Apache Kafka® Administration by Confluent

Exercise Book

Version 7.5.2-v1.0.0

Table of Contents

Copyright & Trademarks

Lab 01 Fundamentals of Apache Kafka

a. Introduction

This document provides Hands-On Exercises for the course **Apache Kafka® Administration by Confluent**. You will use a setup that includes a virtual machine (VM) configured as a Docker host to demonstrate the distributed nature of Apache Kafka.

The main Kafka cluster includes the following components, each running in a Docker container:

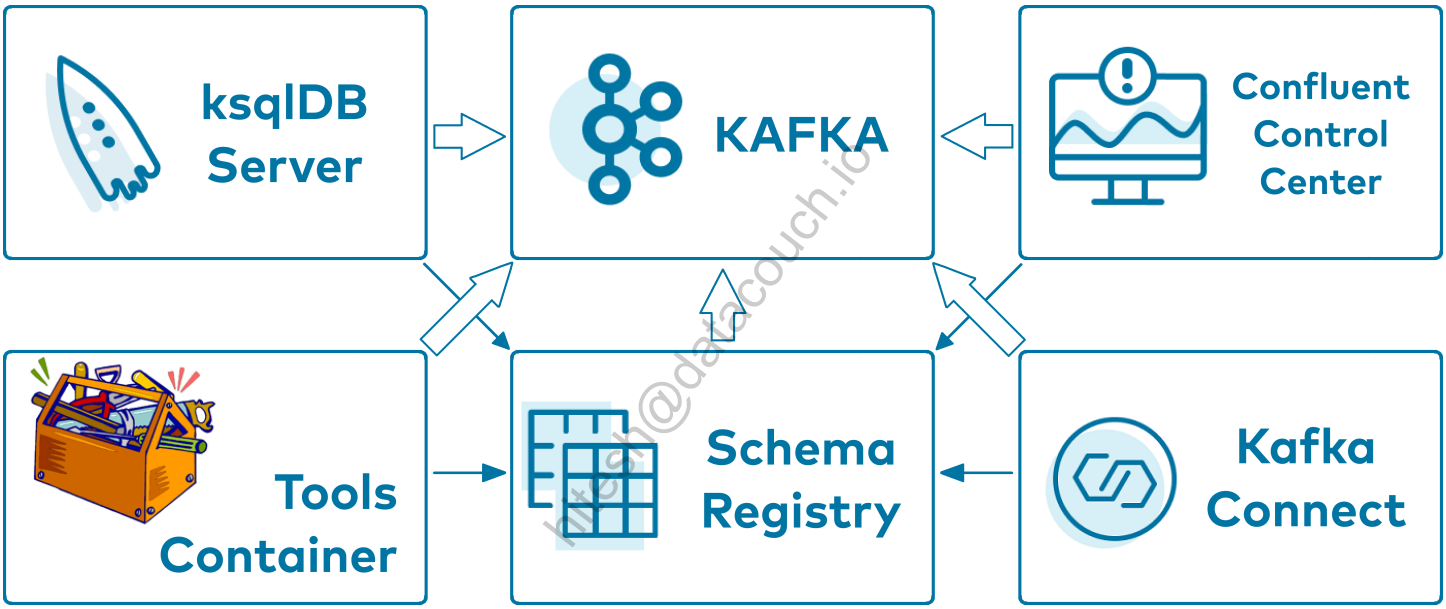


Table 1. Components of the Confluent Platform

Hostname	Description
controller-1	Kafka Controller 1
controller-2	Kafka Controller 2
controller-3	Kafka Controller 3
kafka-1	Kafka Broker 1
kafka-2	Kafka Broker 2
kafka-3	Kafka Broker 3
schema-registry	Schema Registry
kafka-connect	Kafka Connect

Hostname	Description
control-center	Confluent Control Center
ksqlDB-server	ksqlDB Server
tools	A container used to run tools against the cluster

As you progress though the exercises you will selectively turn on parts of your cluster as they are needed.

You will use Confluent Control Center to monitor the main Kafka cluster. To achieve this, we are also running the Control Center service which is backed by the same Kafka cluster.

In this course we are using Confluent Platform version 7.5.2 which includes Kafka 3.5.2.

In production, Control Center should be deployed with its own dedicated Kafka cluster, separate from the cluster with production traffic. Using a dedicated metrics cluster is more resilient because it continues to provide system health monitoring even if the production traffic cluster experiences issues.

Alternative Lab Environments

As an alternative you can also download the VM to your laptop and run it in VirtualBox. Make sure you have the newest version of VirtualBox installed. Download the VM from this link:

- https://s3.amazonaws.com/confluent-training-images-us-east-1/training-ubuntu-20-04-jan2022.ova

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine, then you can run the labs there. But please note that your trainer might not be able to troubleshoot any potential problems if you are running the labs locally. If you choose to do this, follow the instructions at → Running Labs in Docker for Desktop.

Command Line Examples

Most Exercises contain commands that must be run from the command line. These commands will look like this:

```
$ pwd
/home/training
$ echo very long \
  line
very long line
```


Commands you should type are shown after the prompt (`$`) and in bold ; plain text is an example of the output produced as a result of the command. Also, very long commands have the `\` character at the end of lines to indicate command continues on the next line. These multi-line commands must be typed (or pasted) in their entirety **before** hitting `Enter`.

Continued Learning After Class

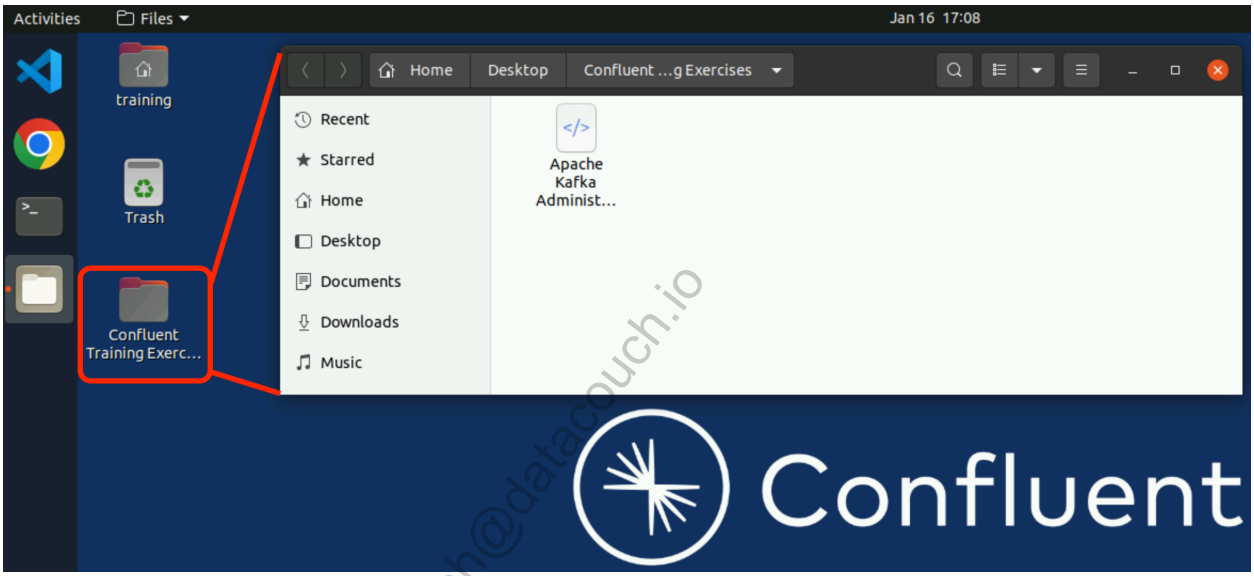
Once the course ends, the VM in the Training Portal will terminate and you will no longer have access to it. However, you can still download the VM onto your own machine or use Docker locally to revisit these materials. We encourage you to bring up your own test environment, explore configuration files, inspect scripts, and perform tests. Here are some activities we encourage to reinforce your learning:

- Revisit the exercises in this exercise book
- Summarize and discuss the student handbook with your peers
- Consult the README in this course's public source repository: https://github.com/confluentinc/training-operations-src

Preparing the Lab

Welcome to your lab environment! You are connected as user **training**, password **training**.

If you haven't already done so, you should open the **Exercise Guide** that is located on the lab virtual machine. To do so, open the **Confluent Training Exercises** folder that is located on the lab virtual machine desktop. Then double-click the shortcut that is in the folder to open the **Exercise Guide**.



Copy and paste works best if you copy from the Exercise Guide on your lab virtual machine.

- Standard Ubuntu keyboard shortcuts will work: `Ctrl+C` → Copy, `Ctrl+V` → Paste
- In a Terminal window: `Ctrl+Shift+C` → Copy, `Ctrl+Shift+V` → Paste.

If you find these keyboard shortcuts are not working you can use the right-click context menu for copy and paste.

1. Open a terminal window
2. Clone the source code repository to the folder `confluent-admin` in your **home** directory and change to that directory:

```
$ cd ~
$ git clone --branch 7.5.2-v1.0.0 \
  https://github.com/confluentinc/training-administration-src.git \
  confluent-admin
```

If you chose to select another folder for the labs then note that many of our samples assume that the lab folder is `~/confluent-admin`. You will have to adjust all those commands to fit your specific environment.

3. Navigate to the `confluent-admin` folder:

```
$ cd ~/confluent-admin
```

4. Start the complete Kafka cluster with the following command:

```
$ docker-compose up -d
[+] Running 9/9
 ⠿ kafka-2 Pulled
 ⠿ controller-2 Pulled
 ⠿ kafka-connect Pulled
 ⠿ schema-registry Pulled
 ⠿ kafka-1 Pulled
 ⠿ controller-3 Pulled
 ⠿ kafka-3 Pulled
 ⠿ control-center Pulled
 ⠿ controller-1 Pulled
[+] Running 10/10
 ⠿ Network confluent-admin_default  Created
 ⠿ Container controller-3           Started
 ⠿ Container controller-1           Started
 ⠿ Container controller-2           Started
 ⠿ Container kafka-2                Healthy
 ⠿ Container kafka-1                Healthy
 ⠿ Container kafka-3                Healthy
 ⠿ Container control-center         Started
 ⠿ Container kafka-connect          Started
 ⠿ Container schema-registry        Started
```


5. Monitor the cluster with:

```
$ docker-compose ps
NAME                COMMAND                  SERVICE
STATUS              PORTS
control-center      "/etc/confluent/dock…"   control-center
running             0.0.0.0:9021->9021/tcp, :::9021->9021/tcp
controller-1        "/etc/confluent/dock…"   controller-1
running             0.0.0.0:19093->19093/tcp, :::19093->19093/tcp
controller-2        "/etc/confluent/dock…"   controller-2
running             0.0.0.0:29093->29093/tcp, :::29093->29093/tcp
controller-3        "/etc/confluent/dock…"   controller-3
running             0.0.0.0:39093->39093/tcp, :::39093->39093/tcp
kafka-1             "/etc/confluent/dock…"   kafka-1
running (healthy)   0.0.0.0:10001->10001/tcp, 0.0.0.0:19092-
>19092/tcp, :::10001->10001/tcp, :::19092->19092/tcp
kafka-2             "/etc/confluent/dock…"   kafka-2
running (healthy)   0.0.0.0:10002->10002/tcp, 0.0.0.0:29092-
>29092/tcp, :::10002->10002/tcp, :::29092->29092/tcp
kafka-3             "/etc/confluent/dock…"   kafka-3
running (healthy)   0.0.0.0:10003->10003/tcp, 0.0.0.0:39092-
>39092/tcp, :::10003->10003/tcp, :::39092->39092/tcp
kafka-connect       "/etc/confluent/dock…"   kafka-connect
running (healthy)   0.0.0.0:8083->8083/tcp, :::8083->8083/tcp
schema-registry     "/etc/confluent/dock…"   schema-registry
running (healthy)   0.0.0.0:8081->8081/tcp, :::8081->8081/tcp
```

All services should have **`Status`** equal to **`running`**.

6. OPTIONAL: You can also observe the stats of Docker on your VM:

```
$ docker stats
CONTAINER ID   NAME              CPU %     MEM USAGE / LIMIT     MEM
%     NET I/O           BLOCK I/O         PIDS
67d1e407cf17   kafka-connect     3.09%     2.433GiB / 15.33GiB
15.87%    961kB / 741kB     1.21GB / 4.1kB    43
36ce65c891e4   schema-registry   3.48%     242.5MiB / 15.33GiB
1.55%     260kB / 190kB     49.9MB / 41kB     35
367e277753db   control-center    19.58%    887.1MiB / 15.33GiB
5.65%     50MB / 30.1MB     103MB / 7.13MB    129
d1dcef5ff1f2   kafka-3           69.30%    782.5MiB / 15.33GiB
4.99%     45.9MB / 50.4MB   18.1MB / 8.65MB   123
74d0602dc5a6   kafka-2           39.39%    797.1MiB / 15.33GiB
5.08%     46.1MB / 52.9MB   27.6MB / 8.73MB   124
b0ffc65be8c5   kafka-1           19.89%    790MiB / 15.33GiB
5.03%     46.5MB / 54.5MB   16.9MB / 8.45MB   124
15df88e2bd32   controller-1      7.65%     337.6MiB / 15.33GiB
2.15%     569kB / 299kB     25.7MB / 4.83MB   59
54b0ab5ede52   controller-2      1.01%     391MiB / 15.33GiB
2.49%     1.64MB / 2.97MB   55.8MB / 4.82MB   61
a474b6f0bdc7   controller-3      5.62%     352.2MiB / 15.33GiB
2.24%     569kB / 298kB     36.5MB / 4.83MB   59
```


Press **Ctrl+C** to exit Docker stats.

7. NOT OPTIONAL: Add to the shell's environment a few shortcut variables:

```
$ cat <<EOF >~/.myvars
export CLUSTERID=$(grep -m1 CLUSTER_ID: \
  ~/confluent-admin/docker-compose.yml \
  | tr -d \ | cut -d: -f2)
export CONTROLLERS="9991@controller-1:19093,\
9992@controller-2:29093,9993@controller-3:39093"
export BOOTSTRAPS="kafka-1:19092,kafka-2:29092,kafka-3:39092"
EOF
$ x='source ~/.myvars' ; grep -qxF "$x" ~/.bashrc || echo $x
>>~/.bashrc
$ source ~/.bashrc
```


Testing the Installation

1. Use the `kafka-metadata-shell` command to verify that all Brokers have registered with the Controller sub-cluster. You should see the three Brokers (1 to 3) listed in the output.

```
$ kafka-metadata-shell --cluster-id $CLUSTERID --controllers
$CONTROLLERS ls image/cluster
1
2
3
```

For those **not** using the VM but instead are using Docker for Desktop, remember to run commands against the cluster from within the `tools` container with `docker-compose exec tools bash`.

OPTIONAL: Analyzing the Docker Compose File

1. Open the file `docker-compose.yml` in your editor and:
   a. locate the various services that are listed in the table earlier in this section
   b. note that the `hostname` (e.g. `controller-1` or `kafka-2`) is used to resolve a particular service
   c. note how each Kafka Controller (controller-1, controller-2, controller-3)
      i. gets a unique ID assigned via environment variable `KAFKA_NODE_ID`
      ii. gets the information about all members of the Controller sub-cluster:

```
KAFKA_CONTROLLER_QUORUM_VOTERS: 9991@controller-
1:19093,9992@controller-2:29093,9993@controller-3:39093
```

   d. note how each Broker (kafka-1, kafka-2, kafka-3) also defines its `KAFKA_NODE_ID` and `KAFKA_CONTROLLER_QUORUM_VOTERS` so it can join the Kafka cluster.

      It also configures the Broker to send metrics to Control Center:

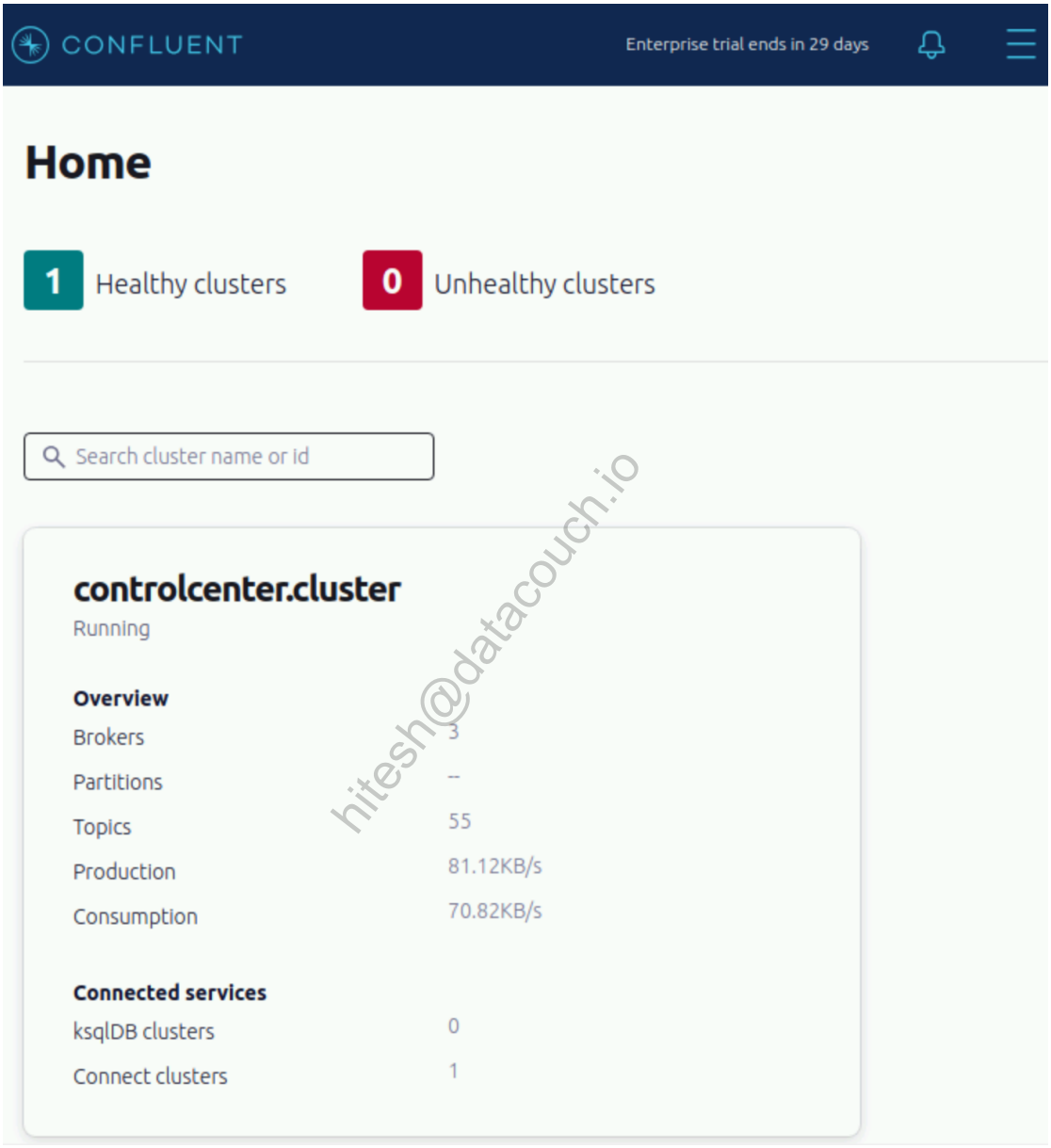
```
KAFKA_METRIC_REPORTERS:
"io.confluent.metrics.reporter.ConfluentMetricsReporter"
KAFKA_CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: kafka-
1:9092,kafka-2:9092,kafka-3:9092
```

e. note how various services use the environment variable **`…_BOOTSTRAP_SERVERS`** to define the list of Kafka Brokers that serve as bootstrap servers:

```
..._BOOTSTRAP_SERVERS: kafka-1:9092,kafka-2:9092,kafka-3:9092
```

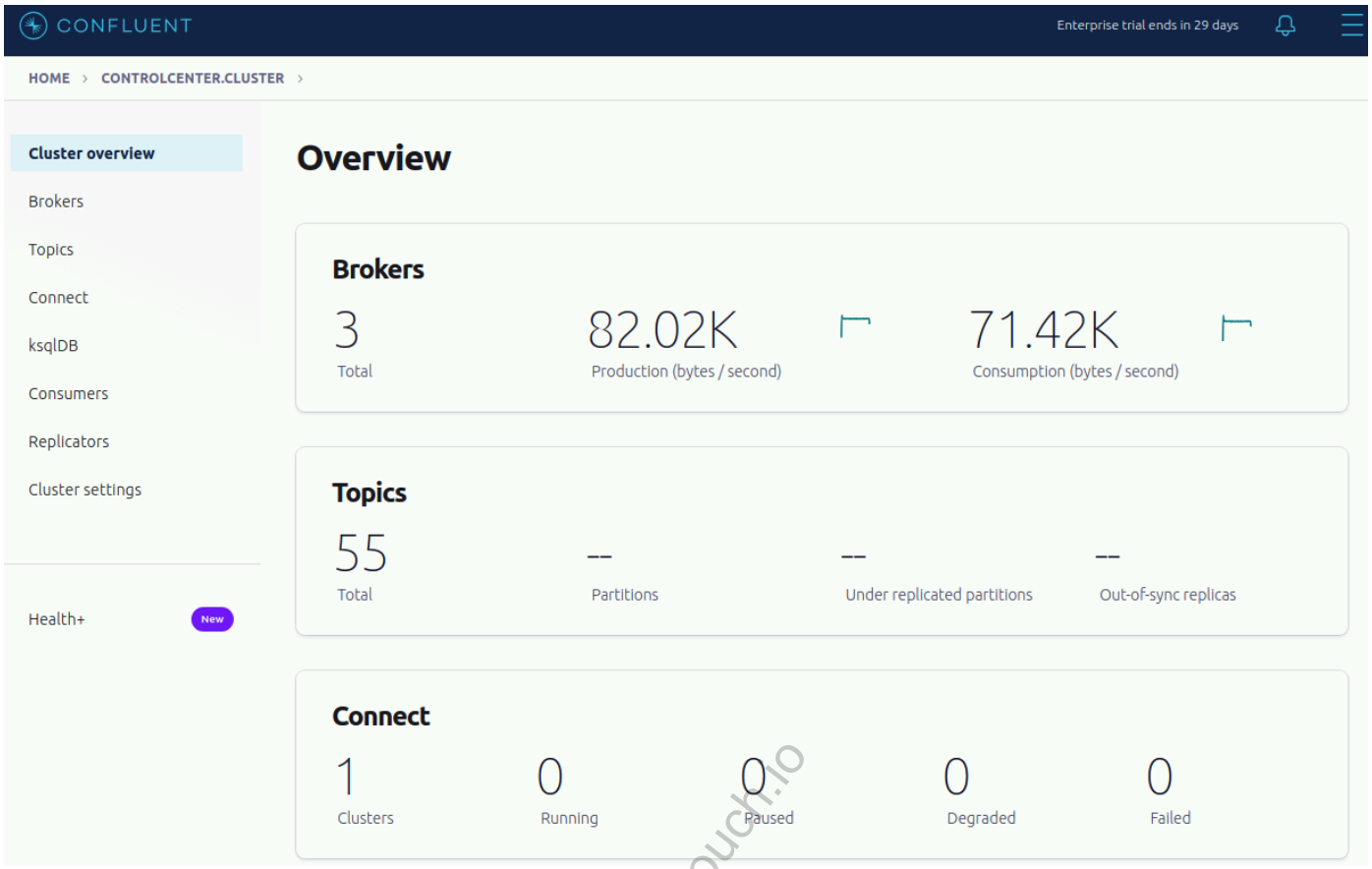

Using Control Center

1. In the VM, open a new browser tab in Google Chrome.
2. Navigate to Control Center at http://localhost:9021:

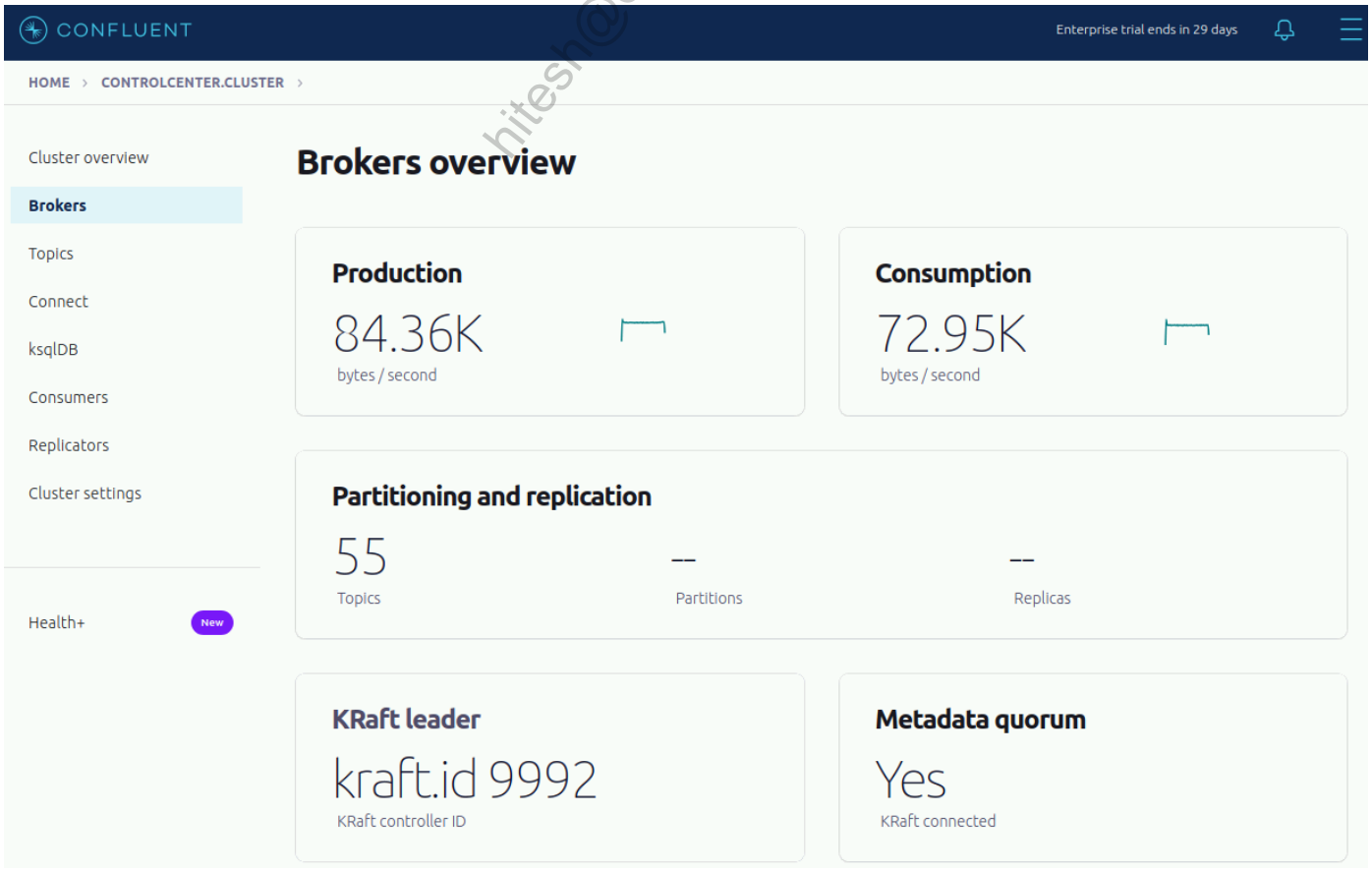


The Control Center environment will take a few minutes to stabilize and report a healthy cluster.

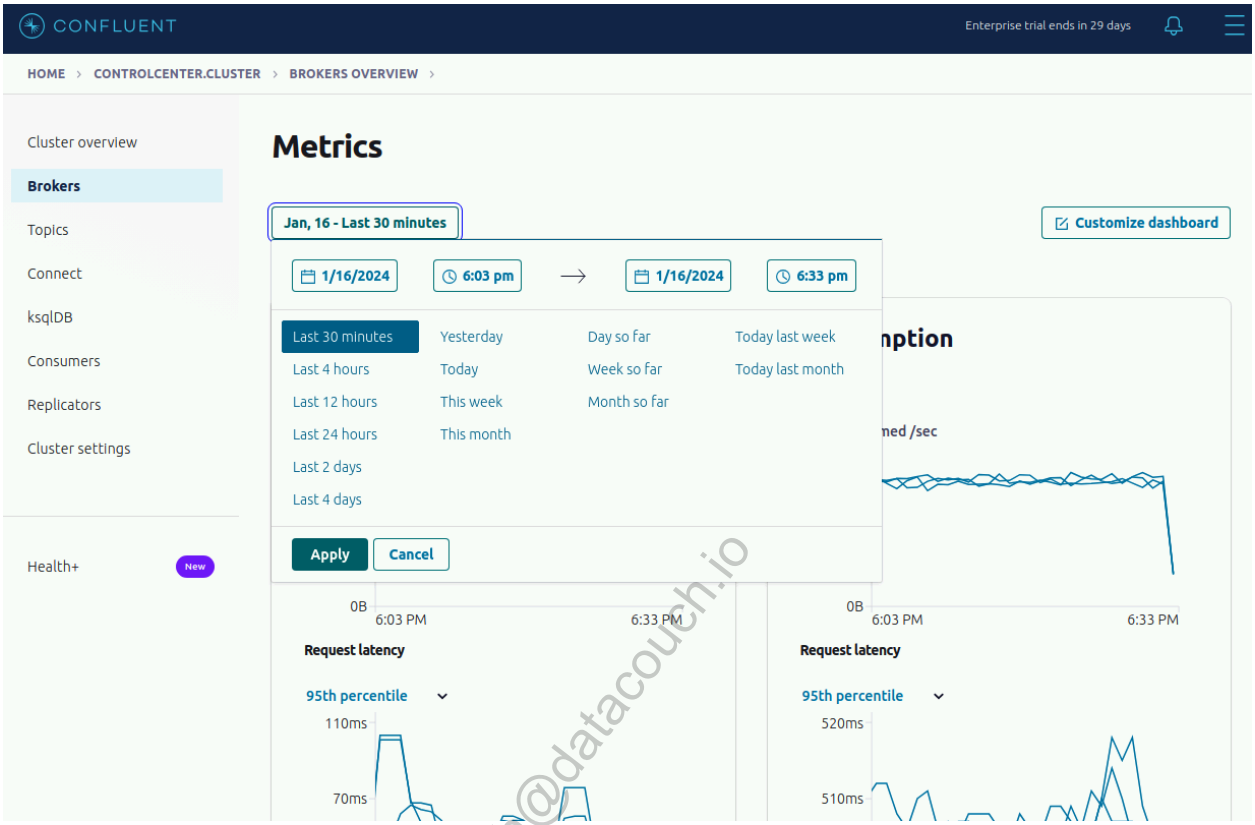
3. Select the cluster **controlcenter.cluster** and you will see this:



4. Click on **Brokers** on the left to go to the **Brokers overview** page.



5. In the **Brokers overview** page, click on **Production** to open the **Metrics** page.
6. Click on the button that shows the current date, and change `Last 4 hours` to `Last 30 minutes`.



7. In the **Production** section of the **Metrics** page, you are looking at metrics for **Produce** requests in your cluster. Hover your mouse over the **Throughput** and **Request latency** graphs.

   Consumption metrics are visible either to the right of or below the **Production** metrics depending upon the Control Center window width. This section of the **Metrics** page displays similar metrics for **Consume** requests.

> In production, Control Center should be deployed with its own dedicated Kafka cluster, separate from the cluster with production traffic. Using a dedicated metrics cluster is more resilient because it continues to provide system health monitoring even if the production traffic cluster experiences issues.

Cleanup

1. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```


b. Using Kafka's Command-Line Tools

In this Hands-On Exercise you will start to become familiar with some of Kafka's command-line tools. Specifically you will:

- Use a tool to **create** a Topic
- Use a console program to **produce** a message
- Use a console program to **consume** a message
- Use a tool to explore Cluster Metadata

Prerequisites

1. Navigate to the `confluent-admin` folder:

```
$ cd ~/confluent-admin
```

2. Run the Kafka cluster, including Control Center:

```
$ docker-compose up -d
```


Console Producing and Consuming

Kafka has built-in command line utilities to produce messages to a Topic and read messages from a Topic. These are extremely useful to verify that Kafka is working correctly, and for testing and debugging.

1. Before we can start writing data to a Topic in Kafka, we need to first create that Topic using a tool called `kafka-topics`. From within the terminal window run the command:

```
$ kafka-topics
```

   This will bring up a list of parameters that the `kafka-topics` program can receive. Take a moment to look through the options.

2. Now execute the following command to create the Topic `testing`:

```
$ kafka-topics --bootstrap-server $BOOTSTRAPS \
    --create \
    --partitions 1 \
    --replication-factor 1 \
    --topic testing
```

We create the Topic with a single Partition and `replication-factor` of one.

If `auto.create.topics.enable` was set to `true` and the previous step had not been completed, the Topic would automatically be created when the first record is written to it using the default topic number of partitions and default replication factor. Enabling auto topic creation is **strongly discouraged** in production. Always create your Topics explicitly!

3. Now let's move on to start writing data into the Topic just created. From within the terminal window run the command:

```
$ kafka-console-producer
```

This will bring up a list of parameters that the `kafka-console-producer` program can receive. Take a moment to look through the options. We will discuss many of their meanings later in the course.

4. Run `kafka-console-producer` again with the required arguments:

```
$ kafka-console-producer --bootstrap-server $BOOTSTRAPS --topic
testing
```

The tool prompts you with a `>`.

5. At this prompt type:

```
> some data
```

And hit **Enter**.

6. Now type:

```
> more data
```

   And hit **Enter**.

7. Type:

```
> final data
```

   And hit **Enter**.

8. Press `Ctrl+D` to exit the `kafka-console-producer` program.

9. Now we will use a Consumer to retrieve the data that was produced. Run the command:

```
$ kafka-console-consumer
```

   This will bring up a list of parameters that the `kafka-console-consumer` can receive. Take a moment to look through the options.

10. Run `kafka-console-consumer` again with the following arguments:

```
$ kafka-console-consumer \
    --bootstrap-server $BOOTSTRAPS \
    --from-beginning \
    --topic testing
```

    After a short moment you should see all the messages that you produced using `kafka-console-producer` earlier:

```
some data
more data
final data
```

11. Press `Ctrl+C` to exit `kafka-console-consumer`.

OPTIONAL: Running Producer and Consumer in Parallel

The `kafka-console-producer` and `kafka-console-consumer` programs can be run at the same time. Run `kafka-console-producer` and `kafka-console-consumer` in separate terminal windows at the same time to see how `kafka-console-consumer` receives the events.

OPTIONAL: Working with record keys

By default, `kafka-console-producer` and `kafka-console-consumer` assume null keys. They can also be run with appropriate arguments to write and read keys as well as values.

1. Re-run the Producer with additional arguments to write (key,value) pairs to the Topic:

```
$ kafka-console-producer \
    --bootstrap-server $BOOTSTRAPS \
    --topic testing \
    --property parse.key=true \
    --property key.separator=,
```

2. Enter a few values such as:

```
> 1,my first record
> 2,another record
> 3,Kafka is cool
```

3. Press `Ctrl+D` to exit the producer.

4. Now run the **Consumer** with additional arguments to print the key as well as the value:

```
$ kafka-console-consumer \
    --bootstrap-server $BOOTSTRAPS \
    --from-beginning \
    --topic testing \
    --property print.key=true

null    some data
null    more data
null    final data
1   my first record
2   another record
3   Kafka is cool
```


Note the `NULL` values for the first 3 records that we entered earlier.

5. Press `Ctrl+C` to exit the Consumer.

The Kafka Metadata Shell

1. Access Kafka's Metadata by using the `kafka-metadatas-shell` command to connect to the Controller sub-cluster:

```
$ kafka-metadata-shell --cluster-id $CLUSTERID --controllers
$CONTROLLERS
Loading...
Starting...
[ Kafka Metadata Shell ]
>>
```

2. From within the `kafka-metadata-shell` application, type `ls /` to view the directory structure of the Metadata. Note the leading `/` is not required.

```
ls /
image  local
```

3. Type `ls /image` to see this next level of the directory structure.

```
ls /image
acls          cluster       encryptor    provenance         tenants
cells         clusterLinks  features     replicaExclusions  topics
clientQuotas  configs       producerIds  scram
```

4. Type `ls /image/cluster` to see the node ids for the Kafka brokers.

```
ls /image/cluster
1 2 3
```

Note the last output `1 2 3`, indicating that we have 3 Brokers with IDs `1`, `2`, `3` in our cluster. The node IDs for the controllers (`9991`, `9992` and `9993`) do **not** appear here.

5. Type `cat /image/cluster/1` to see the metadata for broker 1.


```
cat /image/cluster/1
BrokerRegistration(id=1, epoch=5,
incarnationId=F6vv9ZrFRfGJbOyzSPmzGw,
listeners=[Endpoint(listenerName='DOCKER',
securityProtocol=PLAINTEXT, host='kafka-1', port=9092),
Endpoint(listenerName='EXTERNAL', securityProtocol=PLAINTEXT,
host='kafka-1', port=19092)],
supportedFeatures={confluent.metadata.version: 1-111,
metadata.version: 1-11}, rack=Optional[rack-0], fenced=false,
inControlledShutdown=false, isMigratingZkBroker=false,
degradedComponents=[])
```

6. Type `cat /image/topics/byName/testing/0` to see the metadata for partition 0 of topic `testing`.

```
cat /image/topics/byName/testing/0
PartitionRegistration(replicas=[3], observers=[], isr=[3],
removingReplicas=[], addingReplicas=[], removingObservers=[],
addingObservers=[], leader=3, leaderRecoveryState=RECOVERED,
leaderEpoch=0, partitionEpoch=0, linkedLeaderEpoch=-1,
linkState=NOT_MIRROR)
```

Note: During client startup, it requests cluster metadata from a broker in the `bootstrap.servers` list. The output of the two previous commands reflects a bit of this cluster metadata included in the broker response.

7. Type `exit` to exit the Kafka Metadata Shell.

Cleanup

1. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```

Conclusion

In this lab you have used Kafka command line tools to create a Topic, write and read from this Topic. Finally, you have used the Kafka Metadata Shell tool to access the cluster's metadata.

c. Producing Records with a Null Key

In this Hands-On Exercise, you will create a Topic with **multiple Partitions**, produce records with a `null` key to those Partitions using the default partitioner, and then read it back to observe issues with ordering.

Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Start the Kafka cluster:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Produce to multiple Partitions

1. From within the terminal window create a Topic manually with Kafka's command-line tool, specifying that it should have **two Partitions**:

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic two-p-topic \
    --partitions 2 \
    --replication-factor 1
Created topic two-p-topic.
```

2. You can use the `kafka-topics` tool to describe the details of the topic:

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --describe \
    --topic two-p-topic
Topic: two-p-topic  TopicId: sw5HkmkKQL-Vkbj4fp3Y0Q PartitionCount: 2
ReplicationFactor: 1    Configs:
    Topic: two-p-topic  Partition: 0    Leader: 1   Replicas: 1 Isr:
1  Offline:
    Topic: two-p-topic  Partition: 1    Leader: 2   Replicas: 2 Isr:
2  Offline:
```

Note that Your Partitions might be placed on different Brokers.

3. Use the command-line Producer to write several lines of data to the Topic.

```
$ seq 1 20 |
  kafka-console-producer \
    --bootstrap-server $BOOTSTRAPS \
    --sync --batch-size 1 \
    --topic two-p-topic
```

The `--sync --batch-size 1` options cause the producer to send messages to brokers synchronously, one at a time as they arrive. Using these options in this exercise will result in a better illustration of how the default partitioner assigns records to a partition when they have a `null` key.

4. Use the command-line Consumer to read the messages written to partitions `0` and `1`. Press `Ctrl+C` to exit the Consumer.

```
$ kafka-console-consumer \
    --bootstrap-server \
        $BOOTSTRAPS \
    --from-beginning \
    --partition 0 \
    --topic two-p-topic
1
3
4
5
7
9
10
11
12
13
14
18
```

```
$ kafka-console-consumer \
    --bootstrap-server \
        $BOOTSTRAPS \
    --from-beginning \
    --partition 1 \
    --topic two-p-topic
2
6
8
15
16
17
19
20
```


5. Note the order of the numbers. What do you think is happening as each message is being produced?

Starting with Apache Kafka 3.3.0, the default partitioner for records with `null` keys, as described in KIP-794, tries to balance the amount of records sent to each broker but also

to improve the performance of sending records to one broker each time, which is very different from round-robin (used by Kafka back in the day). The result is that it isn't possible to predict which message arrives where (the results in your exercise may be different from in this guide).

Cleanup

1. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```

Conclusion

You created a Topic with multiple Partitions. You then used the `kafka-console-producer` to write some data to the Topic. Finally, you analyzed the order of the data output by the `kafka-console-consumer` and noticed that the order is not deterministic.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04 Providing Durability

a. Investigating the Distributed Log

In this exercise, you will investigate the distributed log. You will then simulate a Broker failure, and see how to recover the Broker.

Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Make sure your Kafka cluster is started, **otherwise** execute this command:

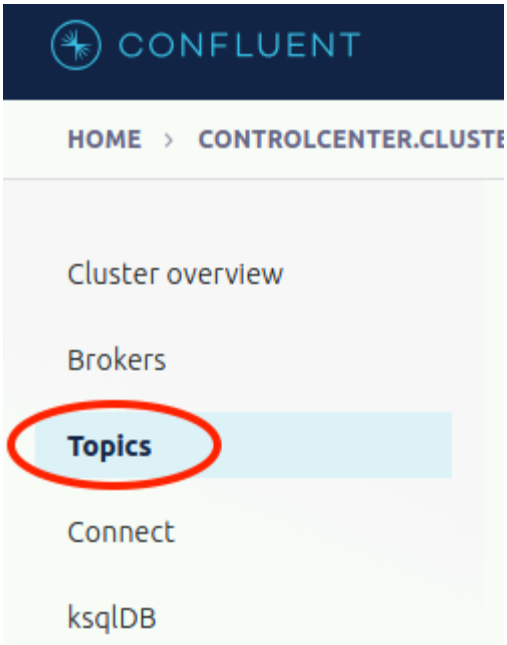
```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Observing the Distributed Log

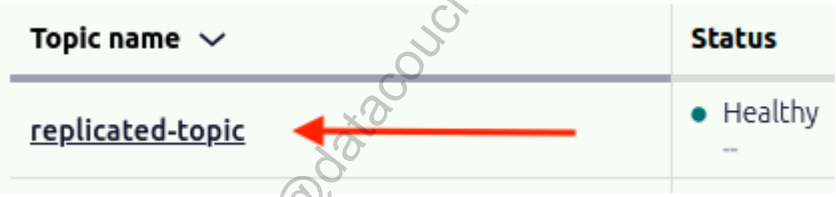
1. Create a new Topic called `replicated-topic` with six Partitions and two replicas:

```
$ kafka-topics \
    --create \
    --bootstrap-server $BOOTSTRAPS \
    --topic replicated-topic \
    --partitions 6 \
    --replication-factor 2
```

2. View the Topic information to see that it has been created correctly. The exact output may vary depending on which Brokers the Topic-Partition replicas were assigned.
   a. Connect to Control Center. If necessary, open a browser tab to the URL http://localhost:9021
   b. In Control Center, click on **Topics**:



c. Scroll through the Topics list until you find `replicated-topic` and click on the topic name.



A detailed view of the topic opens in the right pane with the **Overview** tab selected.

d. Notice the Topic's Partition list and the corresponding Brokers where each Partition replica resides.

Partitions		Replica placement		Replica lag	Offset		Size
Partition id	Status	Leader (broker ID)	Followers (broker IDs)	Followers (max l...	Start	End	Total Size
0	Available	2	3	490s	0	0	0B
1	Available	3	1	490s	0	0	0B
2	Available	1	2	490s	0	0	0B
3	Available	1	2	490s	0	0	0B
4	Available	2	3	490s	0	0	0B
5	Available	3	1	490s	0	0	0B

e. You can also view the same Topic information via Kafka command line tools. In your terminal, describe the Topic:

```
$ kafka-topics \
    --describe \
    --bootstrap-server $BOOTSTRAPS \
    --topic replicated-topic
```

The output should be similar to this:

```
Topic: replicated-topic TopicId: 47vTP8v2T7qOXD_yBF2_bA
PartitionCount: 6   ReplicationFactor: 2    Configs:
    Topic: replicated-topic Partition: 0    Leader: 2   Replicas:
2,3   Isr: 2,3    Offline:
    Topic: replicated-topic Partition: 1    Leader: 3   Replicas:
3,1   Isr: 3,1    Offline:
    Topic: replicated-topic Partition: 2    Leader: 1   Replicas:
1,2   Isr: 1,2    Offline:
    Topic: replicated-topic Partition: 3    Leader: 1   Replicas:
1,2   Isr: 1,2    Offline:
    Topic: replicated-topic Partition: 4    Leader: 2   Replicas:
2,3   Isr: 2,3    Offline:
    Topic: replicated-topic Partition: 5    Leader: 3   Replicas:
3,1   Isr: 3,1    Offline:
```

3. Produce some data to send to the Topic `replicated-topic`. Leave this process running until it finishes. The following command line sends 6000 messages, 100 bytes in size, at a rate of 1000 messages/sec:


```
$ kafka-producer-perf-test \
    --topic replicated-topic \
    --num-records 6000 \
    --record-size 100 \
    --throughput 1000 \
    --producer-props bootstrap.servers=$BOOTSTRAPS
```

The output should look similar to this:

```
5001 records sent, 1000.2 records/sec (0.10 MB/sec), 23.0 ms avg
latency, 905.0 ms max latency.
6000 records sent, 996.843329 records/sec (0.10 MB/sec), 20.16 ms avg
latency, 905.00 ms max latency, 2 ms 50th, 88 ms 95th, 123 ms 99th,
133 ms 99.9th.
```

4. Start the console Producer for the same Topic `replicated-topic`:

```
$ kafka-console-producer \
    --bootstrap-server $BOOTSTRAPS \
    --topic replicated-topic
```

5. At the `>` prompt, type "I heart logs" and press Enter. Add five more messages and then press `Ctrl+D` to exit the console Producer:

```
> I heart logs
> Hello world
> All your base
> Kafka rules
> Don't worry
> Be happy
<Ctrl+D>
```

Examine Checkpoint Files

For each Broker (`kafka-1`, `kafka-2` and `kafka3`) examine the contents of the checkpoint files `recovery-point-offset-checkpoint` and `replication-offset-checkpoint` at `/var/lib/kafka/data`. Let's start with the first Broker called `kafka-1`:

1. The next steps will be run on the Broker. From your host system, open a new terminal and connect to the Broker, here `kafka-1`:

```
$ cd ~/confluent-admin
$ docker-compose exec kafka-1 /bin/bash
[appuser@kafka-1 ~]$
```

2. The `recovery-point-offset-checkpoint` file records the point up to which data has been flushed to disk. This is important as, on hard failure, the Broker needs to scan unflushed data, verify the CRC, and truncate the corrupted log.

```
[appuser@kafka-1 ~]$ grep replicated-topic \
/var/lib/kafka/data/recovery-point-offset-checkpoint
replicated-topic 1 0
replicated-topic 2 0
replicated-topic 3 0
replicated-topic 5 0
```

In the output above, the first number is the Partition number, and the second number is the offset of the last flushing point. In the output above, the offset of the last flushing

point is expected to be 0: the active segment has not been flushed yet because it is still open in read+append mode.

3. The `replication-offset-checkpoint` file is the offset of the last committed offset (e.g. the **high water mark**).

```
[appuser@kafka-1 ~]$ grep replicated-topic \
/var/lib/kafka/data/replication-offset-checkpoint
replicated-topic 3 922
replicated-topic 1 1077
replicated-topic 2 1078
replicated-topic 4 1133
```

In the output above, the first number is the Partition number, and the second number is the offset of the high water mark. The offset of the high water mark is expected to be different on each partition because the number of produced messages, 6006 (6000 + 6) was not evenly distributed across the six Partitions due to the Strictly Uniform Sticky Partitioner.

Your high water marks will likely be slightly different from those shown in the above example.

4. Disconnect from the Broker container by typing `exit`.
5. Repeat the previous steps for the other two Brokers `kafka-2` and `kafka-3`.

Examining Topic Partitions

On each of the Brokers, examine the contents of one of the data directories for at least one of the Topic-Partitions. For example, the directory `replicated-topic-0` has log files for Partition 0 of Topic `replicated-topic`. The following steps will list the log files for Partition 0 on the `kafka-1` Broker.

1. From your host system connect to the `kafka-1` Broker container:

```
$ cd ~/confluent-admin
$ docker-compose exec kafka-1 /bin/bash
[appuser@kafka-1 ~]$
```

2. List the directory **replicated-topic-0**:

```
[appuser@kafka-1 ~]$ ls -l /var/lib/kafka/data/replicated-topic-0
total 96
-rw-r--r-- 1 appuser appuser 10485760 Jan 17 18:00
00000000000000000000.index
-rw-r--r-- 1 appuser appuser    82273 Jan 17 18:00
00000000000000000000.log
-rw-r--r-- 1 appuser appuser 10485756 Jan 17 18:00
00000000000000000000.timeindex
-rw-r--r-- 1 appuser appuser        0 Jan 17 17:47 leader-epoch-
checkpoint
-rw-r--r-- 1 appuser appuser       43 Jan 17 17:47 partition.metadata
```

The command may return the following error:

```
ls: cannot access /var/lib/kafka/data/replicated-topic-0:
No such file or directory
```

Why would this error occur?

The directory **replicated-topic-0** does not exist on all three Brokers in your cluster.

Why not?

Remember that a Broker may not store every Partition for a Topic. As shown in the Partition lists earlier, each Broker only contains a subset of the Partitions for **replicated-topic**. If you get an error when trying to list the **replicated-topic-0** directory, the Partition is on another Broker.

If this error occurs, repeat the command using **replicated-topic-1**.

The data files have the following meaning:

.log	holds the messages and their metadata

`.index`	maps message offsets to their byte position in the log file
`.timeindex`	maps message timestamps to their offset number
`leader-epoch-checkpoint`	maps the leader epoch to its corresponding start offset
`partition.metadata`	internal information about the partition itself

3. On the Brokers, view the contents of the file `leader-epoch-checkpoint`:

```
[appuser@kafka-1 ~]$ cat \
/var/lib/kafka/data/replicated-topic-0/leader-epoch-checkpoint
```


This file tracks the lineage of the Partition leadership by noting which offset starts a new leader generation. The first row denotes the version of the file format, in this case `0`. The second row denotes the number of epochs, which is incremented every leader election. This Partition has only had one leader in its history, so it is still in its first epoch. Each row afterwards shows the epoch ID followed by the offset where the epoch begins. In this case, the current epoch ID is `0` and starts at offset `0`.

```
0
1
0 0
```

4. On the Broker, use the `kafka-dump-log` tool to view details of the `.log` file:

```
[appuser@kafka-1 ~]$ kafka-dump-log \
    --print-data-log \
    --files \
/var/lib/kafka/data/replicated-topic-0/00000000000000000000.log
```

The command output should be similar to (shortened):

```
Dumping /var/lib/kafka/data/replicated-topic-
1/00000000000000000000.log
Log starting offset: 0
baseOffset: 0 lastOffset: 147 count: 148 baseSequence: 0
lastSequence: 147 producerId: 1006 producerEpoch: 0
partitionLeaderEpoch: 0 isTransactional: false isControl: false
deleteHorizonMs: OptionalLong.empty position: 0 CreateTime:
1705514443122 size: 16277 magic: 2 compresscodec: none crc: 71692478
isvalid: true
| offset: 0 CreateTime: 1705514443118 keySize: -1 valueSize: 100
sequence: 0 headerKeys: [] payload:
HHUMDRHTNLSHIJVSGZIKNEQKXWQNDINJYSPHVWESORDCFFWSQYJMXCFFHMVYOJUEDQIAA
VZIEMRCNGZXGMBWPYPSXDXUIASBSEGC
| offset: 1 CreateTime: 1705514443118 keySize: -1 valueSize: 100
sequence: 1 headerKeys: [] payload:
FBYXMZSEUZLHDAUIUARWHVOCWZSRZOWBTPNEXWOWHUFTXJHAKJXVHTGNCYIQRGSMDHEXY
LOEWDBHKLXPZNWBXOWDEGHSVFRERGKU
| offset: 2 CreateTime: 1705514443118 keySize: -1 valueSize: 100
sequence: 2 headerKeys: [] payload:
GZQFBHYAZIUHUIXQPRZNFXCVHYMTKGKUZGUCDQCRTXYIWALPQZNMILBALVEQYPBHOUDYK
NTNUOPBUKSZFIVROVHZLGXHUYFNHVPA
...
baseOffset: 655 lastOffset: 656 count: 2 baseSequence: 655
lastSequence: 656 producerId: 1006 producerEpoch: 0
partitionLeaderEpoch: 0 isTransactional: false isControl: false
deleteHorizonMs: OptionalLong.empty position: 81994 CreateTime:
1705514447213 size: 279 magic: 2 compresscodec: none crc: 1650004841
isvalid: true
| offset: 655 CreateTime: 1705514447213 keySize: -1 valueSize: 100
sequence: 655 headerKeys: [] payload:
HYAPLWQJSRKDUQQTEEKVHCDGPNIUAHQHGRBLUHLUPJOWGFJCEMRYGGRVTZHVLRWDVZEGB
CDNOYWJLVIYKQISLXESZBXWDLVNLYZG
| offset: 656 CreateTime: 1705514447213 keySize: -1 valueSize: 100
sequence: 656 headerKeys: [] payload:
NOOSHZINGLJBUZGAIKYCPNPNMOKVCGXSKMCSSRYJETKKJZSVDEPXBLRTWMUBVNXQTRRZM
THMGYEJCRLANYQOKCSZPYNLMRZTBOIL
```


5. **Questions:**
   a. Are the offsets in the `.log` file sequential?
   b. Can you work out what is the offset of the message "All your base" that you entered earlier? If you can't find the value in the directory `replicated-topic-0`, where else might it be?
6. On the Broker, use the `DumpLogSegments` tool to view the offsets in the `.index` file:

```
[appuser@kafka-1 ~]$ kafka-run-class kafka.tools.DumpLogSegments \
--files \
/var/lib/kafka/data/replicated-topic-0/00000000000000000000.index

Dumping /var/lib/kafka/data/replicated-topic-
0/00000000000000000000.index
offset: 32 position: 4172
offset: 64 position: 8466
offset: 94 position: 12638
...
...
offset: 807 position: 109936
offset: 837 position: 114060
offset: 867 position: 118245
```

Note the difference from one index entry position to the next is typically ~4000 which corresponds to the default `log.index.interval.bytes=4096` setting. In cases where the difference is larger, the cause is batching.

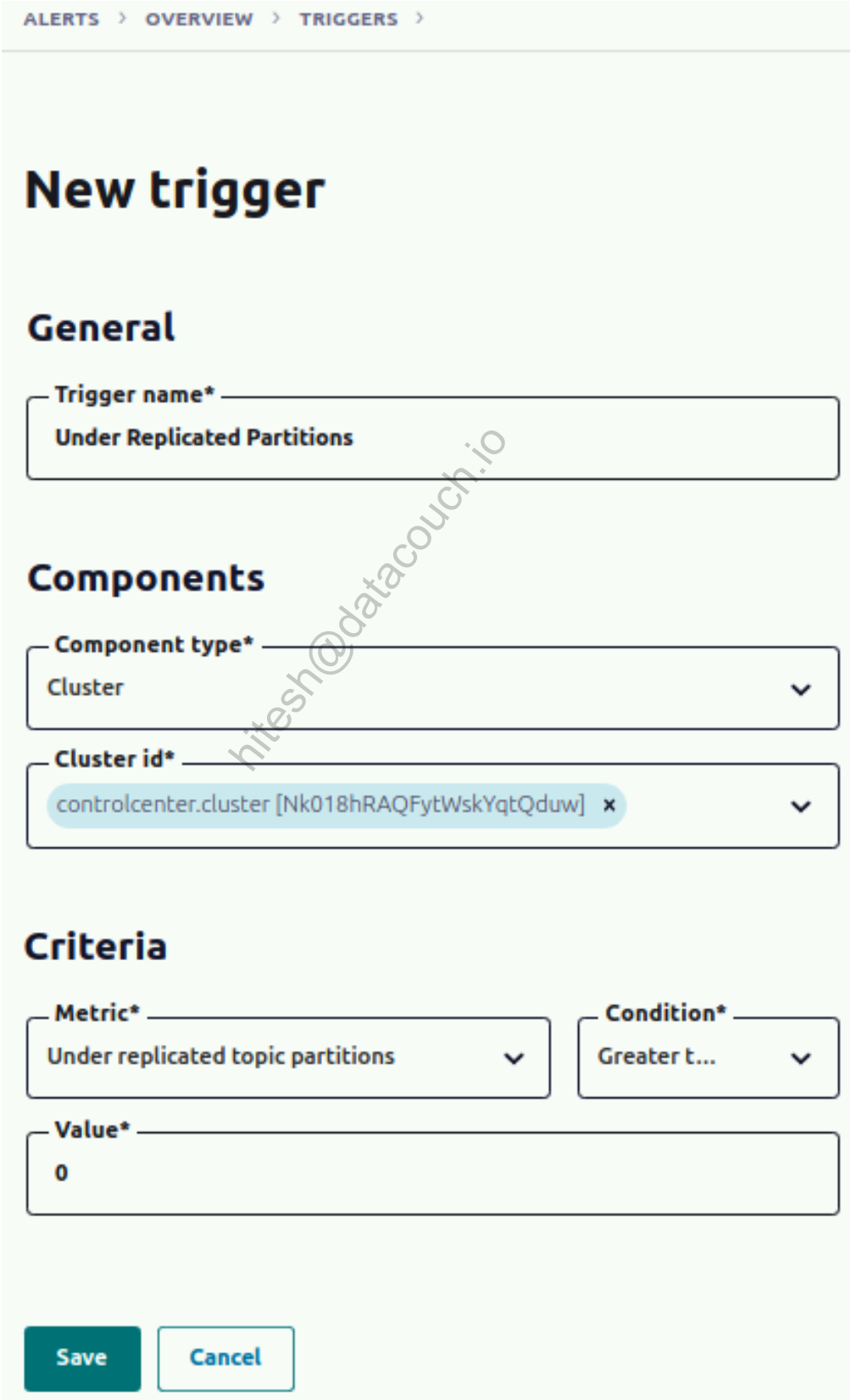
7. Disconnect from the Broker (container) by typing `exit`.
8. Repeat the above steps for the other two Brokers `kafka-2` and `kafka-3`.

Taking a Broker Offline

1. First let's define an **action** in Control Center for when an under replication happens. We will see later why that is important:
   a. Open Control Center
   b. Click the bell icon in the top right corner to navigate to **Alerts**
   c. Click the button **Create trigger**
   d. Complete the dialog using the following settings:

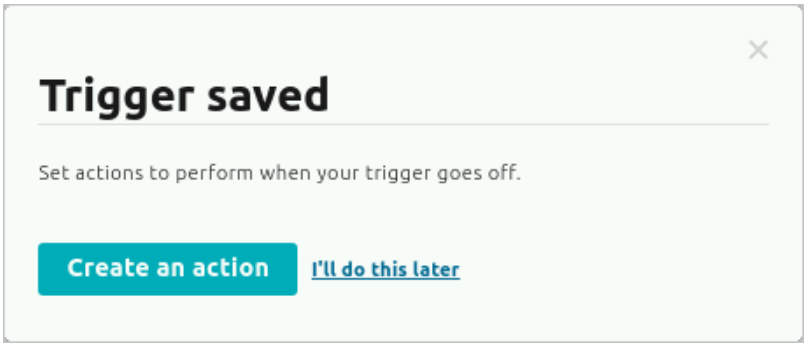
Field	Value
Trigger Name	Under Replicated Partitions
Component type	Cluster
Cluster id	controlcenter.cluster ...
Metric	Under replicated topic partitions

Field	Value
Condition	Greater than
Value	0



e. Click **Save**

f. In the confirmation box, when asked to "Set actions to perform when your trigger goes off", select **Create an action**



g. Complete the dialog using the following settings:

Field	Value
Action Name	My action
Triggers	Under Replicated Partitions
Actions	Send email
Subject	Under replicated partitions
Max send rate	1 Per minute
Recipient email address	anything@anything.com

ALERTS › OVERVIEW › ACTIONS ›

New action

General

Action Name*

My action

Enabled

Triggers

Triggers*

Under Replicated Partitions ×

Actions

Action*

Send email

Subject*

Under replicated partitions

Max send rate*

1

Per minute

Recipient email address*

anything@anything.com

Save Cancel

h. Click **Save**

2. Before taking a broker offline, capture its name in an environment variable, with the following command (you may choose `kafka-2` or `kafka-3` instead:

```
$ export OFFLINE_BROKER=kafka-1
```

3. Let's stop the Broker now. From your host system execute:

```
$ cd ~/confluent-admin
$ docker-compose stop $OFFLINE_BROKER
[+] Running 1/1
 ⠿ Container kafka-1  Stopped
```

4. Verify that the other Brokers are running:

   a. From your host system use the following command:

```
$ docker-compose ps
NAME                COMMAND                  SERVICE
STATUS              PORTS
control-center      "/etc/confluent/dock…"   control-center
running             0.0.0.0:9021->9021/tcp, :::9021->9021/tcp
controller-1        "/etc/confluent/dock…"   controller-1
running             0.0.0.0:19093->19093/tcp, :::19093->19093/tcp
controller-2        "/etc/confluent/dock…"   controller-2
running             0.0.0.0:29093->29093/tcp, :::29093->29093/tcp
controller-3        "/etc/confluent/dock…"   controller-3
running             0.0.0.0:39093->39093/tcp, :::39093->39093/tcp
kafka-1             "/etc/confluent/dock…"   kafka-1
exited (143)
kafka-2             "/etc/confluent/dock…"   kafka-2
running (healthy)   0.0.0.0:10002->10002/tcp, 0.0.0.0:29092-
>29092/tcp, :::10002->10002/tcp, :::29092->29092/tcp
kafka-3             "/etc/confluent/dock…"   kafka-3
running (healthy)   0.0.0.0:10003->10003/tcp, 0.0.0.0:39092-
>39092/tcp, :::10003->10003/tcp, :::39092->39092/tcp
kafka-connect       "/etc/confluent/dock…"   kafka-connect
running (healthy)   0.0.0.0:8083->8083/tcp, :::8083->8083/tcp
schema-registry     "/etc/confluent/dock…"   schema-registry
running (healthy)   0.0.0.0:8081->8081/tcp, :::8081->8081/tcp
```

b. Make sure that the **Status** of the other Brokers marked as **runnning** and stopped Broker as **exited (143)**.

c. From a terminal window use **kafka-metadata-quorum** command-line tool to see which Brokers are registered:

```
$ kafka-metadata-quorum --bootstrap-server $BOOTSTRAPS describe
--status
ClusterId:                Nk018hRAQFytWskYqtQduw
LeaderId:                 9991
LeaderEpoch:              2
HighWatermark:            17339
MaxFollowerLag:           0
MaxFollowerLagTimeMs:     0
CurrentVoters:            [9991,9992,9993]
CurrentObservers:         [2147483647,2,3]
```

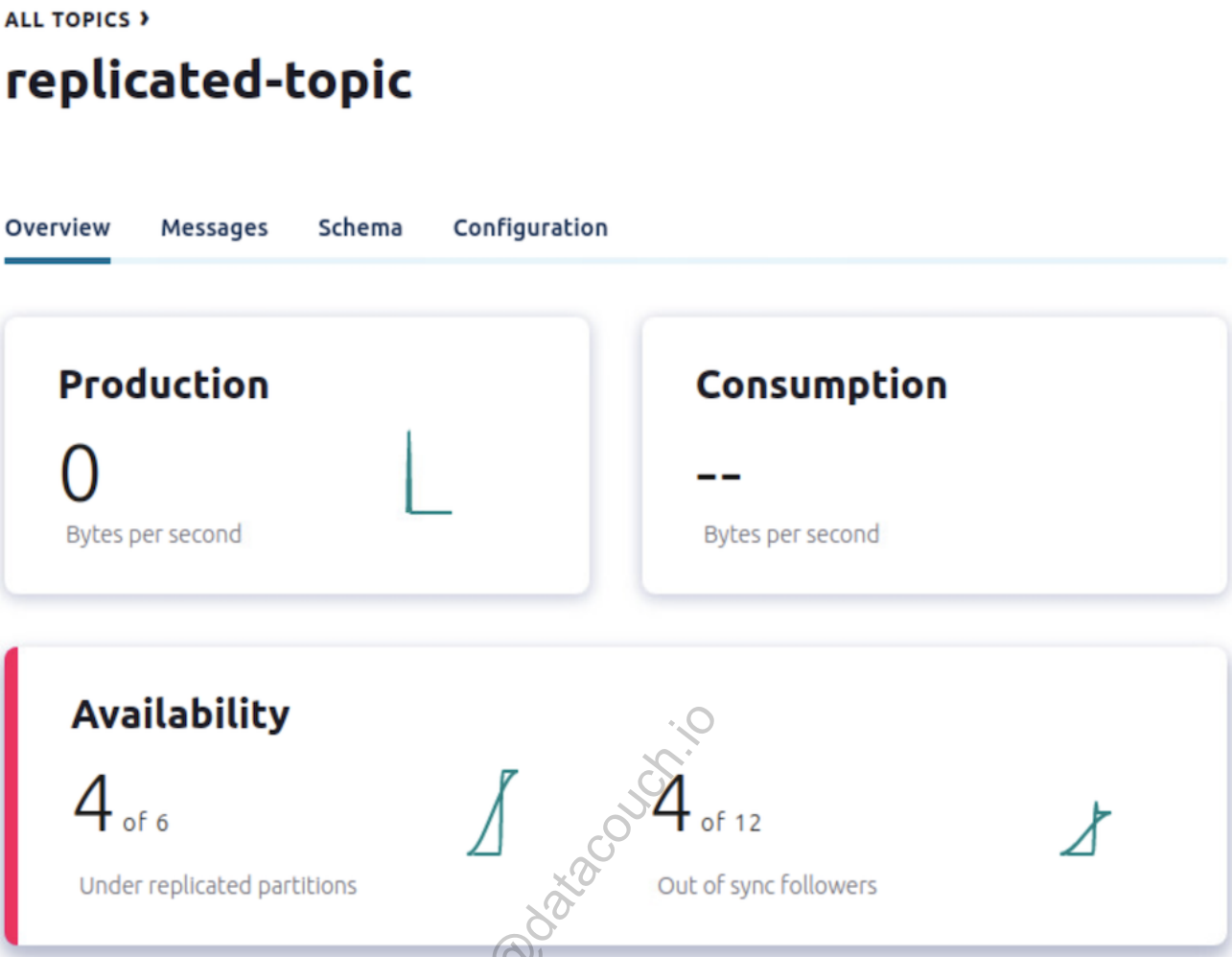
The command outputs the list of current brokers in **CurrentObservers**: in our example, it shows **2** and **3** (plus a "mock" value for the missing **1**).

5. From the host system, review the server log for the offline Broker. Look for confirmation of the controlled shutdown succeeding:

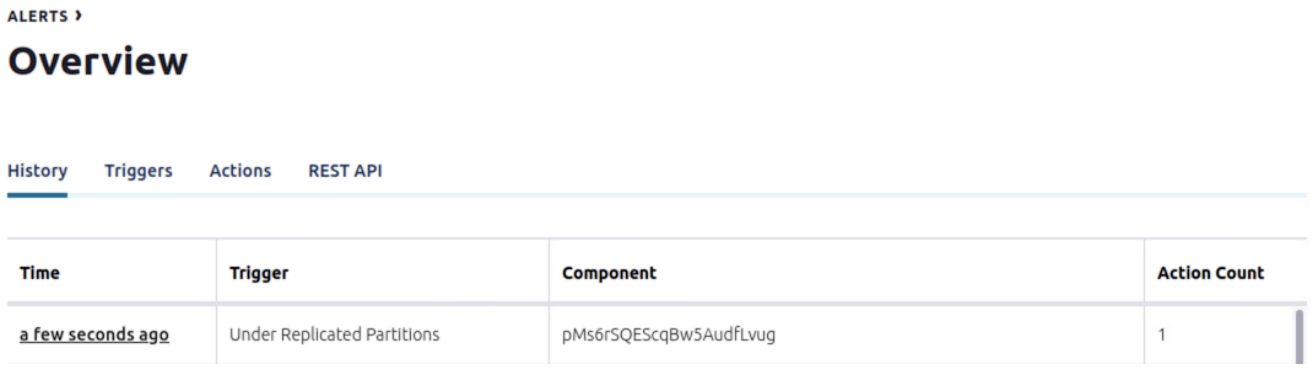
```
$ docker-compose logs $OFFLINE_BROKER | grep -i shut

...
kafka-1  | [2024-01-17 19:42:32,316] INFO [RaftManager id=1]
Beginning graceful shutdown (org.apache.kafka.raft.KafkaRaftClient)
kafka-1  | [2024-01-17 19:42:32,317] INFO [RaftManager id=1] Graceful
shutdown completed (org.apache.kafka.raft.KafkaRaftClient)
kafka-1  | [2024-01-17 19:42:32,317] INFO [kafka-1-raft-io-thread]:
Completed graceful shutdown of RaftClient
(kafka.raft.KafkaRaftManager$RaftIoThread)
kafka-1  | [2024-01-17 19:42:32,317] INFO [kafka-1-raft-io-thread]:
Shutdown completed (kafka.raft.KafkaRaftManager$RaftIoThread)
kafka-1  | [2024-01-17 19:42:32,326] INFO [kafka-1-raft-outbound-
request-thread]: Shutting down (kafka.raft.RaftSendThread)
kafka-1  | [2024-01-17 19:42:32,326] INFO [kafka-1-raft-outbound-
request-thread]: Shutdown completed (kafka.raft.RaftSendThread)
kafka-1  | [2024-01-17 19:42:32,385] INFO [BrokerServer id=1] shut
down completed (kafka.server.BrokerServer)
kafka-1  | [2024-01-17 19:42:32,385] INFO [BrokerServer id=1]
Transition from SHUTTING_DOWN to SHUTDOWN (kafka.server.BrokerServer)
```


All logs produced by the Brokers running inside a container are written to STDOUT and STDERR and reflected in the Docker logs. On a dedicated Broker machine, this server log data will be located at `/var/log/kafka/server.log`.

6. Wait up to five minutes and then observe the cluster in Control Center.
   a. In Control Center click **Cluster 1**.
   b. In the **Broker** panel of **Overview**, observe the **Total** count decrease to **2**:
   c. In the **Topics** panel of **Overview**, observe the **Under replicated partitions** count (should be non-zero).
   d. Click on **Topics**.
   e. Scroll through the topic list to locate `replicated-topic` and click the topic name.
   f. On the **Overview** tab, notice there are now **Under replicated partitions**:



g. Click the bell icon in the top right corner to view the alert history and notice that the broker down event triggered an alert. It is critical to monitor under replicated partitions to ensure message durability in your Kafka cluster:



h. Start the console Producer for the same Topic `replicated-topic`:

```
$ kafka-console-producer \
    --bootstrap-server $BOOTSTRAPS \
    --topic replicated-topic
```

Clients send the cluster metadata request to `bootstrap-server` brokers in a random order. You may see a warning message if the initial metadata request is sent to a broker that is not running.

i. At the `>` prompt, type six more messages and then press `Ctrl+D` to exit the console Producer:

```
> Kafka
> Distributed
> Secure
> Real-time
> Scalable
> Fast
<Ctrl+D>
```


7. From your host system view the impact to leader epoch:

   a. First run `kafka-topics` again to identify which `replicated-topic` partitions had a preferred replica that was the offline broker. It would have been the leader replica when the topic was created. Since we shut this broker down, leader election would have occurred and these partitions would have been assigned a new leader:

```
$ kafka-topics \
    --describe \
    --bootstrap-server $BOOTSTRAPS \
    --topic replicated-topic
```

   The output should be similar to this:

```
Topic: replicated-topic TopicId: 47vTP8v2T7qOXD_yBF2_bA
PartitionCount: 6   ReplicationFactor: 2    Configs:
    Topic: replicated-topic Partition: 0    Leader: 2   Replicas:
2,3   Isr: 2,3    Offline:
    Topic: replicated-topic Partition: 1    Leader: 3   Replicas:
3,1   Isr: 3  Offline: 1
    Topic: replicated-topic Partition: 2    Leader: 2   Replicas:
1,2   Isr: 2  Offline: 1
    Topic: replicated-topic Partition: 3    Leader: 2   Replicas:
1,2   Isr: 2  Offline: 1
    Topic: replicated-topic Partition: 4    Leader: 2   Replicas:
2,3   Isr: 2,3    Offline:
    Topic: replicated-topic Partition: 5    Leader: 3   Replicas:
3,1   Isr: 3  Offline: 1
```

The offline broker was the original leader for partitions in which it appears first in the **`Replicas`** list (preferred replica). In our example, they would be partitions 2 and 3.

b. On either of the other two Brokers, view the contents of the file **`leader-epoch-checkpoint`** in one of the Topic-Partitions subdirectory for which the offline was the previous leader:

```
$ docker-compose exec kafka-2 cat \
/var/lib/kafka/data/replicated-topic-2/leader-epoch-checkpoint
0
2
0 0
1 1077
```


This Partition now reflects that it has had two leaders. Leader epoch **`0`** had an initial offset of **`0`**. Leader epoch **`1`** has an initial offset of **`1077`**.

- Your leader epoch **`1`** may have a slightly different initial offset.
- If you see the following response to the above command, it indicates that no leader election has occurred for that partition.

```
0
1
0 0
```

Bringing a Broker Online

Bringing a Broker online is as simple as restarting the Broker and letting Kafka automatically rebalance the leaders.

1. From your host system restart the initial Controller Broker:

```
$ docker-compose start $OFFLINE_BROKER
```

2. Wait five minutes and then observe the cluster in Control Center:
   a. In Control Center, go to the Home page and select our cluster.
   b. In the **Broker** panel of **Overview**, observe the **Total** count as it returns to **3**.
   c. In the **Topics** panel of **Overview**, observe the **Under replicated partitions** count as it returns to 0.
   d. Click on **Topics**.
   e. Scroll through the topic list to locate `replicated-topic` and click the topic name.
   f. On the **Overview** tab, notice the **Under replicated partitions** has returned to 0.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 05 Configuring a Kafka Cluster

a. Exploring Configuration

In this exercise, you will research a few important Broker properties of a Kafka cluster.

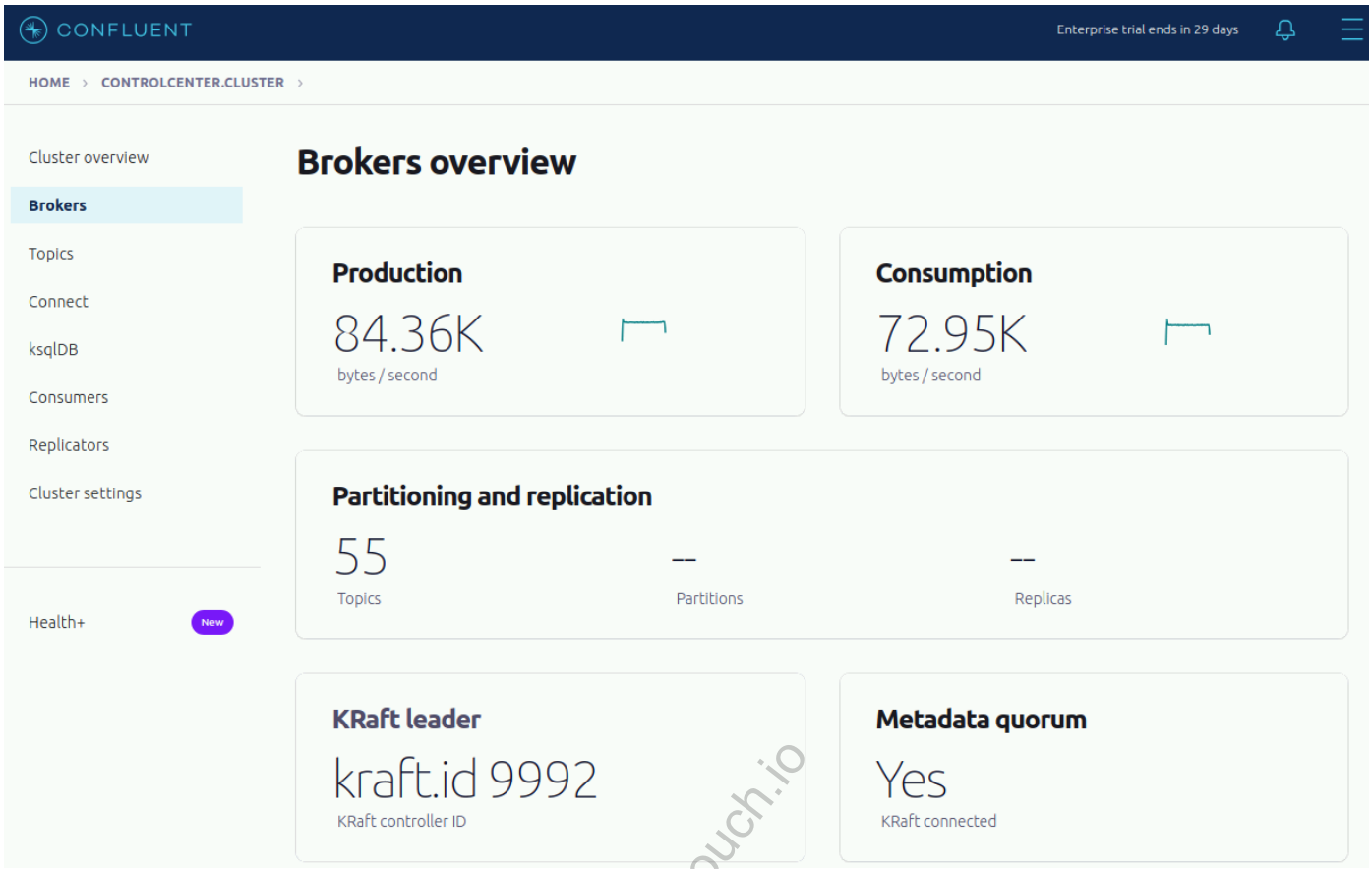
Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Make sure your Kafka cluster is started, **otherwise** execute this command:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```

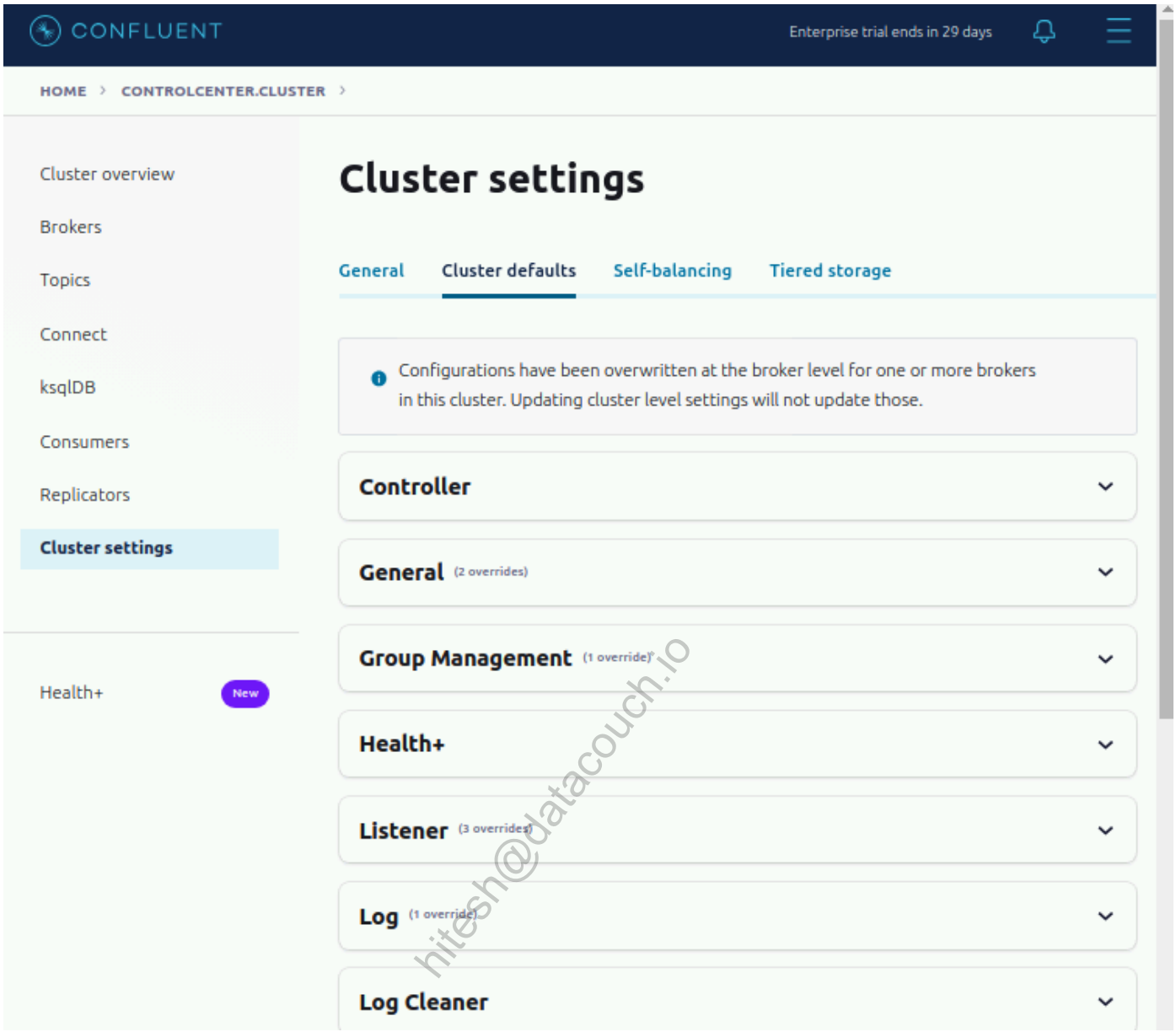

Configuration Jigsaw

1. Connect to Control Center. If necessary, open a browser tab to the URL **http://localhost:9021**.
2. Select the cluster on the home page, then click **Brokers** on the left side, so you will see the **Brokers overview** view:



Observe the metrics you see on this view. Do they correspond to what you'd expect to view at a first glance? Why yes? Why not?

3. On the left side select **Cluster settings** and then select the tab **Cluster defaults**:



4. Spend 5 minutes looking at the different categories of properties, and perhaps read a few of the descriptions provided (the description pops up when hovering your mouse over a property). There are many configuration settings, so don't get too focused on one property in particular.
    - What do you notice? Write down a few observations.
    - What do you wonder? Write down a few questions that arise.

1. Here are some sets of configuration properties. Choose **one** set of properties and take 15 minutes to research what they do, what some reasonable values are, and what some unreasonable (but valid) values might be.
    - Listener:

        - `listeners`
        - `inter.broker.listener.name`
        - `advertised.listeners`
    - Log (*i.e.* cluster default properties for Topics, Partitions, and replicas):
        - `log.dirs`
        - `default.replication.factor`
        - `log.retention.hours`
        - `log.segment.bytes`
        - `log.roll.ms`
        - `num.partitions`
    - Socket Server
        - `connections.max.idle.ms`
        - `max.connections.per.ip`
        - `max.connections.per.ip.overrides`
    - Threads:
        - `num.recovery.threads.per.data.dir`
        - `num.replica.fetchers`
        - `num.network.threads`
        - `num.io.threads`

This is just a first exposure. Many of these properties will be explored in more detail in upcoming material, so don't worry too much if you still have questions at this point. Record your questions and come back to them later towards the end of training to make sure they get answered.

STOP HERE. THIS IS THE END OF THE EXERCISE.

b. Increasing Replication Factor

We all make mistakes. Maybe you accidentally created a mission-critical Topic with a replication factor of 1. Luckily, you notice this before disaster strikes. In this exercise, you will use the `kafka-reassign-partitions` tool to increase the replication factor of a Topic. The `kafka-reassign-partitions` tool is an included utility that is usually used to rebalance the load of Partitions across Brokers. It will be discussed in further detail later in the course.

Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Make sure your Kafka cluster is started, **otherwise** execute this command:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Increasing Replication Factor

1. Create a Topic called `test` with 3 Partitions and replication factor 1.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic test \
    --partitions 3 \
    --replication-factor 1
```

2. View the Topic information. Partitions may land on different Brokers than shown here.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --describe \
    --topic test
Topic: test TopicId: Zs79KLDEToGmy6e9_VbHuw PartitionCount: 3
ReplicationFactor: 1    Configs:
    Topic: test Partition: 0    Leader: 3   Replicas: 3 Isr: 3
Offline:
    Topic: test Partition: 1    Leader: 1   Replicas: 1 Isr: 1
Offline:
    Topic: test Partition: 2    Leader: 2   Replicas: 2 Isr: 2
Offline:
```

3. Create a **json** file called **replicate_topic_test_plan.json** that declares your desired state of Partition replication. Notice that **kafka-reassign-partitions** allows you to set replication factor on a per-Partition basis.

```
$ cat << EOF > replicate_topic_test_plan.json
{"version":1,
          "partitions":[
               {"topic":"test","partition":0,"replicas":[1,2,3]},
               {"topic":"test","partition":1,"replicas":[1,2,3]},
               {"topic":"test","partition":2,"replicas":[2,3]}]
}
EOF
```


Notice that we declare Partition 2 to only have 2 replicas. This shows the granular control of the **kafka-reassign-partitions** tool. Also note the first of each list will become the preferred replica.

4. Use the **--reassignment-json-file** and **--execute** options of the **kafka-reassign-partitions** tool to execute the change.

```
$ kafka-reassign-partitions \
     --bootstrap-server $BOOTSTRAPS \
     --reassignment-json-file replicate_topic_test_plan.json \
     --execute

Current partition replica assignment

{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[
3],"log_dirs":["any"]},{"topic":"test","partition":1,"replicas":[1],"
log_dirs":["any"]},{"topic":"test","partition":2,"replicas":[2],"log_
dirs":["any"]}]}

Save this to use as the --reassignment-json-file option during
rollback
Successfully started partition reassignments for test-0,test-1,test-2
```

5. View the Topic information again to see that the Partitions are now replicated.

```
$ kafka-topics \
    --bootstrap-server kafka-1:19092,kafka-2:29092,kafka-3:39092 \
    --describe \
    --topic test

Topic: test TopicId: Zs79KLDEToGmy6e9_VbHuw PartitionCount: 3
ReplicationFactor: 3    Configs:
    Topic: test Partition: 0    Leader: 3   Replicas: 1,2,3 Isr:
3,1,2  Offline:
    Topic: test Partition: 1    Leader: 1   Replicas: 1,2,3 Isr:
1,2,3  Offline:
    Topic: test Partition: 2    Leader: 2   Replicas: 2,3   Isr: 2,3
Offline:
```


STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 06 Managing a Kafka Cluster

a. Kafka Administrative Tools

In this exercise, you will delete a Topic, reassign Partitions, and simulate a completely failed Broker.

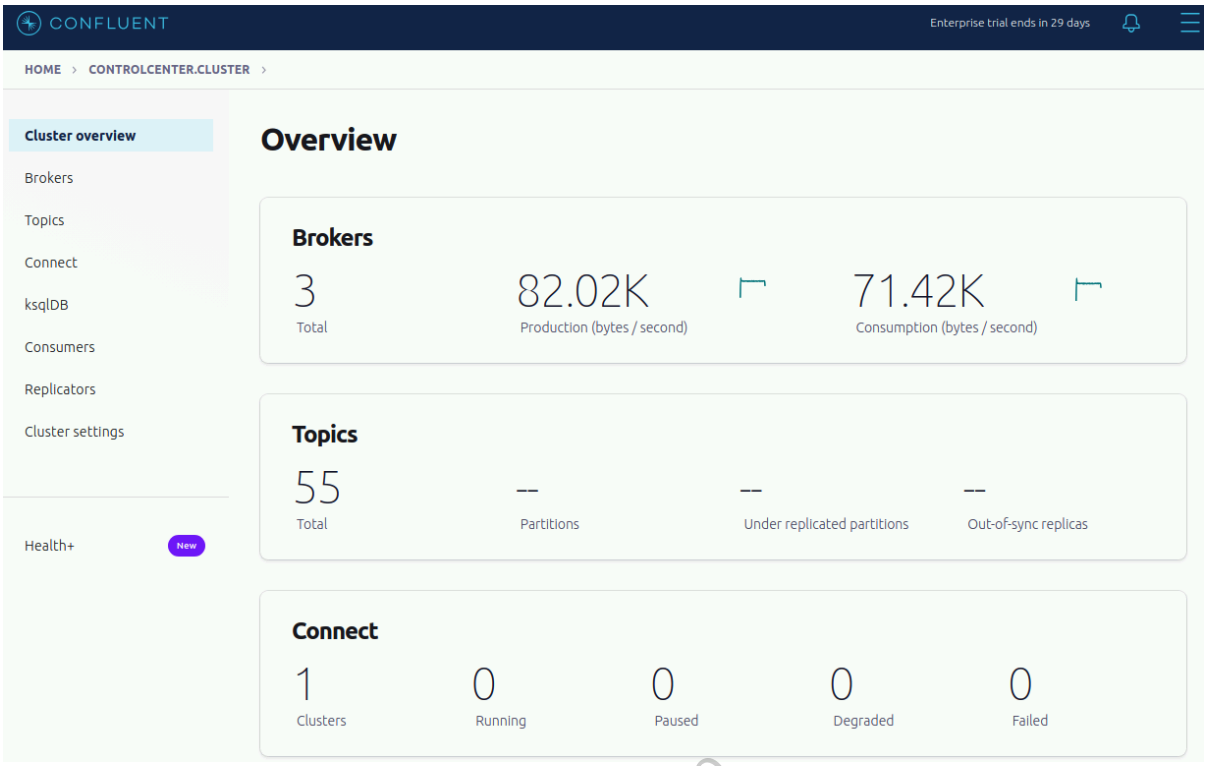
Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Make sure your Kafka cluster is started, **otherwise** execute this command:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Deleting Topics in the Cluster

1. Connect to Control Center. If necessary, open a browser tab to the URL **http://localhost:9021**.
2. Verify that three Brokers are running. In the Control Center **Overview** view, observe the Broker count is three.



3. Delete the Topic `replicated-topic`:

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --delete \
    --topic replicated-topic
```


When deleting a Topic, always make sure you don't have any Producers or Consumers running. Otherwise, the Topic will be re-created automatically. Also, you may have already deleted this Topic when destroying the cluster with `docker-compose down -v`. If so, create the Topic again, verify it exists, and then delete it.

4. In the Control Center, check that the Topic `replicated-topic` is now gone from the list of topics in the Topic Management view.

The Topic may not disappear immediately. If necessary wait a few minutes.

Rebalancing the Cluster

1. Create a new Topic called `moving` with 6 Partitions and 2 replicas, on only Broker 1 and Broker 2 (with IDs `101` and `102`):

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic moving \
    --replica-assignment 1:2,2:1,1:2,2:1,1:2,2:1
```

2. Run the command line tool `kafka-producer-perf-test` to produce 2GB of data to Topic `moving`.

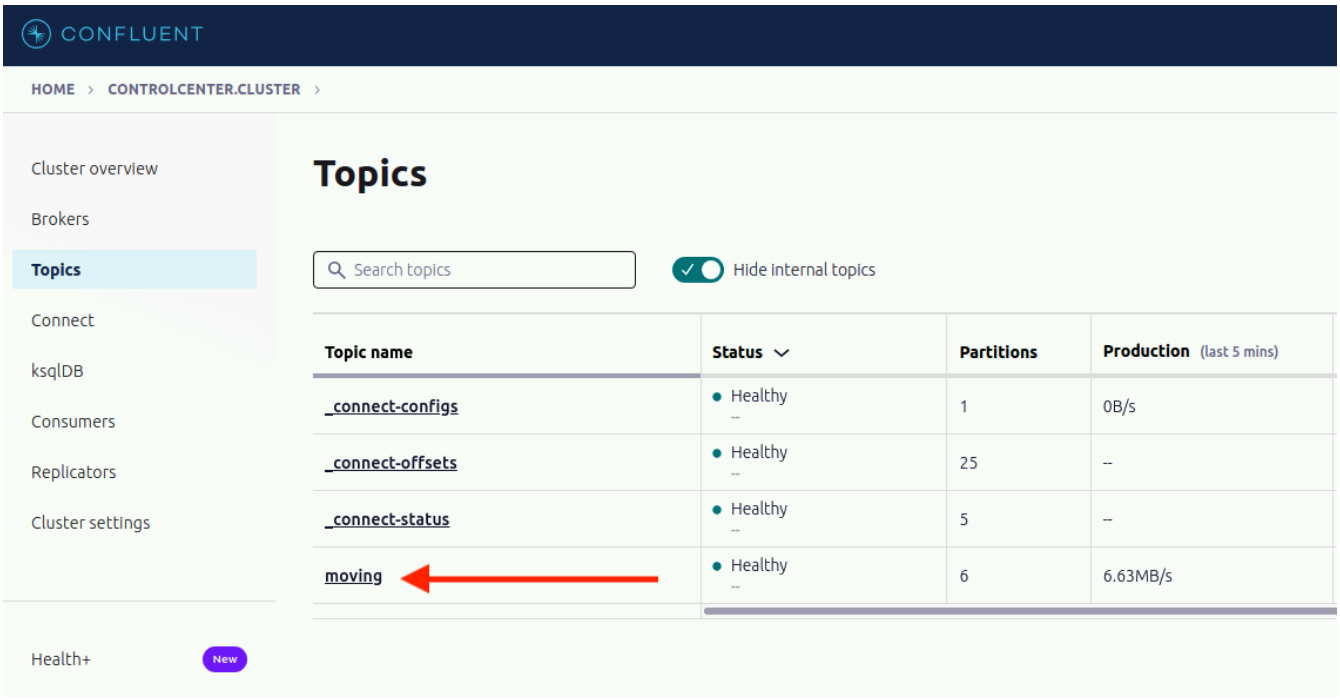
Wait until this command has completed before continuing.

```
$ kafka-producer-perf-test \
    --topic moving \
    --num-records 2000000 \
    --record-size 1000 \
    --throughput 1000000000 \
    --producer-props bootstrap.servers=$BOOTSTRAPS
```

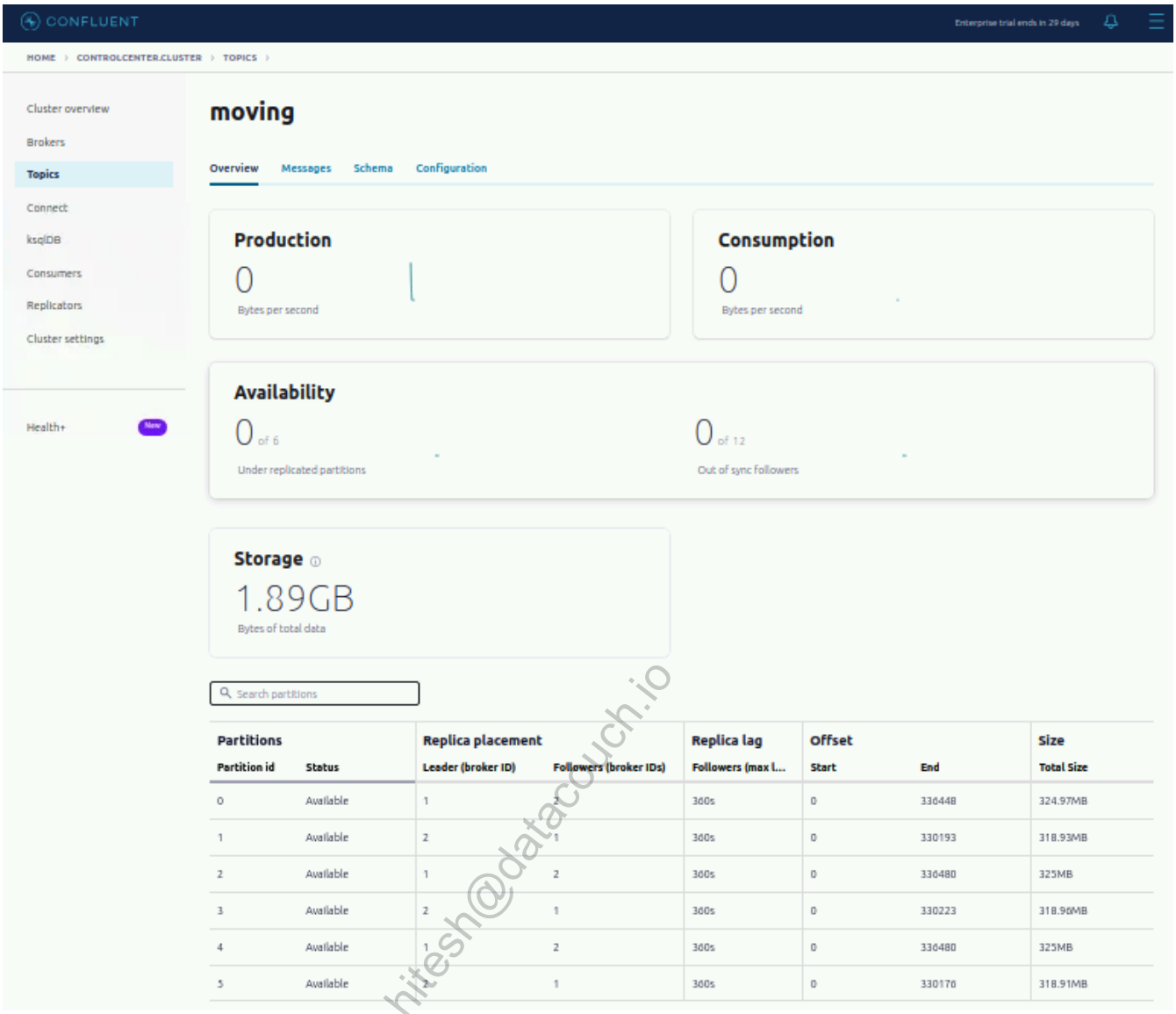

3. Verify which Brokers contain Partitions for the Topic `moving`:

   a. In the Control Center go back to cluster overview*

   b. Then select the tab **Topics**:

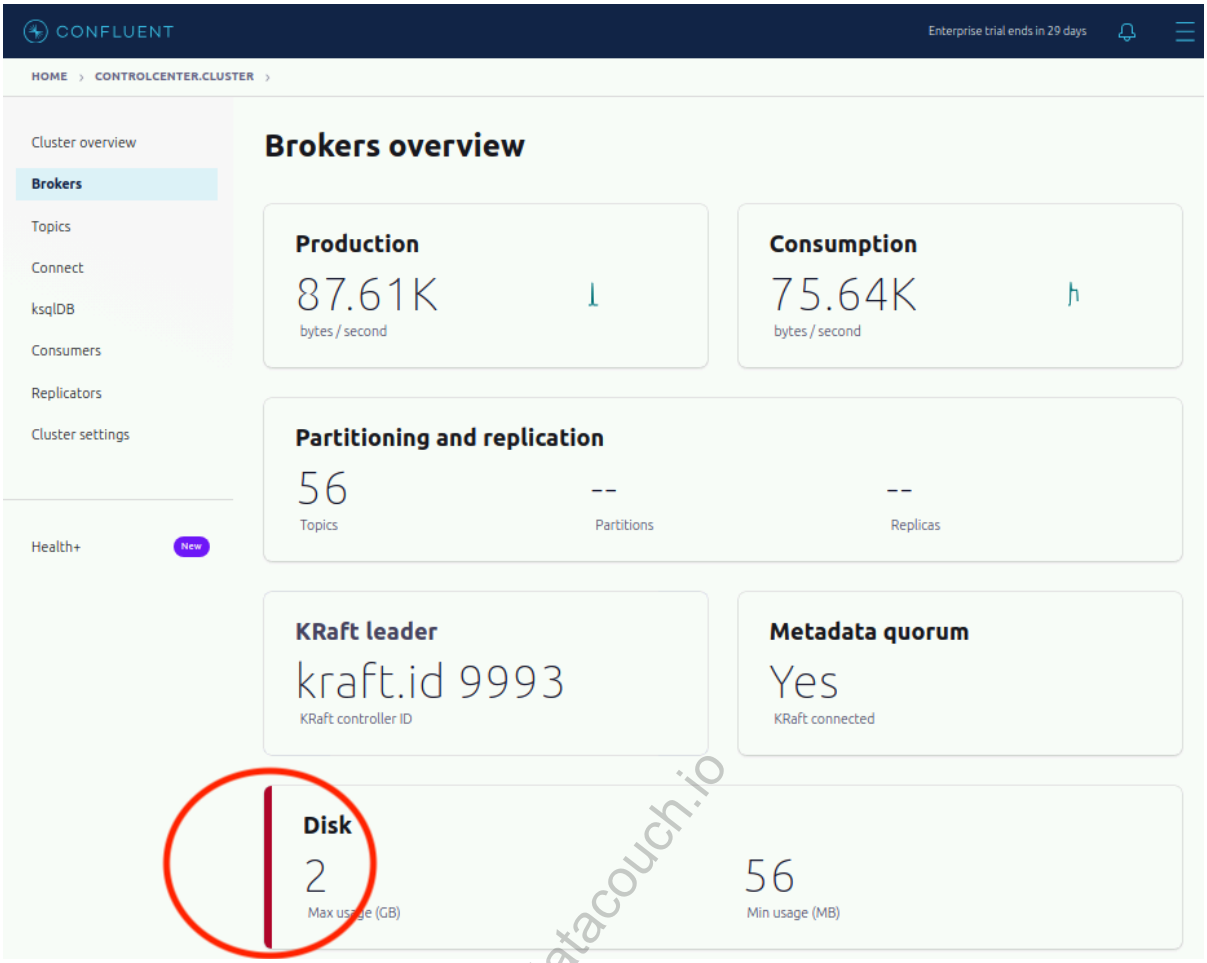


c. In the list of topics select the **`moving`** topic:



Notice the Topic's Partition list and the corresponding Brokers where each Partition replica resides. They are all on Brokers 1 and 2.

4. In Control Center on the left side, select **Brokers**.
5. In the **Brokers overview** view, look at the metric **Disk**. Notice the red bar to indicate a cluster imbalance: (may take a couple of minutes)



6. Consume some data from the Topic **moving**. Leave this consumer running for the duration of the exercise:

```
$ kafka-consumer-perf-test \
    --bootstrap-server $BOOTSTRAPS \
    --topic moving \
    --group test-group \
    --show-detailed-stats \
    --timeout 1000000 \
    --reporting-interval 5000 \
    --messages 10000000

time, threadId, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg,
nMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec,
fetch.nMsg.sec
2024-01-18 14:55:14:718, 0, 0.0010, 0.0000, 1, 0.0356, 27962, 101,
0.0094, 9.9010
2024-01-18 14:55:19:718, 0, 758.2903, 151.6579, 795125, 159024.8000,
0, 5000, 151.6579, 159024.8000
...
```

The next few steps use the Confluent Auto Data Balancer to rebalance the cluster. If you prefer to use the Apache open source tool instead, you may view the Appendix section of this manual called **Reassigning Partitions in a Topic: Alternate Method** (→ Jump here). Note the open source tool requires users to enumerate every Topic to rebalance and does not take into consideration disk utilization per Broker.

7. Run the Confluent Auto Data Balancer to rebalance the cluster. This will distribute the Partitions in the cluster, including the Topic `moving`, so that the Brokers are more evenly utilized. Wait for a few seconds for it to compute the rebalance plan.


```
$ confluent-rebalancer execute \
    --bootstrap-server $BOOTSTRAPS \
    --metrics-bootstrap-server $BOOTSTRAPS \
    --throttle 1000000 \
    --verbose

Computing the rebalance plan (this may take a while) ...
You are about to move 0 replica(s) for 0 partitions to 0 broker(s)
with total size 0 MB.
The preferred leader for 21 partition(s) will be changed.
In total, the assignment for 21 partitions will be changed.
The minimum free volume space is set to 20.0%.

The following brokers will have less than 40% of free volume space
during the rebalance:
    Broker     Current Size (MB)  Size During Rebalance (MB)   Free %
During Rebalance      Size After Rebalance (MB)    Free % After
Rebalance
    3          105.4              105.4                        10.8
105.4                         10.8
    2          2,131              2,131                        10.8
2,131                         10.8
    1          2,131              2,131                        10.8
2,131                         10.8

Min/max stats for brokers (before -> after):
    Type  Leader Count                Replica Count
Size (MB)
    Min   244 (id: 3) -> 263 (id: 1)   783 (id: 3) -> 783 (id: 3)
105.4 (id: 3) -> 105.4 (id: 3)
    Max   275 (id: 2) -> 263 (id: 1)   789 (id: 1) -> 789 (id: 1)
2,131 (id: 1) -> 2,131 (id: 1)

Rack stats (before -> after):
    Rack       Leader Count    Replica Count   Size (MB)
    rack-0     789 -> 789      2361 -> 2361    4,367.4 -> 4,367.4

Broker stats (before -> after):
    Broker     Leader Count    Replica Count   Size (MB)
Free Space (%)
    1          270 -> 263      789 -> 789      2,131 -> 2,131
10.8 -> 10.8
    2          275 -> 263      789 -> 789      2,131 -> 2,131
10.8 -> 10.8
    3          244 -> 263      783 -> 783      105.4 -> 105.4
10.8 -> 10.8

Would you like to continue? (y/n):
```

8. To start the rebalance operation, type **y** and press **Enter**.

The following response will appear:

```
The rebalance has been started, run `status` to check progress.

Warning: You must run the `status` or `finish` command periodically,
until the rebalance completes, to ensure the throttle is removed. You
can also alter the throttle by re-running the execute command passing
a new value.
```

9. Observe the configured throttling limits:

```
$ kafka-configs \
    --bootstrap-server $BOOTSTRAPS \
    --describe \
    --entity-type brokers

Dynamic configs for broker 1 are:
  follower.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Dynamic configs for broker 2 are:
  follower.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Dynamic configs for broker 3 are:
  follower.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Default configs for brokers in the cluster are:
```


10. Verify that throttling is in effect for topic `moving`:

```
$ kafka-configs --bootstrap-server $BOOTSTRAPS --describe --topic
moving

Dynamic configs for topic moving are:
  follower.replication.throttled.replicas=0:3,1:3,2:3,3:3
sensitive=false
synonyms={DYNAMIC_TOPIC_CONFIG:follower.replication.throttled.replica
s=0:3,1:3,2:3,3:3}

leader.replication.throttled.replicas=0:1,0:2,1:1,1:2,2:1,2:2,3:1,3:2
sensitive=false
synonyms={DYNAMIC_TOPIC_CONFIG:leader.replication.throttled.replicas=
0:1,0:2,1:1,1:2,2:1,2:2,3:1,3:2}
```

11. Monitor the progress of the rebalancing:

```
$ confluent-rebalancer status \
    --bootstrap-server $BOOTSTRAPS
Partitions being rebalanced:
    Topic moving: 0,1,2,4
```

Your status may indicate different partitions being rebalanced.

12. Increase the throttle limit configuration to 1GBps by rerunning the `confluent-rebalancer` command with the new throttle limit:

```
$ confluent-rebalancer execute \
    --bootstrap-server $BOOTSTRAPS \
    --metrics-bootstrap-server $BOOTSTRAPS \
    --throttle 1000000000 \
    --verbose
The throttle rate was updated to 1000000000 bytes/sec.
A rebalance is currently in progress for:
    Topic moving: 0,1,2,4
```

13. Note the updated throttle limit configuration values.

```
$ kafka-configs \
    --describe \
    --bootstrap-server $BOOTSTRAPS \
    --entity-type brokers

Dynamic configs for broker 1 are:
  follower.replication.throttled.rate=1000000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Dynamic configs for broker 2 are:
  follower.replication.throttled.rate=1000000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Dynamic configs for broker 3 are:
  follower.replication.throttled.rate=1000000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
```


14. Check the status of the Auto Data Balancer again.

```
$ confluent-rebalancer status \
    --bootstrap-server $BOOTSTRAPS
```

Note that it has completed:

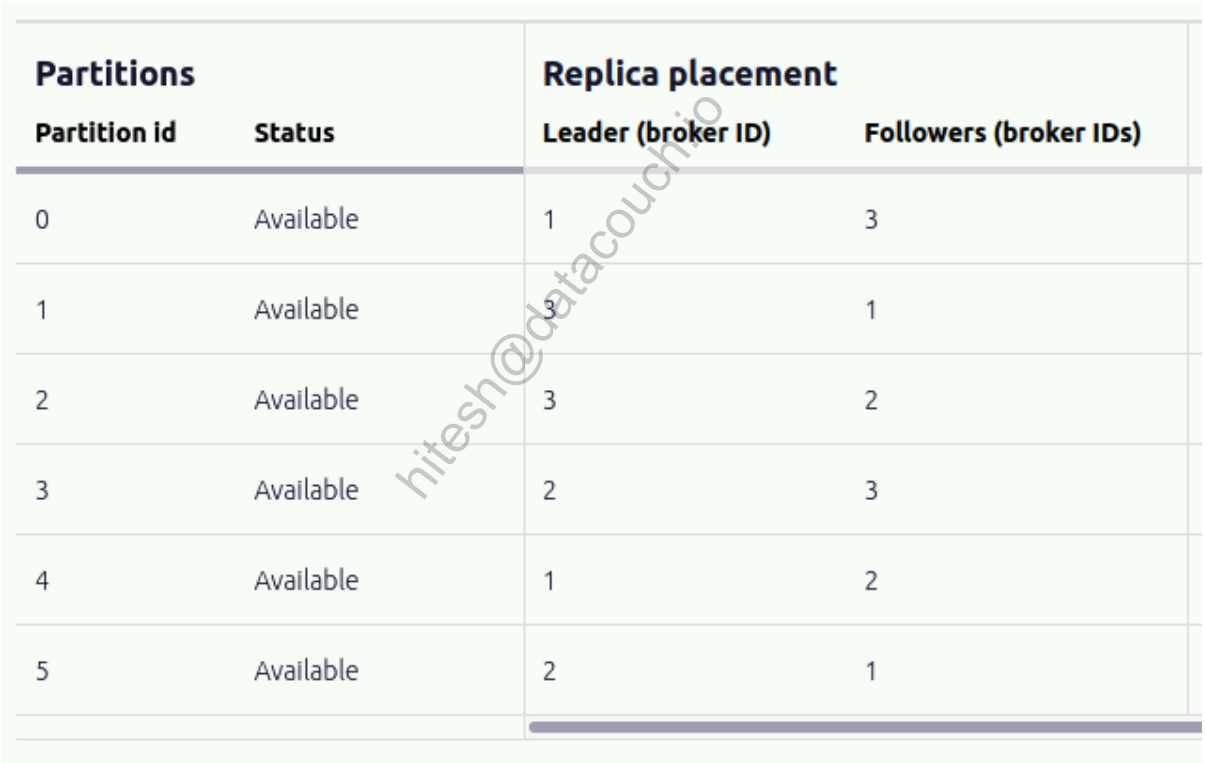
```
Error: No rebalance is currently in progress. If you have called
`status` after a rebalance was started successfully, the rebalance
has completed. Run the `execute` command to check if the cluster is
balanced.
```

15. Check to see if topic `moving` is still throttled:

```
$ kafka-configs \
    --bootstrap-server $BOOTSTRAPS \
    --describe \
    --topic moving

Dynamic configs for topic moving are:
```

16. Wait up to five minutes. In the Control Center `Broker overview`, look at the metric `Disk`. Notice the red bar is no longer present.

17. View the Topic information to see that the Topic `moving` has its Partitions moved across all three Brokers.



Partitions		Replica placement	
Partition id	Status	Leader (broker ID)	Followers (broker IDs)
0	Available	1	3
1	Available	3	1
2	Available	3	2
3	Available	2	3
4	Available	1	2
5	Available	2	1

18. Return to the terminal running the Consumer. Press `Ctrl+C` to exit the Consumer.

Simulate a Completely Failed Broker

In previous exercises, the Broker failures were such that the Broker could be simply restarted to recover. Now you will simulate a completely failed Broker which requires a replacement system.

1. In a new terminal from the host, observe the logs on Broker 1.

```
$ docker-compose exec kafka-1 ls /var/lib/kafka/data
...
moving-2
moving-3
moving-4
moving-5
recovery-point-offset-checkpoint
replication-offset-checkpoint
...
```

2. Stop and remove Broker 1 (this also removes the data of Broker 1).

```
$ docker-compose stop kafka-1 && \
    docker-compose rm kafka-1 && \
    docker volume rm confluent-admin_data-kafka-1

[+] Running 1/1
 ⠿ Container kafka-1  Stopped
? Going to remove kafka-1 Yes
[+] Running 1/0
 ⠿ Container kafka-1  Removed
```


3. In the Control Center under **Overview**, verify that only 2 Brokers are available.

You may have to wait up to five minutes for the Broker to disappear.

4. Recreate Broker 1:

```
$ docker-compose up -d kafka-1

[+] Running 4/4
 ⠿ Container controller-3  Running
 ⠿ Container controller-1  Running
 ⠿ Container controller-2  Running
 ⠿ Container kafka-1       Started
```

5. In the Control Center under **Overview** verify that all three Brokers are running.

You may have to wait up to five minutes for the previously failed Broker to reappear.

6. Notice that there are under replicated and offline Topic Partitions. After a moment of recovery they will return back to zero though.
7. Look at the logs in `/var/lib/kafka/data`. Verify that Broker 1 has not lost any of the data it had in step 1:

```
$ docker-compose exec kafka-1 ls /var/lib/kafka/data
```

Cleanup

1. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```


STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07 Consumer Groups and Load Balancing

a. Modifying Partitions and Viewing Offsets

In this exercise, you will increase the number of Partitions in a Topic and view offsets in an active Consumer Group.

Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Make sure your Kafka cluster is started, **otherwise** execute this command:

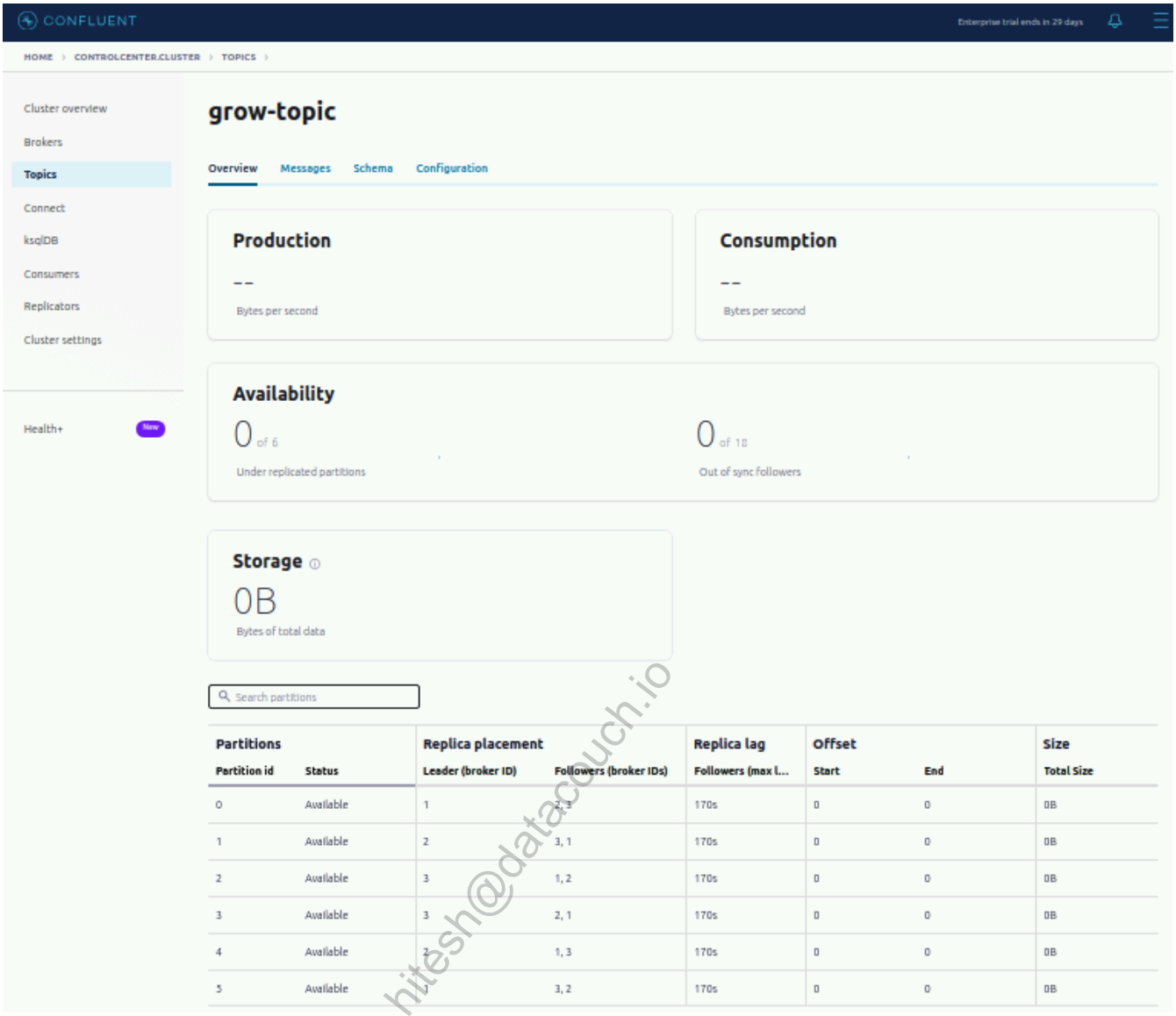
```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Increasing the Number of Partitions in a Topic

1. Create a new Topic called `grow-topic` with six Partitions and three replicas:

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic grow-topic \
    --partitions 6 \
    --replication-factor 3
```

2. View the `grow-topic` configuration and replica placement in the cluster:
   a. In Control Center go to the Home page and select our cluster
   b. Then, on the left side click on `Topics` and then click on the `grow-topic` Topic to see:



3. Start the console Producer for Topic `grow-topic`.

```
$ kafka-console-producer \
    --bootstrap-server $BOOTSTRAPS \
    --topic grow-topic
```

At the `>` prompt, type three messages and then press `Ctrl+D` to exit the console Producer:

```
> ksqlDB
> Streaming
> Engine
<Ctrl+D>
```

4. Start the console Consumer for Topic `grow-topic` specifying the `group.id` property:

```
$ kafka-console-consumer \
    --consumer-property group.id=test-consumer-group \
    --from-beginning \
    --topic grow-topic \
    --bootstrap-server $BOOTSTRAPS

Streaming
ksqlDB
Engine
```

Leave this Consumer running during the next step. Think about why it needs to keep running.

5. From another terminal window, describe the Consumer Group `test-consumer-group`:

```
$ kafka-consumer-groups \
    --bootstrap-server $BOOTSTRAPS \
    --group test-consumer-group \
    --describe

GROUP                TOPIC           PARTITION  CURRENT-OFFSET  LOG-
END-OFFSET  LAG             CONSUMER-ID
HOST            CLIENT-ID
test-consumer-group grow-topic      1          0               0
0               console-consumer-62ff2010-0f28-4b69-941e-1a8b557d9931
/172.19.0.1     console-consumer
test-consumer-group grow-topic      4          0               0
0               console-consumer-62ff2010-0f28-4b69-941e-1a8b557d9931
/172.19.0.1     console-consumer
test-consumer-group grow-topic      0          0               0
0               console-consumer-62ff2010-0f28-4b69-941e-1a8b557d9931
/172.19.0.1     console-consumer
test-consumer-group grow-topic      3          0               0
0               console-consumer-62ff2010-0f28-4b69-941e-1a8b557d9931
/172.19.0.1     console-consumer
test-consumer-group grow-topic      2          0               0
0               console-consumer-62ff2010-0f28-4b69-941e-1a8b557d9931
/172.19.0.1     console-consumer
test-consumer-group grow-topic      5          3               3
0               console-consumer-62ff2010-0f28-4b69-941e-1a8b557d9931
/172.19.0.1     console-consumer
```

6. Alter the Topic to increase the number of Partitions to 12:

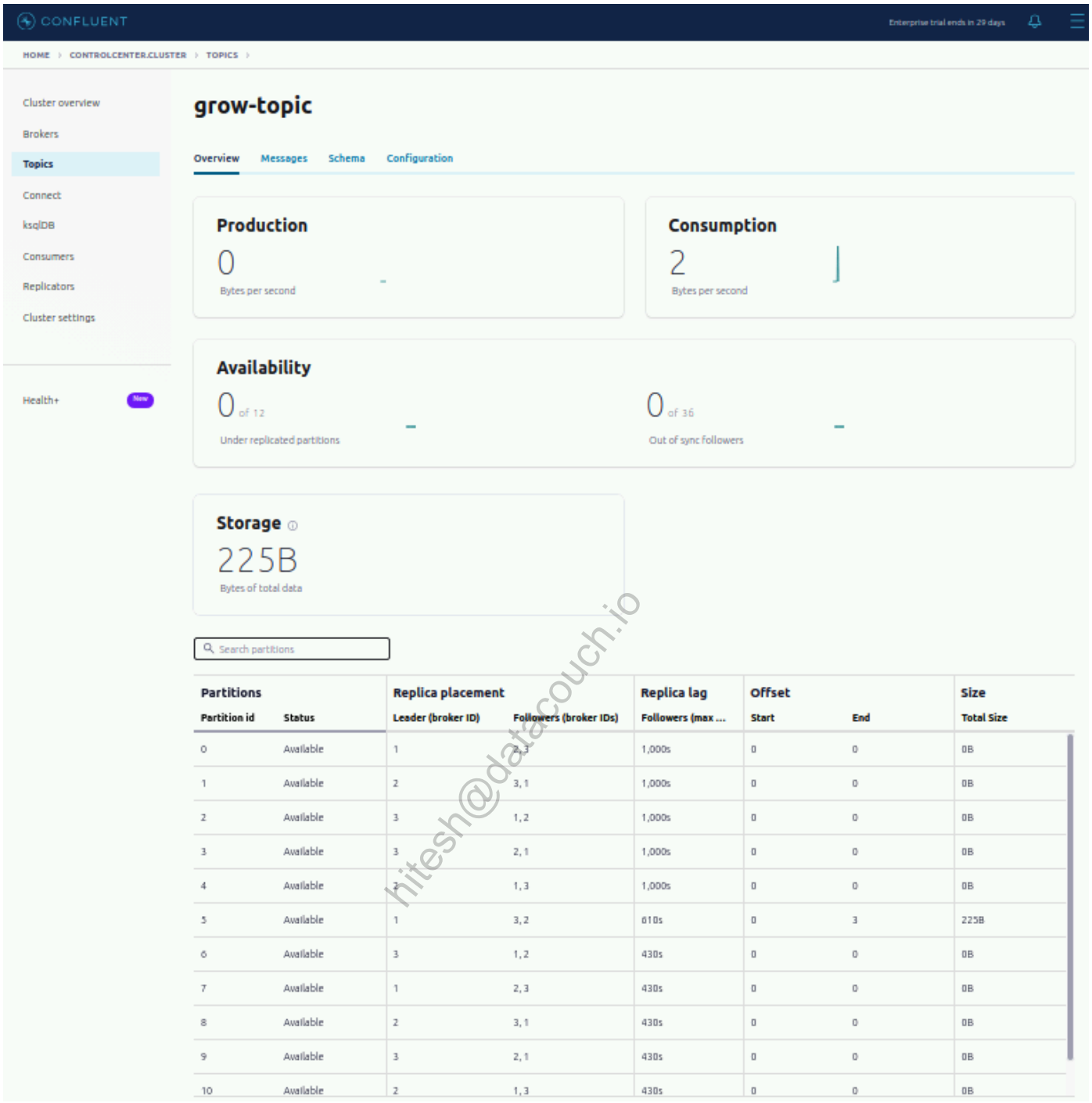
```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --alter \
    --topic grow-topic \
    --partitions 12
```

If partitions are increased for a topic that has a key, the partition logic or ordering of the messages will be affected!

7. Wait for a refresh of the metadata, which happens after `metadata.max.age.ms` (default: 5 minutes). Then describe the Consumer Group `test-consumer-group` again. What has changed?

```
$ kafka-consumer-groups \
    --bootstrap-server $BOOTSTRAPS \
    --group test-consumer-group \
    --describe
```

8. In Control Center, view the `grow-topic` topic configuration from the `Topics` view. You may need to refresh your browser:



You need to scroll the list of partitions to see all partitions.

9. In the terminal window where the Consumer is running, press `Ctrl+C` to terminate the process.

Kafka-based Offset Storage

1. Start the console Producer for Topic `new-topic`:

```
$ kafka-console-producer \
    --bootstrap-server $BOOTSTRAPS \
    --topic new-topic
```

   At the `>` prompt, type some messages into the console. After the first message, you will see a warning message because the Topic `new-topic` did not exist prior to producing to the Topic. Press `Ctrl+D` to return to the command line when you are done:

```
> I
[2024-01-18 21:19:01,528] WARN [Producer clientId=console-producer]
Error while fetching metadata with correlation id 5 : {new-
topic=UNKNOWN_TOPIC_OR_PARTITION}
(org.apache.kafka.clients.NetworkClient)
[2024-01-18 21:19:01,628] WARN [Producer clientId=console-producer]
Error while fetching metadata with correlation id 6 : {new-
topic=UNKNOWN_TOPIC_OR_PARTITION}
(org.apache.kafka.clients.NetworkClient)
> Love
> Kafka
<Ctrl+D>
```

2. Start the console Consumer for Topic `__consumer_offsets` to view the offsets. Leave this running for the duration of the exercise:

```
$ kafka-console-consumer \
    --topic __consumer_offsets \
    --bootstrap-server $BOOTSTRAPS \
    --formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatte
r" \
    | grep new-topic
```

3. Start the console Consumer in a new terminal window for Topic `new-topic` in a Consumer Group called `new-group`:

```
$ kafka-console-consumer \
    --from-beginning \
    --topic new-topic \
    --group new-group \
    --bootstrap-server $BOOTSTRAPS
I
Love
Kafka
```

4. Press `Ctrl+C` to terminate the Consumer once your messages have been displayed.

   a. When reading from the Topic `new-topic`, did you see the messages you typed earlier?

   b. In the other terminal window where you were reading from the Topic `__consumer_offsets`, did you see any `OffsetMetadata` specifically for the Topic `new-topic` such as shown below?

```
[new-group,new-topic,0]::OffsetAndMetadata(offset=3,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1705613176556,
expireTimestamp=None)
...
```

5. If they are currently running, terminate the Producer and Consumers with `Ctrl+C`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 08 Optimizing Kafka's Performance

a. Exploring Producer Performance

Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs

2. Make sure your Kafka cluster is started, **otherwise** execute this command:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Tuning Producer Performance

1. Create a new Topic called `performance` with six Partitions and three replicas.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic performance \
    --partitions 6 \
    --replication-factor 3
```

2. Run Producer performance tests to compare `acks=all`, `acks=1`, and `acks=0`

   Use the following command line and replace `<VARIABLE_HERE>` with the specified options in the table below:

```
$ kafka-producer-perf-test \
    --topic performance \
    --num-records 1000000 \
    --record-size 100 \
    --throughput 10000000 \
    --producer-props \
    bootstrap.servers=$BOOTSTRAPS \
    <VARIABLE_HERE>
```

Let each test run until it terminates and provides a performance summary. Record the throughput and latency results in a table. You will be appending to this table in upcoming questions.

Variables	Throughput (MB/sec)	Latency (ms avg)
acks=1		
acks=all		
acks=0		

3. Questions:
    a. Do you get better throughput with `acks=0`, `acks=1`, or `acks=all`? What is the percentage difference in performance?
    b. Do you get better latency with `acks=0`, `acks=1`, or `acks=all`? Why?
4. You will investigate the effects of tuning `batch.size` and `linger.ms` by investigating **one** question. Try values of `batch.size` between 0 and 1,000,000 and values of `linger.ms` between 0 and 3,000. Hold `acks=all` constant. **Choose one question to investigate**. Consider dividing work amongst peers who are investigating the same question. Record your results in a table.
    a. What is the maximum throughput, no matter the latency?
    b. What is the minimum latency, no matter the throughput?
    c. What is the best balance of throughput and latency? (and defend your decision)
    d. Given a batch size of 100,000 Bytes, what linger time gives best performance? What do you notice? What do you wonder?
    e. Given a linger time of 500 ms, what batch size gives best performance? What do you notice? What do you wonder?

Here is an example of a test you might run:

```
$ kafka-producer-perf-test \
    --topic performance \
    --num-records 1000000 \
    --record-size 100 \
    --throughput 10000000 \
    --producer-props \
    bootstrap.servers=$BOOTSTRAPS \
    acks=all \
    batch.size=400000 \
    linger.ms=500
```

Cleanup

1. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```


STOP HERE. THIS IS THE END OF THE EXERCISE.

b. Performance Tuning

In this exercise, you will observe Kafka performance and use some of Kafka's settings to monitor and optimize Consumers.

Prerequisites

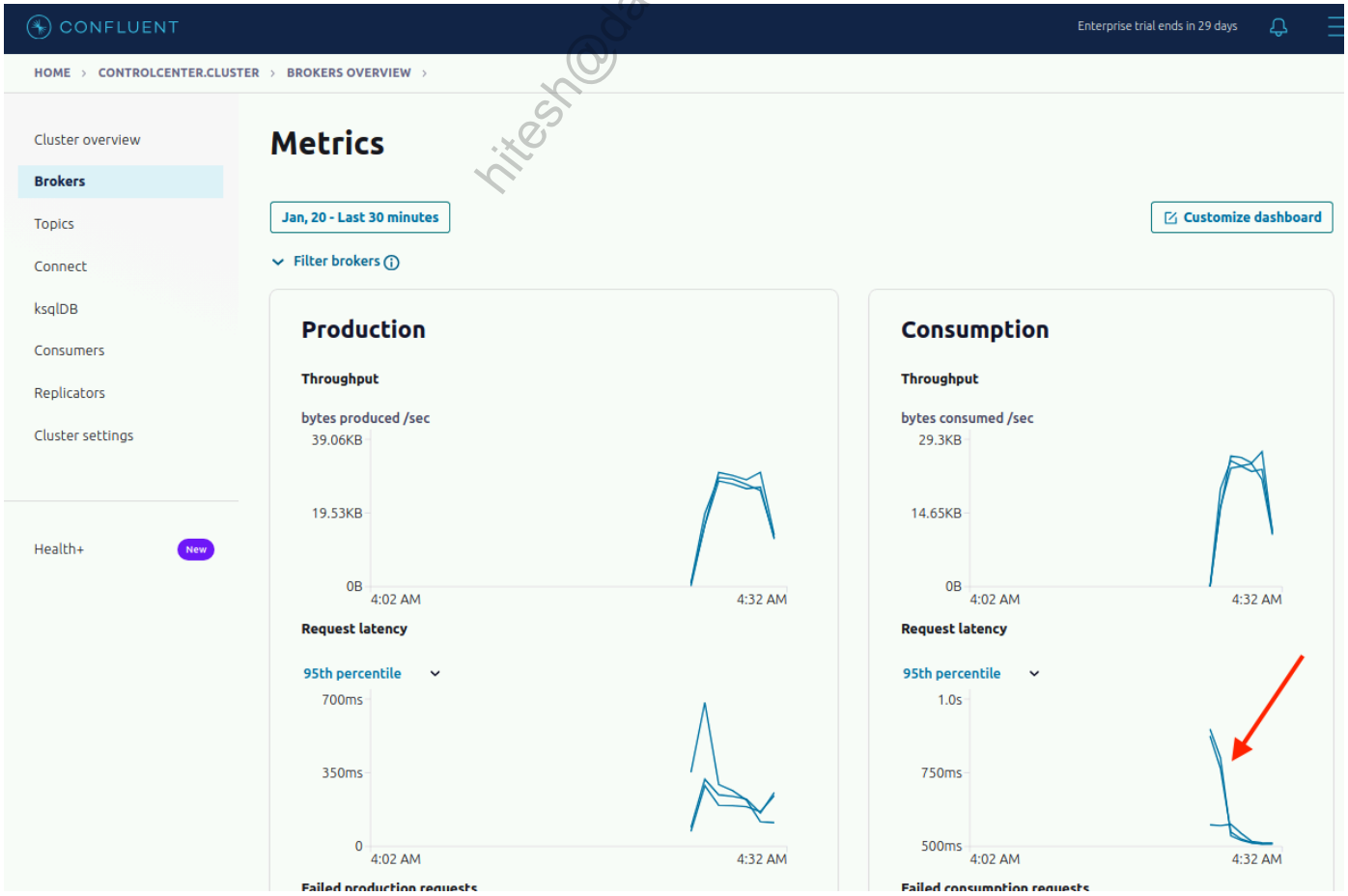
1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Make sure your Kafka cluster is started, **otherwise** execute this command:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```

Observing Replica Fetch Times

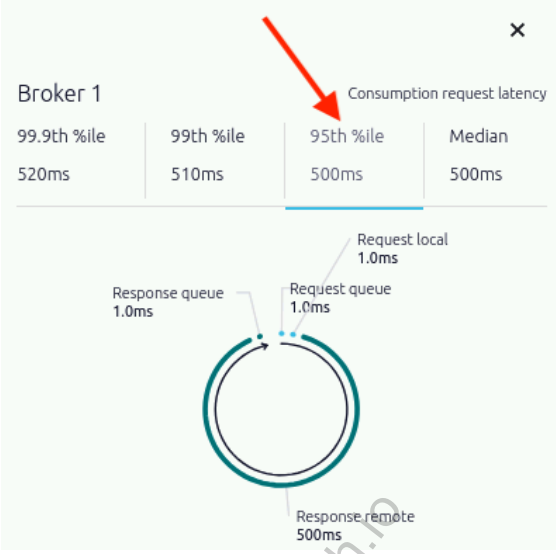
1. Connect to Control Center. If necessary, open a browser tab to the URL **http://localhost:9021**.

1. In the Control Center, go to the Home page → our cluster → **Brokers** → **Consumption**. Click on one of the lines in the consumption request latency line graph to open the breakdown of times in the request latency lifecycle.



Your exact view may differ.

2. In the fetch request latency breakdown, select the **95th %ile** view.



Remember that percentiles don't add associatively, so the numbers around the circle won't generally add up to the overall request time percentile.

3. Observe that most of the fetch request time is in **`Response remote`** time waiting for the **`replica.fetch.wait.max.ms`** timeout. When Producers are not writing records, the fetch request latency will go up to **`replica.fetch.wait.max.ms`** (default 500ms).

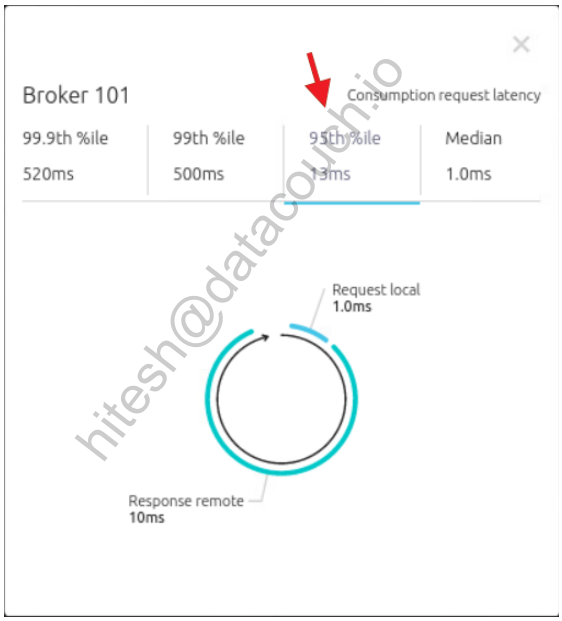
4. Create a new Topic called **`fetch-request`** with six Partitions and three replicas:

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic fetch-request \
    --partitions 6 \
    --replication-factor 3
```

5. Produce some data to send to the Topic **`fetch-request`** and let it run:

```
$ kafka-producer-perf-test \
    --topic fetch-request \
    --num-records 1000000 --record-size 100 --throughput 1000 \
    --producer-props bootstrap.servers=$BOOTSTRAPS

4999 records sent, 999.8 records/sec (0.10 MB/sec), 14.4 ms avg
latency, 538.0 ms max latency.
5002 records sent, 1000.2 records/sec (0.10 MB/sec), 7.8 ms avg
latency, 82.0 ms max latency.
5002 records sent, 1000.2 records/sec (0.10 MB/sec), 3.5 ms avg
latency, 54.0 ms max latency.
...
```

6. Let the producer run for five minutes. In Control Center's fetch request latency breakdown, observe that the **response remote time** at the `95th %ile` drops to a few milliseconds:



Cleanup

1. If it is currently running, terminate the Producer by pressing `Ctrl+C`.
2. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```

Tune Consumers to Decrease Broker CPU Load

1. Re-start the cluster after the previous clean-up:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```

2. Create a new Topic called `i-love-logs` with one Partition and one replica.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic i-love-logs \
    --replica-assignment 1
```

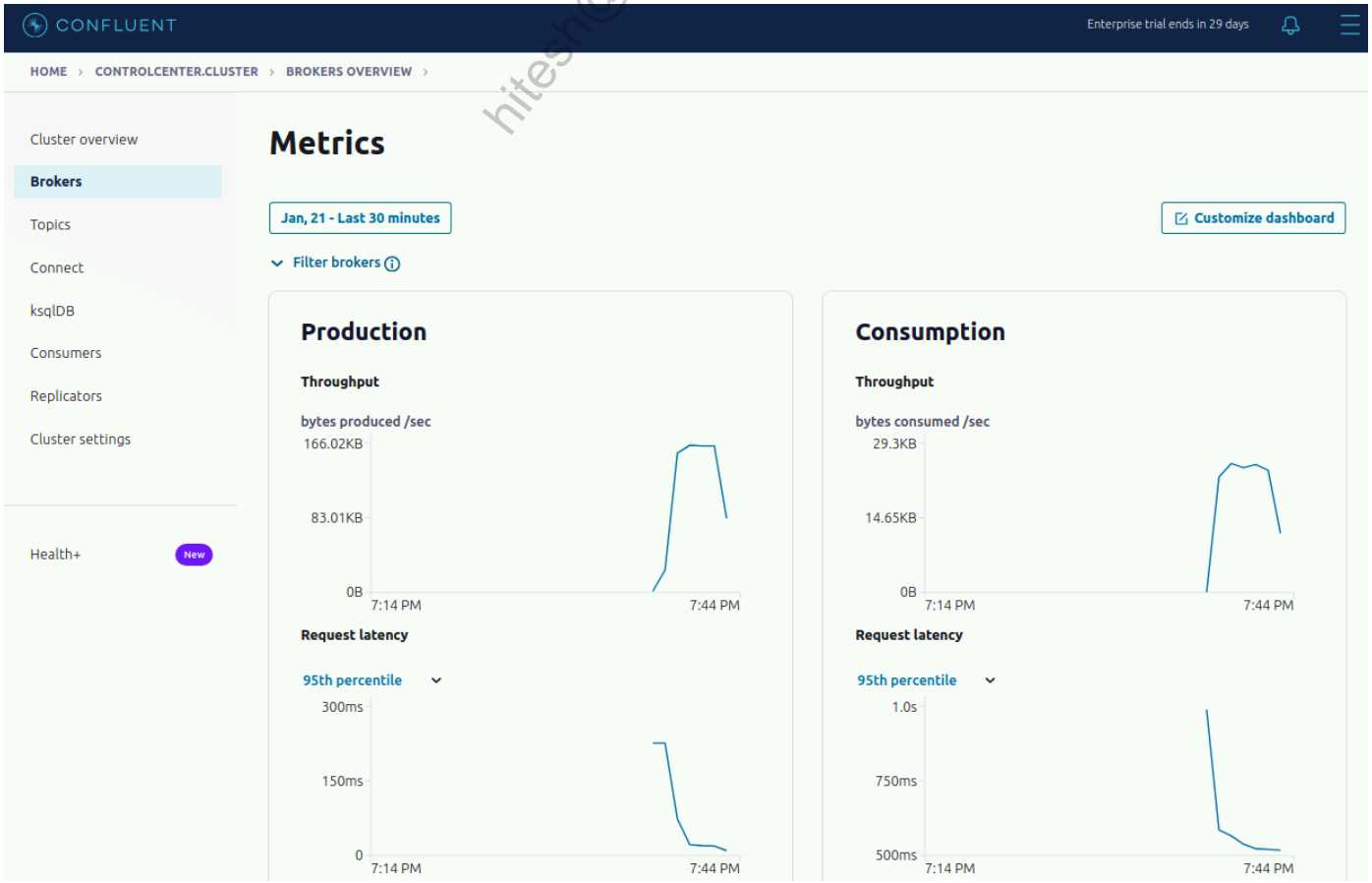
We are using the `--replica-assignment` parameter to ensure that the partition for all students is located on the `kafka-1` broker which is assigned broker id `1`.

3. Produce some data to send to the Topic `i-love-logs`. Leave this process running until instructed to end it.

   The following command line sends 10 million messages, 100 bytes in size, at a rate of 1000 messages/sec. The `NS` environment variable defines interceptors that enable stream monitoring in Control Center.

```
$ PACKAGE=io.confluent.monitoring.clients.interceptor && \
kafka-producer-perf-test \
    --topic i-love-logs \
    --num-records 10000000 \
    --record-size 100 \
    --throughput 1000 \
    --producer-props \
        bootstrap.servers=$BOOTSTRAPS \
        interceptor.classes=$PACKAGE.MonitoringProducerInterceptor

4990 records sent, 998.0 records/sec (0.10 MB/sec), 55.1 ms avg
latency, 1140.0 ms max latency.
5003 records sent, 1000.6 records/sec (0.10 MB/sec), 5.8 ms avg
latency, 99.0 ms max latency.
5002 records sent, 1000.2 records/sec (0.10 MB/sec), 14.7 ms avg
latency, 179.0 ms max latency.
...
```

4. Wait for a few minutes. Go to Control Center → **Brokers** → **Production**. If necessary, adjust the timescale of the data to the last 30 minutes.
5. Select **broker.id 1** in the **Filter brokers** list.
6. Observe changes that occur in the **Throughput** and **Request latency**. After five minutes, the graphs should look similar to this:



7. Open a new terminal and navigate to the **~/confluent-admin** directory:

```
$ cd ~/confluent-admin
```

8. Create a consumer properties file called **data/consumer.properties** that will enable Control Center to monitor Consumer performance of Consumer Group **cg**:

```
$ echo \
"interceptor.classes=io.confluent.monitoring.clients.interceptor.Moni
toringConsumerInterceptor" > data/consumer.properties
```

9. Consume some data from the Topic **i-love-logs** with Consumer Group **cg**.
   Leave this process running until instructed to end it.

```
$ kafka-consumer-perf-test \
        --bootstrap-server $BOOTSTRAPS \
        --topic i-love-logs \
        --group cg \
        --messages 10000000 \
        --show-detailed-stats \
        --reporting-interval 5000 \
        --consumer.config data/consumer.properties

time, threadId, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg,
nMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec,
fetch.nMsg.sec
2024-01-21 19:50:59:314, 0, 68.5235, 13.7020, 718521, 143675.4649,
601, 4400, 15.5735, 163300.2273
2024-01-21 19:51:04:314, 0, 69.0002, 0.0953, 723520, 999.8000, 0,
5000, 0.0953, 999.8000
2024-01-21 19:51:09:314, 0, 69.4771, 0.0954, 728520, 1000.0000, 0,
5000, 0.0954, 1000.0000
...
```


10. In a new terminal, observe the CPU load for the **java** process on Broker **kafka-1**:

```
$ cd ~/confluent-admin
$ docker-compose exec kafka-1 top -n10

top - 19:52:51 up 18 min,  0 users,  load average: 2.93, 2.58, 2.54
Tasks:   2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 17.2 us,  4.1 sy,  0.0 ni, 77.0 id,  0.3 wa,  0.0 hi,  1.4
si,  0.0 st
MiB Mem :  15693.6 total,   3013.9 free,   9252.0 used,   3427.6
buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   6039.9
avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
TIME+ COMMAND
      1 appuser   20   0   20.1g 962976  22504 S  24.9   6.0
4:18.41 java
   1046 appuser   20   0   56328   3996   3372 R   0.0   0.0
0:00.02 top
```

Observe CPU utilization until **top** exits. In the output above, the java process CPU load is 24.9%.

11. Go to the terminal with the Consumer Group called **cg** and kill it with **Ctrl+C**.

12. Add consumer setting **fetch.min.bytes=10485760** to **data/consumer.properties**:

```
$ echo "fetch.min.bytes=10485760" >> data/consumer.properties
```

This configuration tells the Broker to wait for larger amounts of data to accumulate before responding to the Consumer. This generally improves throughput and reduces load on the Broker but can increase latency.

13. Continue consuming data from for the Topic **i-love-logs** with Consumer Group **cg** using the updated **consumer.properties**:

```
$ kafka-consumer-perf-test \
        --bootstrap-server $BOOTSTRAPS \
        --topic i-love-logs \
        --group cg \
        --messages 10000000 \
        --show-detailed-stats \
        --reporting-interval 5000 \
        --consumer.config data/consumer.properties
```

14. Observe again the CPU load for the **java** process on Broker **kafka-1**:

```
$ docker-compose exec kafka-1 top -n10

top - 20:01:20 up 27 min,  0 users,  load average: 1.45, 2.49, 2.64
Tasks:   2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  8.1 us,  2.7 sy,  0.0 ni, 88.4 id,  0.3 wa,  0.0 hi,  0.5
si,  0.0 st
MiB Mem :  15693.6 total,   2592.1 free,   9437.7 used,   3663.8
buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   5855.0
avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
TIME+ COMMAND
      1 appuser   20   0   20.1g 989.7m  23172 S  12.0   6.3
6:10.23 java
   1453 appuser   20   0   56328   4160   3536 R   0.0   0.0
0:00.02 top
```

Indeed, increasing **`fetch.min.bytes`** reduces the CPU load of the Broker.

Defining Over Consumption Trigger and Action

For this exercise, we need to define a trigger and corresponding action in Control Center:

1. Open Control Center
2. Click the bell icon to get to **Alerts**:
3. Click the button **Create trigger**
4. Complete the dialog using the following settings:

Field	Value
Trigger Name	Over Consumption
Component type	Consumer group
Consumer group name	cg [...]
Metric	Consumption difference
Buffer (seconds)	120
Condition	Greater than
Value	0

5. Click **Save**
6. In the confirmation box, when asked to "Set actions to perform when your trigger goes off", select **Create an action**
7. Complete the dialog using the following settings:

Field	Value
Action Name	Over Consumption
Triggers	Over Consumption
Actions	Send email
Subject	Over Consumption
Max send rate	12 Per hour
Recipient email address	anyone@anywhere.com

8. Click **Save**

Simulating Over Consumption

1. Go to the terminal with the Consumer Group called `cg` and kill it with `Ctrl+C`.
2. Identify the current offsets of Consumer Group `cg`:

```
$ kafka-consumer-groups \
        --bootstrap-server $BOOTSTRAPS \
        --group cg \
        --describe

Consumer group 'cg' has no active members.

GROUP           TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-
OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
cg              i-love-logs     0          1847532         1897933
50401           -               -               -
```

3. Reset the offsets of Consumer Group `cg` offset for Partition `0` of Topic `i-log-logs` to `<current offset> - 100`. Using the previous step as an example, this would be, `1847532 - 100 = 1847432`:

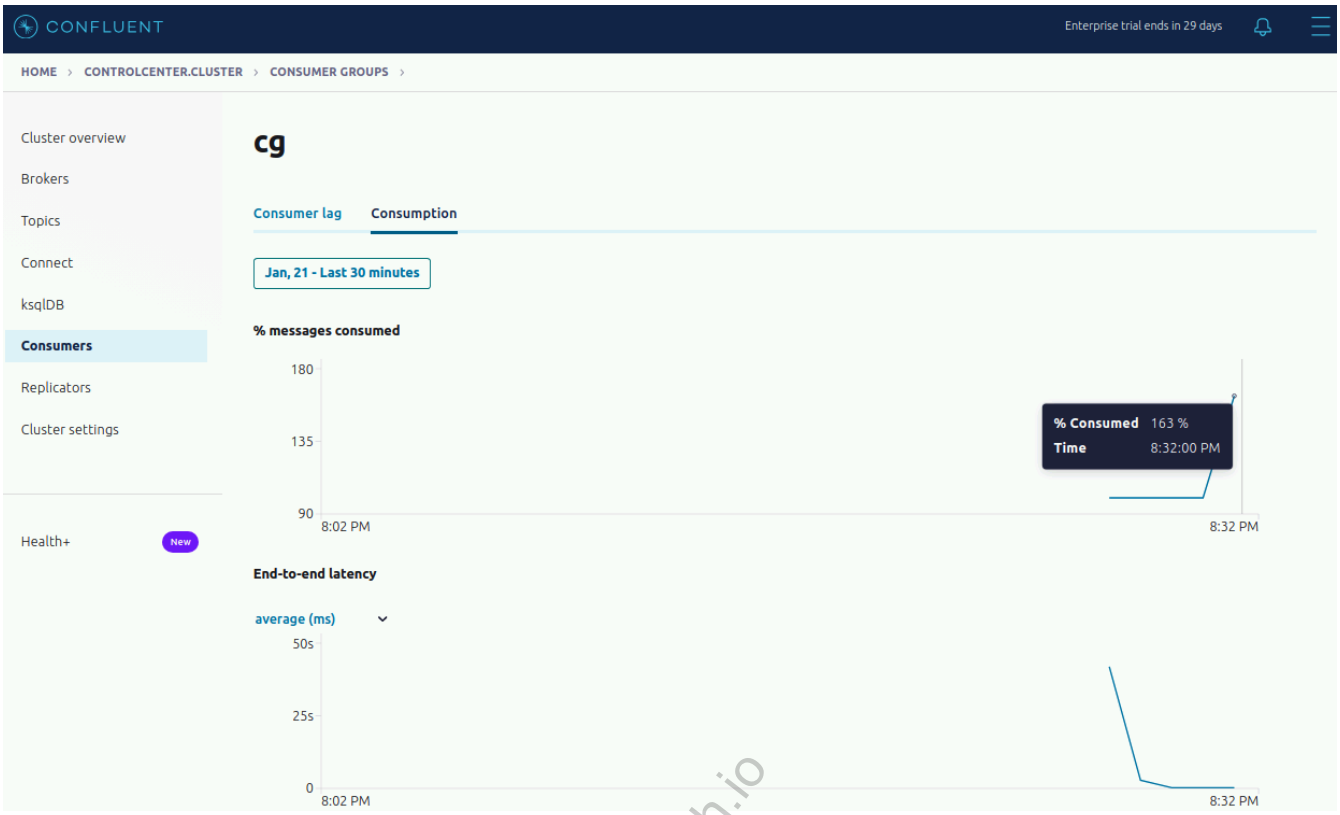
```
$ kafka-consumer-groups \
    --bootstrap-server $BOOTSTRAPS \
    --group cg \
    --reset-offsets \
    --to-offset <value> \
    --topic i-love-logs:0 \
    --execute

GROUP                          TOPIC
PARTITION  NEW-OFFSET
cg                             i-love-logs                    0
1847432
```

4. Continue consuming data from for the Topic **i-love-logs** with Consumer Group **cg**. Since the offsets have been reset to **<current offset> - 100**, the Consumer Group will consume **100** messages from the **i-love-logs** Topic a second time, thereby processing these messages again. This is a good simulation of Consumer Group over-consumption:

```
$ kafka-consumer-perf-test \
        --bootstrap-server $BOOTSTRAPS \
        --topic i-love-logs \
        --group cg \
        --messages 10000000 \
        --show-detailed-stats \
        --reporting-interval 5000 \
        --consumer.config data/consumer.properties
```


5. Observe the cluster in Control Center for a couple of minutes. You may need to adjust your time window.

   a. Go to our cluster → **Consumers** → **cg** → **Consumption** and observe the Consumer Group **cg** for a few minutes:



Observe that:

- The `% Consumed` is over 100%

6. Click the bell icon to go back to the `Alerts Overview`. View the alert history and notice that the over-consumption event triggered an alert. It is critical to monitor applications for message consumption to know how your applications are behaving. Over-consumption may happen intentionally, or it may happen unintentionally, for example if an application crashes before committing processed messages.

Cleanup

1. End all running Producers and Consumers by going to the respective terminal window and pressing `Ctrl+C`.
2. Execute the following command to completely clean up your environment:

```
$ docker-compose down -v
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 09 Kafka Security

a. Securing the Kafka Cluster

In this exercise, you will configure one-way SSL authentication, two-way SSL authentication, and explore the SSL performance impact.

Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Change directory to the `secure-cluster` folder:

```
$ cd ~/confluent-admin/secure-cluster
```

3. Start the secure cluster:

```
$ docker-compose up -d
```

The broker containers have been configured to wait until their respective `kafka-n-creds` directory has been populated with the required certificates. Once the certificates are present, the broker startup continues. These certificates are being created in the next step of the exercise.

Generating Certificates

Next, we generate all the necessary certificates and credentials that are used to create a secure, 3-Broker Kafka cluster:

1. Generate a signed certificate, keystores, and truststores by running a script:

```
$ ./certs-create.sh
```

2. (OPTIONAL) Use VS-code or other editor to inspect the files the script:

```
$ code .
```

The `ca.key` file is the private key that the Certificate Authority uses to sign the certificate `ca.crt`. The script used this certificate to create the Broker keystore and the client truststore. The Broker's keystore is what the Broker uses to send its certificate to the client. The client truststore is what the client uses to check whether a Broker's certificate ought to be trusted - if the checksum of the received certificate is unexpected, then someone tried to alter the certificate and thus the certificate is untrustworthy.

Later during mutual SSL, the client will also need to authenticate with the Broker. Therefore, the script also created a keystore for the client and a truststore for the Broker.

Enabling SSL on the Brokers

1. Open the file `~/confluent-admin/secure-cluster/docker-compose.yml` in your editor and note the following environment variables (in addition to the existing ones) that have been added to each of the three Brokers (`kafka-1`, `kafka-2` and `kafka-3`), e.g. for `kafka-1` they are:

```
KAFKA_LISTENERS: SSL://kafka-1:19093, DOCKER://kafka-1:9092,
EXTERNAL://kafka-1:19092
KAFKA_INTER_BROKER_LISTENER_NAME: DOCKER
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
CONTROLLER:PLAINTEXT,DOCKER:PLAINTEXT,EXTERNAL:PLAINTEXT,SSL:SSL
KAFKA_SSL_KEYSTORE_FILENAME: kafka.kafka-1.keystore.jks
KAFKA_SSL_KEYSTORE_CREDENTIALS: kafka-1_keystore_creds
KAFKA_SSL_KEY_CREDENTIALS: kafka-1_sslkey_creds
KAFKA_SSL_TRUSTSTORE_FILENAME: kafka.kafka-1.truststore.jks
KAFKA_SSL_TRUSTSTORE_CREDENTIALS: kafka-1_truststore_creds
KAFKA_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM: "HTTPS"
```

The variable names shown above might not look familiar at first. Your trainer has shown you variable names in the configuration files for Brokers such as **`ssl.keystore.filename`** or **`ssl.truststore.credentials`**. A startup script inside each container converts the environment variable names by a standard algorithm into the Kafka specific variable names and adds them to the respective configuration file. Upper case is converted to lowercase and underscores are converted to periods. Furthermore, each component (Kafka, REST Proxy, Kafka Connect, etc.) has a well-defined prefix that is removed from the Docker variable. In the above case it is **`KAFKA_`**.

2. Each of the three Brokers maps the credential folder into **`/etc/kafka/secrets`**. For example, **`kafka-1`** has:

```
volumes:
    ...
    - $PWD/kafka-1-creds:/etc/kafka/secrets
```


3. Please answer the following questions. Feel free to discuss with peers:
   a. Why is **`KAFKA_LISTENERS`** configured for both **`SSL`** and **`PLAINTEXT`**?
   b. What would happen if **`KAFKA_LISTENERS`** were configured for just **`SSL`**?
   c. What does **`KAFKA_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM`** do?

Verifying SSL is Working

1. From a terminal window verify that the keystore and truststore of each Broker are set up properly. The **`openssl`** command below should return a key and certificate:

```
$ openssl s_client -connect kafka-1:19093 -tls1_3

CONNECTED(00000003)
Can't use SSL_get_servername
depth=1 CN = ca1.test.confluent.io, OU = TEST, O = CONFLUENT, L =
PaloAlto, C = US
verify error:num=19:self signed certificate in certificate chain
verify return:1
depth=1 CN = ca1.test.confluent.io, OU = TEST, O = CONFLUENT, L =
PaloAlto, C = US
verify return:1
depth=0 C = US, ST = Ca, L = PaloAlto, O = CONFLUENT, OU = TEST, CN =
kafka-1
verify return:1
---
Certificate chain
 0 s:C = US, ST = Ca, L = PaloAlto, O = CONFLUENT, OU = TEST, CN =
kafka-1
   i:CN = ca1.test.confluent.io, OU = TEST, O = CONFLUENT, L =
PaloAlto, C = US
 1 s:CN = ca1.test.confluent.io, OU = TEST, O = CONFLUENT, L =
PaloAlto, C = US
   i:CN = ca1.test.confluent.io, OU = TEST, O = CONFLUENT, L =
PaloAlto, C = US
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDTjCCAjYCFFwPFVdSbplifyZrfVpOZOyWmwceMA0GCSqGSIb3DQEBCwUAMGMx
...
```


The line `verify error:num=19 …` is expected since there are self-signed certificates during the certificate creation process. Press `Ctrl+C` to stop the above command.

2. Repeat the same for the other two Brokers:

```
$ openssl s_client -connect kafka-2:29093 -tls1_3
$ openssl s_client -connect kafka-3:39093 -tls1_3
```

If you see a `Connection refused` error, then the Broker is not properly set up for SSL or may need to be restarted with `docker-compose restart <broker>`. We simulate this error here by providing the wrong port:

```
$ openssl s_client -connect kafka-1:9094 -tls1_3

140257314772288:error:0200206F:system
library:connect:Connection
refused:../crypto/bio/b_sock2.c:110:
140257314772288:error:2008A067:BIO
routines:BIO_connect:connect
error:../crypto/bio/b_sock2.c:111:
connect:errno=111
```

3. Let's try to produce messages to the cluster via the SSL port.

   a. Create a new Topic called `ssl-topic` with one Partition and three replicas:

```
$ kafka-topics \
    --bootstrap-server kafka-1:19093,kafka-2:29093,kafka-3:39093 \
    --create \
    --topic ssl-topic \
    --partitions 1 \
    --replication-factor 3
```


The Admin Client is using the PLAINTEXT listener at port 19092 to create the Topic because it doesn't have any SSL/TLS client configuration.

   b. Start the console Producer for Topic `ssl-topic`. Here the client is connecting to the SSL port `19093` rather than the PLAINTEXT port `19092`:

```
$ kafka-console-producer \
    --bootstrap-server kafka-1:19093,kafka-2:29093,kafka-3:39093 \
    --topic ssl-topic

[2024-01-23 08:28:03,288] WARN [Producer clientId=console-
producer] Bootstrap broker kafka-1:19093 (id: -1 rack: null)
disconnected (org.apache.kafka.clients.NetworkClient)
[2024-01-23 08:28:03,289] WARN [Producer clientId=console-
producer] Bootstrap broker kafka-2:29093 (id: -2 rack: null)
disconnected (org.apache.kafka.clients.NetworkClient)
...
```

c. Press **Ctrl+C** to stop the producer.

This leads to a connection error. Why?

When clients communicate with Kafka brokers using SSL, we must provide the required configuration settings.

4. We will now configure the Kafka clients to use the truststore.

   a. First, let's examine the **client-ssl.properties** file that we need to specify when we run the **kafka-console-producer and kafka-console-consumer** clients when connecting to the broker using SSL:

```
$ cat client-creds/client_ssl.properties
security.protocol=SSL
ssl.truststore.location=client-creds/kafka.client.truststore.jks
ssl.truststore.password=confluent
```

   b. Try to produce messages again using the **client_ssl.properties** file:

```
$ kafka-console-producer \
    --bootstrap-server kafka-1:19093,kafka-2:29093,kafka-3:39093 \
    --topic ssl-topic \
    --producer.config client-creds/client_ssl.properties
```

   c. At the **>** prompt, type "Security is good" and press Enter. Add a couple more messages and then press **Ctrl+D** to exit the console Producer.

```
> Security is good
> Certificates
> Keys
<Ctrl+D>
```

   d. Start the console Consumer for Topic **ssl-topic** using the **client_ssl.properties** file. You should see the messages you typed above. Press **Ctrl+C** once your messages have been displayed.

```
$ kafka-console-consumer \
    --consumer.config client-creds/client_ssl.properties \
    --from-beginning \
    --topic ssl-topic \
    --bootstrap-server kafka-1:19093,kafka-2:29093,kafka-3:39093

Security is good
Certificates
Keys
```

Press **Ctrl+C** to stop the Consumer.

Enabling Mutual SSL Authentication

1. In the **docker-compose.yml** file in folder **~/confluent-admin/secure-cluster**, uncomment the following environment variable for each Broker:

```
KAFKA_SSL_CLIENT_AUTH: "required"
```

2. Restart all 3 Brokers by executing:

```
$ cd ~/confluent-admin/secure-cluster
$ docker-compose up -d

[+] Running 6/6
 ✔ Container controller-3  Running
 ✔ Container controller-1  Running
 ✔ Container controller-2  Running
 ✔ Container kafka-1       Started
 ✔ Container kafka-2       Started
 ✔ Container kafka-3       Started
```

Docker will realize that the definition of the 3 Broker definitions have changed and restart the respective containers.

3. Try to run the **kafka-console-consumer**:

```
$ kafka-console-consumer \
    --consumer.config client-creds/client_ssl.properties \
    --from-beginning \
    --topic ssl-topic \
    --bootstrap-server kafka-1:19093,kafka-2:29093,kafka-3:39093

[2024-01-23 08:48:49,584] ERROR [Consumer clientId=console-consumer,
groupId=console-consumer-53527] Connection to node -3 (kafka-
3/127.0.0.1:39093) failed authentication due to: Failed to process
post-handshake messages, SNI host name: empty
(org.apache.kafka.clients.NetworkClient)
...
```

The command reports an authentication failure. Why?

Previously, the Broker had to authenticate with the Clients. Now that we configured the Brokers to require mutual SSL, the Clients must also authenticate with Brokers.

The `certs-create.sh` script already created a keystore in the `client-creds` folder and corresponding truststore in each `kafka-{1|2|3}-creds` folders. We need to add the related keystore configuration settings to `client_ssl.properties`.

4. Add the required lines to `client_ssl.properties`:

```
$ cat <<EOF >>client-creds/client_ssl.properties
ssl.keystore.location=client-creds/kafka.client.keystore.jks
ssl.keystore.password=confluent
EOF
```

   a. Let's confirm `client-ssl.properties` now contains all the required settings:

```
$ cat client-creds/client_ssl.properties
security.protocol=SSL
ssl.truststore.location=client-creds/kafka.client.truststore.jks
ssl.truststore.password=confluent
ssl.keystore.location=client-creds/kafka.client.keystore.jks
ssl.keystore.password=confluent
```

5. Start the console Consumer for Topic `ssl-topic` again, this time passing in the updated `client_ssl.properties` file. Now it should succeed and you should see the messages you typed earlier:

```
$ cd ~/confluent-admin/secure-cluster
$ kafka-console-consumer \
    --consumer.config client-creds/client_ssl.properties \
    --from-beginning \
    --topic ssl-topic \
    --bootstrap-server kafka-1:19093,kafka-2:29093,kafka-3:39093

Security is good
Certificates
Keys
```

Press **Ctrl+C** to quit the Consumer once your messages have been displayed.

SSL Performance Impact

In this section, you will run two performance tests and compare the results:

- Connecting to the cluster with no SSL via the configured **PLAINTEXT** port (19092/29092)
- Connecting to the cluster with SSL via the configured **SSL** port (19093/29093)

1. Create a new Topic called **no-ssl-topic** with one Partition and three replicas.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic no-ssl-topic \
    --partitions 1 \
    --replication-factor 3
```

2. Produce a high rate of data to Topic **no-ssl-topic** without SSL. Notice that the command line doesn't include **client_ssl.properties** nor configures **security.protocol** so it's using the **PLAINTEXT** listener.

```
$ kafka-producer-perf-test \
    --topic no-ssl-topic \
    --num-records 400000 \
    --record-size 1000 \
    --throughput 1000000 \
    --producer-props bootstrap.servers=$BOOTSTRAPS

78367 records sent, 15673.4 records/sec (14.95 MB/sec), 1281.7 ms avg
latency, 1903.0 ms max latency.
130690 records sent, 26138.0 records/sec (24.93 MB/sec), 1368.9 ms
avg latency, 1893.0 ms max latency.
161120 records sent, 32172.5 records/sec (30.68 MB/sec), 1007.9 ms
avg latency, 1069.0 ms max latency.
400000 records sent, 25257.308834 records/sec (24.09 MB/sec), 1180.49
ms avg latency, 1903.00 ms max latency, 1045 ms 50th, 1711 ms 95th,
1821 ms 99th, 1901 ms 99.9th.
```

Performance results will vary.

a. What is the average throughput? Look in the last line for the value associated with **`MB/sec`**.

b. What is the average latency? Look in the last line for the value associated with **`ms avg latency`**.

3. Now produce a high rate of data to Topic **`ssl-topic`** with SSL. Notice that you now connect to the SSL port because that's what is configured in **`client_ssl.properties`**. Bootstrap servers also need to go to the SSL ports because bootstrapping uses the same security protocol as the client configuration.

```
$ kafka-producer-perf-test \
    --topic ssl-topic \
    --num-records 400000 \
    --record-size 1000 \
    --throughput 1000000 \
    --producer-props bootstrap.servers=kafka-1:19093,kafka-
2:29093,kafka-3:39093 \
    --producer.config client-creds/client_ssl.properties

42481 records sent, 8496.2 records/sec (8.10 MB/sec), 1956.4 ms avg
latency, 2942.0 ms max latency.
85632 records sent, 17116.1 records/sec (16.32 MB/sec), 2021.4 ms avg
latency, 2463.0 ms max latency.
140496 records sent, 28076.7 records/sec (26.78 MB/sec), 1226.5 ms
avg latency, 1730.0 ms max latency.
400000 records sent, 21007.300037 records/sec (20.03 MB/sec), 1408.21
ms avg latency, 2942.00 ms max latency, 1139 ms 50th, 2317 ms 95th,
2817 ms 99th, 2934 ms 99.9th.
```

Performance results will vary.

a. What is the average throughput? Look in the last line for the value associated with `MB/sec`. Is this higher or lower than without SSL?

b. What is the average latency? Look for the value associated with `ms avg latency`. Is this higher or lower than without SSL?

Cleanup

1. Execute the following command to completely clean up your environment:

```
$ cd ~/confluent-admin/secure-cluster
$ docker-compose down -v
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 10 Data Pipelines with Kafka Connect

a. Running Kafka Connect

In this exercise, you will run Connect in distributed mode, and use the JDBC source Connector and File sink Connector. You will configure monitors using the REST API as well as the Control Center UI. You will use Control Center to monitor the connectors as well.

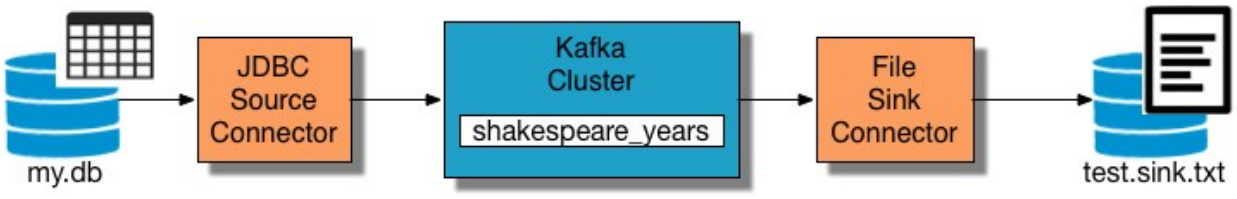
Prerequisites

1. Please make sure you have prepared your environment by following → Preparing the Labs
2. Start the Kafka cluster:

```
$ cd ~/confluent-admin
$ docker-compose up -d
```


Connect Pipeline

In this section, you will run Connect in distributed mode with two Connectors: a JDBC source Connector and a file sink Connector. The JDBC source Connector writes the contents of a database table to a Kafka Topic. The file sink Connector reads data from the same Kafka Topic and writes those messages to a file. It will update when new rows are added to the database.



Prerequisites

1. Create the SQLite database `my.db` in the folder `~/confluent-admin/data`: (You may have to install SQLite - first command below)

```
$ sudo apt install sqlite3
$ sqlite3 data/my.db
```

2. Run the following statements in `sqlite>` :

```
create table years(id INTEGER PRIMARY KEY AUTOINCREMENT, name
VARCHAR(50), year INTEGER);
insert into years(name,year) values('Hamlet',1600);
insert into years(name,year) values('Julius Caesar',1599);
insert into years(name,year) values('Macbeth',1605);
insert into years(name,year) values('Merchant of Venice',1595);
insert into years(name,year) values('Othello',1604);
insert into years(name,year) values('Romeo and Juliette',1594);
insert into years(name,year) values('Anthony and Cleopatra',1606);
```

3. Make sure the data is there:

```
sqlite> SELECT * FROM years;

1|Hamlet|1600
2|Julius Caesar|1599
3|Macbeth|1605
4|Merchant of Venice|1595
5|Othello|1604
6|Romeo and Juliette|1594
7|Anthony and Cleopatra|1606
```


4. Type `.quit` to exit SQLite.

5. Create a new Topic called `shakespeare-years` with one Partition and one replica.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --create \
    --topic shakespeare-years \
    --partitions 1 \
    --replication-factor 1
```

Install the Kafka Connect JDBC Connector

We use the Kafka Connect JDBC connector in this exercise, so we need to install it on the worker.

1. Install the connector:

```
$ docker-compose exec -u root kafka-connect confluent-hub install
--component-dir  confluentinc/kafka-connect-jdbc:10.7.4

The component can be installed in any of the following Confluent
Platform installations:
  1. / (installed rpm/deb package)
  2. / (where this tool is installed)
Choose one of these to continue the installation (1-2):
Do you want to install this into /usr/share/confluent-hub-components?
(yN)
```

2. At the prompts above, choose **1** (and **Enter**) then **Y** (and **Enter**).

3. At the prompt, type **y** and press **Enter**.

```
Component's license:
Confluent Community License
https://www.confluent.io/confluent-community-license
I agree to the software license agreement (yN)
```

4. At the prompt, type **y** and press **Enter**.

```
Downloading component Kafka Connect JDBC 10.7.4, provided by
Confluent, Inc. from Confluent Hub and installing into
/usr/share/confluent-hub-components
Detected Worker's configs:
  1. Standard: /etc/kafka/connect-distributed.properties
  2. Standard: /etc/kafka/connect-standalone.properties
  3. Standard: /etc/schema-registry/connect-avro-
distributed.properties
  4. Standard: /etc/schema-registry/connect-avro-
standalone.properties
  5. Used by Connect process with PID : /etc/kafka-connect/kafka-
connect.properties
Do you want to update all detected configs? (yN)
```

The installation completes.

```
Adding installation directory to plugin path in the following files:
  /etc/kafka/connect-distributed.properties
  /etc/kafka/connect-standalone.properties
  /etc/schema-registry/connect-avro-distributed.properties
  /etc/schema-registry/connect-avro-standalone.properties
  /etc/kafka-connect/kafka-connect.properties

Completed
```

5. To complete the installation, we need to restart the `kafka-connect` container:

```
$ docker-compose restart kafka-connect
```

6. Verify that the Connect Worker successfully restarted prior to continuing to the next step:

```
$ docker-compose logs kafka-connect | grep -i "INFO .* Finished
starting connectors and tasks"

kafka-connect  | [2024-01-21 22:00:37,065] INFO [Worker
clientId=connect-1, groupId=kafka-connect] Finished starting
connectors and tasks
(org.apache.kafka.connect.runtime.distributed.DistributedHerder)
```


If the message doesn't appear, wait and repeat this command until it does.

Configuring the Source Connector

1. Add a new source connector to read data from the database `my.db` and write to the Kafka Topic `shakespeare-years` by using the Confluent Center:

a. Open Control Center at http://localhost:9021:

b. Select the `controlecenter.cluster` cluster

c. In the sidebar click **Connect**

d. In the **Connect Clusters** view select the (only available) entry **connect**

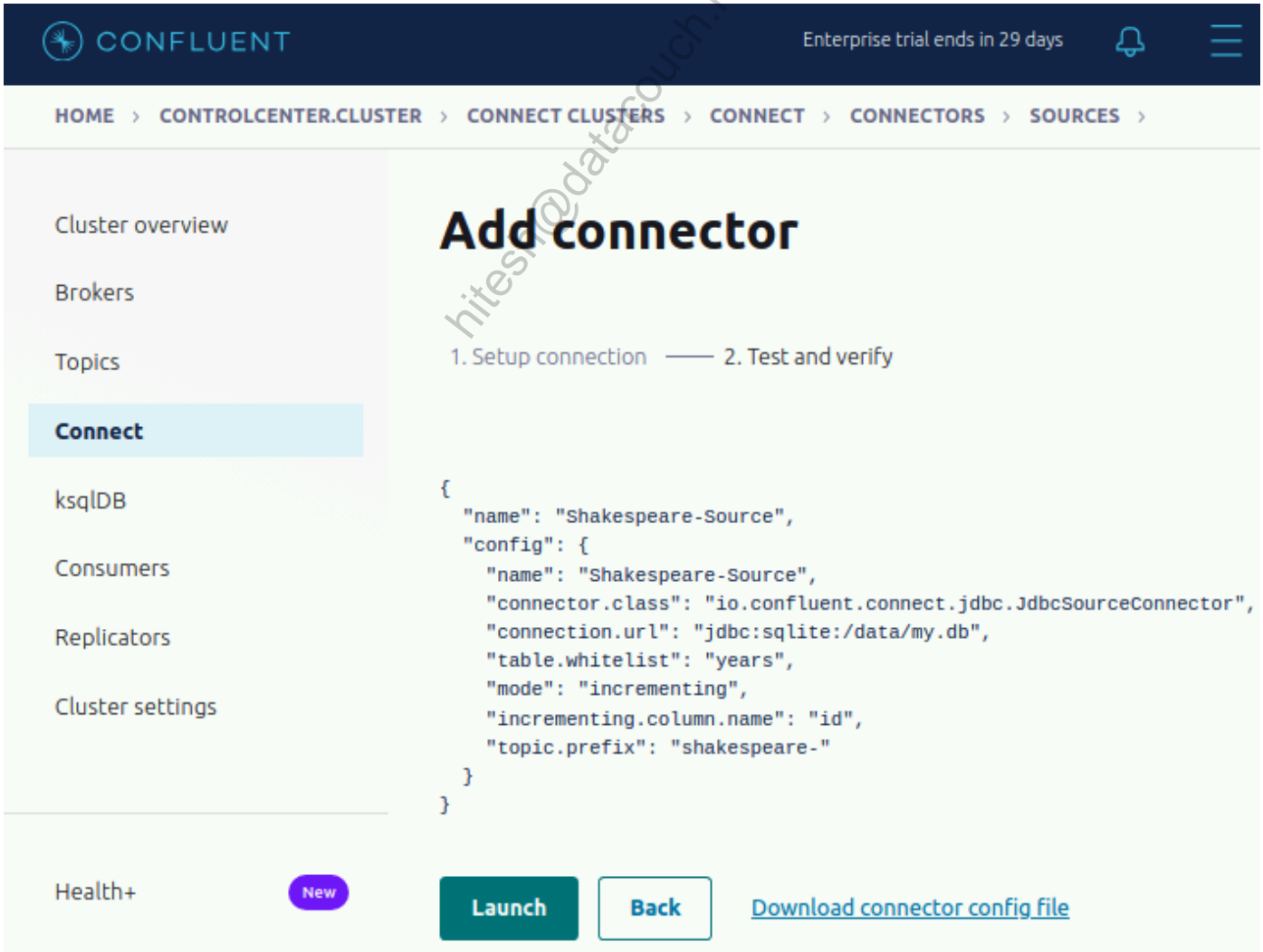
e. Click **Add connector**

f. In the **Browse** overview select the **JdbcSourceConnector Source** tile

g. Configure the JDBC Source Connector

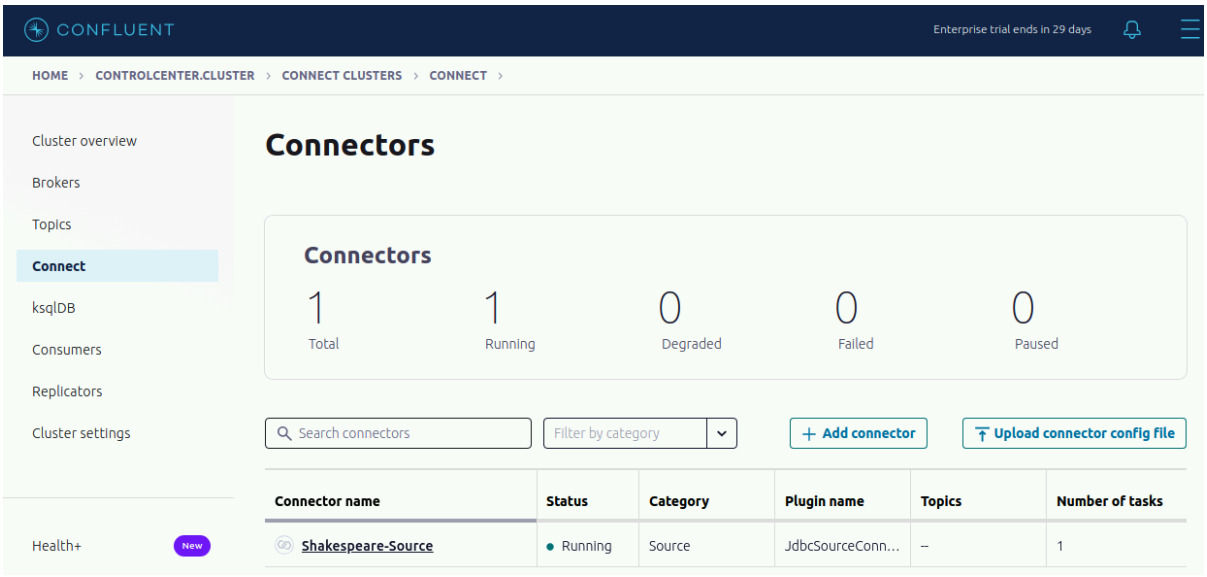
Section	Field	Value
	Name	Shakespeare-Source
Database	JDBC URL	jdbc:sqlite:/data/my.db
Database	Table Whitelist	years
Mode	Table Loading Mode	incrementing
Mode	Incrementing Column Name	id
Connector	Topic Prefix	shakespeare-

h. Click **Continue**



i. Click **Launch**

j. Verify you see the new connector **JDBC-Source-Connector** running



k. **Optional**: As an alternative to using Control Center, you could instead add the source connector via command line using Kafka Connect's **REST API**:


```
$ curl -s -X POST \
      -H "Content-Type: application/json" \
      --data '{
          "name": "Shakespeare-Source",
          "config": {
              "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
              "connection.url": "jdbc:sqlite:/data/my.db",
              "table.whitelist": "years",
              "mode": "incrementing",
              "incrementing.column.name": "id",
              "topic.prefix": "shakespeare-"
          }
      }' http://connect:8083/connectors
```

1. Launch another terminal and start the console Consumer for Topic `shakespeare-years`:

```
$ kafka-avro-console-consumer \
    --bootstrap-server $BOOTSTRAPS \
    --property schema.registry.url=http://schema-registry:8081 \
    --from-beginning \
    --topic shakespeare-years

{"id":1,"name":{"string":"Hamlet"},"year":{"long":1600}}
{"id":2,"name":{"string":"Julius Caesar"},"year":{"long":1599}}
{"id":3,"name":{"string":"Macbeth"},"year":{"long":1605}}
{"id":4,"name":{"string":"Merchant of
Venice"},"year":{"long":1595}}
{"id":5,"name":{"string":"Othello"},"year":{"long":1604}}
{"id":6,"name":{"string":"Romeo and
Juliette"},"year":{"long":1594}}
{"id":7,"name":{"string":"Anthony and
Cleopatra"},"year":{"long":1606}}
...
```

Leave the Consumer running until instructed to terminate it. See what messages have been and will be produced.

Configuring the Sink Connector

1. Open a second terminal window and navigate to the **confluent-admin** folder:

```
$ cd ~/confluent-admin
```

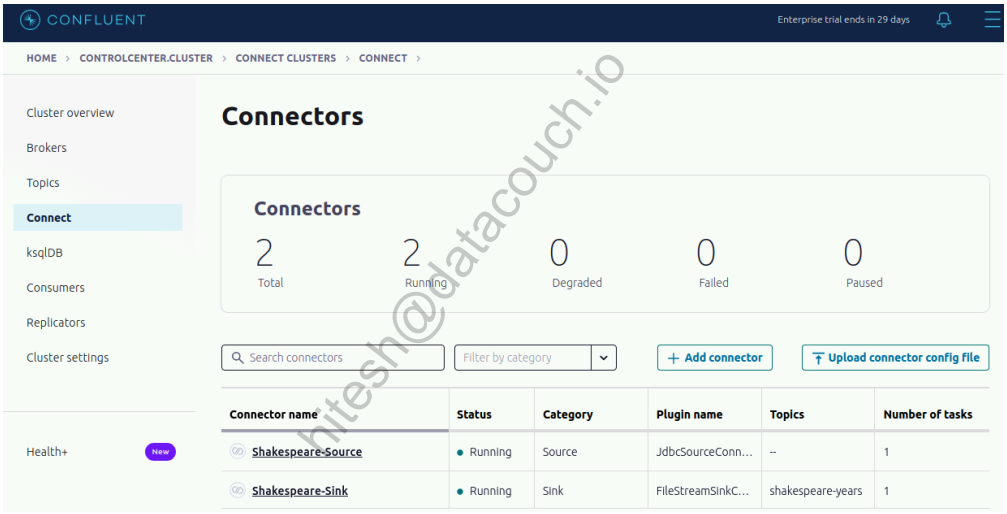
2. Update permissions for the **data** directory to allow access by the **connect** container:

```
$ chmod 777 data
```

3. Add a new sink connector to read data from the Kafka Topic **shakespeare-years** and write to the file **data/test.sink.txt** on the Connect worker system:

```
$ curl -s -X POST \
        -H "Content-Type: application/json" \
        --data '{
            "name": "Shakespeare-Sink",
            "config": {
                "topics": "shakespeare-years",
                "connector.class":
"org.apache.kafka.connect.file.FileStreamSinkConnector",
                "value.converter":
"io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url":
"http://schema-registry:8081",
                "file": "/data/test.sink.txt"
            }
        }' http://connect:8083/connectors
```

4. Use Control Center to verify the creation of the connector.



5. Verify that a new file called `test.sink.txt` has been created in the folder `~/confluent-admin/data`. View this file to confirm that the sink connector worked:

```
$ cat data/test.sink.txt

Struct{id=1,name=Hamlet,year=1600}
Struct{id=2,name=Julius Caesar,year=1599}
Struct{id=3,name=Macbeth,year=1605}
Struct{id=4,name=Merchant of Venice,year=1596}
Struct{id=5,name=Othello,year=1604}
Struct{id=6,name=Romeo and Juliet,year=1594}
Struct{id=7,name=Antony and Cleopatra,year=1606}
```

This file was created by the **File Sink Connector** in folder `/data/` on the `connect` container. Since this container folder is mapped to the folder `~/confluent-admin/data` of your host system, we can see it there too.

6. Insert a few new rows into the table `years`:

   a. From your host, run SQLite3:

   ```
   $ sqlite3 data/my.db
   ```

   b. Insert two records:

   ```
   INSERT INTO years(name,year) VALUES('Tempest',1611);
   INSERT INTO years(name,year) VALUES('King Lear',1605);
   ```

   c. In the window where the Consumer is still running observe that two new records have been output:

   ```
   {"id":8,"name":{"string":"Tempest"},"year":{"long":1611}}
   {"id":9,"name":{"string":"King Lear"},"year":{"long":1605}}
   ```

   d. Quit SQLite3 by typing `.quit`.

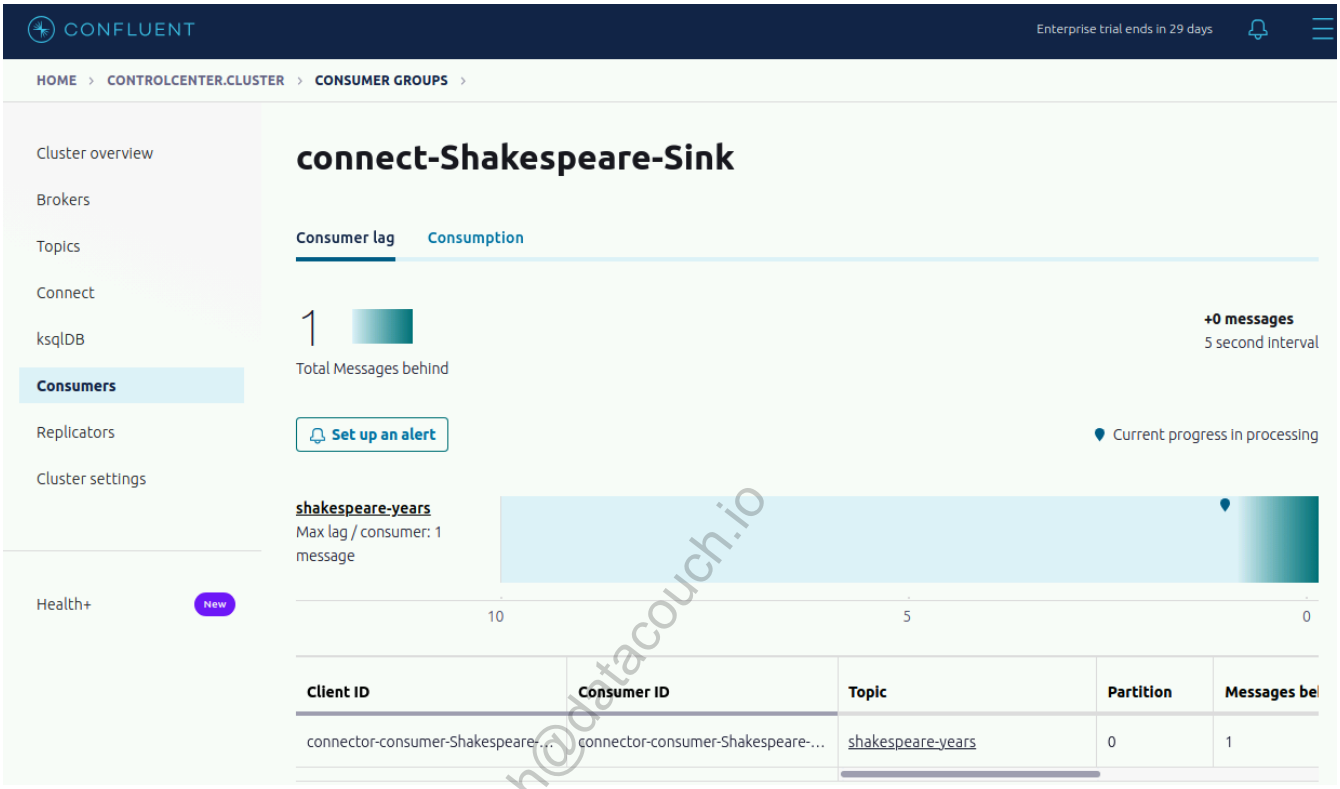
7. View the sink file, `test.sink.txt` again and notice that the new log lines are added to it:

```
$ cat data/test.sink.txt

Struct{id=1,name=Hamlet,year=1600}
Struct{id=2,name=Julius Caesar,year=1599}
Struct{id=3,name=Macbeth,year=1605}
Struct{id=4,name=Merchant of Venice,year=1596}
Struct{id=5,name=Othello,year=1604}
Struct{id=6,name=Romeo and Juliet,year=1594}
Struct{id=7,name=Antony and Cleopatra,year=1606}
Struct{id=8,name=Tempest,year=1611}
Struct{id=9,name=King Lear,year=1605}
```

8. In Control Center, observe the Consumer Group performance for Kafka Connect. You could use this to ensure that Kafka Connect is performing well in your cluster, just like other production traffic.

a. In Control Center, make sure that you have selected **controlcenter.cluster** and then click on **Consumers**

b. In the list of Consumer Groups select **connect-Shakespeare-Sink**. You can select it to see the **Consumer lag** view



9. In the terminal window where the consumer is running, press `Ctrl+C` to terminate the process.

Cleanup

1. Execute the following command to completely clean up your environment.

```
$ docker-compose down -v
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix A: Reassigning Partitions in a Topic - Alternate Method

In the **Kafka Administrative Tools** exercise, one of the sections uses the Confluent Auto Data Balancer to rebalance the cluster and to reassign Partitions for the Topic `moving`. If you prefer to use the Apache open source tool `kafka-reassign-partitions` instead, you may follow the instructions below.

This appendix assumes that the Topic `moving` exists and has been populated as described in the **Kafka Administrative Tools** exercise. If this is not the case then please first follow the instructions → Rebalancing the Cluster.

1. Change to the `data` folder:

```
$ cd ~/confluent-admin/data
```

2. In that folder create a file `topics-to-move.json` as follows:

```
$ echo '{"topics": [{"topic": "moving"}],"version":1}' > topics-to-
move.json
```

3. Generate the reassignment plan.

```
$ kafka-reassign-partitions \
    --bootstrap-server $BOOTSTRAPS \
    --topics-to-move-json-file topics-to-move.json \
    --broker-list "1,2,3" \
    --generate > reassignment.json
```

The content of the file `reassignment.json` should look like this:


```
$ cat reassignment.json

Current partition replica assignment
{"version":1,"partitions":[{"topic":"moving","partition":0,"replicas"
:[1,2],"log_dirs":["any","any"]},{"topic":"moving","partition":1,"rep
licas":[2,1],"log_dirs":["any","any"]},{"topic":"moving","partition":
2,"replicas":[1,2],"log_dirs":["any","any"]},{"topic":"moving","parti
tion":3,"replicas":[2,1],"log_dirs":["any","any"]},{"topic":"moving",
"partition":4,"replicas":[1,2],"log_dirs":["any","any"]},{"topic":"mo
ving","partition":5,"replicas":[2,1],"log_dirs":["any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"moving","partition":0,"replicas"
:[3,1],"log_dirs":["any","any"]},{"topic":"moving","partition":1,"rep
licas":[1,2],"log_dirs":["any","any"]},{"topic":"moving","partition":
2,"replicas":[2,3],"log_dirs":["any","any"]},{"topic":"moving","parti
tion":3,"replicas":[3,2],"log_dirs":["any","any"]},{"topic":"moving",
"partition":4,"replicas":[1,3],"log_dirs":["any","any"]},{"topic":"mo
ving","partition":5,"replicas":[2,1],"log_dirs":["any","any"]}]}
```

4. Open `reassignment.json` and edit it so it only includes the "Proposed partition reassignment configuration" in JSON (i.e., just the last line).

   The resulting file should look like this:

```
$ cat reassignment.json

{"version":1,"partitions":[{"topic":"moving","partition":0,"replicas"
:[3,1],"log_dirs":["any","any"]},{"topic":"moving","partition":1,"rep
licas":[1,2],"log_dirs":["any","any"]},{"topic":"moving","partition":
2,"replicas":[2,3],"log_dirs":["any","any"]},{"topic":"moving","parti
tion":3,"replicas":[3,2],"log_dirs":["any","any"]},{"topic":"moving",
"partition":4,"replicas":[1,3],"log_dirs":["any","any"]},{"topic":"mo
ving","partition":5,"replicas":[2,1],"log_dirs":["any","any"]}]}
```

5. Observe the current throttling limits configured:

```
$ kafka-configs \
    --describe \
    --bootstrap-server $BOOTSTRAPS \
    --entity-type brokers
Dynamic configs for broker 1 are:
Dynamic configs for broker 2 are:
Dynamic configs for broker 3 are:
Default configs for brokers in the cluster are:
```


kafka-configs by default only reports dynamic configuration settings. Since none have been set at this time, the dynamic configs list for each broker should be empty.

6. Execute the reassignment plan with throttling set to 1MBps.

```
$ kafka-reassign-partitions \
    --bootstrap-server $BOOTSTRAPS \
    --reassignment-json-file reassignment.json \
    --execute \
    --throttle 1000000

Current partition replica assignment

{"version":1,"partitions":[{"topic":"moving","partition":0,"replicas"
:[1,2],"log_dirs":["any","any"]},{"topic":"moving","partition":1,"rep
licas":[2,1],"log_dirs":["any","any"]},{"topic":"moving","partition":
2,"replicas":[1,2],"log_dirs":["any","any"]},{"topic":"moving","parti
tion":3,"replicas":[2,1],"log_dirs":["any","any"]},{"topic":"moving",
"partition":4,"replicas":[1,2],"log_dirs":["any","any"]},{"topic":"mo
ving","partition":5,"replicas":[2,1],"log_dirs":["any","any"]}]}

Save this to use as the --reassignment-json-file option during
rollback
Warning: You must run --verify periodically, until the reassignment
completes, to ensure the throttle is removed.
The inter-broker throttle limit was set to 1000000 B/s
Successfully started partition reassignments for moving-0,moving-
1,moving-2,moving-3,moving-4,moving-5
```

7. Observe the new throttling limits configured.

```
$ kafka-configs \
    --describe \
    --bootstrap-server $BOOTSTRAPS \
    --entity-type brokers

Dynamic configs for broker 1 are:
  follower.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Dynamic configs for broker 2 are:
  follower.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Dynamic configs for broker 3 are:
  follower.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:follower.replication.throttled.rate=1
000000,
DEFAULT_CONFIG:follower.replication.throttled.rate=922337203685477580
7}
  leader.replication.throttled.rate=1000000 sensitive=false
synonyms={DYNAMIC_BROKER_CONFIG:leader.replication.throttled.rate=100
0000,
DEFAULT_CONFIG:leader.replication.throttled.rate=9223372036854775807}
Default configs for brokers in the cluster are:
```


8. Monitor the progress of the reassignment.

   a. Run the **`kafka-reassign-partitions`** command with the **`--verify`** option and note that some reassignments are “still in progress”.

```
$ kafka-reassign-partitions \
    --bootstrap-server $BOOTSTRAPS \
    --reassignment-json-file reassignment.json \
    --verify

Status of partition reassignment:
Reassignment of partition moving-0 is still in progress.
Reassignment of partition moving-1 is completed.
Reassignment of partition moving-2 is still in progress.
Reassignment of partition moving-3 is still in progress.
Reassignment of partition moving-4 is still in progress.
Reassignment of partition moving-5 is completed.
```

b. Run the **kafka-topics** command with the **--describe** option and notice the listed replicas for the Partitions that are still in progress for reassignment.

```
$ kafka-topics \
    --bootstrap-server $BOOTSTRAPS \
    --describe \
    --topic moving

Topic: moving   TopicId: YfY2SoNWRG6JmYrpjU5nBQ PartitionCount: 6
ReplicationFactor: 2    Configs:
leader.replication.throttled.replicas=0:1,0:2,1:1,1:2,2:1,2:2,3:1,
3:2,4:1,4:2,5:1,5:2,follower.replication.throttled.replicas=0:3,2:
3,3:3,4:3
    Topic: moving   Partition: 0    Leader: 1   Replicas: 3,1,2
Isr: 1,2Offline:    Adding Replicas: 3  Removing Replicas: 2
    Topic: moving   Partition: 1    Leader: 2   Replicas: 1,2
Isr: 2,1Offline:
    Topic: moving   Partition: 2    Leader: 1   Replicas: 2,3,1
Isr: 1,2Offline:    Adding Replicas: 3  Removing Replicas: 1
    Topic: moving   Partition: 3    Leader: 2   Replicas: 3,2,1
Isr: 2,1Offline:    Adding Replicas: 3  Removing Replicas: 1
    Topic: moving   Partition: 4    Leader: 1   Replicas: 1,3,2
Isr: 1,2Offline:    Adding Replicas: 3  Removing Replicas: 2
    Topic: moving   Partition: 5    Leader: 2   Replicas: 2,1
Isr: 2,1Offline:
```


9. Increase the throttle limit configuration to 1GBps by rerunning the **kafka-reassign-partitions** command with the new throttle limit.

```
$ kafka-reassign-partitions \
    --bootstrap-server $BOOTSTRAPS \
    --reassignment-json-file reassignment.json \
    --execute \
    --throttle 1000000000 \
    --additional

Current partition replica assignment

{"version":1,"partitions":[{"topic":"moving","partition":0,"replicas"
:[3,1,2],"log_dirs":["any","any","any"]},{"topic":"moving","partition
":1,"replicas":[1,2],"log_dirs":["any","any"]},{"topic":"moving","par
tition":2,"replicas":[2,3,1],"log_dirs":["any","any","any"]},{"topic"
:"moving","partition":3,"replicas":[3,2,1],"log_dirs":["any","any","a
ny"]},{"topic":"moving","partition":4,"replicas":[1,3,2],"log_dirs":[
"any","any","any"]},{"topic":"moving","partition":5,"replicas":[2,1],
"log_dirs":["any","any"]}]}

Save this to use as the --reassignment-json-file option during
rollback
Warning: You must run --verify periodically, until the reassignment
completes, to ensure the throttle is removed.
The inter-broker throttle limit was set to 1000000000 B/s
Successfully started partition reassignments for moving-0,moving-
1,moving-2,moving-3,moving-4,moving-5
```

10. Run the **kafka-reassign-partitions** command with the **--verify** option again. Note the "completed successfully" output and "Throttle was removed".

```
$ kafka-reassign-partitions \
    --bootstrap-server $BOOTSTRAPS \
    --reassignment-json-file reassignment.json \
    --verify

Status of partition reassignment:
Reassignment of partition moving-0 is completed.
Reassignment of partition moving-1 is completed.
Reassignment of partition moving-2 is completed.
Reassignment of partition moving-3 is completed.
Reassignment of partition moving-4 is completed.
Reassignment of partition moving-5 is completed.
```

11. Confirm that the throttle limit is now removed.

```
$ kafka-configs \
    --describe \
    --bootstrap-server $BOOTSTRAPS\
    --entity-type brokers
Dynamic configs for broker 1 are:
Dynamic configs for broker 2 are:
Dynamic configs for broker 3 are:
Default configs for brokers in the cluster are:
```

12. Return to (the end of) the exercise **Rebalancing the Cluster** (→ Simulate a Completely Failed Broker).

STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix B: Running Labs in Docker for Desktop

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine you are able to complete the course by building and running your applications from the command line.

- Increase the memory available to Docker Desktop to a minimum of 8 GiB. See the advanced settings for Docker Desktop for Mac, and Docker Desktop for Windows.
- Follow the instructions at → Preparing the Labs to `git clone` the source code.
- Edit `docker-compose.yml` and uncomment all the lines of the `tools` section (remove the `#` at the beginning of each line)
- Launch the cluster containers with `docker-compose`.
- In each exercise, open a bash shell in the **tools container** to run commands against the cluster. All the command line instructions will work from the `tools` container. This container has been preconfigured with all the tools you use in the exercises, e.g. `kafka-topics`, `confluent-rebalancer`, and `kafka-configs.` You can think of this container as a "bastion" inside your distributed architecture.


```
$ docker-compose exec tools bash
root@tools:#
```

The `~/confluent-admin/data` folder is mapped to `/apps/data` in the `tools` container, so any directions that use `~/confluent-admin/data` should be executed from `/apps/data` inside the `tools` container.

- Anywhere you are instructed to open additional terminal windows, you can open additional bash shells on the `tools` container with the same command as above on your host machine.
- Any subsequent `docker` or `docker-compose` instructions should be run on your host machine.