running docker compose ps command for status.

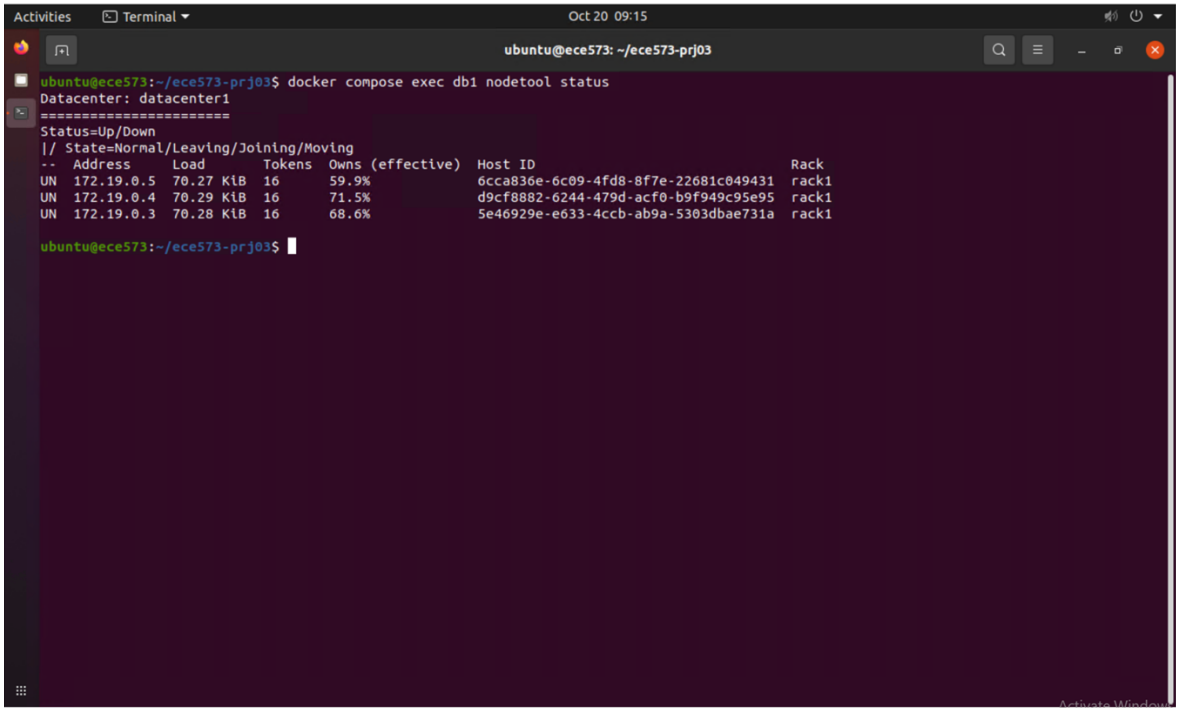

Figure 6: Running docker compose exec db1 nodetool status command.

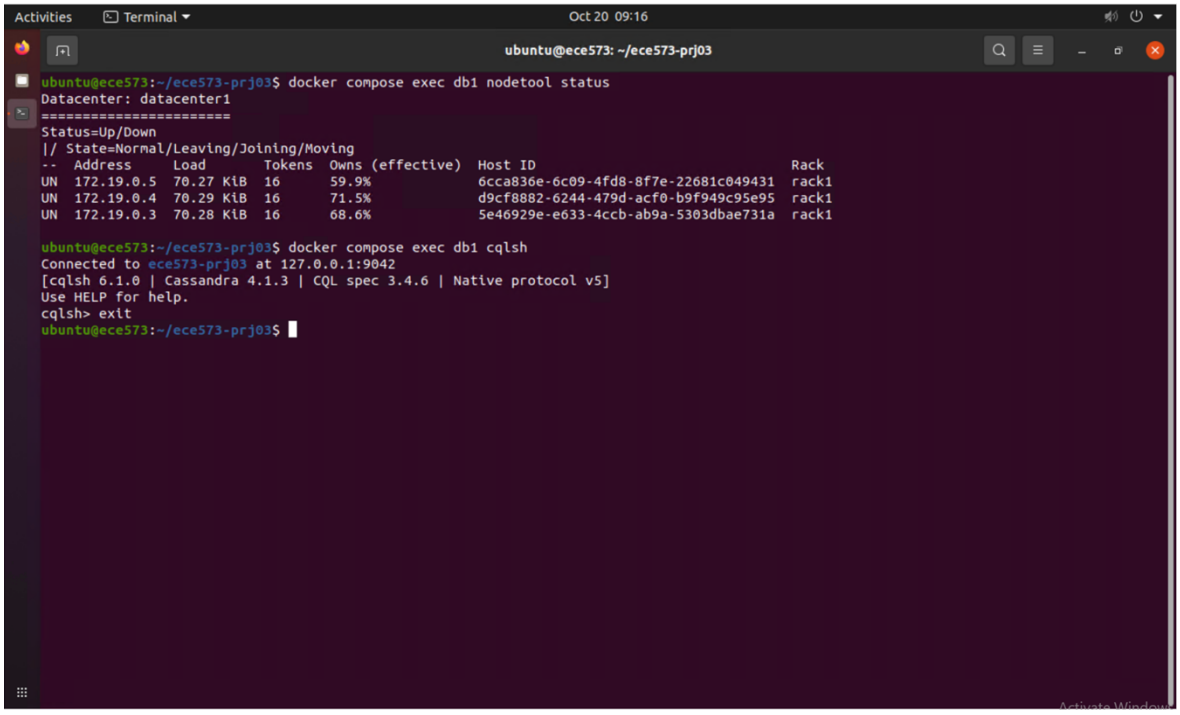

Figure 7: Running docker compose exec db1 cqlsh command and exiting the CQL command line with exit command.

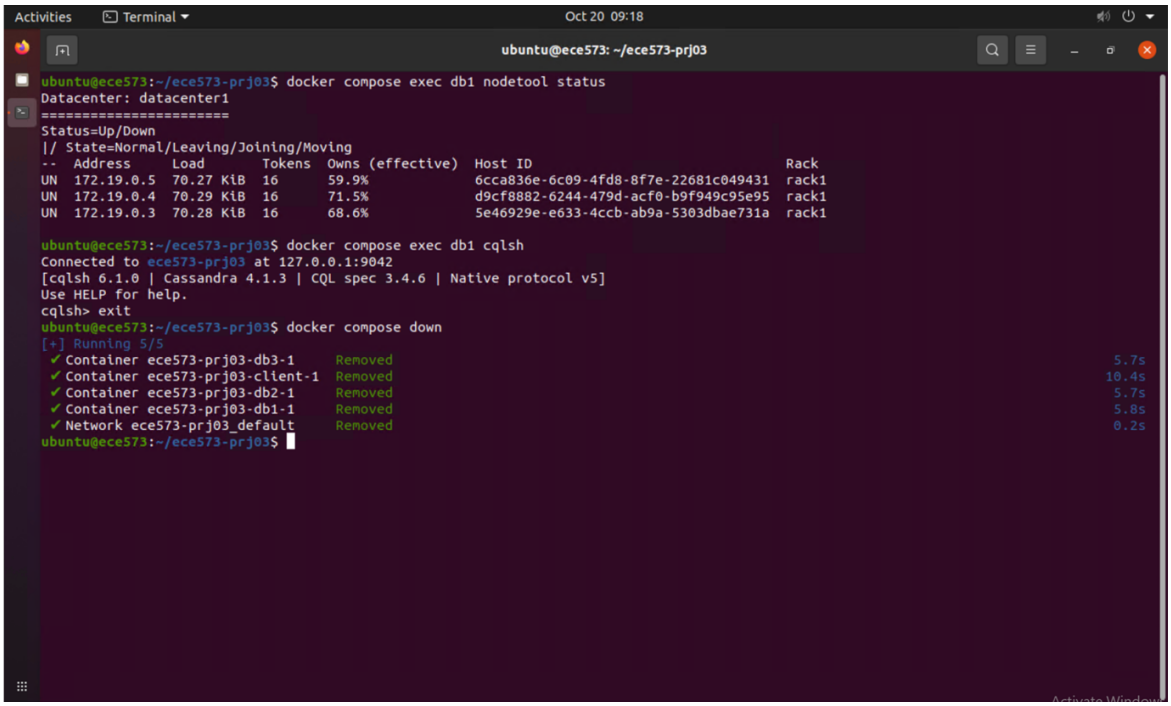

Figure 8: Running docker compose down command.

2 Tunable Consistency

2.1 Question 1

Q: Keep both reader and writer running. In a few minutes you should be able to see that reader reports missing data. Provide screenshots and briefly explain what is happening.

When the commands,

```
docker compose exec client writer test ONE db1
docker compose exec client reader test ONE db2
```

are run in separate terminal windows and left for a while, the following is observed.

In the below Figure 9, the terminal window on the left is running the writer code and rows are being inserted. Similarly, the right terminal window is running the reader code and after a while of the code running, there is evidence of missing data which the reader is pointing out to.

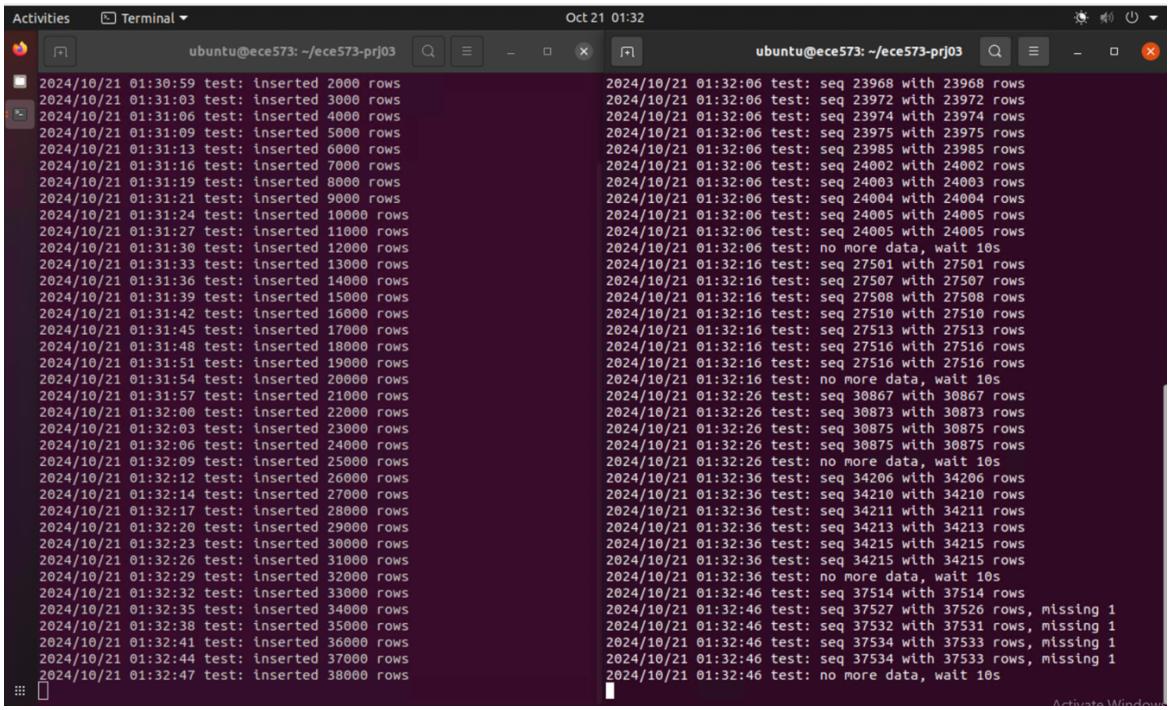

Figure 9: The two terminal windows running each the reader and the writer.

The reader would probably indicate missing data if both the writer and the reader were using consistency level ONE. The relaxed consistency level and Cassandra's eventual consistency mechanism are directly responsible for this behavior.

As soon as a write operation is acknowledged by even one duplicate, the writer, with consistency level ONE, deems it successful. This implies that initially, the data may only be written to one cluster node. In the meantime, the reader obtains information from any one copy that answers its query first, also with consistency level ONE.

A race situation is created by this configuration in which:

1. The writer may write to node A.
2. Before the data has been duplicated from A to B, the reader may read from node B.
3. The freshly written content is thus overlooked by the reader.

The gaps in the reader's reported value sequence would represent the missing data. These inconsistencies won't completely disappear under continuous write operations with this consistency level, but they should gradually diminish as Cassandra's internal processes (such as read repair and implied handoff) strive to spread the data across all replicas.

2.2 Question 2

Q: You can terminate the reader by Ctrl+C now and then restart it with the same arguments. Does it report any missing data? Provide screenshots and briefly explain what is happening.

The following Figure 10 shows the stopping of the reader using the Ctrl+C command and then restarting it with the command,

```
docker compose exec client reader test ONE db2
```

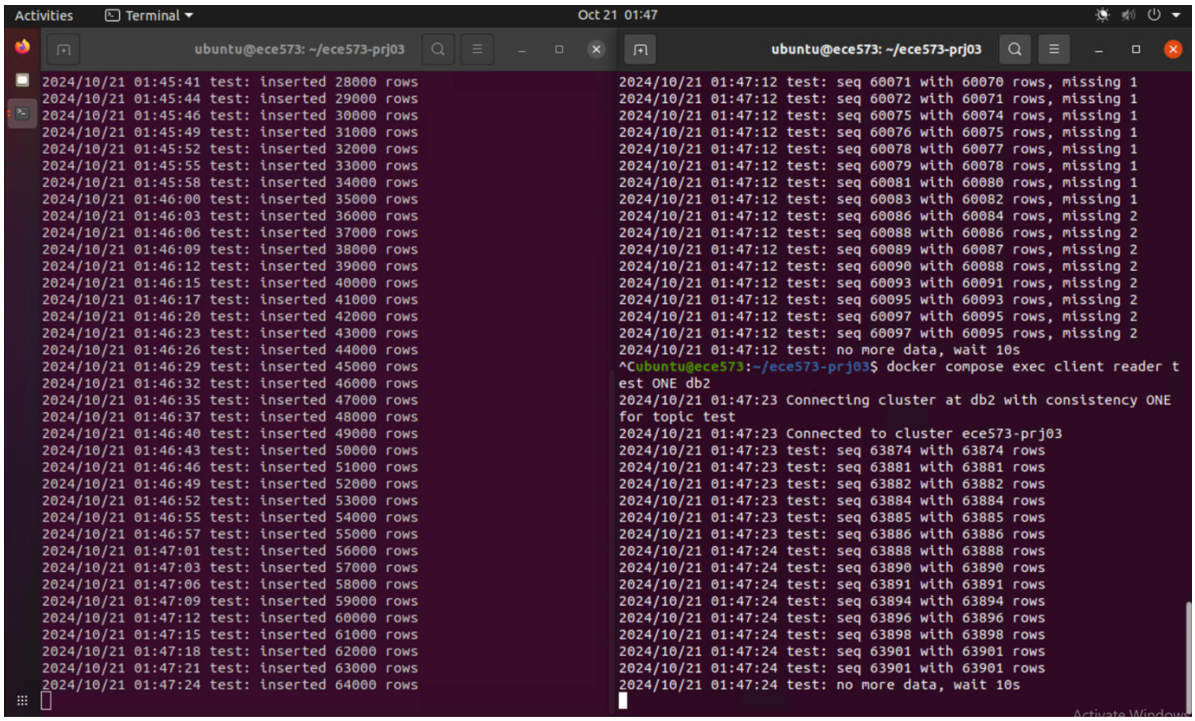

Figure 10: The reader in the terminal window on the right stopped and restarted again immediately.

The reader may not detect missing data right away after restarting. There are several reasons for this behavior:

1. Catch-up period: Cassandra's background actions, such as read repair and suggested handoff, have a chance to spread data among all replicas during the reader's halt, free from the disruption of continuous read operations.
2. Starting point: Since the reader most likely picks up where it left off, it is now reading latest data that has had more time to spread throughout the cluster.
3. Random node selection: When consistency level ONE is used, the reader may unintentionally reconnect to a more recent replica.

It's crucial to remember, though, that this seeming stability is frequently fleeting. New inconsistencies could still appear as the reader keeps working, particularly if write operations are still going on. This is because the reader is still only querying

one copy for each read operation, which may or may not have the most current data, which is the basic problem with the ONE consistency level.

With time, one may notice:

1. At certain points there seems to be consistency and no missing data.
2. Suddenly missing data appears when the reader queries less recent replicates.
3. Missing data gradually disappears as the cluster converges to consistency.

This behavior highlights Cassandra's ultimate consistency mechanism and the availability vs. consistency trade-off when employing lower consistency levels.

2.3 Question 3

Q: Stop both the reader and writer and restart them with a different topic and the QUORUM consistency. Let them run for a few minutes. Is the reader reporting any missing data and does that match your expectation? Provide screenshots and briefly explain why.

After stopping both the reader and writer executions in both the terminals, the following commands in each different terminal are run,

```
docker compose exec client writer newTopic QUORUM db1
docker compose exec client reader newTopic QUORUM db2
```

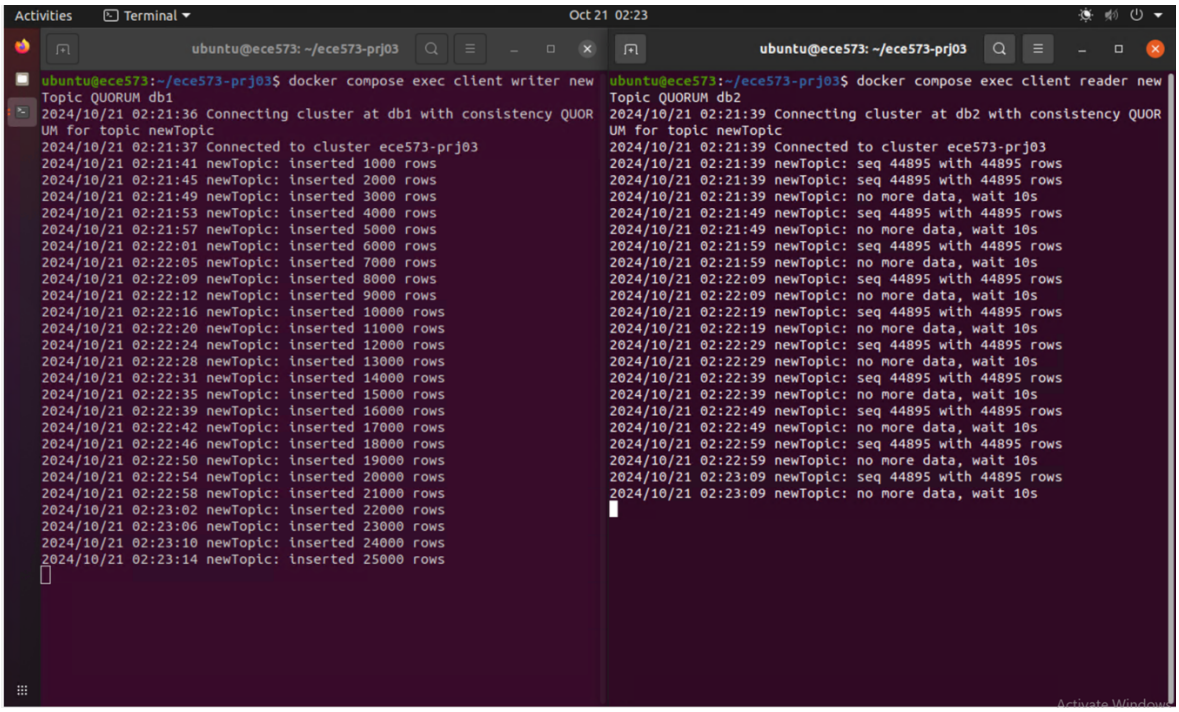

Figure 11: The run of different topic with QUORUM consistency.

The above Figure 11 illustrates through a screenshot of the way that different topic and QUORUM consistency is handled.

The reader should report little or no missing data when there is QUORUM consistency for both the writer and the reader. The stronger assurances offered by the QUORUM consistency level are to blame for this notable increase in data consistency:

1. Writer behavior: To be deemed effective with QUORUM consistency, a writer must receive recognition from most replicas. This implies that a minimum of two nodes in a cluster consisting of three must verify the write.
2. Reader behavior: In a similar vein, a reader who practices QUORUM consistency will contact most replicas, compare their answers, and then provide the outcome. Based on timestamps, it will provide the most recent version of the information.

A better likelihood of consistent readings is ensured by this configuration because:

- Before validating the write, the writer made sure that most of the cluster had the data.
- Before delivering a result, the reader is consulting with most of the cluster.

It's crucial to remember that although QUORUM greatly minimizes discrepancies, it doesn't always totally eradicate them. Rarely, you could still notice slight discrepancies because of things like:

- Network partitions.
- Issues with timing in a highly concurrent setting.
- Temporary node failures.

Because the system must wait for replies from most nodes rather than just one, there is a minor increase in latency for both read and write operations as a trade-off for this enhanced consistency.

2.4 Question 4

Q: Repeat the above experiment using writer with the ALL consistency and reader with the ONE consistency. Don't forget to change to a different topic. Provide a screen shot and briefly explain your observations.

The following Figure 12 represents the tow terminals running each a writer and a reader operation by starting a different topic with the ALL consistency by the writer and the ONE consistency by the reader.

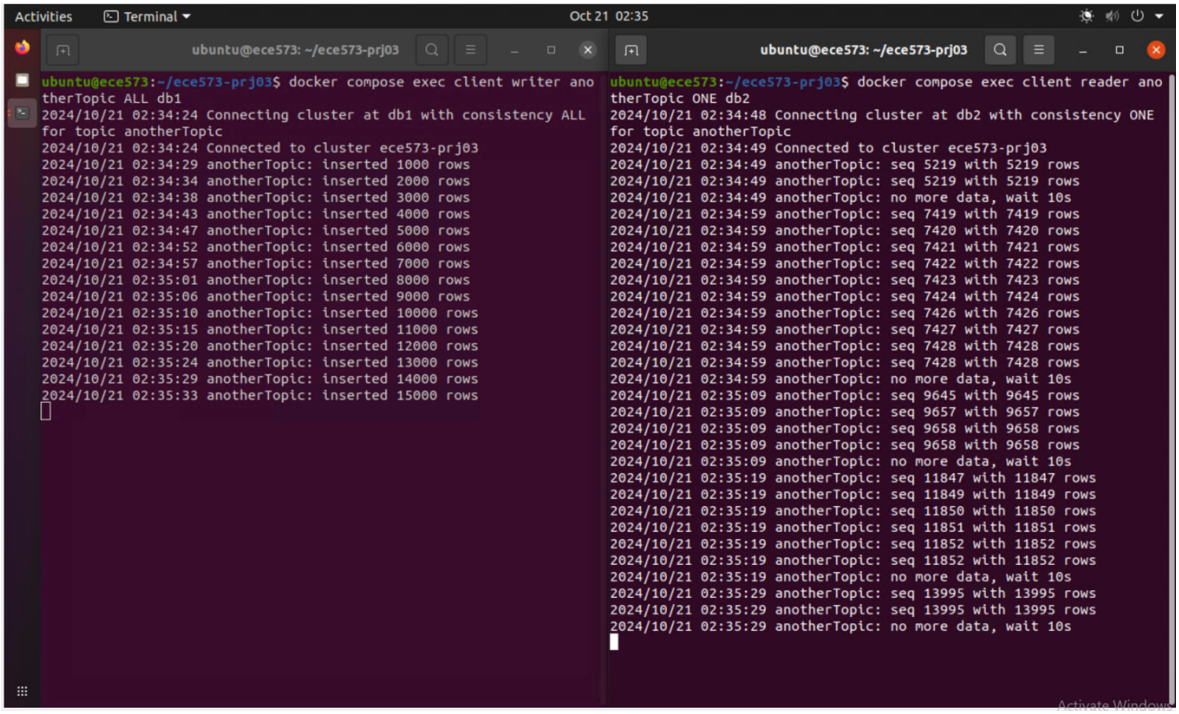

Figure 12: The run of a writer with ALL consistency and a reader with ONE consistency for a different topic on both.

In this case, when the reader uses ONE consistency and the writer uses ALL, you should see that the reader has not reported any missing data. Strong write consistency and relaxed read consistency are produced by this combination:

- Writer behavior (ALL): Before deeming a write operation successful, the writer needs confirmation from every replica in the cluster. This guarantees that, prior to the write being validated, all nodes inside the cluster possess the most recent data.
- Reader behavior (ONE): To satisfy a read request, the reader only must query one single replica.

There are no missing data with this setup because:

- Before starting any write operation, make sure all nodes have the data.
- The reader may query any single node, and it will always provide the most recent information.

Principal findings:

1. Because the writer must wait for every node to acknowledge the write, the ALL consistency level will result in a greater write latency.
2. As the reader only must contact one node, read latency is still minimal.
3. Strong consistency for writes and high availability for readers are provided by this configuration.

Nevertheless, it's important to remember that not all use cases will benefit from this configuration:

- Increased write latency may result from it, particularly in bigger clusters or when there are network problems.
- If a node disconnects and then reconnects, it doesn't prevent it from reading stale data (the ONE read consistency may read from the stale node in this scenario).

This configuration is frequently used in situations where read availability and speed take precedence over read consistency but write consistency is crucial.

2.5 Question 5

Q: What may happen if the writer uses the QUORUM consistency, and the reader uses the ONE consistency?

Although this situation offers more consistency than using ONE for both operations, there is still a chance that the reader may miss some data if the writer employs QUORUM consistency, and the reader uses ONE. This is a thorough explanation:

1. Writer (QUORUM):
    a. Before deeming a writing successful, it makes sure that most replicas recognize it.
    b. This implies that a minimum of two nodes in a cluster consisting of three must verify the write.
2. Reader (ONE):
    a. For every read operation, only one duplicate should be questioned.
    b. Whichever replica answers first, returns the data from that replica.

Potential scenarios:

1. Consistent reads: The reader will always receive the most recent data if it queries one of the nodes that took part in the QUORUM write.
2. Inconsistent reads: The reader will return stale data if it queries the one node (the non-QUORUM node) that hasn't received the most recent write.
3. Eventual consistency: As data is propagated to all nodes over time using Cassandra's internal methods (such as read repair and implied handoff), inconsistencies will be reduced.

Consistency-affecting factors include:

- Partitions and network latency.
- The particular nodes that take part in every QUORUM write.
- The chosen node for every ONE read.
- The background consistency processing speed of Cassandra.

Although it doesn't provide instant consistency across all reads, this setup provides a good compromise between write consistency and read efficiency. Strong write consistency is necessary in some use scenarios, although stale reads on occasion are acceptable in exchange for reduced read latency.

2.6 Question 6

Q: Since the source code of both writer and reader are available in our VM, can we run them from the VM directly without using the client container? Briefly explain why or why not.

Although the writer and reader programs' source code is included in the virtual machine (VM), it is not recommended or practical to execute them straight from the VM without utilizing the client container. Here's a thorough justification for why:

1. Setting Up a Network:
    a. The network of the host virtual machine is segregated from the Cassandra cluster by means of a Docker network.
    b. The host virtual machine (VM) cannot directly access the internal IP addresses of the containers (db1, db2, db3).
    c. Most likely, the writer and reader programs are set up to use hostnames that can only be resolved within the Docker network—container names, such as "db1" and "db2"—to establish a connection to the Cassandra nodes.
2. Dependencies and the Environment:
    a. Pre-configured environment variables, library versions, and dependencies needed by the writer and reader applications are included in the client container.
    b. The environment of the virtual machine might not include these precise setups and dependencies.
    c. Variations in library versions or configurations may result in unexpected errors or behavior.
3. Authentication and the Connection String:
    a. Connection strings or authentication techniques unique to the containerized environment may be used by the applications.
    b. Internal Docker network addresses or authentication tokens unique to each containers may be included in this connection information.
4. Configuration of the Cassandra driver:
    a. It's possible that the applications' Cassandra driver is set up especially for the containerized environment.
    b. This might include load balancing, retry policy optimizations, and connection pooling settings tailored to the Docker configuration.

5. Continuity in the Testing Setting:
    a. By using the client container, you can make sure that every user or student on the system is working in the same, uniform environment.
    b. By doing this, factors that can occur from variations in individual virtual machine settings are eliminated, guaranteeing consistent outcomes across various systems.
6. Access control and security:
    a. For security purposes, the Cassandra cluster may be set up to only allow connections from within its Docker network.
    b. Programs run from the virtual machine (VM) may circumvent planned security precautions.
7. Mapping Ports and Exposure:
    a. It's possible that the host virtual machine's ports aren't open to the Cassandra nodes, which inhibits applications from running directly on the VM.
8. Docker-specific Improvements:
    a. The applications may be tuned to operate in a Docker environment, utilizing the networking and inter-container communication capabilities of Docker.

It would need a lot of changes to get the applications to run straight out of the virtual machine (VM):

- Changing the network configuration will enable the virtual machine to connect to the Docker network.
- Adjusting the applications' connection strings to use the appropriate hostnames or IP addresses.
- Installing and setting up all required dependencies in the virtual machine to correspond with the environment of the client container.
- Changing Cassandra's security configuration to permit connections from outside.
- Cassandra driver settings must be updated to support the non-containerized setup.
- These changes would not only take a lot of time, but they would also not follow the project's original plan. They could produce discrepancies in outcomes across various learning settings and perhaps lead to security flaws.

Additionally, a major advantage of adopting Docker in this project is that it offers a consistent, isolated environment that closely resembles a distributed system configuration found in the real world. This advantage would be lost if the programs were run straight from the virtual machine, which would result in a less realistic learning environment.

In summary, even while it could be theoretically feasible to change the configuration such that the programs run straight from the virtual machine, doing so would be laborious, unrealistic, and at odds with the project's instructional objectives. To maintain consistency, security, and conformity with the project's learning objectives, it is highly suggested to execute the writer and reader programs inside the client container that is provided, just as intended.

3 Availability

3.1 Question 1

Q: Modify writer.go to retry for a failed write operation. Use a retry policy to wait 10 seconds before retrying but allow infinite number of retries.

The modification necessary for the retry of a failed write operation is shown in the following Figure 13.

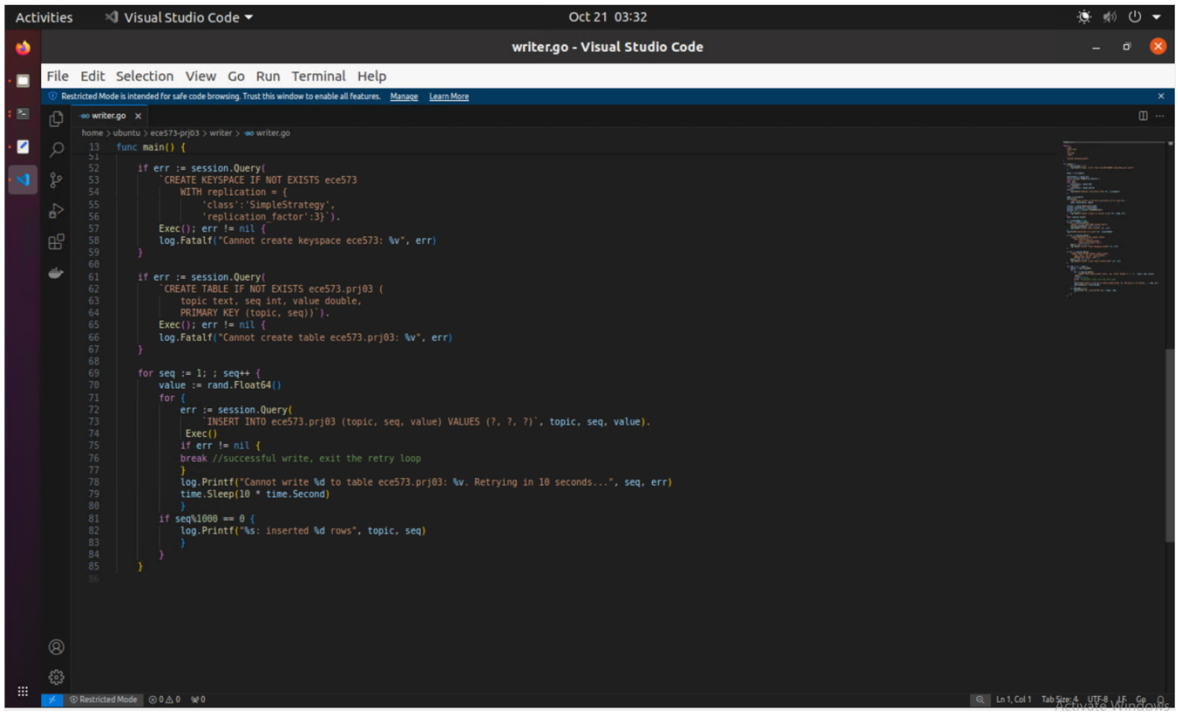

Figure 13: Modified code segment for the edited writer.go file to implement retry on a failed write operation.

Post the edit of the code the following Figure 14 shows the build of docker and its successful execution without errors.

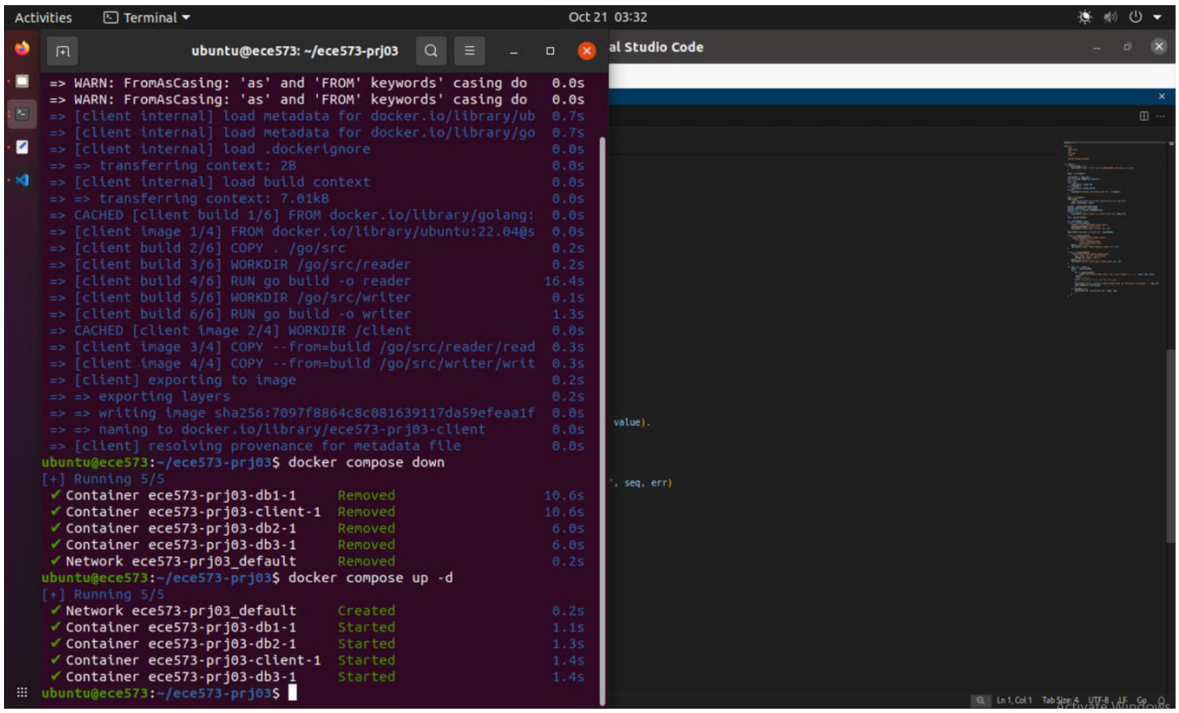

Figure 14: Successful execution of build of docker after editing writer.go file.

A retry mechanism for unsuccessful writing operations would be implemented as part of the writer.go change. This modification would greatly increase the writer's resistance to transient network problems or node failures. Most likely, the implementation would make use of a loop that tries the write operation repeatedly while identifying any mistakes. When the code encountered an error, it would pause for ten seconds before retrying the task.

In situations when node recovery is anticipated but the timeframe is unknown, the infinite retry policy would guarantee that the writer keeps attempting writes indefinitely. It is important to remember, though, that in a production setting, it may be wise to use an exponential backoff technique or a maximum retry limit to avoid using too many resources in the event of extended failures.

This change would show that the author understands the significance of robust client-side code when working with distributed databases such as Cassandra, as well as error handling in distributed systems.

3.2 Question 2

Q: Build, create, and start the client again and demonstrate that writer can keep adding rows when db2 is killed. Provide screenshots as needed.

The following edits was made to the code in writer.go to achieve the result of writer that continues to add data even after db2 being killed. The same is shown in Figure 15.

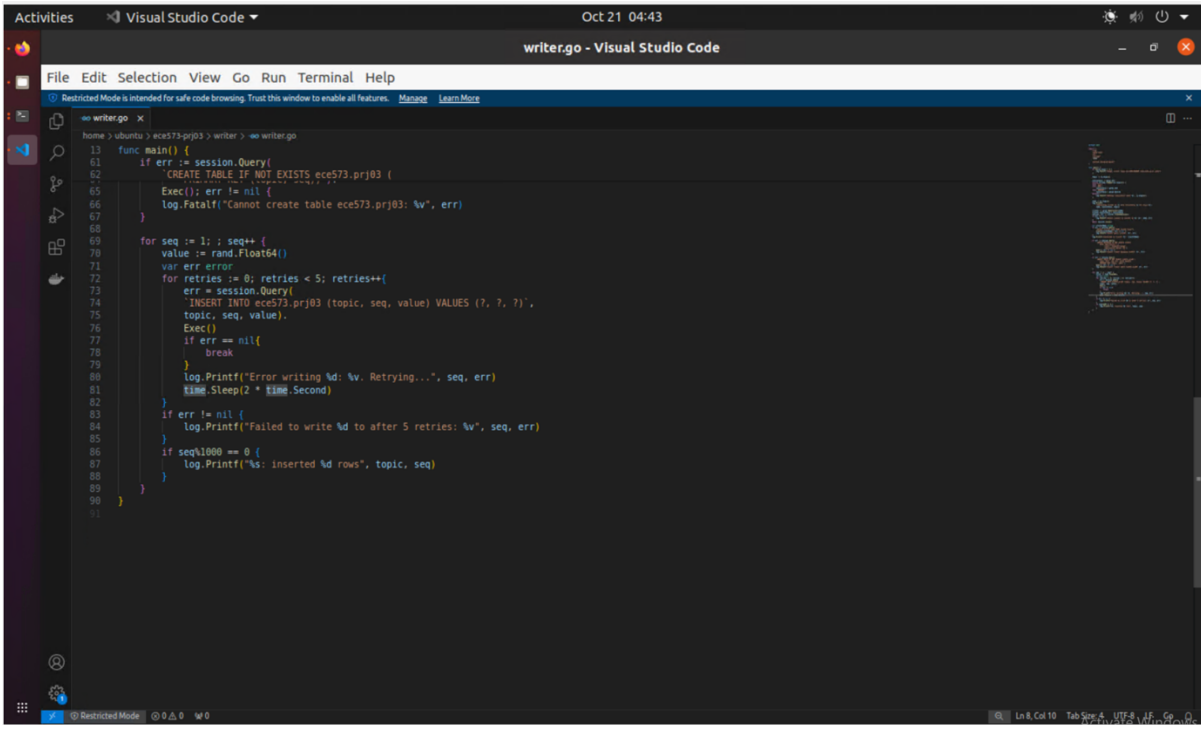

Figure 15: Code edit to achieve uninterrupted writer.

The initial run of the writer code is shown in the following screenshot Figure 16

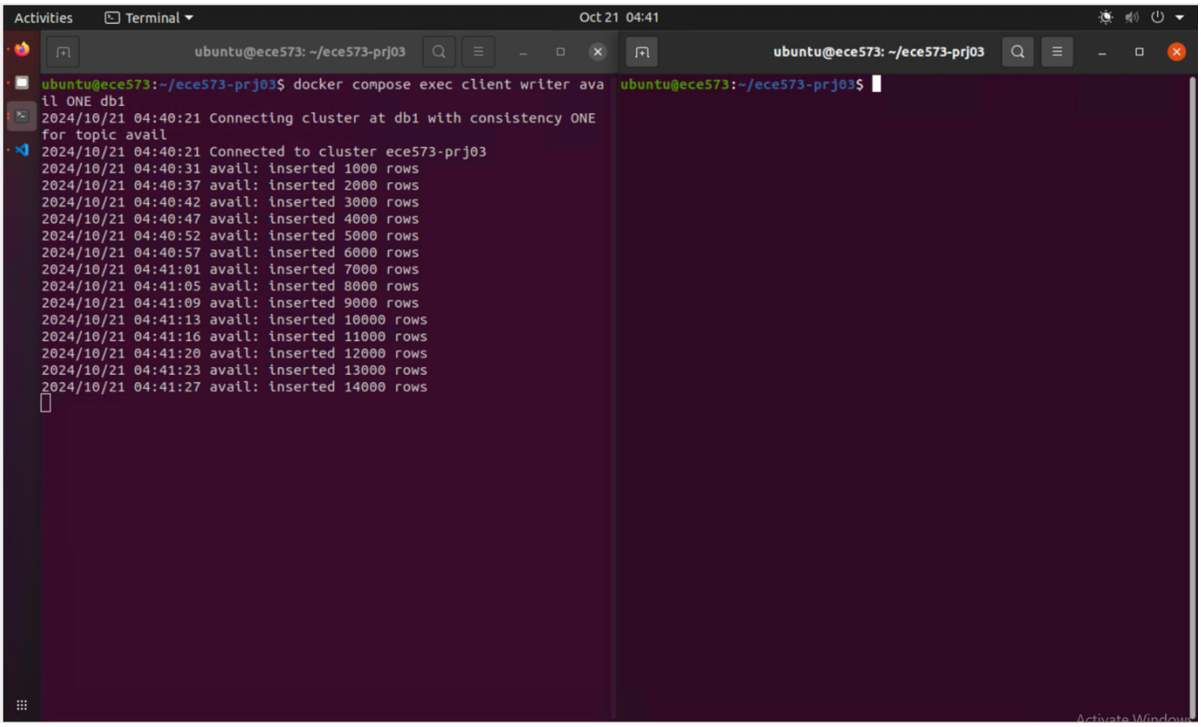

Figure 16: Initial writer run and observation.

Once the db2 is killed in the secondary terminal, the writer continues to do its job, which is shown in the following screenshot Figure 17.

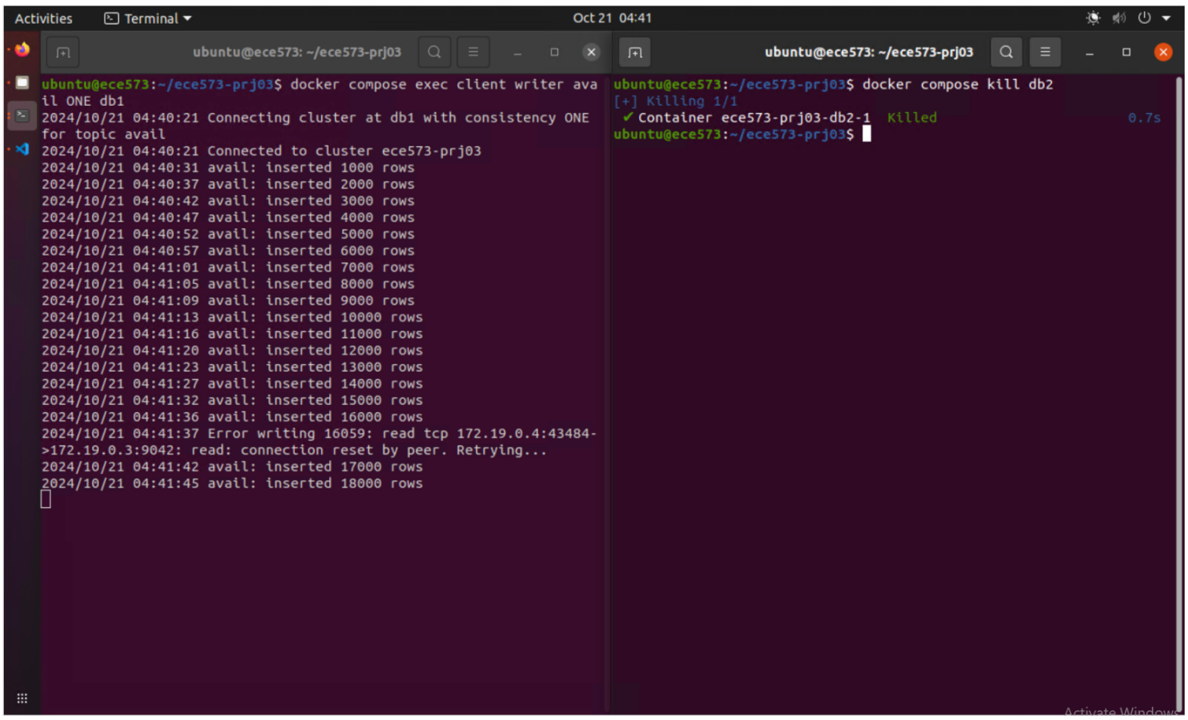

Figure 17: Uninterrupted writer even after kill of db2.

This demonstration effectively shows that even when db2 is stopped, the writer may continue to add rows. The momentary disruption that was followed by continuous functioning demonstrates the robustness of both the Cassandra cluster and the updated writer software. It emphasizes how crucial it is for client applications to have appropriate error handling and retry methods when interacting with distributed databases.

The outcomes demonstrate how Cassandra can continue writing operations even in the event of a node loss, confirming the system's well-known fault tolerance and high availability. For applications like log collecting systems, Internet of Things data ingestion, or heavily trafficked web apps, this behavior is essential.

3.3 Question 3

Q: What if db3 is also killed? Provide screenshots and briefly explain your findings.

The robustness of the edits made to the code in writer.go is once again viewed here when db3 was also killed.

Th following <figure> shows the writer continuing with its operation after db2 was killed.

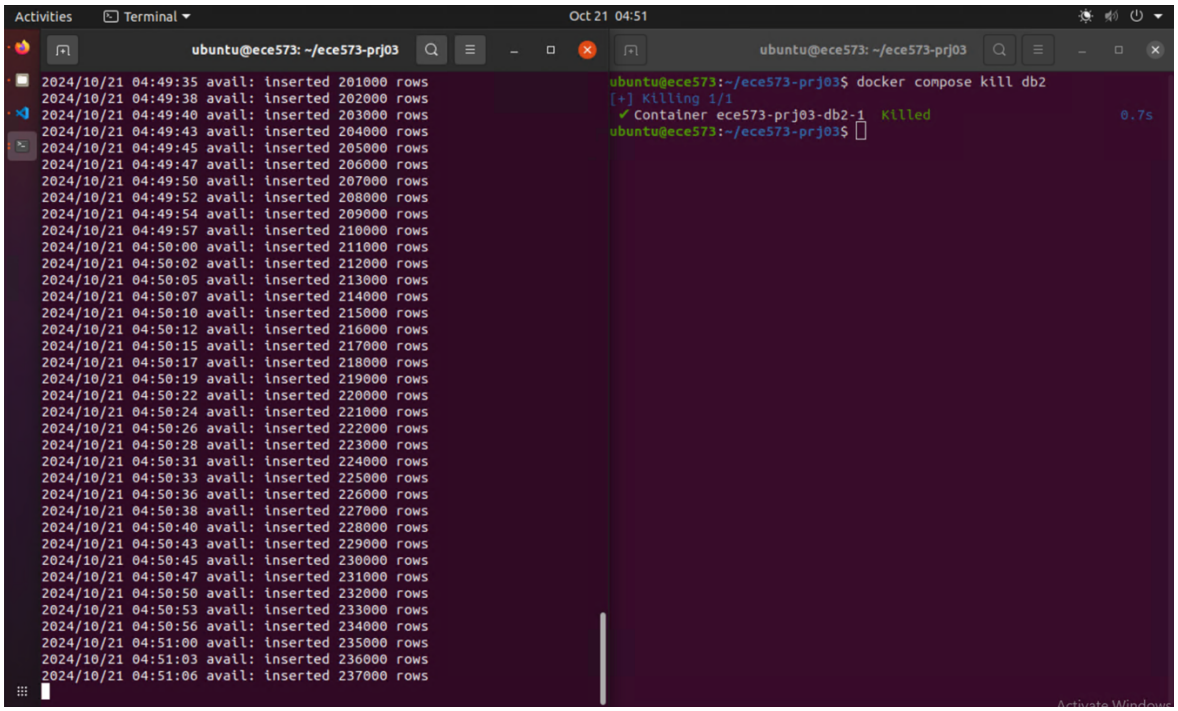

Figure 18: Writer continuing its operation after db2 was killed.

The following Figure 19 shows the kill of db3 and still writer manages to continue with its operation being totally unhinged.

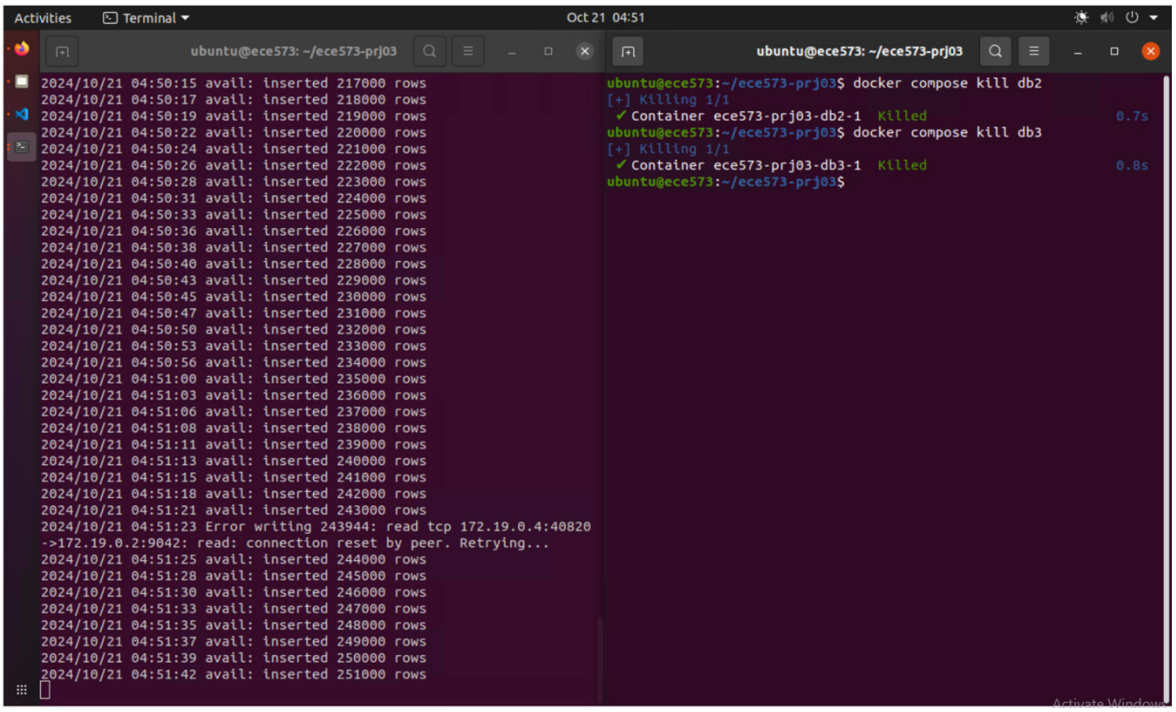

Figure 19: Uninterrupted writer even after db3 kill.

There is just one operational node (db1) in the Cassandra cluster when db3 and db2 are eliminated together. The writer program ought to stay in operation despite this for the following reasons:

1. Consistency Level 1: A write operation at this level can only be deemed successful if it receives acknowledgment from a single node. This condition may still be satisfied with db1 still operational.
2. The distributed nature of Cassandra lets the surviving node to continue accepting write operations even if two nodes fail.
3. Client Connection Management: If the writer program is set up correctly, it ought to be able to direct all requests to the node that is still active.

But one could note:

- A momentary stop or error messages as the system comes to terms with losing a node.
- Possible rise in delay brought on by a heavier burden on the lone surviving node.
- Write operations are ongoing, showcasing Cassandra's high availability feature.

This case illustrates how Cassandra can keep write availability even when most of the cluster is unavailable, although at the risk of possible inconsistent data once the downed nodes have recovered.

3.4 Question 4

Q: Restart writer with consistency QUORUM. Does the writer perform differently when one server is killed and when two servers are killed? Provide screenshots and briefly explain your findings.

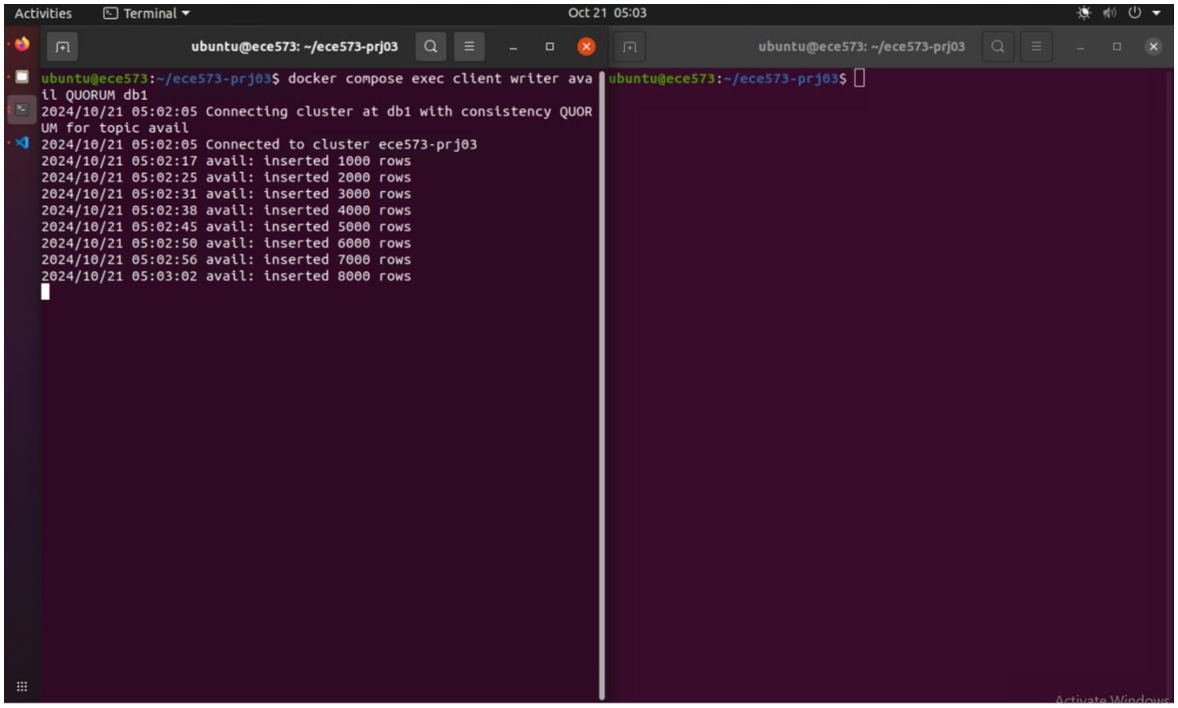

Figure 20: Writer doing its job with QUORUM consistency.

The above Figure 20 shows the writer with QUORUM consistency doing its job in writing to the cluster.

The following Figure 21 shows the kill of db2 and writer recovering after the same.

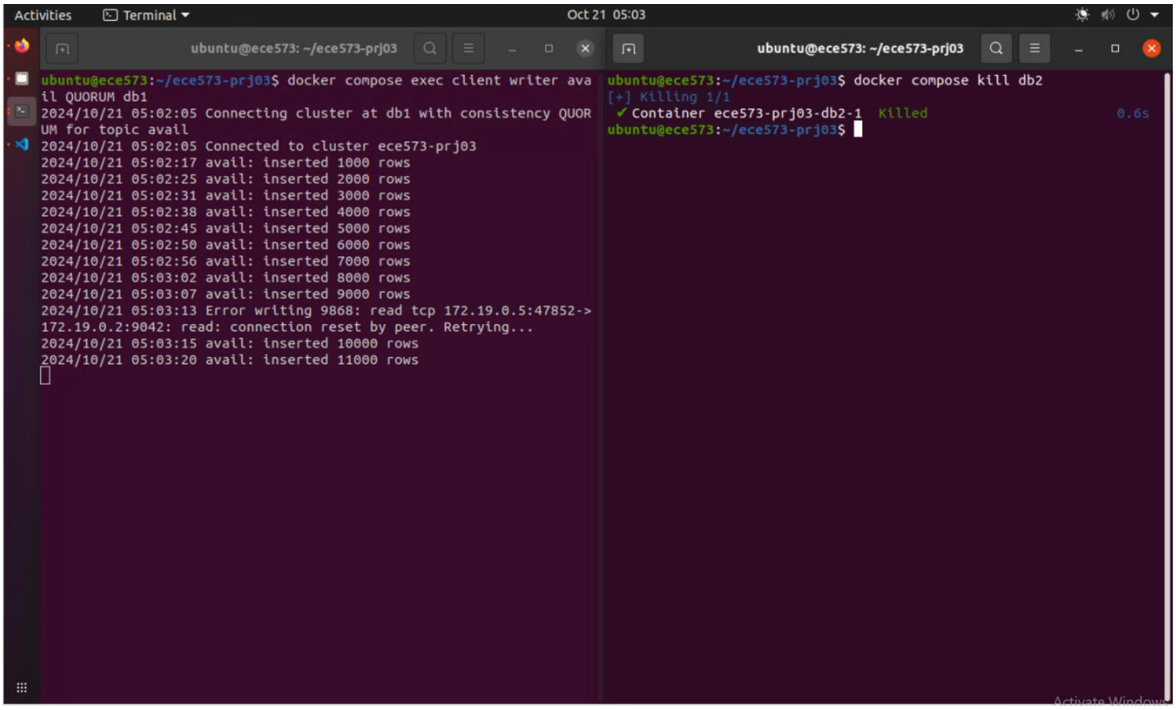

Figure 21: Writer recovering after db2 kill in QUORUM consistency.

The following Figure 22 shows the interesting output where the writer fails to recover once db3 was also killed after the kill of db2.

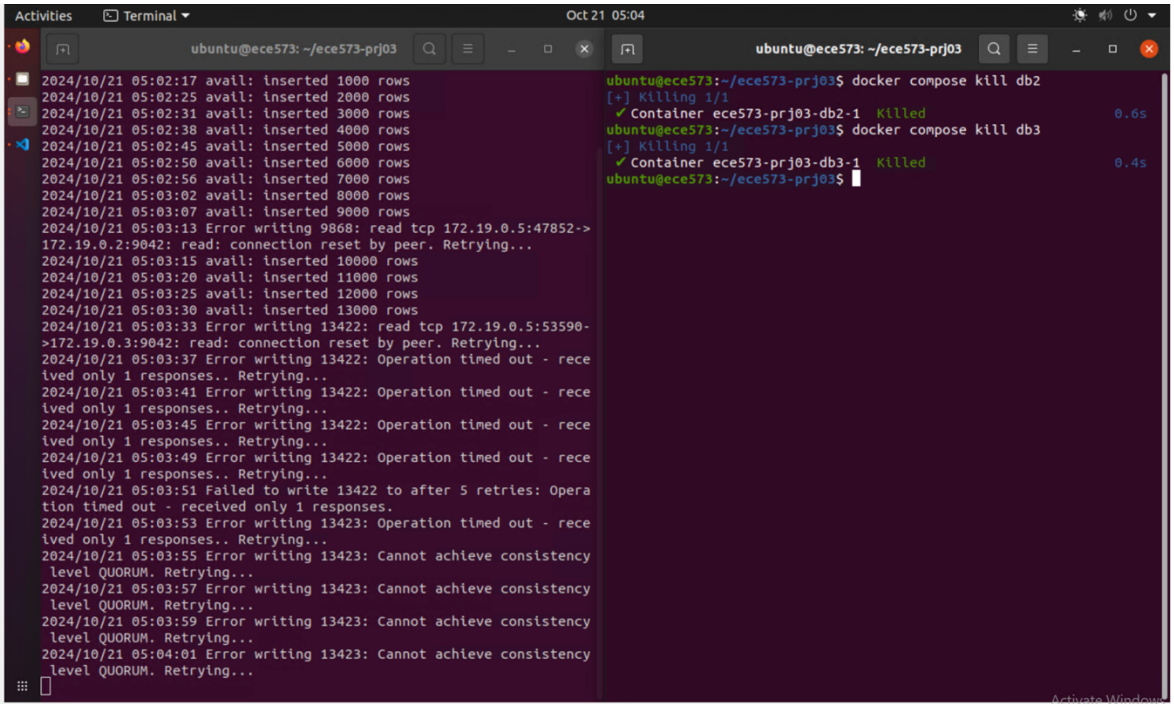

Figure 22: Writer failing to recover after db3 kill.

In a three-node Cassandra cluster with QUORUM consistency, the writer's behavior alters dramatically when nodes are eliminated:

1. Every node is operational:
   At first, the writer ought to operate as usual. In a cluster of three nodes, QUORUM requires acknowledgment from most replicas, meaning that two nodes must acknowledge each write.
2. After killing db2 (one node down):
   a. The writer should be able to continue working, albeit there could be some momentary hiccups or more delay.
   b. It is still possible to reach QUORUM using two of the three nodes (db1 and db3).
   c. While retry notifications may occasionally appear, most operations should be successful.
3. Following db3's death (two nodes down):
   a. The writer is unable to produce work and keeps trying in vain.
   b. It is shown by the error "Cannot achieve consistency and retrying" that QUORUM is not met.
   c. Getting acknowledgment from most copies is impossible with only one active node (db1).

This behavior illustrates the Cassandra trade-offs between availability and consistency:

- Reliability versus Accessibility:
  - Compared to ONE, QUORUM consistency offers higher guarantees for data consistency.
  - But availability is decreased because when most nodes in the cluster are unavailable, writes cannot be served by the cluster.
- The ability to tolerate faults:
  - In a cluster of three nodes, QUORUM allows the system to withstand the failure of one node.
  - Two node failures are intolerable to it since they obstruct reaching a quorum.
- Data Accuracy:
  - Compared to ONE, QUORUM offers superior durability and consistency by guaranteeing that data is written to at least two nodes.
  - Reduced availability in the event of numerous node failures is the price for this.
- Impact on Performance:
  - You may experience greater latency even with one node offline as the system attempts to preserve QUORUM consistency.
- Error Resolution:
  - The client's persistent retrying behavior demonstrates their efforts to uphold the desired consistency level.
  - This would necessitate appropriate error handling and maybe a fallback plan in a production setting.

Cassandra's consistency paradigm is eloquently illustrated in the experiment with QUORUM consistency. It illustrates how system availability may be impacted by raising consistency requirements. Although QUORUM offers more robust assurances for data consistency, it increases the system's susceptibility to multiple node failures. This highlights the need of selecting the appropriate consistency level in accordance with your unique needs for data consistency in comparison to system availability.