

# Table of Contents

<b>Solutions.....</b>	<b>1</b>
<b>1    kind Cluster Setup .....</b>	<b>1</b>
1.1 <i>Setup Completion .....</i>	<i>1</i>
1.2 <i>Questions.....</i>	<i>2</i>
1.2.1 <i>Question 1.....</i>	<i>2</i>
1.2.2 <i>Question 2.....</i>	<i>3</i>
<b>2    The Cassandra Service.....</b>	<b>4</b>
2.1 <i>Setup Completion .....</i>	<i>4</i>
2.2 <i>Questions.....</i>	<i>5</i>
2.2.1 <i>Question 1.....</i>	<i>5</i>
2.2.2 <i>Question 2.....</i>	<i>6</i>
2.2.3 <i>Question 3.....</i>	<i>7</i>
2.2.4 <i>Question 4.....</i>	<i>8</i>
<b>3    Build and Deploy and Application.....</b>	<b>9</b>
3.1 <i>Setup Completion .....</i>	<i>9</i>
3.2 <i>Questions.....</i>	<i>10</i>
3.2.1 <i>Question 1.....</i>	<i>10</i>
3.2.2 <i>Question 2.....</i>	<i>11</i>
3.2.3 <i>Question 3.....</i>	<i>13</i>
<b>4    Stateless Application .....</b>	<b>14</b>
4.1 <i>Questions.....</i>	<i>14</i>
4.1.1 <i>Question 1.....</i>	<i>14</i>
4.1.2 <i>Question 2.....</i>	<i>16</i>

## List of Figures

Figure 1: Creation of the kind cluster with 4 worker nodes and 1 control node.	1
Figure 2: Verification of the creation of the nodes.	1
Figure 3: Indicating the nodes as docker containers.	1
Figure 4: Running of K8 containers inside.	2
Figure 5: Cluster deletion.	2
Figure 6: Creation of 6 worker nodes.	2
Figure 7: The creation of the cluster and verification of the number of nodes.	3
Figure 8: Running crictl ps to show the K8 containers running inside.	3
Figure 9: Reset of the cluster is done.	4
Figure 10: Application of changes to the configuration file.	4
Figure 11: Verification of the status of the objects.	4
Figure 12: Executing commands to check Cassandra containers inside docker container nodes.	5
Figure 13: Killing the Cassandra nodes and its Pods.	5

Figure 14: Code snippet showing the edit of number of replicas.	5
Figure 15: Verification of change of Cassandra replicas.	6
Figure 16: Code snippet of the cassandra.yml file showing the definition of the resources.	6
Figure 17: Code snippet of the env section.	7
Figure 18: Code snippet from docker-compose.yml from Project 3.	7
Figure 19: Cassandra Service Definition.	8
Figure 20: Cassandra statefulset definition.	8
Figure 21: Running build.sh file.	9
Figure 22: Application of the changes made to the writer.yml file and checking status.	9
Figure 23: Successful write of cassandra.	11
Figure 24: Docker build and replacing writer with new.	14
Figure 25: Current running log of row insertion being done.	16
Figure 26: Stopping pod and restarting from the previous top point.	16

## List of Code Snippets

Code 1: Edited writer.yml file.	10
Code 2: Edited writer.go file.	11
Code 3: Snippet from the cassandra.yml file.	12
Code 4: Code snippet from writer.yml focusing on env.	13
Code 5: Edited and updated code from writer.go.	15

## Solutions

### 1 kind Cluster Setup

#### 1.1 Setup Completion

The following set of screenshots shows the successful setup of the kind cluster.

```
ubuntu@ece573:~/ece573-prj04$ kind create cluster --config cluster.yml
Creating cluster "kind" ...
  ✓ Ensuring node image (kindest/node:v1.30.0) 🚢
  ✓ Preparing nodes 📦 📦 📦 📦
  ✓ Writing configuration 📄
  ✓ Starting control-plane 🚦
  ✓ Installing CNI 🛠️
  ✓ Installing StorageClass 💾
  ✓ Joining worker nodes 🤖
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a nice day! 🙌
ubuntu@ece573:~/ece573-prj04$
```

Figure 1: Creation of the kind cluster with 4 worker nodes and 1 control node.

The successful setup of kind cluster with 4 worker nodes and 1 control node is elucidated in the above Figure 1.

```
ubuntu@ece573:~/ece573-prj04$ kind get nodes
kind-worker3
kind-worker2
kind-worker
kind-worker4
kind-control-plane
ubuntu@ece573:~/ece573-prj04$
```

Figure 2: Verification of the creation of the nodes.

The setup of the nodes is shown in Figure 2 where it shows the active status of 4 worker nodes and 1 control node.

```
ubuntu@ece573:~/ece573-prj04$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
568f3c7a85fc   kindest/node:v1.30.0   "/usr/local/bin/entr..." 4 minutes ago   Up 4 minutes   127.0.0.1:35415->6443/tcp   kind-worker3
4e2baaf27d7e   kindest/node:v1.30.0   "/usr/local/bin/entr..." 4 minutes ago   Up 4 minutes   127.0.0.1:35415->6443/tcp   kind-worker2
5ad142acd364   kindest/node:v1.30.0   "/usr/local/bin/entr..." 4 minutes ago   Up 4 minutes   127.0.0.1:35415->6443/tcp   kind-worker
5f33b22395ce   kindest/node:v1.30.0   "/usr/local/bin/entr..." 4 minutes ago   Up 4 minutes   127.0.0.1:35415->6443/tcp   kind-worker4
977b6464bf44   kindest/node:v1.30.0   "/usr/local/bin/entr..." 4 minutes ago   Up 4 minutes   127.0.0.1:35415->6443/tcp   kind-control-plane
ubuntu@ece573:~/ece573-prj04$
```

Figure 3: Indicating the nodes as docker containers.

In Figure 3, it is understood that the 4 nodes created are indeed docker containers. This is validated by running the `docker ps` command.

```
ubuntu@ece573:~/ece573-prj04$ docker exec -it kind-worker crictl ps
CONTAINER          IMAGE                                     CREATED          STATE          NAME          ATTEMPT          POD_ID          POD
2373af161e2f4      4950bb10b3f87                          6 minutes ago    Running        kindnet-cni    0                decbc58cbecb   kindnet-x8zfq
ff1c1ee18aec4      c4e12eee82f28                          6 minutes ago    Running        kube-proxy    0                a100a271ccab1  kube-proxy-mjdx8
ubuntu@ece573:~/ece573-prj04$
```

Figure 4: Running of K8 containers inside.

Figure 4 shows the running of K8 containers inside of the worker node using the `crictl ps` command.

```
ubuntu@ece573:~/ece573-prj04$ kind delete cluster
Deleting cluster "kind" ...
Deleted nodes: ["kind-worker3" "kind-worker2" "kind-worker" "kind-worker4" "kind-control-plane"]
ubuntu@ece573:~/ece573-prj04$
```

Figure 5: Cluster deletion.

The kind cluster is deleted using the command as shown in Figure 5.

## 1.2 Questions

### 1.2.1 Question 1

Q: Modify cluster.yml to include 6 worker nodes. Create the cluster and verify that it is running.

The modification done to the cluster.yml file is shown in the following Figure 6.

```
! cluster.yml
1  kind: Cluster
2  apiVersion: kind.x-k8s.io/v1alpha4
3  nodes:
4    - role: control-plane
5    - role: worker
6    - role: worker
7    - role: worker
8    - role: worker
9    - role: worker
10   - role: worker
```

Figure 6: Creation of 6 worker nodes.

```

ubuntu@ece573:~/ece573-prj04$ kind create cluster --config cluster.yml
Creating cluster "kind" ...
  ✓ Ensuring node image (kindest/node:v1.30.0)
  ✓ Preparing nodes
  ✓ Writing configuration
  ✓ Starting control-plane
  ✓ Installing CNI
  ✓ Installing StorageClass
  ✓ Joining worker nodes
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Not sure what to do next? Check out https://kind.sigs.k8s.io/docs/user/quick-start/
ubuntu@ece573:~/ece573-prj04$ kind get nodes
kind-worker5
kind-worker3
kind-worker2
kind-worker4
kind-worker
kind-worker6
kind-control-plane
ubuntu@ece573:~/ece573-prj04$ docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED        STATUS        PORTS                               NAMES
74d90afcfeac   kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes                               kind-worker5
2d32e2622f70   kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes                               kind-worker3
f2da47a4d92    kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes                               kind-worker2
fcfd934e5b32   kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes                               kind-worker4
297a2ebf9f5f   kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes                               kind-worker
95285634f563   kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes                               kind-worker6
66742ffd86c0   kindest/node:v1.30.0   "/usr/local/bin/entr..." 2 minutes ago   Up 2 minutes   127.0.0.1:45305->6443/tcp   kind-control-plane
ubuntu@ece573:~/ece573-prj04$

```

Figure 7: The creation of the cluster and verification of the number of nodes.

Figure 7 shows the creation of 6 nodes as per the edits made to the cluster.yml file and it is verified using the `kind get nodes` command. Additionally, it is also validated that they are running as docker containers.

## 1.2.2 Question 2

**Q:** Run `crictl ps` in the control plane node to show K8s containers running inside. Name two K8s control plane components from the list.

The following Figure 8 shows the run of `crictl ps` command where the output shows the K8 containers running inside.

```

ubuntu@ece573:~/ece573-prj04$ docker exec -it kind-control-plane bash
root@kind-control-plane:/# crictl ps
CONTAINER ID   IMAGE               CREATED        STATE      NAME                ATTEMPT     POD ID          POD
4cfff70d9334   cbb01a7bd410d      4 minutes ago   Running    coredns             0           1d552b0100967   coredns-7db6d8ff4d-jpccq
90b911ce928d    cbb01a7bd410d      4 minutes ago   Running    coredns             0           66a0072326c9e   coredns-7db6d8ff4d-6t46
5bf29ad3f67af   0500518ebaa68      4 minutes ago   Running    local-path-provisioner 0           21d8e07e59fc6   local-path-provisioner-988d74bc-m6xq9
f902d85649df7   4950bb10b3f87      4 minutes ago   Running    kindnet-cni         0           118e5a463cbbf   kindnet-mlm2
378d91ea021ad   c4e12ee82f28       4 minutes ago   Running    kube-proxy          0           542240b731380   kube-proxy-zbgm4
d930250ba0e1e   3061cfd7c04c       5 minutes ago   Running    etcd                0           3333c00001e5e   etcd-kind-control-plane
18fb279db8bb9   7f6c51674d5ef      5 minutes ago   Running    kube-apiserver      0           fc4701f7308a0   kube-apiserver-kind-control-plane
0fe85984032c3   6abc94235f022      5 minutes ago   Running    kube-controller-manager 0           a693101be76c4   kube-controller-manager-kind-control-plane
d4fc665d68cd7   6c97f001b028e      5 minutes ago   Running    kube-scheduler      0           abc5a4a0f6c71   kube-scheduler-kind-control-plane
root@kind-control-plane:/#

```

Figure 8: Running `crictl ps` to show the K8 containers running inside.

Among the listed containers the two K8 control plane components running are:

### 1. kube-apiserver-kind-control-plane:

This is the Kubernetes control plane's front end. It manages both internal and external queries to the cluster and makes the Kubernetes API available.

### 2. kube-scheduler-kind-control-plane:

This part oversees arranging the pods to operate on available nodes in accordance with resource needs and other limitations.

## 2 The Cassandra Service

### 2.1 Setup Completion

The following set of screenshots show the follow of the procedure to perform the default tasks.

```
ubuntu@ece573:~/ece573-prj04$ ./reset_cluster.sh
4.1.3: Pulling from library/cassandra
Digest: sha256:7cbcec0086ac3d448537f1bdd6762f7a69fe8f570f00d45f4f719f6c2cdd4d5c
Status: Image is up to date for cassandra:4.1.3
docker.io/library/cassandra:4.1.3
Deleting cluster "kind" ...
Deleted nodes: ["kind-worker5" "kind-worker3" "kind-worker2" "kind-worker4" "kind-worker" "kind-worker6" "kind-control-plane"]
Creating cluster "kind" ...
  ✓ Ensuring node image (kindest/node:v1.30.0)
  ✓ Preparing nodes
  ✓ Writing configuration
  ✓ Starting control-plane
  ✓ Installing CNI
  ✓ Installing StorageClass
  ✓ Joining worker nodes
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a nice day!
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-worker5", loading...
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-worker", loading...
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-worker4", loading...
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-worker6", loading...
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-worker3", loading...
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-worker2", loading...
Image: "cassandra:4.1.3" with ID "sha256:c5a661a601b9297c727a7e4f0f330eef30ba06af1b8be0b8ca48f64eb6416e19" not yet present on node "kind-control-plane", loading...
```

Figure 9: Reset of the cluster is done.

Figure 9 represents the reset of the cluster where existing cluster is deleted, and new nodes are created. This new change is to be applied to the configuration file which is done and shown in the following Figure 10

```
ubuntu@ece573:~/ece573-prj04$ kubectl apply -f cassandra.yml
service/cassandra-service created
statefulset.apps/cassandra created
ubuntu@ece573:~/ece573-prj04$
```

Figure 10: Application of changes to the configuration file.

```
ubuntu@ece573:~/ece573-prj04$ kubectl get services
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
cassandra-service   ClusterIP   None         <none>        9042/TCP   27s
kubernetes          ClusterIP   10.96.0.1    <none>        443/TCP    3m10s

ubuntu@ece573:~/ece573-prj04$ kubectl get statefulsets
NAME        READY   AGE
cassandra  3/3     47s

ubuntu@ece573:~/ece573-prj04$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
cassandra-0   1/1     Running   0           61s
cassandra-1   1/1     Running   0           43s
cassandra-2   1/1     Running   0           34s
```

Figure 11: Verification of the status of the objects.

Figure 11 shows the verification of the created objects done using 3 different commands: `kubectl get services`, `kubectl get statefulsets` and, `kubectl get pods`.

```
ubuntu@ece573:~/ece573-prj04$ kubectl exec cassandra-0 -- nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
-- Address      Load          Tokens  Owns (effective)  Host ID                               Rack
UN  10.244.5.3    ?             16      100.0%            92ec5ae4-5b5c-4e9c-a333-d18b477f2ad6  rack1
UN  10.244.3.3    109.39 KiB    16      100.0%            9ee86c17-bec5-480a-8177-63285f8db179  rack1

ubuntu@ece573:~/ece573-prj04$ kubectl exec -it cassandra-0 -- cqlsh
Connected to ece573-prj04 at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.3 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> exit
ubuntu@ece573:~/ece573-prj04$
```

Figure 12: Executing commands to check Cassandra containers inside docker container nodes.

Figure 12 shows the successful running of Cassandra containers within Docker container nodes using commands that run from the Cassandra containers.

```
ubuntu@ece573:~/ece573-prj04$ kubectl delete -f cassandra.yml
service "cassandra-service" deleted
statefulset.apps "cassandra" deleted
ubuntu@ece573:~/ece573-prj04$
```

Figure 13: Killing the Cassandra nodes and its Pods.

In Figure 13, we remove the Cassandra service and its Pods that follows.

## 2.2 Questions

### 2.2.1 Question 1

**Q:** Modify `cassandra.yml` to include 5 Cassandra replicas. Create the service and verify that Cassandra is running properly.

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
spec:
  serviceName: cassandra-service
  replicas: 5 #Changed to 5 from 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
```

Figure 14: Code snippet showing the edit of number of replicas.

The modification of the cassandra.yml file is shown in the above Figure 14. Hereafter, the changes made is applied to the configuration file like as performed in Figure 10.

```
ubuntu@ece573:~/ece573-prj04$ kubectl get statefulsets
NAME          READY   AGE
cassandra     2/5     32s
ubuntu@ece573:~/ece573-prj04$
ubuntu@ece573:~/ece573-prj04$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
cassandra-0   1/1     Running   0           46s
cassandra-1   1/1     Running   0           30s
cassandra-2   1/1     Running   0           21s
cassandra-3   1/1     Running   0           11s
cassandra-4   0/1     Pending   0           4s
ubuntu@ece573:~/ece573-prj04$
```

Figure 15: Verification of change of Cassandra replicas.

Thus, the changes made to the number of replicas being successful is shown in the above Figure 15.

### 2.2.2 Question 2

**Q:** Explain the "resources" section for the Cassandra container from cassandra.yml. What are the difference between "limits" and "requests"?

The Cassandra container's CPU and RAM allocation in Kubernetes is specified in the resources section of the cassandra.yml file. It guarantees that every Cassandra pod has the resources required for optimal operation.

```
spec:
  containers:
  - name: cassandra
    image: cassandra:4.1.3
    ports:
    - containerPort: 9042
    resources:
      limits:
        cpu: "500m"
        memory: 1Gi
      requests:
        cpu: "500m"
        memory: 1Gi
```

Figure 16: Code snippet of the cassandra.yml file showing the definition of the resources.



From Figure 16, The request specifies the bare minimum of resources that Kubernetes ensures the container will have. The scheduler will use these resources to choose which node to put the pod on. The container's maximum resource consumption is specified by the limit. Kubernetes may throttle (for CPU) or terminate (for memory) the container if it attempts to go beyond these limitations. In this case (from the image), the quantitative allocation for the limit and request is the same.

The key difference between Requests and Limit is that for requests the minimal resources that the container is guaranteed to get are called requests. This is used by Kubernetes to schedule the pod. On the other hand, Limits are the most resources that can be used by the container. It may be throttled or stopped if it goes above certain limits.

### 2.2.3 Question 3

**Q:** Below the "resources" section you will find the section "env" to setup environment variables. Where was the corresponding part for Project 3? Explain the difference between the two.

Cassandra's cluster name, data center, and rack configurations are configured using the environment variables. The following Figure 17 represents the code snippet of the same from the cassandra.yml file.

```
env:
  - name: MAX_HEAP_SIZE
    value: 512M
  - name: HEAP_NEWSIZE
    value: 128M
  - name: CASSANDRA_SEEDS
    value: "cassandra-0.cassandra-service.default.svc.cluster.local"
  - name: CASSANDRA_CLUSTER_NAME
    value: "ece573-prj04"
```

Figure 17: Code snippet of the env section.

The environment variables in Project 3, where we used Docker Compose to install Cassandra, would have been specified in the docker-compose.yml file's environment section.

```
environment:
  - HEAP_NEWSIZE=128M
  - MAX_HEAP_SIZE=512M
  - CASSANDRA_CLUSTER_NAME=ece573-prj03
  - CASSANDRA_SEEDS=db1,db2,db3
```

Figure 18: Code snippet from docker-compose.yml from Project 3.

The difference between them and their varying declaration is because, In Project 4, we are installing Cassandra on Kubernetes, while in Project 3, we defined environment variables for containers using Docker Compose. Like Docker Compose's environment part, Kubernetes' "env" section is used to create environment variables. The primary distinction is the orchestration tool used: Kubernetes for this project and Docker Compose for Project 3.

#### 2.2.4 Question 4

**Q:** How does the Cassandra service know which Pods are part of the service?

Using labels and a label selector, a service in Kubernetes chooses which pods to direct traffic to. Both the Cassandra service and the statefulset utilize labels in this cassandra.yml file to determine which pods are part of the service.

```
apiVersion: v1
kind: Service
metadata:
  name: cassandra-service
spec:
  clusterIP: None
  ports:
  - port: 9042
  selector:
    app: cassandra
```

Figure 19: Cassandra Service Definition.

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
spec:
  serviceName: cassandra-service
  replicas: 5 #Changed to 5 from 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
```

Figure 20: Cassandra statefulset definition.

Figure 19 and Figure 20 shows the Cassandra Service and statefulset definitions in the file cassandra.yml file.

The label app, Cassandra, is used by the Service's selector to find pods. Also, Cassandra is used by the StatefulSet to generate pods. This label matching is how Kubernetes links the pods to the service. Any pod with the label app: cassandra will be seen as a component of the Cassandra service, and traffic will be sent to it by the service. This is how Cassandra knows which pods are of what service.

## 3 Build and Deploy and Application

### 3.1 Setup Completion

The following set of screenshots show the follow of the procedure to perform the default tasks.

```
ubuntu@ece573:~/ece573-prj04$ ./build.sh
[+] Building 26.7s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 483B
=> WARN: FromAsCasing: 'as' and 'FROM' keywords casing do not match (line 3)
=> WARN: FromAsCasing: 'as' and 'FROM' keywords casing do not match (line 13)
=> [internal] load metadata for docker.io/library/golang:1.21
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 60.31kB
=> CACHED [build 1/4] FROM docker.io/library/golang:1.21@sha256:4746d26432a9117a5f58e95cb9f954ddfd0e128e9d581688651499316e4a2fb
=> [build 2/4] COPY . /go/src
=> [build 3/4] WORKDIR /go/src/writer
=> [build 4/4] RUN CGO_ENABLED=0 GOOS=linux go build -o writer
=> [image 1/1] COPY --from=build /go/src/writer/writer .
=> exporting to image
=> => exporting layers
=> => writing image sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4
=> naming to docker.io/library/ece573-prj04-writer:v1
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-control-plane", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-worker4", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-worker2", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-worker5", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-worker", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-worker3", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:3e595e7b4c098cdc201ef4f61923efb5737d82ad38af2bb099b865ea26fd4" not yet present on node "kind-worker6", loading...
ubuntu@ece573:~/ece573-prj04$
```

Figure 21: Running build.sh file.

As stated in the project 4 webpage, the build.sh file is convenient in performing the two steps: building a building a docker image and making it available for the K8 cluster. This action is shown in Figure 21.

```
ubuntu@ece573:~/ece573-prj04$ kubectl apply -f writer.yml
deployment.apps/ece573-prj04-writer created
ubuntu@ece573:~/ece573-prj04$ kubectl get deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
ece573-prj04-writer 0/1     1             0           16s
ubuntu@ece573:~/ece573-prj04$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
cassandra-0                         1/1     Running   0           37m
cassandra-1                         1/1     Running   1 (35m ago) 37m
cassandra-2                         1/1     Running   0           37m
cassandra-3                         1/1     Running   1 (35m ago) 36m
cassandra-4                         1/1     Running   1 (35m ago) 36m
ece573-prj04-writer-d64665785-9t7lw 0/1     Error     2 (21s ago) 24s
ubuntu@ece573:~/ece573-prj04$
```

Figure 22: Application of the changes made to the writer.yml file and checking status.

In Figure 22, the changes made is reflected and as expected there is an error that's spotted with the writer application.

## 3.2 Questions

### 3.2.1 Question 1

**Q:** Correct writer.yml as mentioned above and verify everything is running properly.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ece573-prj04-writer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ece573-prj04-writer
  template:
    metadata:
      labels:
        app: ece573-prj04-writer
    spec:
      containers:
        - name: writer
          image: ece573-prj04-writer:v1
          env:
            - name: TOPIC
              value: "project04"
            - name: CASSANDRA_SEEDS
              value: "cassandra-service.default.svc.cluster.local"
            - name: CASSANDRA_KEYSPACE
              value: "ece573"
            - name: CONSISTENCY
              value: "ONE"
```

Code 1: Edited writer.yml file.

Initially, the writer didn't perform as expected and so the writer.yml and writer.go is edited as shown in **Error! Reference source not found.** and **Error! Reference source not found.**.

From the changes made the same is applied to the writer file as shown in Figure 22 and after the corresponding operation. The next successful output is obtained in Figure 23.

```

package main

import (
    "log"
    "math/rand"
    "os"
    "strings"
)

const githubComGocqlGocql = "github.com/gocql/gocql"

func main() {
    // Check the topic environment variable
    topic := os.Getenv("TOPIC")
    if topic == "" {
        log.Fatalf("Unknown topic: TOPIC environment variable is not set")
    }
    log.Printf("Using topic: %s", topic)

    // Check the consistency level environment variable
    cs := os.Getenv("CONSISTENCY")
    var consistency gocql.Consistency
    switch strings.ToUpper(cs) {
    case "ALL":
        consistency = gocql.All
    case "ONE":
        consistency = gocql.One
    case "QUORUM":
        consistency = gocql.Quorum
    default:
        log.Fatalf("Unknown consistency level %s", cs)
    }

    // Get the Cassandra seed node from the environment variable
    seed := os.Getenv("CASSANDRA_SEEDS")
    log.Printf("Connecting cluster at %s with consistency %s for topic %s", seed, consistency, topic)

    // Create a new Cassandra session
    cluster := gocql.NewCluster(seed)
    cluster.Consistency = consistency
    session, err := cluster.CreateSession()
    if err != nil {
        log.Fatalf("Cannot connect to cluster at %s: %v", seed, err)
    }
    defer session.Close()

    // Query the Cassandra cluster name to verify the connection
    var clusterName string
    if err := session.Query("SELECT cluster_name FROM system.local").Scan(&clusterName); err != nil {
        log.Fatalf("Cannot query cluster: %v", err)
    }
    log.Printf("Connected to cluster %s", clusterName)

    // Create the keyspace and tables if they don't already exist
    if err := session.Query(
        "CREATE KEYSPACE IF NOT EXISTS ece573\nWITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3}").Exec(); err != nil {
        log.Fatalf("Cannot create keyspace ece573: %v", err)
    }

    if err := session.Query(
        "CREATE TABLE IF NOT EXISTS ece573.prj04 (\n    topic text, seq int, value double,\n    PRIMARY KEY (topic, seq))").Exec(); err != nil {
        log.Fatalf("Cannot create table ece573.prj04: %v", err)
    }

    if err := session.Query(
        "CREATE TABLE IF NOT EXISTS ece573.prj04_last_seq (\n    topic text, seq int,\n    PRIMARY KEY (topic))").Exec(); err != nil {
        log.Fatalf("Cannot create table ece573.prj04_last_seq: %v", err)
    }

    log.Printf("Tables ece573.prj04 and ece573.prj04_last_seq ready.")

    // Modify code below to read lastSeq from ece573.prj04_last_seq
    lastSeq := 0

    log.Printf("%s: start from lastSeq=%d", topic, lastSeq)
    for seq := lastSeq + 1; seq++ {
        value := rand.Float64()

        // Insert a new record into the prj04 table
        err := session.Query(
            "INSERT INTO ece573.prj04 (topic, seq, value) VALUES (?, ?, ?)",
            topic, seq, value).Exec()
        if err != nil {
            log.Fatalf("Cannot write %d to table ece573.prj04: %v", seq, err)
        }

        // Update the last sequence number for the topic
        err = session.Query(
            "INSERT INTO ece573.prj04_last_seq (topic, seq) VALUES (?, ?)",
            topic, seq).Exec()
        if err != nil {
            log.Fatalf("Cannot write %d to table ece573.prj04_last_seq: %v", seq, err)
        }

        // Log progress every 1000 records
        if seq%1000 == 0 {
            log.Printf("%s: inserted rows to seq %d", topic, seq)
        }
    }
}

```

Code 2: Edited writer.go file.

```

ubuntu@ece573:~/ece573-prj04$ kubectl logs -l app=ece573-prj04-writer -f
2024/11/04 05:30:51 Connected to cluster ece573-prj04
2024/11/04 05:31:52 gocql: cluster schema versions not consistent: [54e17321-3f2e-37ca-9b08-d91ba7bdd369 e1c5cf32-8845-32ab-a7de-14009e177c7f]
2024/11/04 05:32:53 gocql: cluster schema versions not consistent: [c8d5fa2f-cbe6-3339-9f0e-32c6de67aec4 54e17321-3f2e-37ca-9b08-d91ba7bdd369]
2024/11/04 05:33:53 gocql: cluster schema versions not consistent: [54e17321-3f2e-37ca-9b08-d91ba7bdd369 02965675-1d88-32d6-ac7c-2cabdcf428f6]
2024/11/04 05:33:53 Tables ece573.prj04 and ece573.prj04_last_seq ready.
2024/11/04 05:33:53 project04: start from lastSeq=0
2024/11/04 05:34:16 project04: inserted rows to seq 1000
2024/11/04 05:34:43 project04: inserted rows to seq 2000
2024/11/04 05:35:00 project04: inserted rows to seq 3000
2024/11/04 05:35:16 project04: inserted rows to seq 4000
2024/11/04 05:35:31 project04: inserted rows to seq 5000
c^C
ubuntu@ece573:~/ece573-prj04$

```

Figure 23: Successful write of cassandra.

### 3.2.2 Question 2

**Q:** How does the writer deployment connect to the Cassandra service? In particular, where does "cassandra-service.default.svc.cluster.local" come from?

The internal DNS system of Kubernetes, which automatically assigns DNS names to services inside the cluster, is used by the writer deployment to connect to the Cassandra server. In accordance with Kubernetes' DNS naming policy for services, the name "cassandra-service.default.svc.cluster.local" enables pods to interact with one another using a consistent DNS name.

The service developed in Kubernetes to expose the Cassandra pods is called cassandra-service. The service is described in the cassandra.yml file as:

```
apiVersion: v1
kind: Service
metadata:
  name: cassandra-service
```

Code 3: Snippet from the cassandra.yml file.

For other pods (such as the writer pod) to connect to Cassandra, the Service offers a reliable IP address and port. The service's DNS doesn't change even if the underlying Cassandra pods (such as cassandra-0, cassandra-1) are restarted or relocated to other worker nodes.

.default: This is the namespace in which the pods and service are operating. Unless specifically specified in a different namespace, all Kubernetes resources are initially produced in the default namespace.

.svc.cluster.local is the default domain name for Kubernetes cluster services. Kubernetes is assisted in resolving service addresses within the cluster using the entire internal DNS suffix.

The writer connects to cassandra through the following:

- **Kubernetes Service:** The Cassandra service (cassandra-service) is a ClusterIP service that allows other apps running in the same Kubernetes cluster to access the Cassandra pods. This stable endpoint serves as the conduit between the writer pod and the Cassandra service.
- **DNS Resolution:** Using its own DNS system, Kubernetes resolves cassandra-service.default.svc.cluster.local to the ClusterIP of the Cassandra service, which subsequently routes the request to one of the Cassandra pods when the writer deployment tries to connect to it.
- **Environment Variable:** The CASSANDRA\_SEEDS environment variable in the writer.yml file instructs the writer pod to establish a connection to this service as shown in Code 4. This means that the writer application connects to Cassandra by using this DNS name, which resolves to the Cassandra service.

```
env:
  - name: TOPIC
    value: "project04"
  - name: CASSANDRA_SEEDS
    value: "cassandra-service.default.svc.cluster.local"
  - name: CASSANDRA_KEYSPACE
    value: "ece573"
  - name: CONSISTENCY
    value: "ONE"
```

Code 4: Code snippet from writer.yml focusing on env.

- Load balancing and service discovery: The Cassandra service makes sure that traffic is distributed evenly across the Cassandra pods that are accessible. The writer application can remain oblivious to the real IP addresses of the Cassandra pods thanks to Kubernetes' integrated service discovery feature.

### 3.2.3 Question 3

**Q:** We add retrying logic to the writer program in Project 3. Do we need it for Project 4?

Yes, even if Kubernetes has fault-tolerance techniques such as pod restart capabilities, Project 4 still requires retrying logic.

The following elucidates the need for the retrying logic:

- Temporary Network Outages:
  - Network communication between pods or services may encounter transitory problems in a distributed system such as Kubernetes (e.g., momentary loss of connection, packet drops, etc.).
  - The writer pod may not connect to Cassandra on the first try if there is a temporary problem with Cassandra or the network. The writer would crash instantly without retry logic, necessitating needless pod restarts.
  - By trying to reconnect several times before giving up, retrying logic enables the writer to gracefully tolerate brief network outages.
- Restarting the Cassandra Pod:
  - The writer pod may briefly lose connectivity to Cassandra if one or more Cassandra pods are momentarily unavailable (for example, during restarts or scale-up/scale-down procedures).
  - Even if the problem is just momentary, the writer pod will crash if it lacks retry logic while Cassandra is unavailable. The writer can wait and try to connect again when Cassandra becomes accessible by using retrying logic.
- Preventing Needless Pod Restarts:
  - If the writer pod crashes (because of CrashLoopBackOff), Kubernetes will restart it immediately, although this causes needless downtime.

The writer pod may manage brief failures internally with retry logic, negating the need for a complete restart.

- Retry logic will make the writer more robust and enable it to bounce back from brief connection problems without completely failing.

The functioning of the retrying logic in project 4 is as follows:

We included retry logic in Project 3 to deal with Cassandra connection failures. This reasoning is still useful in Project 4 because:

- If the connection fails, the writer pod will try to reconnect to Cassandra many times rather than quitting right away.
- To make sure the writer doesn't overload the network or Cassandra with too many connection attempts in a short amount of time, the retry logic can be set up (for example, using exponential backoff).

## 4 Stateless Application

### 4.1 Questions

#### 4.1.1 Question 1

**Q:** Modify writer.go as mentioned above and verify everything is running properly. You will need to delete Pods to trigger writer to restart and show log messages indicating lastSeq reading from Cassandra.

The following Figure 24 is the set of steps where ./build.sh is run again and then the existing writer is deleted. There after a fresh and new writer is deployed.

```
ubuntu@ece573:~/ece573-prj04$ ./build.sh
[+] Building 54.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 483B
=> WARN: FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 3)
=> WARN: FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 13)
=> [internal] load metadata for docker.io/library/golang:1.21
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 8.98kB
=> CACHED [build 1/4] FROM docker.io/library/golang:1.21@sha256:4746d26432a9117a5f58e95cb9f954ddf0de128e9d5816886514199316e4a2fb
=> [build 2/4] COPY . /go/src
=> [build 3/4] WORKDIR /go/src/writer
=> [build 4/4] RUN CGO_ENABLED=0 GOOS=linux go build -o writer
=> [image 1/1] COPY --from=build /go/src/writer/writer .
=> exporting to image
=> => exporting layers
=> => writing image sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8
=> => naming to docker.io/library/ece573-prj04-writer:v1

2 warnings found (use docker --debug to expand):
- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 3)
- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 13)
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-control-plane", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-worker4", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-worker2", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-worker5", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-worker", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-worker3", loading...
Image: "ece573-prj04-writer:v1" with ID "sha256:71e2e006e4250c34aeb578a7922784e185f4437a60980552f9769a9e8db5d1f8" not yet present on node "kind-worker6", loading...
ubuntu@ece573:~/ece573-prj04$ kubectl delete -f writer.yml
deployment.apps "ece573-prj04-writer" deleted
ubuntu@ece573:~/ece573-prj04$ kubectl apply -f writer.yml
deployment.apps/ece573-prj04-writer created
ubuntu@ece573:~/ece573-prj04$
```

Figure 24: Docker build and replacing writer with new.



The following Code 5 is updated code snippet of the writer.go file.

```
package main

import (
    "log"
    "math/rand"
    "os"
    "strings"

    "github.com/gocql/gocql"
)

func main() {
    // Check topic environment variable
    topic := os.Getenv("TOPIC")
    if topic == "" {
        log.Fatalf("Unknown topic: TOPIC environment variable is not set")
    }
    log.Printf("Using topic: %s", topic)

    // Set consistency level from environment variable
    cs := os.Getenv("CONSISTENCY")
    var consistency gocql.Consistency
    switch strings.ToUpper(cs) {
    case "ALL":
        consistency = gocql.All
    case "ONE":
        consistency = gocql.One
    case "QUORUM":
        consistency = gocql.Quorum
    default:
        log.Fatalf("Unknown consistency level %s", cs)
    }

    // Get Cassandra seed node from environment variable
    seed := os.Getenv("CASSANDRA_SEEDS")
    log.Printf("Connecting cluster at %s with consistency %s for topic %s", seed, consistency, topic)

    // Create a new Cassandra session
    cluster := gocql.NewCluster(seed)
    cluster.Consistency = consistency
    session, err := cluster.CreateSession()
    if err != nil {
        log.Fatalf("Cannot connect to cluster at %s: %v", seed, err)
    }
    defer session.Close()

    // Verify connection by querying the cluster name
    var clusterName string
    if err := session.Query("SELECT cluster_name FROM system.local").Scan(&clusterName); err != nil {
        log.Fatalf("Cannot query cluster: %v", err)
    }
    log.Printf("Connected to cluster %s", clusterName)

    // Create keyspace and tables if they don't exist
    err = session.Query(
        `CREATE KEYSPACE IF NOT EXISTS ece573
        WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3}`).Exec()
    if err != nil {
        log.Fatalf("Cannot create keyspace ece573: %v", err)
    }

    err = session.Query(
        `CREATE TABLE IF NOT EXISTS ece573.prj04 (
        topic text, seq int, value double,
        PRIMARY KEY (topic, seq))`).Exec()
    if err != nil {
        log.Fatalf("Cannot create table ece573.prj04: %v", err)
    }

    err = session.Query(
        `CREATE TABLE IF NOT EXISTS ece573.prj04_last_seq (
        topic text, seq int,
        PRIMARY KEY (topic))`).Exec()
    if err != nil {
        log.Fatalf("Cannot create table ece573.prj04_last_seq: %v", err)
    }

    // Log that tables are ready
    log.Printf("Tables ece573.prj04 and ece573.prj04_last_seq ready.")

    // Fetch the last sequence from Cassandra
    var lastSeq int
    err = session.Query(
        `SELECT seq FROM ece573.prj04_last_seq WHERE topic = ? LIMIT 1`, topic).Scan(&lastSeq)
    if err != nil {
        if err == gocql.ErrNotFound {
            // If no record is found, start from lastSeq = 0
            lastSeq = 0
            log.Printf("No last sequence found for topic %s, starting from seq=0", topic)
        } else {
            log.Fatalf("Error reading last sequence: %v", err)
        }
    }
    log.Printf("%s: Resuming from lastSeq=%d", topic, lastSeq)

    // Start the loop from lastSeq + 1
    for seq := lastSeq + 1; ; seq++ {
        value := rand.Float64()

        // Insert new row into prj04 table
        err := session.Query(
            `INSERT INTO ece573.prj04 (topic, seq, value) VALUES (?, ?, ?)`,
            topic, seq, value).Exec()
        if err != nil {
            log.Fatalf("Cannot write %d to table ece573.prj04: %v", seq, err)
        }

        // Update the last sequence number for the topic
        err = session.Query(
            `INSERT INTO ece573.prj04_last_seq (topic, seq) VALUES (?, ?)`,
            topic, seq).Exec()
        if err != nil {
            log.Fatalf("Cannot write %d to table ece573.prj04_last_seq: %v", seq, err)
        }

        // Log progress every 1000 rows
        if seq%1000 == 0 {
            log.Printf("%s: inserted rows to seq %d", topic, seq)
        }
    }
}
```

Code 5: Edited and updated code from writer.go.

```

ubuntu@ece573:~/ece573-prj04$ kubectl logs -l app=ece573-prj04-writer -f
2024/11/04 06:11:33 project04: inserted rows to seq 206000
2024/11/04 06:11:43 project04: inserted rows to seq 207000
2024/11/04 06:11:53 project04: inserted rows to seq 208000
2024/11/04 06:12:03 project04: inserted rows to seq 209000
2024/11/04 06:12:13 project04: inserted rows to seq 210000
2024/11/04 06:12:23 project04: inserted rows to seq 211000
2024/11/04 06:12:33 project04: inserted rows to seq 212000
2024/11/04 06:12:43 project04: inserted rows to seq 213000
2024/11/04 06:12:53 project04: inserted rows to seq 214000
2024/11/04 06:13:03 project04: inserted rows to seq 215000

```

Figure 25: Current running log of row insertion being done.

Figure 25 shows the current condition with the logging of row insertion and Figure 26 shows the stopping of the pod and restarting making it start from the previous left off row.

```

ubuntu@ece573:~/ece573-prj04$ kubectl logs -l app=ece573-prj04-writer -f
2024/11/04 06:11:33 project04: inserted rows to seq 206000
2024/11/04 06:11:43 project04: inserted rows to seq 207000
2024/11/04 06:11:53 project04: inserted rows to seq 208000
2024/11/04 06:12:03 project04: inserted rows to seq 209000
2024/11/04 06:12:13 project04: inserted rows to seq 210000
2024/11/04 06:12:23 project04: inserted rows to seq 211000
2024/11/04 06:12:33 project04: inserted rows to seq 212000
2024/11/04 06:12:43 project04: inserted rows to seq 213000
2024/11/04 06:12:53 project04: inserted rows to seq 214000
2024/11/04 06:13:03 project04: inserted rows to seq 215000
2024/11/04 06:13:14 project04: inserted rows to seq 216000
2024/11/04 06:13:24 project04: inserted rows to seq 217000
2024/11/04 06:13:35 project04: inserted rows to seq 218000
2024/11/04 06:13:45 project04: inserted rows to seq 219000
2024/11/04 06:13:55 project04: inserted rows to seq 220000
2024/11/04 06:14:06 project04: inserted rows to seq 221000
2024/11/04 06:14:16 project04: inserted rows to seq 222000
2024/11/04 06:14:26 project04: inserted rows to seq 223000
2024/11/04 06:14:36 project04: inserted rows to seq 224000
2024/11/04 06:14:46 project04: inserted rows to seq 225000
2024/11/04 06:14:56 project04: inserted rows to seq 226000
^C
ubuntu@ece573:~/ece573-prj04$ kubectl delete pod -l app=ece573-prj04-writer
pod "ece573-prj04-writer-6fd86df6db-gnqbt" deleted
ubuntu@ece573:~/ece573-prj04$ kubectl logs -l app=ece573-prj04-writer -f
2024/11/04 06:15:07 Using topic: project04
2024/11/04 06:15:07 Connecting cluster at cassandra-service.default.svc.cluster.local with consistency ONE for topic project04
2024/11/04 06:15:07 Connected to cluster ece573-prj04
2024/11/04 06:15:07 Tables ece573.prj04 and ece573.prj04_last_seq ready.
2024/11/04 06:15:07 project04: Resuming from lastSeq=226880
2024/11/04 06:15:09 project04: inserted rows to seq 227000
2024/11/04 06:15:20 project04: inserted rows to seq 228000

```

Figure 26: Stopping pod and restarting from the previous top point.

#### 4.1.2 Question 2

**Q:** What happens if consistency is broken so that the writer doesn't obtain the most recent last seq? Will this cause an issue for the writer?

Consistency is the assurance that every node in the cluster has the same data at any given time in a distributed system such as Cassandra. The writer may get an out-of-date or inaccurate lastSeq value from one of the Cassandra nodes if consistency is compromised, which indicates that certain nodes are not completely synced.

Duplicate data will be added to the ece573.prj04 table if the writer receives an out-of-date lastSeq (that is, a sequence number that is less than the actual last sequence

written). Duplicate entries for the same sequence number, maybe with different values, result from the writer beginning to insert rows from a sequence number that has already been used.

The following are the potential issues that may arise:

1. **Data Integrity:** Inconsistent application behavior may result from duplicate rows having the same sequence number. Other apps may have trouble processing the data if they depend on the sequence numbers being accurate.
2. **Data Overwriting:** If the lastSeq is inaccurate, the writer may overwrite rows with wrong sequence numbers that already exist in the ece573.prj04\_last\_seq table, which would result in even more discrepancies.

The following are the ways to mitigate the issues:

1. **Employ a Stricter Consistency Level:** When querying Cassandra, the writer can employ a more stringent consistency level, such as QUORUM or ALL, to prevent reading out-of-date data. This guarantees that before the writer moves forward, the majority (or all) of the nodes concur on the returned data.
2. **Introduce Conflict Resolution:** The application may use conflict resolution techniques to deal with duplicate rows or inconsistencies if more stringent consistency standards are not practical because of performance issues.

This indeed could cause issues for the writer if it retrieves an outdated lastSeq due to broken consistency. This can lead to duplicate rows and inconsistent data in the Cassandra tables.

By guaranteeing that the writer accesses the most recent data from the cluster, a higher consistency level, such as QUORUM or ALL, can help to alleviate these problems.