Confluent Advanced Skills for Optimizing Apache Kafka®

Exercise Book

Version 6.0.0-v2.0.1

CONFLUENT

Table of Contents

Copyright & Trademarks

Lab 03 Generic Monitoring

a. Introduction

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

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 6.0.0 which includes Kafka 2.6.0.

!	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-18.04-may2020.ova

Command Line Examples

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

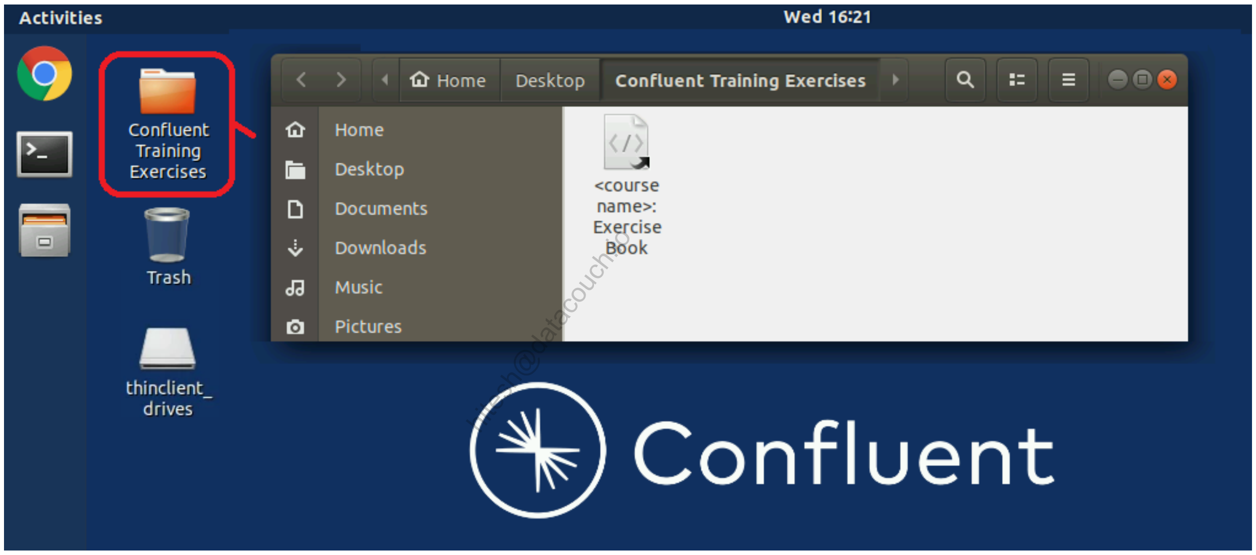
```
$ pwd
/home/training
```

Commands you should type are shown in **bold**; non-bold text is an example of the output produced as a result of the command.

Preparing the Labs

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-cao` in your **home** directory:

```
$ git clone --branch 6.0.0-v2.0.1 \
    https://github.com/confluentinc/training-cao-src.git \
    ~/confluent-cao
```

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

b. Monitoring via JMX

In this exercises we are going to monitor metrics of all involved components in our streaming platform that are exported via the Java Management Extensions (JMX).

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/mo-gen/jmx
```

2. Run the Kafka cluster, including Confluent Control Center:

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

3. Create the topic `cc-authorization-topic` with 6 partitions on Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic cc-authorization-topic
Created topic "cc-authorization-topic".
```

4. Build the sample consumer and producer service from source code:

```
$ docker-compose -f docker-compose-clients.yml build
```

This may take a while. Please be patient.

5. Run the two Kafka client services, producer and consumer:

```
$ docker-compose -f docker-compose-clients.yml \
    -p jmx up -d
```

You should see an output similar to this (you can safely ignore the warning):

```
WARNING: Found orphan containers (ksqldb-server, connect, control-
center, rest-proxy, kafka, schema-registry, zookeeper) for this
project. If you removed or renamed this service in your compose file,
you can run this command with the --remove-orphans flag to clean it
up.
Creating consumer ... done
Creating producer ... done
```

You might want to have a look into the source code of the producer and consumer.

- The producer `producer/src/main/java/clients/AuthorizationProducer.java` simulates a stream of credit card authorizations against a set of 10000 credit cards. Randomly there are some failed attempts, sometimes in a short sequence. The authorization attempts are written to the topic `cc-authorization-topic`
- The consumer `consumer/src/main/java/clients/FraudConsumer.java` consumes all events from the topic `CC_POTENTIAL_FRAUD_COUNTS` and writes them to the console. Each event represents a discovered attempt for fraud. The latter topic is fed by the ksqlDB queries that we will define in the next section.

6. Verify all Docker containers are running:

```
$ docker ps
CONTAINER ID    ...    STATUS                   PORTS
NAMES
ca57e0b2f376    ...    Up 21 seconds            0.0.0.0:4445-
>4445/tcp                                                 consumer
eaafc19d5e68    ...    Up 21 seconds            0.0.0.0:4444-
>4444/tcp                                                 producer
ef151b8e7df1    ...    Up 6 minutes             0.0.0.0:9092-
>9092/tcp, 0.0.0.0:49999->49999/tcp                       kafka
8c4fe0a49f5b    ...    Up 6 minutes             0.0.0.0:8088-
>8088/tcp, 0.0.0.0:59998->59998/tcp                       ksqldb-
server
110736241d13    ...    Up 6 minutes             0.0.0.0:9021-
>9021/tcp                                                 control-
center
e9c367f33ea1    ...    Up 6 minutes             2888/tcp,
0.0.0.0:2181->2181/tcp, 0.0.0.0:39999->39999/tcp, 3888/tcp
zookeeper
3c96839222bd    ...    Up 6 minutes (healthy)   0.0.0.0:8083-
>8083/tcp, 0.0.0.0:59997->59997/tcp, 9092/tcp             connect
6ca2a7b41af5    ...    Up 6 minutes             0.0.0.0:8081-
>8081/tcp, 0.0.0.0:59999->59999/tcp                       schema-
registry
```

If any containers are not running, run step 2 or step 5 again needed.

Create ksqlDB Queries

You will now create ksqlDB queries whose JMX metrics can be examined in the section that follows. These queries will do the following:

- Create a stream `cc_authorizations` from the topic `cc-authorization-topic`
- Create a table `CC_POTENTIAL_FRAUD` that contains the filtered malicious authorization attempts (the formula is of course overly simplified)
- Convert the table `CC_POTENTIAL_FRAUD` back to a stream `CC_POTENTIAL_FRAUD_STREAM` (this stream represents the change log of the table)
- Create a stream `CC_POTENTIAL_FRAUD_STREAM` which filters out the relevant event from the change log stream (where ROWTIME IS NOT NULL). At the same time this will create an underlying topic in Kafka that the consumer will then consume.

1. Run the ksqlDB CLI:

```
$ ksql http://ksqldb-server:8088
```

2. Execute the following DDL scripts in ksqlDB CLI:

```
ksql> CREATE STREAM cc_authorizations(
    credit_card_tkn STRING,
    auth_time BIGINT,
    status STRING)
    WITH(KAFKA_TOPIC='cc-authorization-topic',
         VALUE_FORMAT='AVRO');
```

```
ksql> CREATE TABLE CC_POTENTIAL_FRAUD AS
    SELECT credit_card_tkn as rowkey,
           AS_VALUE(credit_card_tkn) as credit_card_tkn,
           COUNT( * ) attempts
    FROM cc_authorizations
    WINDOW TUMBLING (SIZE 3 SECONDS)
    WHERE status='FAILED'
    GROUP BY credit_card_tkn
    HAVING COUNT( * )>=3;
```

```
ksql> CREATE STREAM CC_POTENTIAL_FRAUD_STREAM (
    credit_card_tkn string, attempts bigint)
    WITH (kafka_topic='CC_POTENTIAL_FRAUD', value_format='AVRO');
```

```
ksql> CREATE STREAM CC_POTENTIAL_FRAUD_COUNTS
    WITH (PARTITIONS=3,REPLICAS=1) AS
    SELECT * FROM CC_POTENTIAL_FRAUD_STREAM WHERE ROWTIME IS NOT
NULL;
```

Technically, when using AVRO you would not have to specify the column names in a `CREATE STREAM/TABLE` query. But often it makes sense to explicitly state them for readability.

3. Exit the ksqlDB CLI by pressing `Ctrl+D`.

Examine JMX Metrics

1. Observe the JMX metrics of the producer:

   a. Open a `jconsole` connection to port `4444` which is the JMX port for the producer.

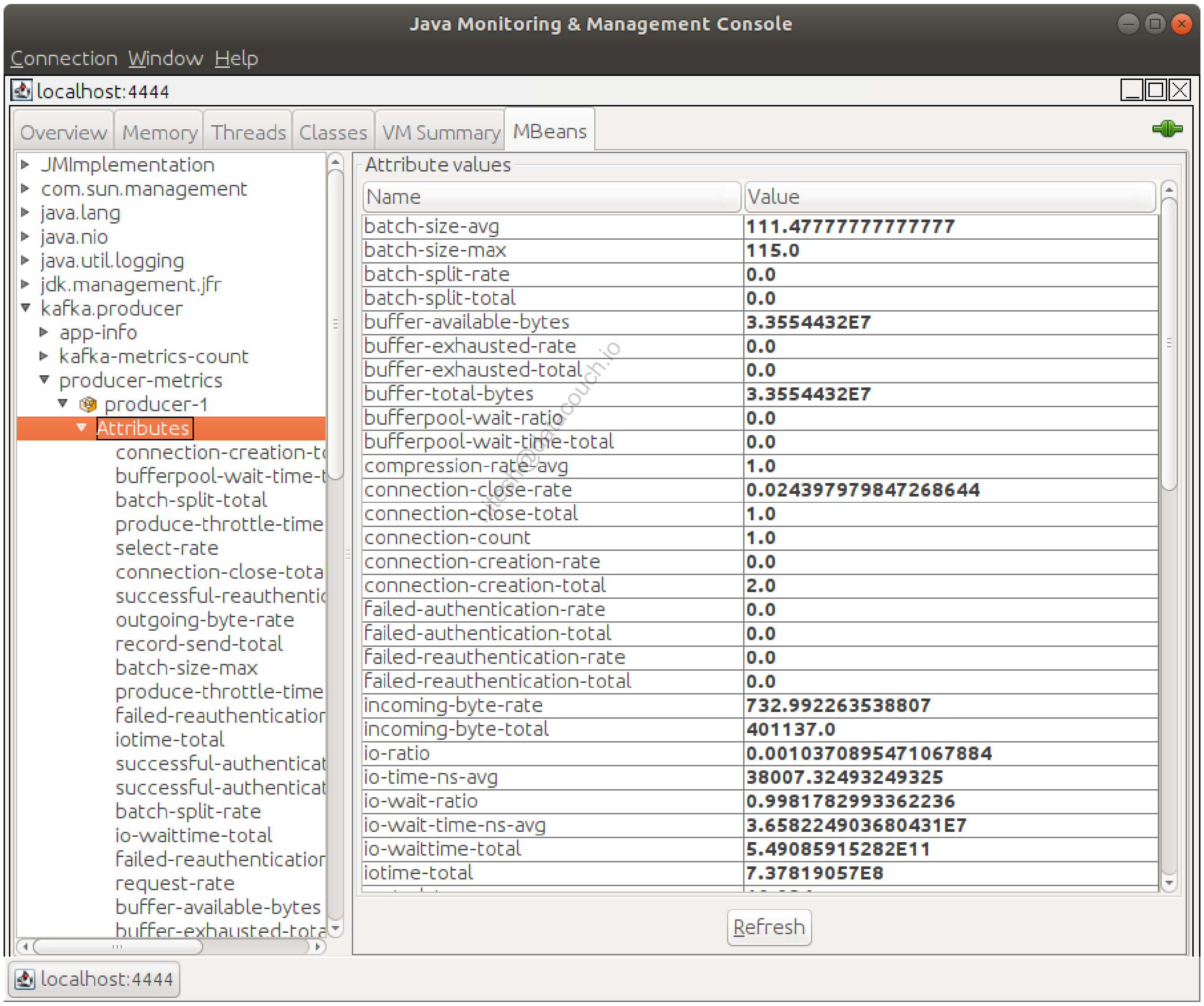
```
$ jconsole localhost:4444 &
```

b. Select **Insecure connection** when asked

c. Navigate to the **MBeans** tab

d. Locate the node called `kafka.producer` and navigate to `kafka.producer` → `producer-metrics` → `producer-1` → `Attributes`

e. Click **Refresh** occasionally to see how the numbers change



Name	Value
batch-size-avg	111.47777777777777
batch-size-max	115.0
batch-split-rate	0.0
batch-split-total	0.0
buffer-available-bytes	3.3554432E7
buffer-exhausted-rate	0.0
buffer-exhausted-total	0.0
buffer-total-bytes	3.3554432E7
bufferpool-wait-ratio	0.0
bufferpool-wait-time-total	0.0
compression-rate-avg	1.0
connection-close-rate	0.024397979847268644
connection-close-total	1.0
connection-count	1.0
connection-creation-rate	0.0
connection-creation-total	2.0
failed-authentication-rate	0.0
failed-authentication-total	0.0
failed-reauthentication-rate	0.0
failed-reauthentication-total	0.0
incoming-byte-rate	732.992263538807
incoming-byte-total	401137.0
io-ratio	0.0010370895471067884
io-time-ns-avg	38007.32493249325
io-wait-ratio	0.9981782993362236
io-wait-time-ns-avg	3.658224903680431E7
io-waittime-total	5.49085915282E11
iotime-total	7.37819057E8

f. Locate some of the metrics that have an immediate meaning to you

g. Double click the value in the **Value** column of any metric of interest to see a graph of it

2. Observe the JMX metrics of the consumer:

a. Open a `jconsole` connection to port `4445` which is the JMX port for the consumer.

```
$ jconsole localhost:4445 &
```

b. Select **Insecure connection** when asked

c. Navigate to the **MBeans** tab

d. Locate the node called `kafka.consumer` and drill down

e. Click **Refresh** occasionally to see how the numbers change

3. To observe the JMX metrics of ZooKeeper, do the same as above for port `39999`.

Drill down into the MBean `org.apache.ZooKeeperService` and notice that we expose `StandaloneServer_port2181` since we're running a single instance of ZooKeeper. If ZooKeeper is run as an ensemble of 3 or 5 instances then we would see nodes `ReplicatedServer_id<ID>` instead, where `<ID>` is the ID of the respective ZooKeeper instance in the ensemble.

4. To observe the JMX metrics of the Kafka (broker), do the same as above for port `49999`.

Drill down (at least) into the MBeans `kafka.log`, `kafka.network` and `kafka.server` and familiarize yourself with the immense amount of metrics exposed to you.

5. To observe the JMX metrics of ksqlDB Server, do the same as above for port `59998`.

Observe that there are the three MBeans `kafka.admin.client`, `kafka.consumer`, `kafka.producer` and `kafka.streams`. This indicates that the ksqlDB Server is using the Kafka client library internally to create producers and consumers. It also uses the Kafka Admin client and the Kafka streams library. Also note that there are some ksqlDB specific metrics at `io.confluent.ksql.metrics`.

6. To observe the JMX metrics of the Schema Registry, do the same as above for port

59999.

Observe that there are the three MBeans `kafka`, `kafka.consumer`, `kafka.producer` and `kafka.schema.registry`. This indicates that the Schema Registry is using the Kafka client library internally to create producers and consumers.

Cleanup

1. Close **JConsole**
2. Shut down the producer and consumer services:

```
$ docker-compose -f docker-compose-clients.yml -p jmx down -v
```

You can safely ignore the `WARNING:` and `ERROR:`. It is because the client services are using the same network as the rest of the Kafka cluster. Docker Compose tries to remove that network, which at this time is still in use.

3. Tear down the Kafka cluster and its data:

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

Summary

In this exercises you have monitored JMX metrics of all components - ZooKeeper, broker, ksqlDB server, Kafka producer, and Kafka consumer - of your streaming platform. You have used `jconsole` for this purpose.

c. Monitoring librdkafka based Clients

In this exercise we are showing how a `librdkafka` based Kafka client can be instrumented with Confluent monitoring interceptors. This will make those clients appear in Confluent Control Center such as that they can be monitored the same way any Java based client application can be monitored.

Monitoring interceptors for **Confluent Control Center** will be introduced in more detail in the next module **Monitoring with Confluent Control Center**.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/mo-gen/librdkafka/dotnet
```

2. Run the Kafka cluster, including Confluent Control Center:

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

3. Create the topic `test-topic` with 6 partitions on Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic test-topic
Created topic "test-topic".
```

4. Run the .NET based Kafka producer and consumer:

```
$ docker-compose -f docker-compose-clients.yml \
    -p librdkafka up -d
```

This may take a minute or so, since the images need to be downloaded from Docker Hub first.

You should see an output similar to this (you can safely ignore the warning):

```
WARNING: Found orphan containers (zookeeper, control-center, kafka,
ksqldb-server, schema-registry, connect) for this project. If you
removed or renamed this service in your compose file, you can run
this command with the --remove-orphans flag to clean it up.
Creating librdkafka_producer_1 ... done
Creating librdkafka_consumer_1 ... done
```

5. Make sure the 2 services are running:

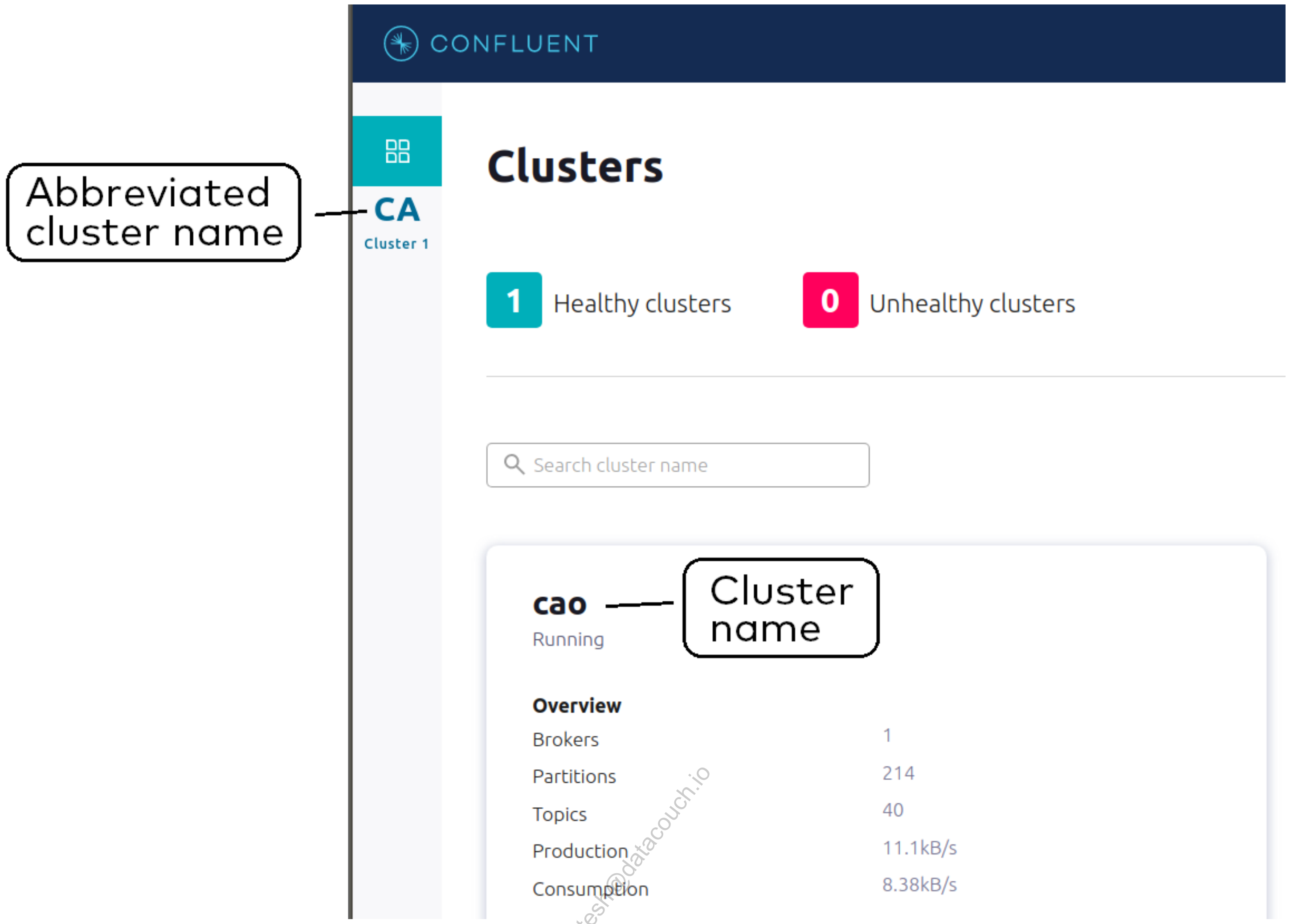
```
$ docker-compose -f docker-compose-clients.yml -p librdkafka ps
      Name                Command        State   Ports
--------------------------------------------------
librdkafka_consumer_1   dotnet app.dll   Up
librdkafka_producer_1   dotnet app.dll   Up
```

6. **OPTIONAL:** Analyze the code of the producer (`producer/Program.cs`) and consumer (`consumer/Program.cs`) and try to find out what they are exactly doing. Also refer to the section **Analyzing the Configuration** towards the end of this exercise.

Monitoring Consumer Lag

1. Open **Control Center** at http://localhost:9021

2. Wait until Control Center is initialized and the **Clusters** view displays the **cao** cluster with 1 broker running.

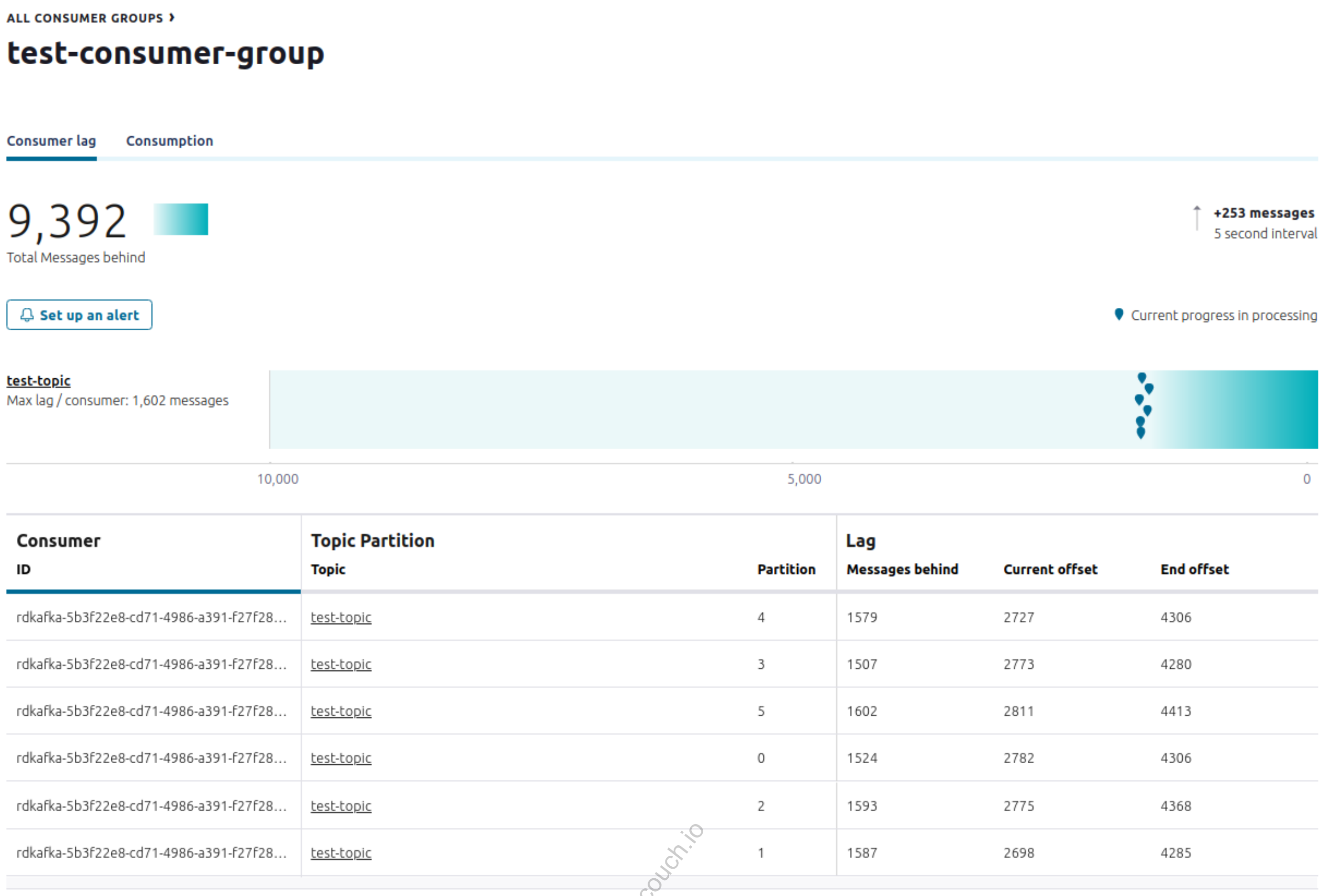


cao
Running

Overview

Brokers	1
Partitions	214
Topics	40
Production	11.1kB/s
Consumption	8.38kB/s

1. Click the **cao** cluster.
2. On the left hand side click the **Consumers** view option.
3. In the list of consumer groups select the item **test-consumer-group** to open the the **Consumer lag** tab of the consumer group view. You should see something like this:



Consumer	Topic Partition		Lag		
ID	Topic	Partition	Messages behind	Current offset	End offset
rdkafka-5b3f22e8-cd71-4986-a391-f27f28...	test-topic	4	1579	2727	4306
rdkafka-5b3f22e8-cd71-4986-a391-f27f28...	test-topic	3	1507	2773	4280
rdkafka-5b3f22e8-cd71-4986-a391-f27f28...	test-topic	5	1602	2811	4413
rdkafka-5b3f22e8-cd71-4986-a391-f27f28...	test-topic	0	1524	2782	4306
rdkafka-5b3f22e8-cd71-4986-a391-f27f28...	test-topic	2	1593	2775	4368
rdkafka-5b3f22e8-cd71-4986-a391-f27f28...	test-topic	1	1587	2698	4285

In the top you can see a graphical representation of the consumer lag for each of the 6 partitions of the topic. In the lower part you see the list of partitions and which consumer instance is handling the respective partition. At the moment there is only a single consumer instance. In my case the **Consumer Id** is `rdkafka-39e2a4b7-3ea5-4a4b-b531-c3467d037faa`

4. Please note that the consumer lag is ever growing. This is because the consumer has purposefully been coded to be slow. This gives us an opportunity to show how scaling the consumer provides more throughput.

5. Scale the consumer group to 2 instances by executing the following command:

```
$ docker-compose -f docker-compose-clients.yml \
    -p librdkafka up -d --scale consumer=2
...
Starting librdkafka_consumer_1 ...
Starting librdkafka_consumer_1 ... done
Creating librdkafka_consumer_2 ... done
```

Note that in the above output I have omitted the warning message (that we can ignore).

6. Switch to the Control Center and notice:
    - that we have now two consumer instances listed on the consumer lag view
    - that the consumer lag does not increase anymore but slightly decreases.
7. Scale the consumer group to 3 consumer instances. Observe that the consumer lag now decreases more quickly.

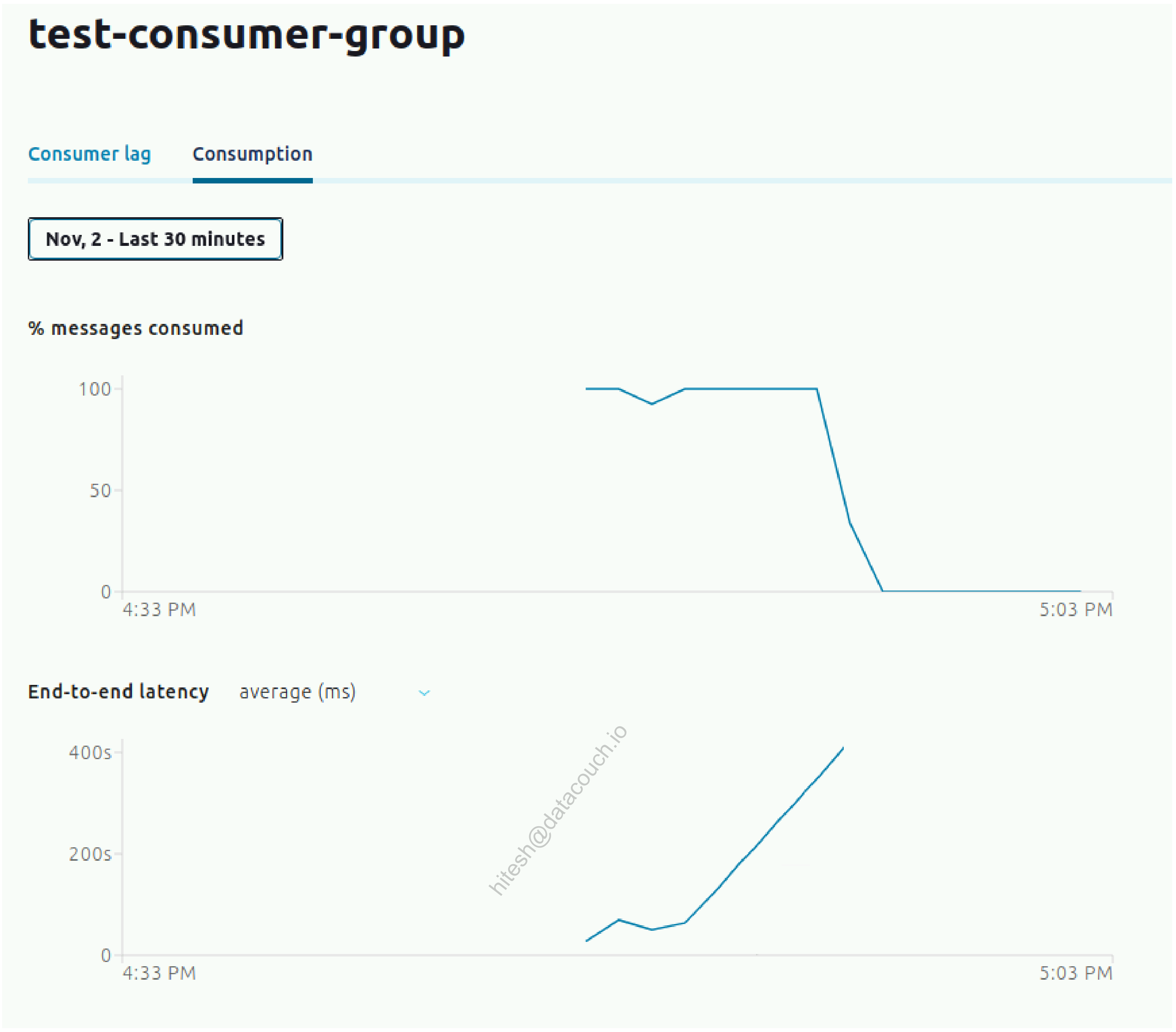
```
$ docker-compose -f docker-compose-clients.yml \
    -p librdkafka up -d --scale consumer=3
```

8. Scale the consumer group even more.
    - Can you ever reach zero in the consumer lag?
    - To how many consumer instances can you maximally scale?
9. Scale down to 1 consumer instance and notice how the consumer lag starts growing again.

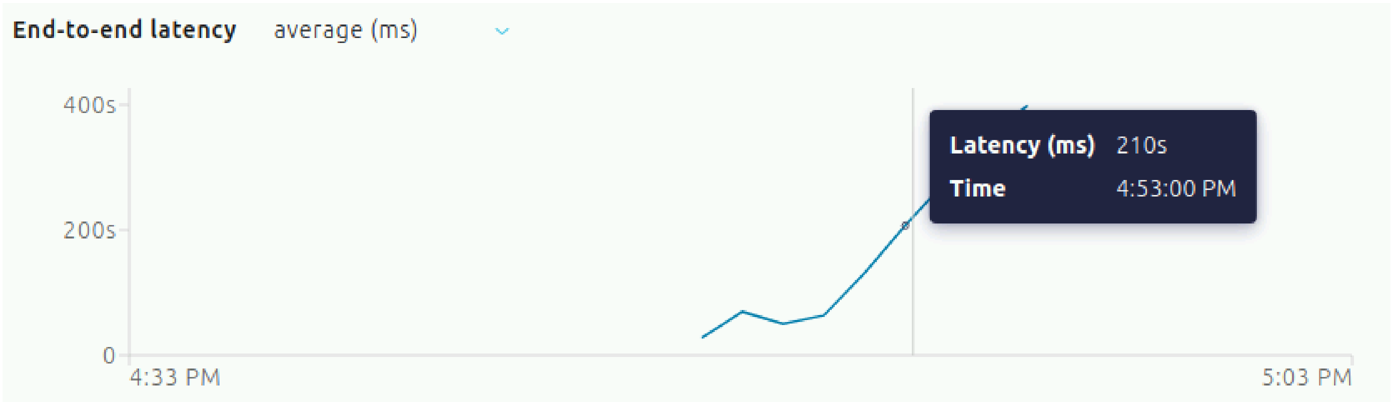
```
$ docker-compose -f docker-compose-clients.yml \
    -p librdkafka up -d --scale consumer=1
```

Monitoring Data Streams

1. In Control Center, click the **Consumption** tab.
2. At the top of the **Consumption** view, click the time period and set it to **Last 30 minutes**.
3. Examine the metrics for **% messages consumed** and **End-to-end latency**:



4. Hover the mouse over one of the graphs and notice how the exact numbers for time, consumption rate and average latency are displayed at the time you are with the mouse:



Cleanup

1. Execute the following commands to completely cleanup your environment:

```
$ docker-compose -f docker-compose-clients.yml -p librdkafka down
$ docker-compose down -v
```

Analyzing the Configuration

1. Navigate to the subfolder `producer` and open the file `Program.cs` in an editor. Note line 20 and 21 in the file.

   - Line 20 defines the list of bootstrap servers the producer shall use. This corresponds to the setting `bootstrap.servers`. In our case the value is `kafka:9092`
   - Line 21 defines the monitoring interceptors. The corresponding setting is `plugin.library.paths` and the value is `monitoring-interceptor`.

   Here the settings are hard-coded for this simple example. In a production ready application the developer would load the settings from a properties file such as `producer.properties`.

2. Navigate to the subfolder `consumer` and also open the file `Program.cs`. There the relevant configuration for the monitoring interceptors is on line 16.
3. Now still in the folder `consumer` locate the file `Dockerfile` and open it. Note lines 2 through 5 which contain the instructions on how to install the package `confluent-librdkafka-plugins` on a Debian Linux based system. This package is provided by Confluent and contains among others the monitoring interceptors. The actual interceptors will be installed as `/usr/lib/x86_64-linux-gnu/monitoring-interceptor.so.1`.

Other Metrics

Other metrics available to `librdkafka`-based applications through the **Statistics** functionality (https://github.com/edenhill/librdkafka/blob/master/STATISTICS.md).

For example, in .NET: https://docs.confluent.io/current/clients/confluent-kafka-dotnet/api/Confluent.Kafka.Consumer.html#Confluent_Kafka_Consumer_OnStatistics

Summary

We have run 2 `librdkafka` based Kafka clients, a producer and a consumer. We have instrumented both clients with monitoring interceptors that are found in the package `confluent-librdkafka-plugins` provided by Confluent. With the help of those interceptors any non Java client that is based on `librdkafka` can be monitored in Confluent Control Center the same way as any standard Java client can be monitored. Note that the monitoring interceptors are only required for the Data Streams view. The Consumer Lag view does not require interceptors.

d. Monitoring with Prometheus

In this exercise you will be using Grafana to monitor a simple Kafka cluster. You will use the tool `kafka-producer-perf-test` to produce some data into a topic `test-topic`, and the tool `kafka-console-consumer` to consume the data.

Prerequisites

1. Navigate to the project folder:

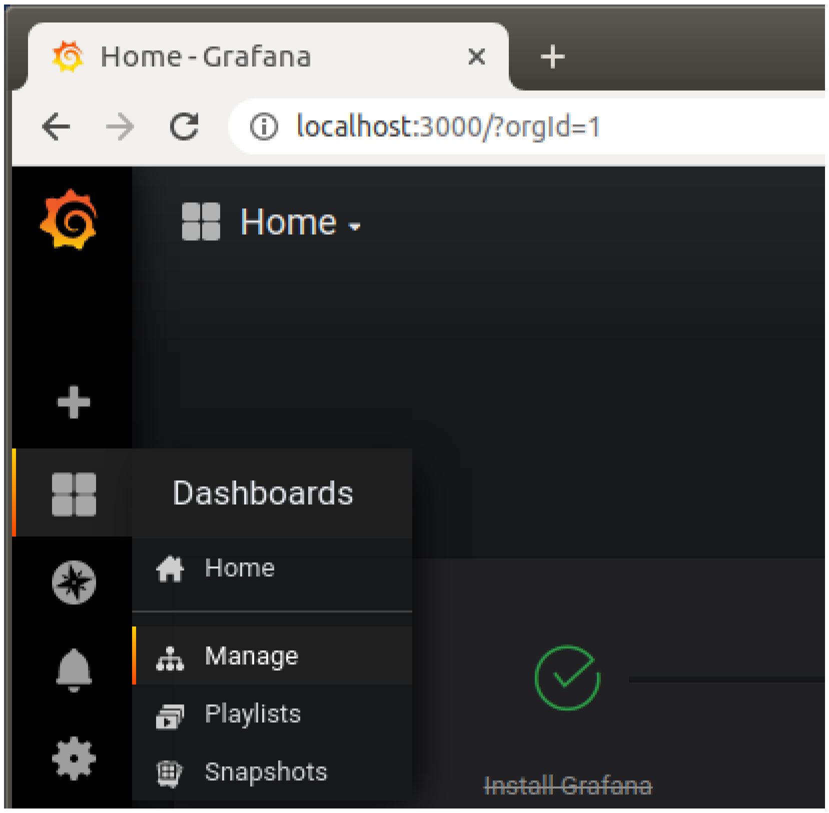
```
$ cd ~/confluent-cao/solution/mo-gen/prometheus
```

2. Run the Kafka cluster, including Prometheus and Grafana:

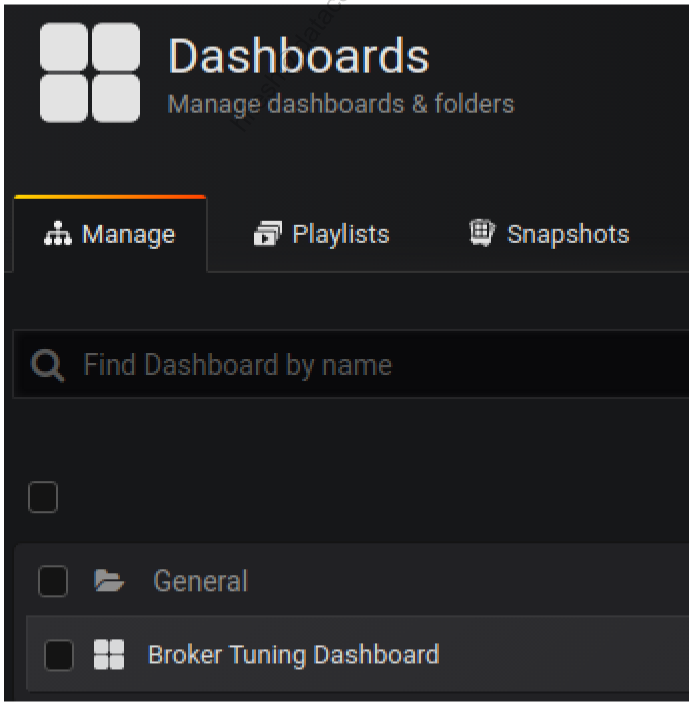
```
$ docker-compose up -d
```

Configuring Grafana

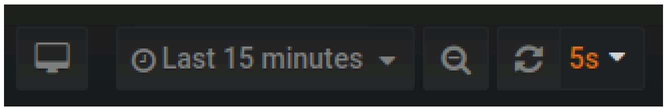
1. Access Grafana at http://localhost:3000
2. The first time you login with **username** `admin` and **password** `admin`. When prompted, you can change the default password or select **Skip**.
3. On the home page, mouse over the dashboard icon and click **Manage**:



4. In the dashboard list, click **Broker Tuning Dashboard**:



5. Set the display time interval and the refresh rate. In the upper right corner:
    a. Click on the clock icon, under **Relative time ranges** select **Last 15 minutes**
    b. Beside the refresh icon, select **5s**



Producing and Consuming Data

1. Create a new Topic called **test-topic** with six Partitions and one replica:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic test-topic \
    --partitions 6 \
    --replication-factor 1

Created topic "test-topic"
```

2. Produce a lot of records:

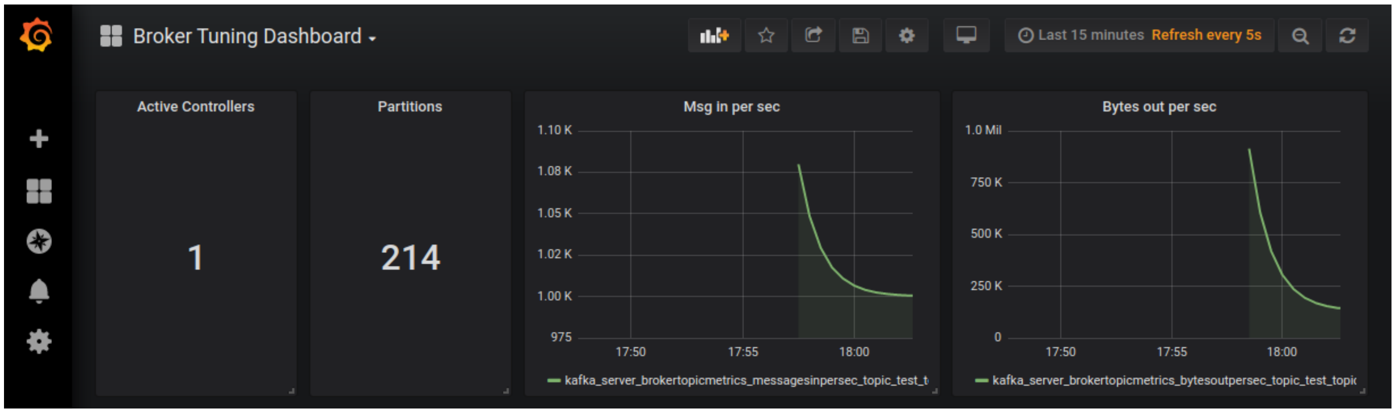
```
$ kafka-producer-perf-test \
    --topic test-topic \
    --num-records 1000000 --record-size 100 --throughput 1000 \
    --producer-props bootstrap.servers=kafka:9092

4988 records sent, 997.6 records/sec (0.10 MB/sec), 19.3 ms avg
latency, 278.0 max latency.
5004 records sent, 999.2 records/sec (0.10 MB/sec), 10.5 ms avg
latency, 83.0 max latency.
5028 records sent, 1005.4 records/sec (0.10 MB/sec), 12.6 ms avg
latency, 123.0 max latency.
...
```

3. Open another terminal window and run the console consumer for the topic **test-topic**:

```
$ export JMX_PORT=8886 && kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --topic test-topic \
    --from-beginning
```

4. Observe the dashboard in Grafana



5. Play with the producer and consumer by stopping and restarting them and observe the dashboard as you do so.
6. Try to add more metrics to the dashboard. If you are not familiar with Grafana try to peek over the shoulders of your peers, on how to do this.

Cleanup

1. Close the Grafana browser window
2. Exit the producer with `Ctrl+C`
3. Exit the consumer with `Ctrl+C`
4. Execute the following command to completely cleanup your environment:

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

Conclusion

In this exercise we have used Grafana to monitor a simple Kafka cluster. We used the tool `kafka-producer-perf-test` to produce some data into a topic `test-topic`, and we used the tool `kafka-console-consumer` to consume the data.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04 Monitoring with Confluent Control Center

a. Monitoring Data Streams

In this lab we're going to use the Confluent Control Center to monitor data streams end to end.

Prerequisites

1. Navigate to the project folder:

   ```
   $ cd ~/confluent-cao/solution/mo-c3
   ```

2. Build the temperatures producer:

   ```
   $ docker image build -t temperatures-producer producer
   ```

   The producer is a simple Java application that produces random temperatures for a given set of meteorological stations. Temperature readings are produced "as fast as possible".

   The building of the Docker image for the producer takes a while, mostly due to the fact that we build an **uber-Jar** for the producer.

3. Run the Kafka cluster:

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

4. Verify the **State** of all containers is **Up** and the `connect` container is **healthy**:

```
$ docker-compose ps
     Name                    Command                State
Ports
---------------------------------------------------------------------
----------------------------------------------
connect           /etc/confluent/docker/run   Up (healthy)
0.0.0.0:8083->8083/tcp, 9092/tcp
control-center    /etc/confluent/docker/run   Up
0.0.0.0:9021->9021/tcp
kafka-1           /etc/confluent/docker/run   Up
0.0.0.0:19092->19092/tcp, 9092/tcp
kafka-2           /etc/confluent/docker/run   Up
0.0.0.0:29092->29092/tcp, 9092/tcp
kafka-3           /etc/confluent/docker/run   Up
0.0.0.0:39092->39092/tcp, 9092/tcp
ksqldb-server     /etc/confluent/docker/run   Up
0.0.0.0:8088->8088/tcp
schema-registry   /etc/confluent/docker/run   Up
0.0.0.0:8081->8081/tcp
zk-1              /etc/confluent/docker/run   Up
0.0.0.0:12181->12181/tcp, 2181/tcp, 2888/tcp, 3888/tcp
zk-2              /etc/confluent/docker/run   Up
2181/tcp, 0.0.0.0:22181->22181/tcp, 2888/tcp, 3888/tcp
zk-3              /etc/confluent/docker/run   Up
2181/tcp, 2888/tcp, 0.0.0.0:32181->32181/tcp, 3888/tcp
```

The `schema-registry` container may end with an `Exit 1` time out error waiting for the brokers to respond. If this occurs, repeat step 3.

5. Create the topic `temperatures-topic` with 6 partitions and 3 replicas:

```
$ kafka-topics --bootstrap-server kafka-1:19092 \
    --create \
    --partitions 6 \
    --replication-factor 3 \
    --topic temperatures-topic
Created topic "temperatures-topic".
```

6. Run the **Temperatures** producer

```
$ docker container run --rm \
    --name producer \
    --net mo-c3_confluent \
    -e MAX_INDEX=999 \
    temperatures-producer:latest
```

7. Open 2 terminal windows.

8. Run an instances of the tool `kafka-avro-console-consumer` with `<client-id>` equal to `consumer-1` and `consumer-2` respectively. Use the following commands to do so. In the first terminal window run:

```
$ export NS=io.confluent.monitoring.clients.interceptor && \
    kafka-avro-console-consumer \
    --bootstrap-server kafka-1:19092 \
    --group my-consumer-group \
    --consumer-property client.id=consumer-1 \
    --from-beginning \
    --consumer-property
interceptor.classes=${NS}.MonitoringConsumerInterceptor \
    --property print.key=true \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic temperatures-topic
```

and in the second terminal window:

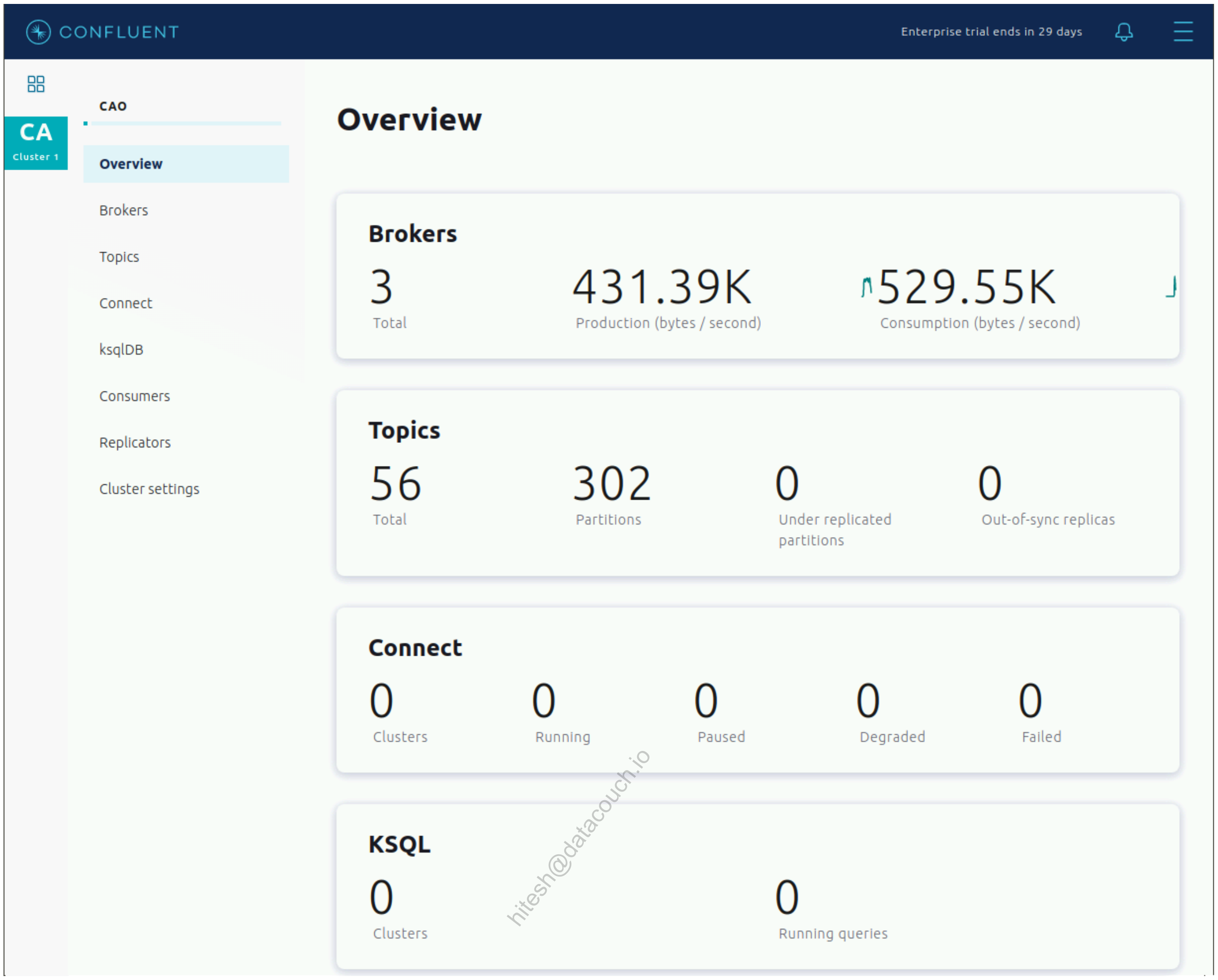
```
$ export NS=io.confluent.monitoring.clients.interceptor && \
    kafka-avro-console-consumer \
    --bootstrap-server kafka-1:19092 \
    --group my-consumer-group \
    --consumer-property client.id=consumer-2 \
    --from-beginning \
    --consumer-property
interceptor.classes=${NS}.MonitoringConsumerInterceptor \
    --property print.key=true \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic temperatures-topic
```

Monitoring System Health

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

2. Wait until Control Center is initialized and the **Clusters** view displays the **cao** cluster with 1 broker running.

3. Click the **cao** cluster and examine the cluster wide metrics.



Overview

Brokers

3 Total

431.39K Production (bytes / second)

529.55K Consumption (bytes / second)

Topics

56 Total

302 Partitions

0 Under replicated partitions

0 Out-of-sync replicas

Connect

0 Clusters

0 Running

0 Paused

0 Degraded

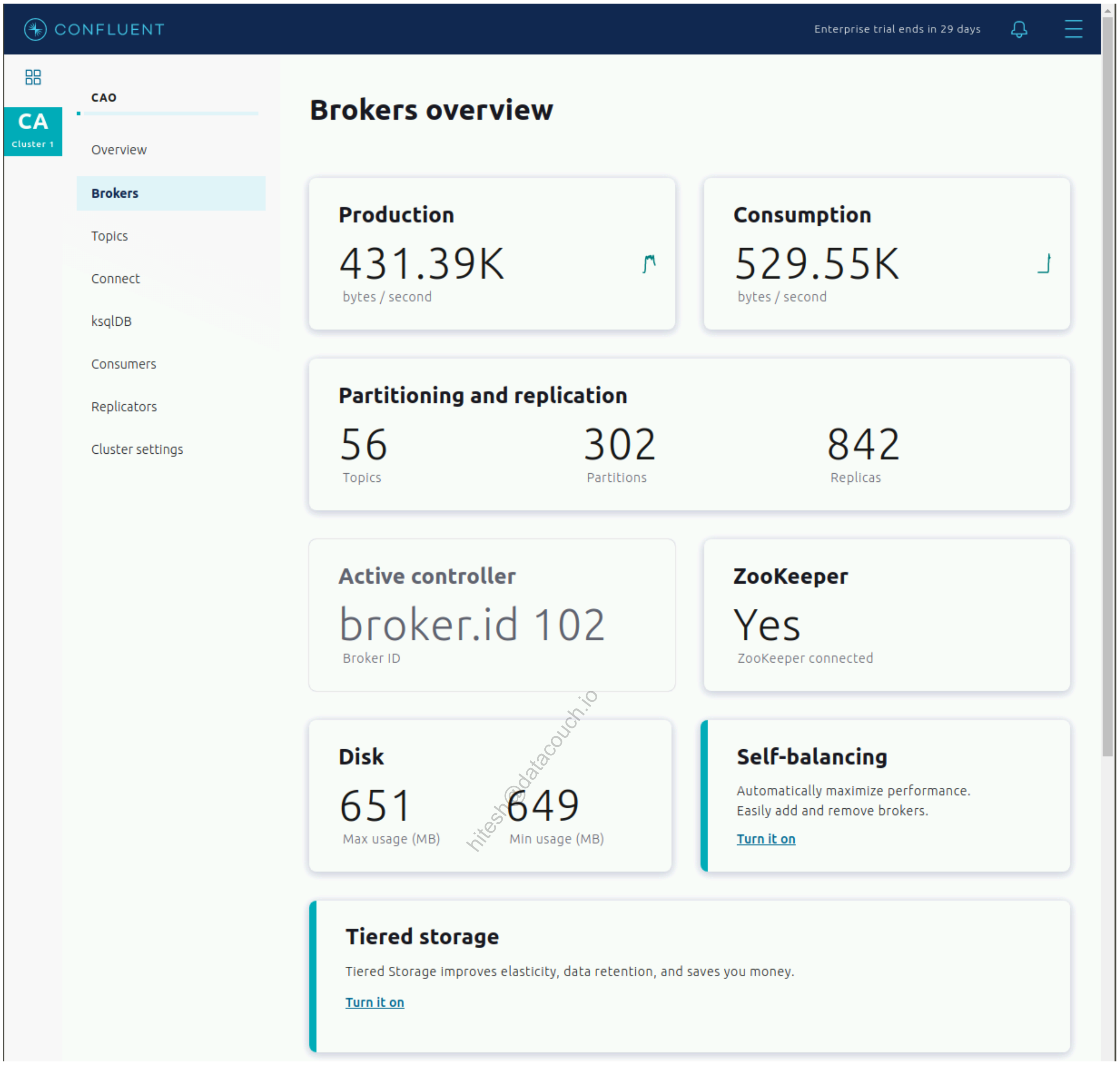
0 Failed

KSQL

0 Clusters

0 Running queries

4. Click **Brokers** and examine the key information provided including the **Active controller** and **Zookeeper** status.



Brokers overview

Production
431.39K
bytes / second

Consumption
529.55K
bytes / second

Partitioning and replication
56 Topics
302 Partitions
842 Replicas

Active controller
broker.id 102
Broker ID

ZooKeeper
Yes
ZooKeeper connected

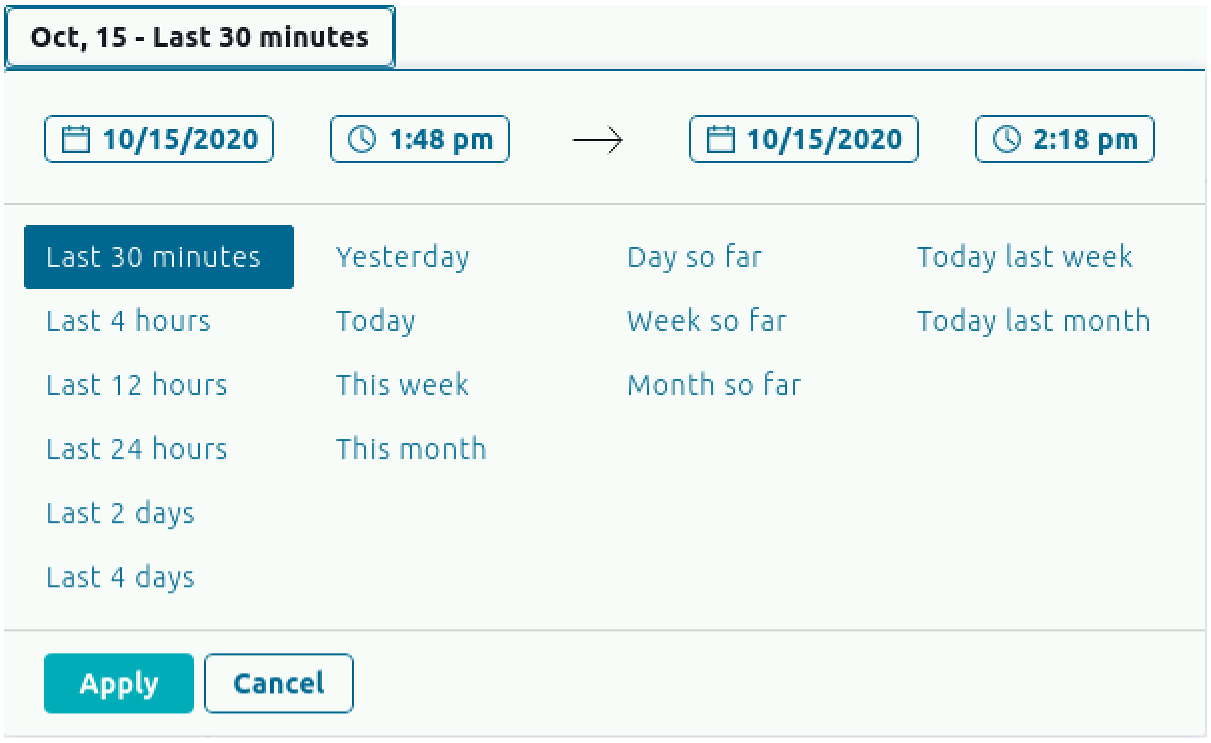
Disk
651 Max usage (MB)
649 Min usage (MB)

Self-balancing
Automatically maximize performance.
Easily add and remove brokers.
Turn it on

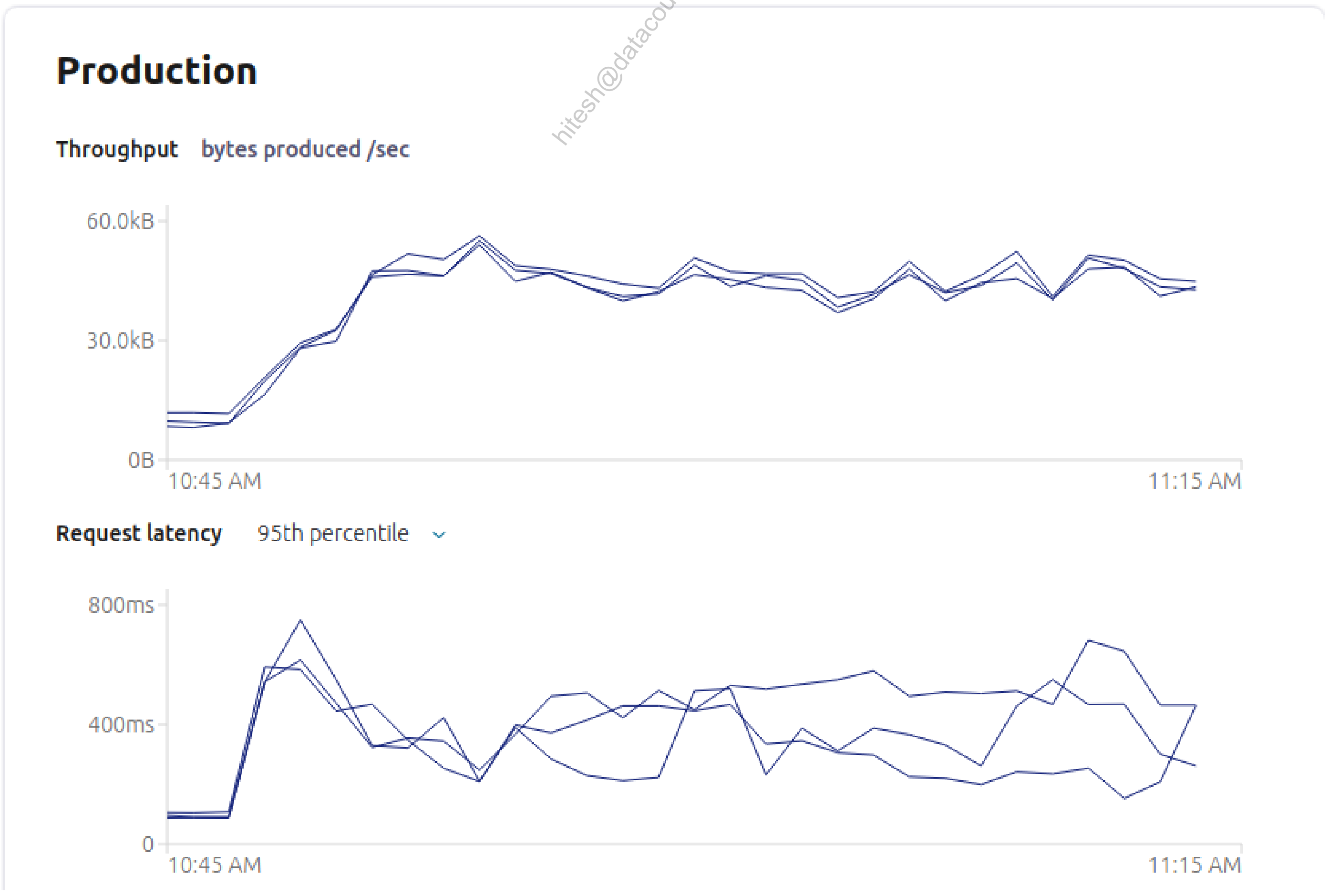
Tiered storage
Tiered Storage improves elasticity, data retention, and saves you money.
Turn it on

5. Click **Production** to open the **Metrics** view.

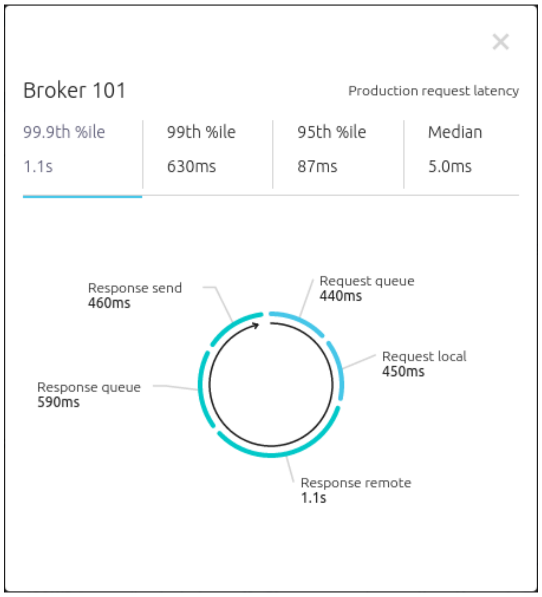
6. At the top of the **Metrics** view, click the time period and set it to **Last 30 minutes**.



7. Scroll through the **Metrics** view and examine the Production and Consumption details, Broker and Zookeeper uptime, Partition replica details, and System and Disk usage.



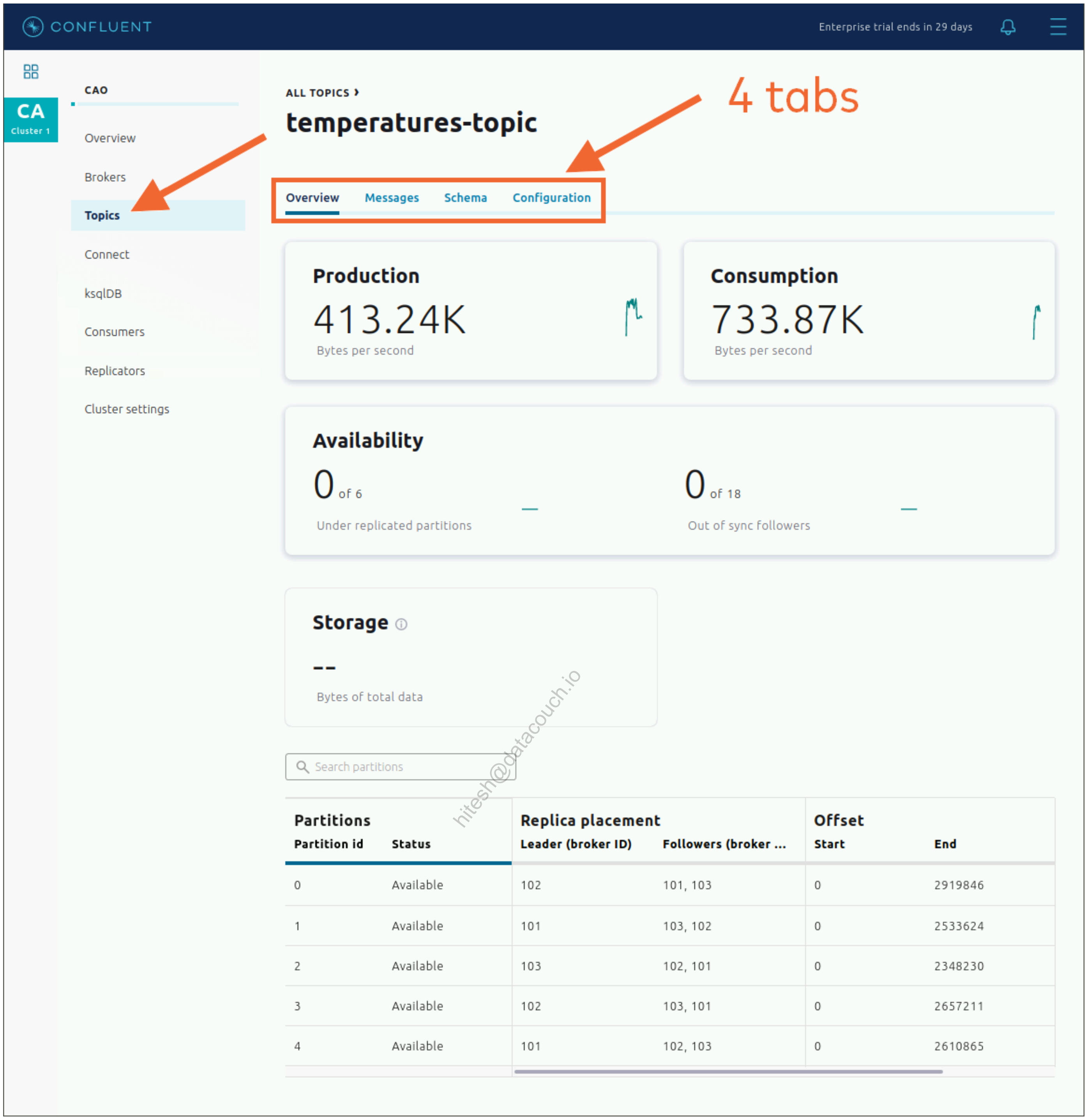
In **Production** metrics, click one of the lines in the **Request latency** chart. A popup similar to the following should appear:



Request queue	request is waiting for IO thread
Request local	request is being processed locally by leader
Response remote	request is waiting on other brokers
Response queue	request is in the response queue waiting for a network thread
Response send	request is being sent back to the producer

Monitoring Topics

1. In Control Center navigate to **Topics**.
2. In the list of topics, click **temperatures-topic**.
3. Notice the topic view has Overview, Messages, Schema, and Configuration tabs:



Partitions		Replica placement		Offset	
Partition id	Status	Leader (broker ID)	Followers (broker ...	Start	End
0	Available	102	101, 103	0	2919846
1	Available	101	103, 102	0	2533624
2	Available	103	102, 101	0	2348230
3	Available	102	103, 101	0	2657211
4	Available	101	102, 103	0	2610865

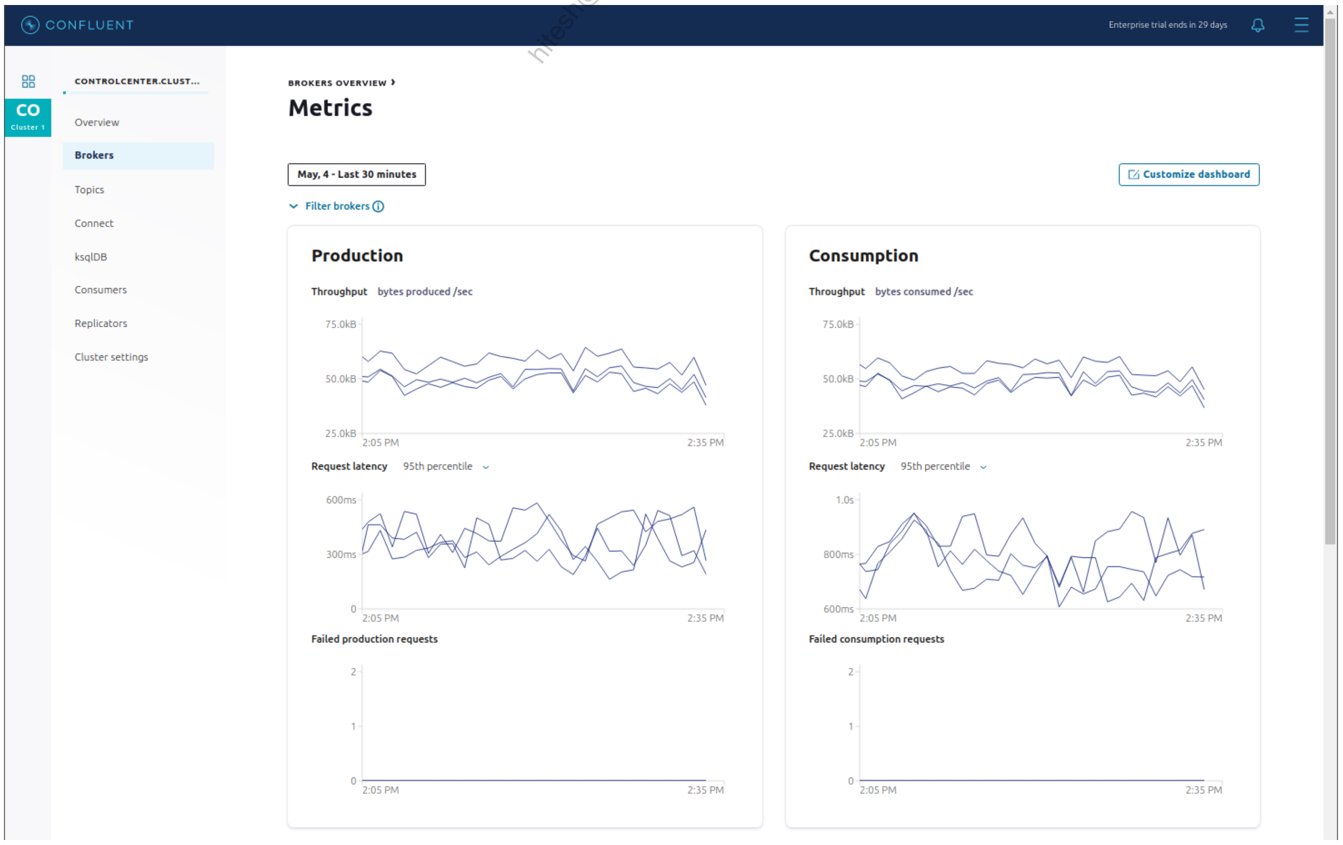
- On the **Overview** tab you can see for each partition of the topic on which brokers replicas have been allocated and which broker is the preferred broker (the first one in the list). The fact that all replicas show blue signifies that all replicas are in sync.

In a later exercise we will simulate failing brokers or out of sync replication. This view will then be useful for a quick overview on what's happening.

- On the **Messages** tab we can see individual messages flowing into the topic. Note, you can only see something here if the topic has currently some inflow of messages.
- On the **Schema** tab we can see the schema used for the key and the value of the messages in this topic. Note, this works for data formats `AVRO`, `JSON`, and `PROTO`.
- On the **Configuration** tab we can see (and edit) the properties of the selected topics. Note that you can only modify a limited set of properties of the topic such as the **retention time** or the **cleanup policy**.

Monitoring Consumption

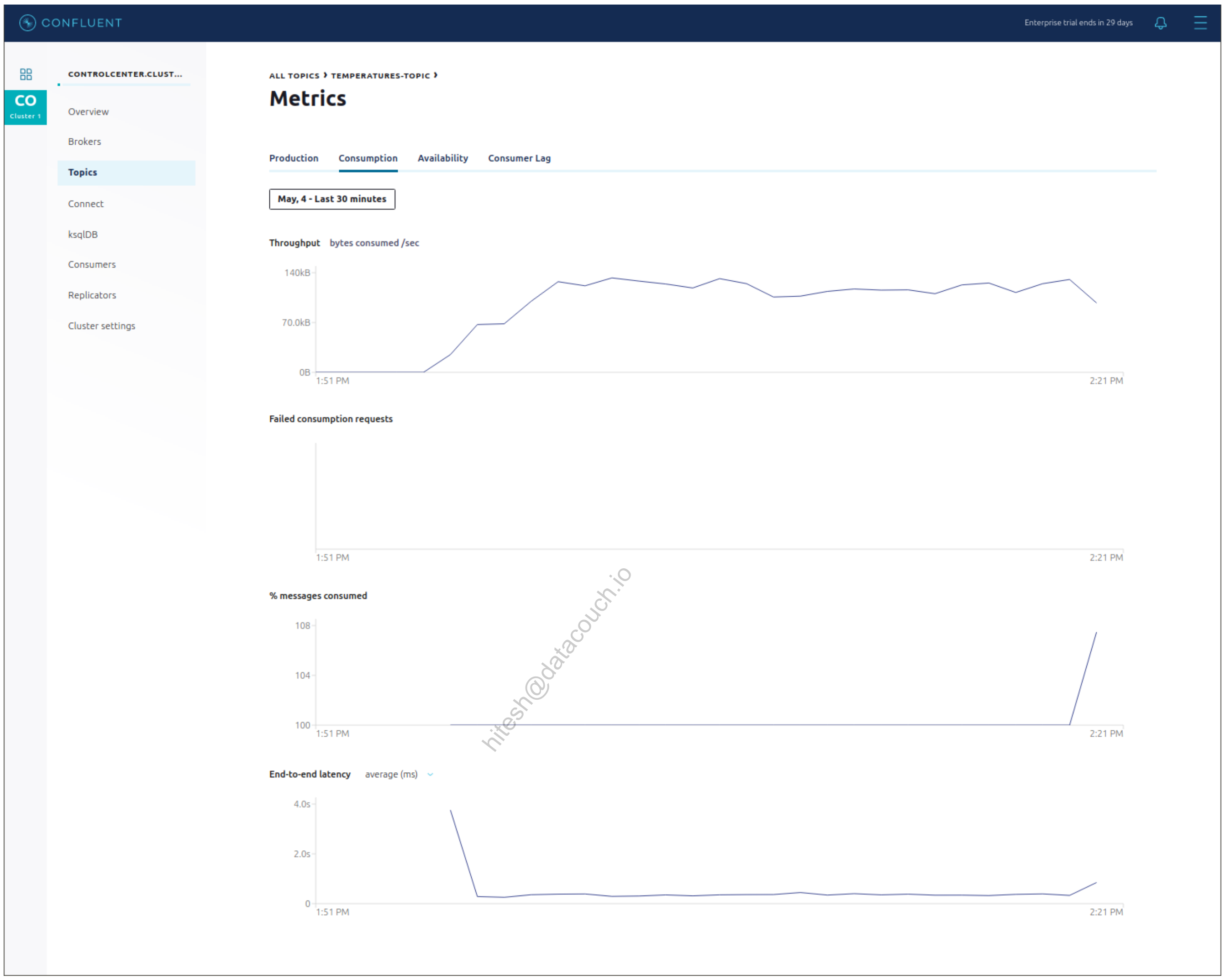
1. In the left side of Control Center, click **Brokers** and then in **Brokers overview** that appears, click **Consumption**.
2. Notice the following elements for Consumption on the view:
   - Throughput
   - Request latency
   - Failed consumption requests



3. You can also view consumption for an individual topic. To do so for **temperatures-topic**:

   a. Switch to the **Topics** view and click **temperatures-topic**.

   b. In the **temperatures-topic** view, navigate to the **Overview** tab and click **Consumption**.

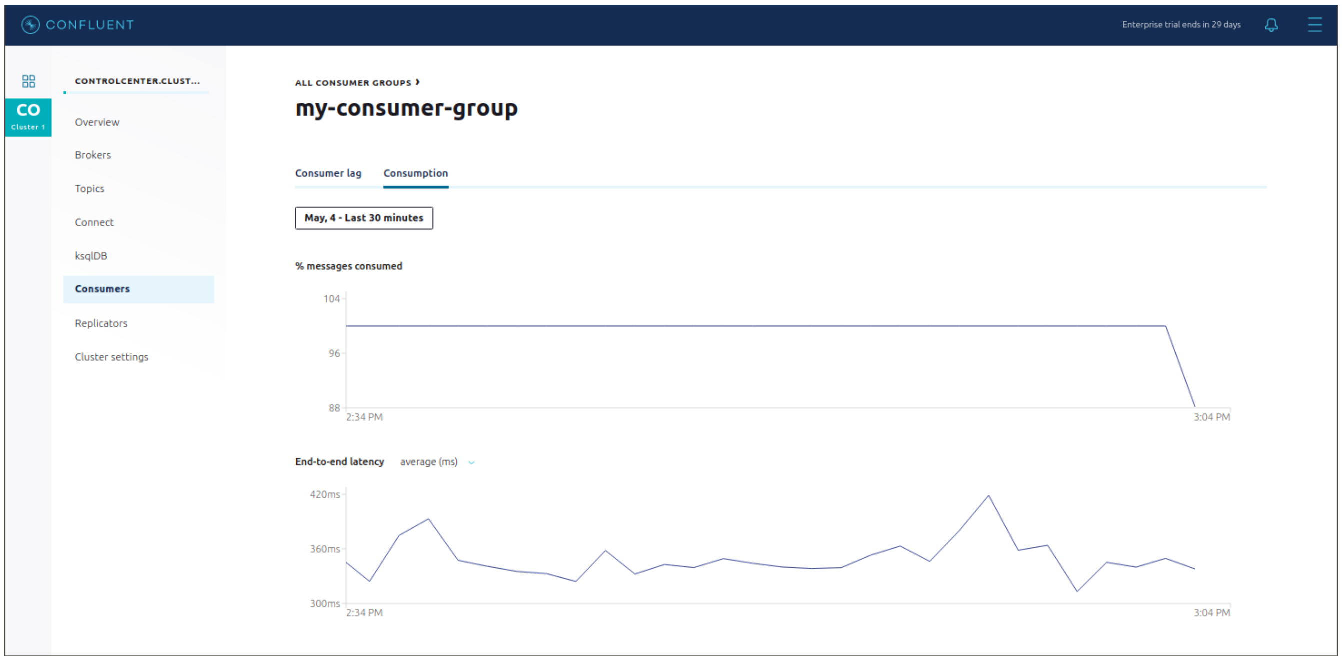


4. And you can view consumption for an individual consumer group. To do so for the consumer group that is currently consuming from **temperatures-topic**:

   a. Switch to the **Consumers** view and click **my-consumer-group**.

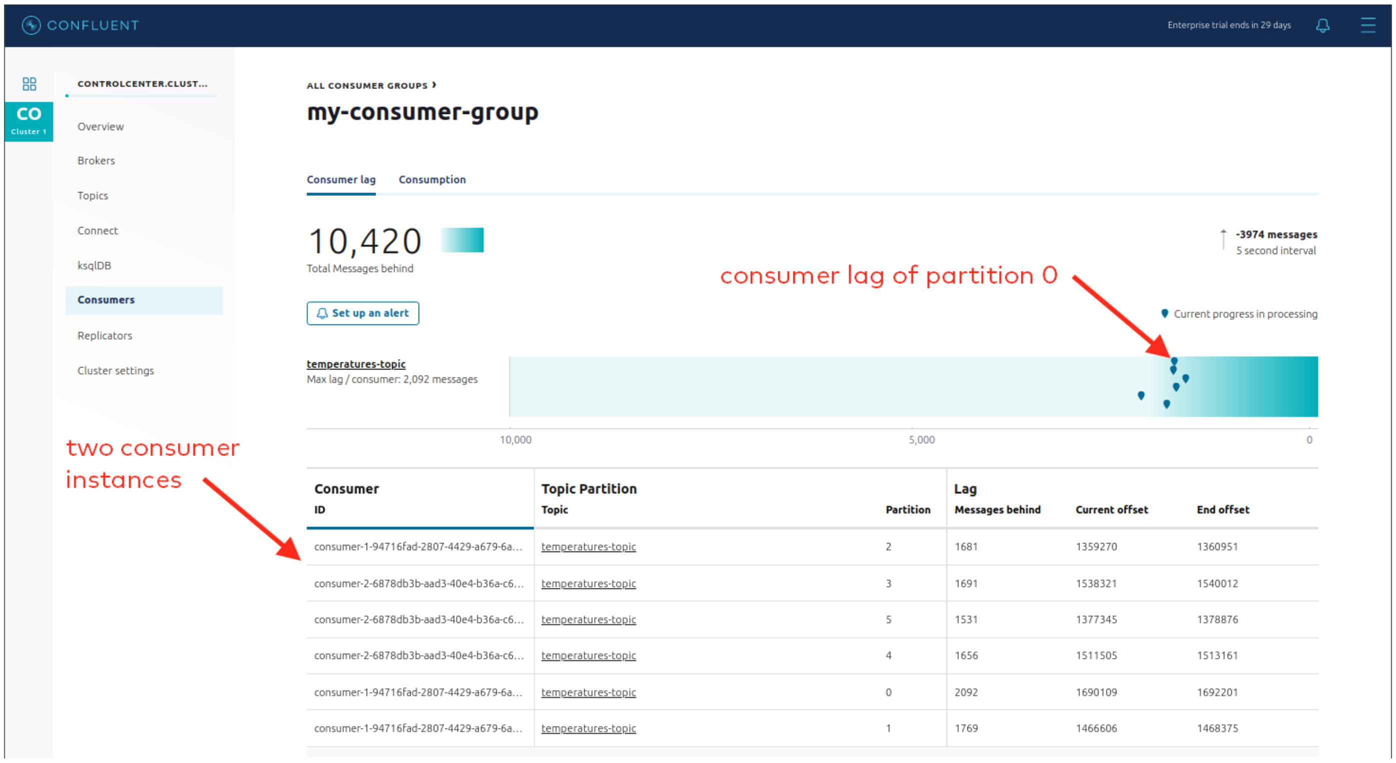
   b. In the All consumer groups view, click **my-consumer-group**.

   c. In the my-consumer-group view, click the **Consumption** tab.



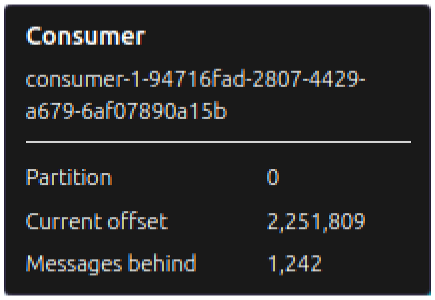
Monitoring Consumer Lag

1. In the my-consumer-group view, click the **Consumer lag** tab.
2. In the upper part of the view locate the graphical representation of the consumer lag per topic partition. In the lower part of the view you can see a table listing important properties per topic partition. Notice that the consumer group includes two consumer instances.



Consumer	Topic Partition		Lag		
ID	Topic	Partition	Messages behind	Current offset	End offset
consumer-1-94716fad-2807-4429-a679-6a...	temperatures-topic	2	1681	1359270	1360951
consumer-2-6878db3b-aad3-40e4-b36a-c6...	temperatures-topic	3	1691	1538321	1540012
consumer-2-6878db3b-aad3-40e4-b36a-c6...	temperatures-topic	5	1531	1377345	1378876
consumer-2-6878db3b-aad3-40e4-b36a-c6...	temperatures-topic	4	1656	1511505	1513161
consumer-1-94716fad-2807-4429-a679-6a...	temperatures-topic	0	2092	1690109	1692201
consumer-1-94716fad-2807-4429-a679-6a...	temperatures-topic	1	1769	1466606	1468375

3. Hover over one of the pin icons in the chart and notice that additional details get displayed in a popup window.


Consumer

consumer-1-94716fad-2807-4429-a679-6af07890a15b

Partition	0
Current offset	2,251,809
Messages behind	1,242

4. Notice that the consumer lag steadily grows.
5. Open a new terminal window and stop/kill the producer:

```
$ docker container kill producer
```

   Back in Control Center, notice that the consumers are catching up.

6. Start the producer again to see that the consumer lag increases again.

```
$ docker container run --rm \
    --name producer \
    --net mo-c3_confluent \
    -e MAX_INDEX=999 \
    temperatures-producer:latest
```

Cleanup

1. Stop/kill the producer as done before

```
$ docker container kill producer
```

2. Stop each consumer instance by pressing `Ctrl+C`.
3. Tear down the cluster and prune the data volumes with:

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

Conclusion

In this lab you have used the Confluent Control Center to monitor a data stream, containing temperature data, end to end.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 05 Troubleshooting Toolbelt

a Troubleshooting Toolbelt

In this exercise we're going to use a few of the typical tools operations engineers use when troubleshooting issues.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-toolbelt
```

2. Run the Confluent Platform:

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

Investigating the foundation

1. If you're running your Confluent Platform containerized then it is a good thing to look at the stats provided by the container daemon:

```
$ docker stats
CONTAINER ID    NAME              CPU %      MEM USAGE / LIMIT
MEM %       NET I/O            BLOCK I/O          PIDS
2648f433affc    control-center    7.17%      410.8MiB / 9.758GiB
4.11%       2.54MB / 1.61MB    8.19kB / 5.21MB    118
a13abdd703f4    base              0.00%      1.098MiB / 9.758GiB
0.01%       1.34kB / 0B        0B / 0B            1
bd6dd14e0454    kafka-1           2.79%      422.2MiB / 9.758GiB
4.23%       2.89MB / 3.23MB    0B / 791kB         75
f2f66e25730e    kafka-2           9.64%      412.1MiB / 9.758GiB
4.12%       3.05MB / 3.21MB    65.5kB / 786kB     75
b7ba1c1a0c6b    kafka-3           3.37%      420.2MiB / 9.758GiB
4.20%       3.14MB / 3.38MB    0B / 791kB         78
9cd535f101c3    zookeeper         0.10%      86.65MiB / 9.758GiB
0.87%       380kB / 552kB      0B / 2.21MB        39
```

The result shows us how much CPU, Memory, Network and I/O each container consumes.

This is a good way to find rogue containers. Please note that you can also discover containers that run unrestricted e.g. on memory. In my case all containers have no limit and thus will take whatever is available (max. 7.8 GB). In a production system each container should always be limited in its use of CPU, memory and more.

2. A nice alternative to `docker stats` is `ctop` (https://github.com/bcicen/ctop). Optional: try to install and run `ctop`.

3. Let's use the `ps` tool to investigate how much resources broker 101 is using while it is more or less idle:

```
$ docker-compose exec -u root kafka-1 ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
appuser        1 13.2  4.4 7942308 453832 ?      Ssl  02:14   0:35 java
-Xmx1G -Xms1G -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35 -XX:+Explic
root         159  0.0  0.0  45844  3684 pts/0    Rs+  02:18   0:00 ps
aux
```

The broker is represented by the **java** process and uses very little CPU and memory at this time.

4. Let's stress the brokers a bit:

   a. Create a topic:

```
$ kafka-topics --bootstrap-server kafka-1:19092 \
    --create \
    --topic test-topic \
    --partitions 6 \
    --replication-factor 3
Created topic "test-topic".
```

   b. Run the perf test tool:

```
$ docker-compose exec -d base kafka-producer-perf-test \
    --topic test-topic \
    --num-records 10000000 \
    --record-size 100 \
    --throughput 10000 \
    --producer-props \
        bootstrap.servers=kafka-1:19092,kafka-2:29092
```

5. Let's see whether the perf tool is running inside the `base` container:

```
$ docker-compose exec base ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
appuser      1  0.0  0.0  19204  3592 pts/0    Ss+  15:01   0:00
/bin/sh
appuser      7 71.6  0.9 3647152 153668 pts/1  Ssl+ 15:07   0:10 java
-Xmx512M -
appuser     46  0.0  0.0  51780  3744 pts/2    Rs+  15:07   0:00 ps
aux
```

   Indeed it is, it is the java process with PID 7.

6. Let's use the `ps` tool again to investigate how much resources broker 101 is using now:

```
$ docker-compose exec -u root kafka-1 ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
appuser      1 23.6  5.1 10312860 830916 ?     Ssl  15:01   1:54 java
-Xmx1G -Xm
root       198  0.0  0.0  51780  3796 pts/0    Rs+  15:09   0:00 ps
aux
```

   The broker's consumption of CPU and memory has changed since we are writing loads of data to it.

7. If we use `docker stats` once more we will see, that now the brokers are really busy and consuming loads of NET I/O. This is due to the heavy writing through the perf test tool and the replication that's happening:

```
$ docker stats
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %
NET I/O             BLOCK I/O     PIDS
1f85db18cd5f   base                9.82%     258.6MiB / 15.35GiB   1.64%
32.7MB / 338MB      16.4kB / 0B   24
9817f43ae545   kafka-2             41.19%    768.9MiB / 15.35GiB   4.89%
441MB / 340MB       0B / 2.97MB   82
8165246c10e7   kafka-3             43.11%    1.028GiB / 15.35GiB   6.70%
440MB / 341MB       0B / 2.97MB   79
27409fcf926d   kafka-1             38.27%    824.9MiB / 15.35GiB   5.25%
441MB / 345MB       0B / 2.99MB   79
8f350b1fac42   zookeeper           0.07%     101.1MiB / 15.35GiB   0.64%
620kB / 957kB       0B / 2.55MB   46
ad38e487f626   control-center      1.15%      76.5MiB / 15.35GiB   3.67%
24.4MB / 15.1MB     0B / 25MB      139
```

8. Stop `docker stats` by pressing `Ctrl+C.

9. We can use `iostat` to report CPU statistics and input/output statistics for devices, partitions and network filesystems.

   a. Run the `netshoot` container in interactive mode:

   ```
   $ docker container run -it --rm --privileged training/netshoot:2.0
   ```

   b. Run `iostat` every 2 seconds in the `netshoot` container:

   ```
   root @ /
    [1] 🕙  → iostat 2
   avg-cpu:  %user   %nice %system %iowait  %steal   %idle
             32.58    0.00    9.69    0.13    0.00   57.61

   Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read
   Blk_wrtn
   loop0             0.00         0.00         0.00          0
   0
   loop1             0.00         0.00         0.00          0
   0
   loop2             0.00         0.00         0.00          0
   0
   loop3             0.00         0.00         0.00          0
   0
   nvme0n1           4.98         0.00       784.08          0
   1576
   nvme0n1p1         4.98         0.00       784.08          0
   1576
   ```

   Everything looks very quiet on my system. Note that `iostat` reports the node level

activity.
Leave `iostat` running for the moment.

c. In a new terminal window run the `stress` container to cause some stress on the machine:

```
$ docker container run --rm -it training/stress:2.0 --hdd 5
```

It spawns 5 workers spinning on write()/unlink(). You should see a significant increase in the numbers produced by `iostat`. Specifically note the value for `%iowait` and `%idle`. In my case, on my system, I see:

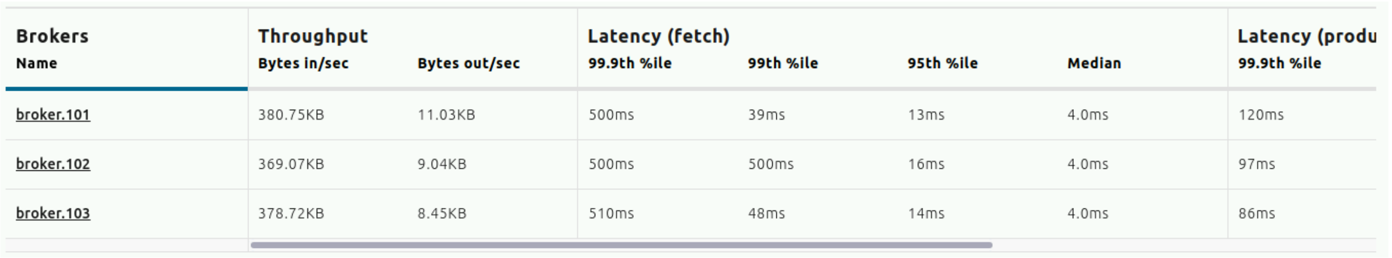
```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          25.78    0.00   10.75   55.96    0.00    7.51

Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read
Blk_wrtn
loop0             0.00          0.00          0.00           0
0
loop1             0.00          0.00          0.00           0
0
loop2             0.00          0.00          0.00           0
0
loop3             0.00          0.00          0.00           0
0
nvme0n1         596.50       3124.00     259560.00        6248
519120
nvme0n1p1       596.50       3124.00     259560.00        6248
519120
```

Using Confluent Control Center

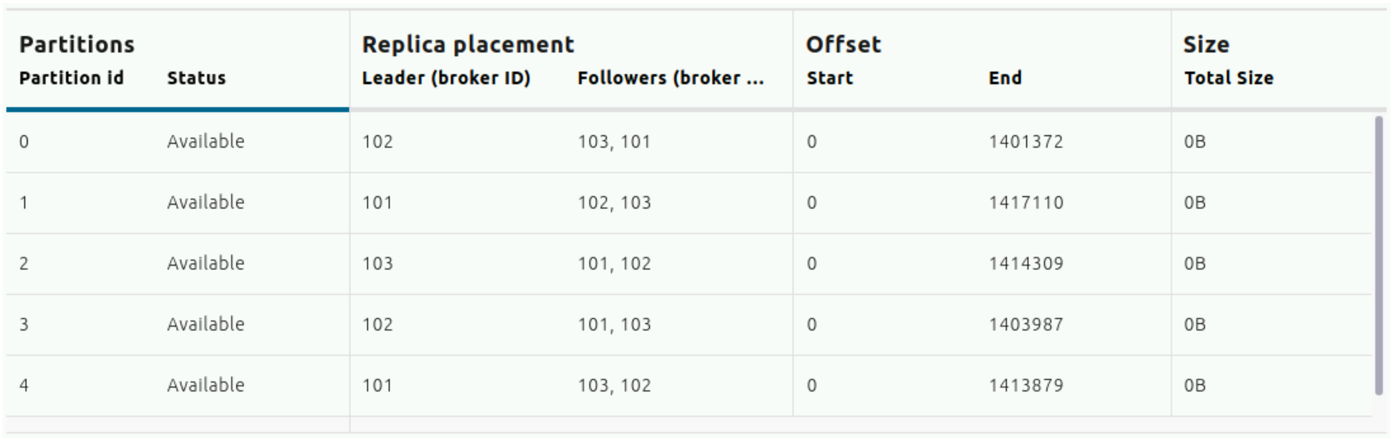
1. Open the Control Center at http://localhost:9021

2. On the **Brokers overview** page, scroll down and analyze the load on the three brokers:



Brokers	Throughput		Latency (fetch)				Latency (produ
Name	Bytes in/sec	Bytes out/sec	99.9th %ile	99th %ile	95th %ile	Median	99.9th %ile
broker.101	380.75KB	11.03KB	500ms	39ms	13ms	4.0ms	120ms
broker.102	369.07KB	9.04KB	500ms	500ms	16ms	4.0ms	97ms
broker.103	378.72KB	8.45KB	510ms	48ms	14ms	4.0ms	86ms

3. Select **TOPICS → test-topic** and scroll down to see the layout and replication status of the topic and its partitions:



Partitions		Replica placement		Offset		Size
Partition id	Status	Leader (broker ID)	Followers (broker ...	Start	End	Total Size
0	Available	102	103, 101	0	1401372	0B
1	Available	101	102, 103	0	1417110	0B
2	Available	103	101, 102	0	1414309	0B
3	Available	102	101, 103	0	1403987	0B
4	Available	101	103, 102	0	1413879	0B

4. Explore more of the views that Control Center allows you into your Kafka powered pipeline.

Cleanup

1. Stop `iostat` by pressing `Ctrl+C` and exit the `netshoot` container with `Ctrl+D`.
2. Stop the `stress` container by pressing `Ctrl+C`.
3. Clean up your environment by running the following command:

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

Summary

In this exercise you have been playing with a few of the typical tools operations engineers use when troubleshooting issues around their Kafka cluster.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 06 Where are my System Log Files?

a. Where are my log files?

In this exercise we're going to extract various crucial information from the logs that the Confluent Platform produces.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-log
```

2. Run the Confluent Platform:

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

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 connect confluent-hub install
confluentinc/kafka-connect-jdbc:10.0.0
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):
```

2. Type `1` and press **Enter**.

3. At the prompt, type `N` and press **Enter**.

```
Do you want to install this into /usr/share/confluent-hub-components?
(yN)
```

4. At the prompt, type `/usr/share/java/kafka` and press **Enter**.

```
Specify installation directory:
```

5. 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)
```

6. At the prompt, type `y` and press **Enter**.

```
Downloading component Kafka Connect JDBC 10.0.0, provided by
Confluent, Inc. from Confluent Hub and installing into
/usr/share/java/kafka
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
```

7. To complete the installation, we need to restart the `connect` container:

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

8. Modify permissions for the `data` directory so that the JDBC Source Connector has access. If you are prompted for a password type `training` as the password:

```
$ sudo chmod 644 data
```

Finding broker specific info

1. Give the brokers some time to startup. This may take a minute or so.

2. Assert that the 3 brokers have all started successfully:

```
$ for N in 1 2 3; do \
      docker-compose logs kafka-${N} | \
        grep -i "started (kafka.server.KafkaServer)"; \
    done
kafka-1 | [2020-10-15 21:39:53,710] INFO [KafkaServer id=101] started
(kafka.server.KafkaServer)
kafka-2 | [2020-10-15 21:39:54,311] INFO [KafkaServer id=102] started
(kafka.server.KafkaServer)
kafka-3 | [2020-10-15 21:39:54,164] INFO [KafkaServer id=103] started
(kafka.server.KafkaServer)
```

If you don't get this output, then something went wrong with your brokers upon startup.

3. Let's see if there is any error reported by say `broker-1`:

```
$ docker-compose logs kafka-1 | grep ERROR
```

It turns out that there are some, but they all happened during the startup of the Kafka cluster, and they all concern the `ReplicaFetcher` and the fetching of metrics by Confluent Control Center. We can safely ignore those errors for the moment.

In this exercise we're dealing with Kafka running in containers. Consequently all logs generated by brokers are redirected to `STDOUT`.
When installing Kafka natively then by default the brokers generate various logfiles on the file system of the host computer. The exact layout of the files is described in the log4j properties file for the broker (https://github.com/apache/kafka/blob/trunk/config/log4j.properties). On a production system though you might want to change this configuration and have the logs forwarded to a central log aggregator.

Finding ZooKeeper specific info

1. Find out who is the leader in the ZK ensemble:

```
$ for N in 1 2 3; do \
      docker-compose logs zk-${N} | grep -i "INFO LEADING"; \
    done
```

In my case I see this:

```
zk-2 | [2020-10-15 21:39:48,288] INFO LEADING
(org.apache.zookeeper.server.quorum.QuorumPeer)
zk-2 | [2020-10-15 21:39:48,302] INFO LEADING - LEADER ELECTION TOOK
- 14 MS (org.apache.zookeeper.server.quorum.Leader)
```

Consequently (in my case) `zk-2` is the leader and `zk-1` and `zk-3` are followers.

2. Capture the leader number in an environment variable. Using the leader identified in the previous step as an example, the command would be `export INITIAL_LEADER=3`:

```
$ export INITIAL_LEADER=<leader number>
```

3. Confirm that indeed, the other two ZK instances are followers:

```
$ for N in 1 2 3; do \
      docker-compose logs zk-${N} | grep -i "INFO FOLLOWING"; \
    done
zk-1 | [2020-10-15 21:39:48,286] INFO FOLLOWING
(org.apache.zookeeper.server.quorum.QuorumPeer)
zk-1 | [2020-10-15 21:39:48,302] INFO FOLLOWING - LEADER ELECTION
TOOK - 17 MS (org.apache.zookeeper.server.quorum.Learner)
zk-3 | [2020-10-15 21:39:48,687] INFO FOLLOWING
(org.apache.zookeeper.server.quorum.QuorumPeer)
zk-3 | [2020-10-15 21:39:48,700] INFO FOLLOWING - LEADER ELECTION
TOOK - 13 MS (org.apache.zookeeper.server.quorum.Learner)
```

and indeed, we see only log output from `zk-1` and `zk-3`.

4. Now let's observe a new leader election by stopping the current leader:

```
$ docker-compose stop zk-${INITIAL_LEADER}
Stopping ts-log_zk-3_1 ... done
```

5. Let's look for new elections:

```
$ for N in zk-1 zk-2 zk-3; do
    if [ "$N" != "zk-$INITIAL_LEADER" ]; then
        docker-compose logs $N | grep -i "INFO New election"
    fi
  done
```

which in my case gives me:

```
zk-1 | [2020-10-15 21:39:47,854] INFO New election. My id =  1,
proposed zxid=0x0
(org.apache.zookeeper.server.quorum.FastLeaderElection)
zk-1 | [2020-10-15 21:50:49,528] INFO New election. My id =  1,
proposed zxid=0x100000372
(org.apache.zookeeper.server.quorum.FastLeaderElection)
zk-3 | [2020-10-15 21:39:48,655] INFO New election. My id =  3,
proposed zxid=0x0
(org.apache.zookeeper.server.quorum.FastLeaderElection)
zk-3 | [2020-10-15 21:50:49,507] INFO New election. My id =  3,
proposed zxid=0x100000372
(org.apache.zookeeper.server.quorum.FastLeaderElection)
```

Indicating that around `2020-10-15 21:50:49` we had the last **new election**. That is around the time I stopped the previous leader.

6. We can confirm that `zk-2` is indeed shutdown:

```
$ docker-compose ps | grep zk-${INITIAL_LEADER}
zk-2              /etc/confluent/docker/run   Exit 143
```

The status of the `zk-2` instance is `Exit 143`, which is expected.

7. Let's find out who is the new leader then:

```
$ for N in zk-1 zk-2 zk-3; do
    if [ "$N" != "zk-$INITIAL_LEADER" ]; then
        docker-compose logs $N | grep -i "INFO LEADING"
    fi
  done
zk-3 | [2020-10-15 21:50:49,936] INFO LEADING
(org.apache.zookeeper.server.quorum.QuorumPeer)
zk-3 | [2020-10-15 21:50:49,940] INFO LEADING - LEADER ELECTION TOOK
- 3 MS (org.apache.zookeeper.server.quorum.Leader)
```

We can see that `zk-3` is the new leader.

8. Before we continue let's restart `zk-2` and have it join the ensemble.

```
$ docker-compose start zk-${INITIAL_LEADER}
Starting zk-2 ... done
```

9. Verify that `zk-3` is now a follower:

```
$ docker-compose logs zk-${INITIAL_LEADER} | grep -i "INFO FOLLOWING"
zk-2 | [2020-10-15 21:55:58,797] INFO FOLLOWING
(org.apache.zookeeper.server.quorum.QuorumPeer)
zk-2 | [2020-10-15 21:55:58,811] INFO FOLLOWING - LEADER ELECTION
TOOK - 14 MS (org.apache.zookeeper.server.quorum.Learner)
```

If the previous command returns no output, it may be the ZK instance is still starting. Wait a minute or two to allow for the start up to complete and repeat the command.

10. We can also find the effect on the brokers that the shutdown had. Let me take as an example `kafka-1`:

```
$ docker-compose logs kafka-1 | grep -i "server zk-
${INITIAL_LEADER}:${INITIAL_LEADER}2181"
```

This gives in my case a series of WARN entries around the time I shut down `zk-3`, the last one being:

```
kafka-1 | [2020-10-15 21:50:51,888] WARN Session 0x200005197980000
for server zk-2:22181, unexpected error, closing socket connection
and attempting reconnect (org.apache.zookeeper.ClientCnxn)
```

Finding Kafka Connect specific info

1. Let's look if Connect started successfully:

```
$ docker-compose logs connect | grep -i "INFO .* Finished starting
connectors and tasks"
connect | [2020-10-15 21:40:34,312] INFO [Worker clientId=connect-1,
groupId=connect] Finished starting connectors and tasks
(org.apache.kafka.connect.runtime.distributed.DistributedHerder:1236)
```

and indeed, startup was completed at `21:40:34`.

2. Add a JDBC source connector via the **REST API** of Connect:

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

the answer should be something like this:

```
{"name":"JDBC-Source-
Connector","config":{"connector.class":"io.confluent.connect.jdbc.Jdb
cSourceConnector","connection.url":"jdbc:sqlite:/data/my.db","table.w
hitelist":"years","mode":"incrementing","incrementing.column.name":"i
d","table.types":"TABLE","topic.prefix":"shakespeare_","name":"JDBC-
Source-Connector"},"tasks":[],"type":"source"}
```

That seems to look fine...

3. Let's see how the configuration of the connector went:

```
$ docker-compose logs connect | grep -i "JDBC-Source-
Connector\|JdbcSourceConnector"
```

Everything looks good except for the last line.

```
...
connect | [2020-10-16 02:39:47,125] WARN [JDBC-Source-
Connector|worker] No tasks will be run because no tables were found
(io.confluent.connect.jdbc.JdbcSourceConnector:150)

...
```

This error tells us that the table we configured the connector to import data from into Kafka does not exist. This missing table was intentional to illustrate that things are not always as they appear. The output from the `curl` seemed to indicate that the connector had been successfully added to the Kafka Connect cluster. Upon closer examination of the log, we discovered that it was actually unsuccessful due to no tables being found in the database.

If we needed to examine the log further to narrow down what might be causing the error, we could optionally use the `less` command to scroll through the log and search for additional information.

```
$ docker-compose logs connect | less
```

4. To prepare for the next part of the exercise clean up your environment by running the following command:

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

Enable Request and Authorizer Logs

In this part we are going to enable the **Request** and the **Authorizer** logs which is useful when debugging issues related to e.g. request authorizations. For this we're going to use our own custom `log4j.properties` file for the Kafka broker.

1. Open the file `logging/log4j.properties` in your favorite editor
   a. In relation to the default `log4j.properties` file that is normally used by the Kafka brokers running inside a container we have added two appenders called `authorizerAppender` and `requestAppender`. This is not strictly needed in our case since they look the same as the default `stdout` appender. But you could change them if you wanted.
   b. Then on line 25 and 26 we define a new log level for the `request` and `authorizer` logger. The new log levels are now `TRACE` for the former and `DEBUG` for the latter.
2. Now open the file `docker-compose-custom.yml` and note how on line 36 we override the default `log4j` configuration with our own:

```
KAFKA_LOG4J_OPTS: "-
Dlog4j.configuration=file:/logging/log4j.properties"
```

   Line 24 and 25 are used to mount our custom `log4j` configuration into the container to make them available to the Kafka broker.

3. Run the Confluent Platform:

```
$ docker-compose -f docker-compose-custom.yml up -d
```

4. Verify that the **Request** appender now produces INFO, DEBUG or TRACE logs:

```
$ docker-compose -f docker-compose-custom.yml logs kafka | grep
"request.logger"
```

5. Clean up your environment by running the following command:

```
$ docker-compose -f docker-compose-custom.yml down -v
```

Summary

In this exercise you have learned how to use the logs generated by some of the components of the Confluent Platform to troubleshoot issues.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07 Troubleshooting ZooKeeper

a. Troubleshooting ZooKeeper

In this exercise we're going to look into various means of troubleshooting a ZooKeeper ensemble.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-zk
```

2. Run the Confluent Platform:

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

Sanity Checks

1. First we will locate the ZooKeeper ensemble leader:

```
$ for N in {1..3}; do
  echo mntr | nc zk-${N} ${N}2181 | \
    grep zk_server_state | xargs -I '{}' echo "zk-$N {}"; \
  done
zk-1 zk_server_state    follower
zk-2 zk_server_state    follower
zk-3 zk_server_state    leader
```

In the above example, `zk-3` is the leader.

2. Capture the leader number in an environment variable so that we can use it to identify the Zookeeper leader in steps that follow in this exercise. Using the leader identified in the previous step as an example, the command would be `export LEADER=3`:

```
$ export LEADER=<leader number>
```

3. Lets use the **four letter word** `mntr` to get a list of variables that could be used for monitoring the health of the cluster:

```
$ echo mntr | nc zk-1 12181
zk_version  3.5.8-f439ca583e70862c3068a1f2a7d4d068eec33315, built on
05/04/2020 15:53 GMT
zk_avg_latency  0
zk_max_latency  10
zk_min_latency  0
zk_packets_received 545
zk_packets_sent 544
zk_num_alive_connections    2
zk_outstanding_requests 0
zk_server_state follower
zk_znode_count  712
zk_watch_count  2
zk_ephemerals_count 4
zk_approximate_data_size    105620
zk_open_file_descriptor_count   127
zk_max_file_descriptor_count    1048576
```

Note the `zk_server_state` which indicates if this instance is the **leader** or a **follower**. Also note its current number of alive connections.

4. Execute the same 4 letter word for `zk-2` and `zk-3` and compare the variable values among the 3 ZK instances.

Each ZK instance is assigned a unique client port:

- `zk-1` client port is `12181`
- `zk-2` client port is `22181`
- `zk-3` client port is `32181`

When sending commands to the ZK instances, you will need to include its unique client port in the command.

5. Verify that there is only one leader in the entire ZooKeeper ensemble.

6. Verify that all followers are in sync with the leader :

```
$ echo mntr | nc zk-${LEADER} ${LEADER}2181 | grep
zk_synced_followers
zk_synced_followers 2
```

should be equal to:

```
$ echo mntr | nc zk-${LEADER} ${LEADER}2181 | grep zk_followers
zk_followers    2
```

The values reported are **2** since we have an ensemble of 3 with one leader and two followers.

7. Use the `conf` command to get details about the instances configuration:

```
$ echo conf | nc zk-1 12181
clientPort=12181
secureClientPort=-1
dataDir=/var/lib/zookeeper/data/version-2
dataDirSize=67108880
dataLogDir=/var/lib/zookeeper/log/version-2
dataLogSize=543
tickTime=2000
maxClientCnxns=60
minSessionTimeout=4000
maxSessionTimeout=40000
serverId=1
initLimit=5
syncLimit=2
electionAlg=3
electionPort=3888
quorumPort=2888
peerType=0
membership:
server.1=zk-1:2888:3888:participant
server.2=zk-2:2888:3888:participant
server.3=zk-3:2888:3888:participant
```

8. Use the `cons` command to list the full connection/session details for all clients connected to a particular ZK instance:

```
$ echo cons | nc zk-1 12181
 /172.18.0.5:33054[1](queued=0,recved=418,sent=419,sid=0x100000636550
000,lop=PING,est=1602872221368,to=18000,lcxid=0x176,lzxid=0xffffffff
ffffffff,lresp=691098,llat=0,minlat=0,avglat=0,maxlat=10)
 /172.18.0.3:58870[1](queued=0,recved=359,sent=360,sid=0x100000636550
001,lop=PING,est=1602872221374,to=18000,lcxid=0x13a,lzxid=0xffffffff
ffffffff,lresp=693808,llat=0,minlat=0,avglat=0,maxlat=12)
 /172.18.0.1:58734[0](queued=0,recved=1,sent=0)
```

9. The above output only shows the IP addresses of the clients. To find out what service is behind a particular IP address inspect the Docker network on which everything is running:

   a. Open another terminal window

   b. Navigate to the project folder

   ```
   $ cd ~/confluent-cao/solution/ts-zk
   ```

   c. Inspect the network `ts-zk_confluent`:

   ```
   $ docker network inspect ts-zk_confluent | jq ".[0].Containers"
   {

   "15e85616a9c6725429f0210798c40de28313cde8d4c7eec4b93afdf10a111f2a"
   : {
       "Name": "zk-1",
       "EndpointID":
   "27b6d1532608cb49fdfecc4838745306d4510a770f463d28222dff5d04645e10"
   ,
       "MacAddress": "02:42:ac:12:00:09",
       "IPv4Address": "172.18.0.9/16",
       "IPv6Address": ""
     },

   "2217c9b3cfbe21008a65f7d1a1775620315b7cd70cd14dabdf1c1e594db0f5a7"
   : {
       "Name": "schema-registry",
       "EndpointID":
   "36804bead4132b1b018cca039678a083c53580a376ef5f19aedf7108c11c77cb"
   ,
       "MacAddress": "02:42:ac:12:00:02",
       "IPv4Address": "172.18.0.2/16",
       "IPv6Address": ""
     },

   "2c2ac2af3b9923c9cea91a7f8b40d8d79fc14eb8ad252ad5c6aaafed763349ec"
   : {
   ```

```
    "Name": "control-center",
    "EndpointID":
"0094c24ab46d165dc79528a9ccf3d07f431ba0dc3fcfda53071d24ed1aebf413"
,
    "MacAddress": "02:42:ac:12:00:04",
    "IPv4Address": "172.18.0.4/16",
    "IPv6Address": ""
  },

"3c86dd0f3ff887ef532657c6bc962e822539e489bd8212a4a4372c5cf07fae10"
: {
    "Name": "kafka-1",
    "EndpointID":
"8df88338bbfd8810ed06f35998768e2a10089e8880c99178c5f40f25a459a7c6"
,
    "MacAddress": "02:42:ac:12:00:06",
    "IPv4Address": "172.18.0.6/16",
    "IPv6Address": ""
  },

"45eefe71f35ff2a64592e143cbe7ca1aec2347144fc20430390979f6266d257a"
: {
    "Name": "zk-3",
    "EndpointID":
"695e702c5de1fdf3f23e86791f02c5f2a589fc3ac410e67adcd1922e72650dfd"
,
    "MacAddress": "02:42:ac:12:00:08",
    "IPv4Address": "172.18.0.8/16",
    "IPv6Address": ""
  },

"86fa3f2545ef0782087a1ae5fc5b9736d5b09d796f8809149b1e9a85c19b1aba"
: {
    "Name": "kafka-2",
    "EndpointID":
"3114a991785238c1947b5343e5b4e1c910122cb68ff49f7ddb281ff9053efb19"
,
    "MacAddress": "02:42:ac:12:00:03",
    "IPv4Address": "172.18.0.3/16",
    "IPv6Address": ""
  },

"8f89b1108f15eff93de83585e0cdb7418e46e48d5316572f68ae28aa8956da69"
: {
    "Name": "zk-2",
    "EndpointID":
"850c90d9bf39346e00b3b8eb1be866cf262a33718f8a6741faa7ddd9d9bb6bf3"
,
    "MacAddress": "02:42:ac:12:00:07",
    "IPv4Address": "172.18.0.7/16",
    "IPv6Address": ""
  },

"e2e462ba5882a3412cd33e50b34103a5af2c319145d65e18fb7181fe34b24e0d"
```

```
: {
      "Name": "kafka-3",
      "EndpointID":
"1403e96a1ecafa9f95bd344d6522f7800d27e4675711782a4324ded7914b6dec"
,
      "MacAddress": "02:42:ac:12:00:05",
      "IPv4Address": "172.18.0.5/16",
      "IPv6Address": ""
    }
  }
```

Comparing this with the previous output we can see that the 3 connections to ZK stem from the local host (172.18.0.1) and the 2 containers `kafka-2` (172.18.0.3) and `kafka-3` (172.18.0.5).

A more concise result could be achieved with the following command:

```
$ docker network inspect ts-zk_confluent | \
    jq '.[0].Containers[] | .Name + " => " +
.IPv4Address'
"zk-1 => 172.18.0.9/16"
"schema-registry => 172.18.0.2/16"
"control-center => 172.18.0.4/16"
"kafka-1 => 172.18.0.6/16"
"zk-3 => 172.18.0.8/16"
"kafka-2 => 172.18.0.3/16"
"zk-2 => 172.18.0.7/16"
"kafka-3 => 172.18.0.5/16"
```

d. Close this terminal window.

10. Test if your leader is running in a non-error state using the `ruok` command. ZooKeeper will respond with `imok` if it is running. Otherwise it will not respond at all:

```
$ echo ruok | nc zk-${LEADER} ${LEADER}2181
imok
```

Simulate Total Server Loss

Let's simulate a total loss of the **leader** to make things a bit interesting…

1. Create a topic:

```
$ kafka-topics \
    --bootstrap-server kafka-1:19092 \
    --create \
    --topic test-topic \
    --partitions 6 \
    --replication-factor 3
Created topic "test-topic".
```

2. Open a new terminal window and run a sample producer to produce a load of data:

```
$ export NS=io.confluent.monitoring.clients.interceptor &&
  kafka-producer-perf-test \
    --topic test-topic \
    --num-records 10000000 \
    --record-size 100 \
    --throughput 10000 \
    --producer-props \
        bootstrap.servers=kafka-1:19092,kafka-2:29092 \
        interceptor.classes=${NS}.MonitoringProducerInterceptor
```

3. Open another new terminal window run a consumer to consume that data:

```
$ export NS=io.confluent.monitoring.clients.interceptor &&
  kafka-console-consumer \
    --group sample-consumer-group \
    --bootstrap-server kafka-1:19092 \
    --from-beginning \
    --topic test-topic \
    --consumer-property
"interceptor.classes=${NS}.MonitoringConsumerInterceptor"
```

4. Open Confluent Control Center at http://localhost:9021 and monitor the consumer group `sample-consumer-group` in the **Consumption** view. Keep that browser window open as we proceed.

5. Return to the initial terminal window from which you created the topic and execute the following to stop the leader:

```
$ docker-compose stop zk-${LEADER} && \
  docker-compose rm -f zk-${LEADER} && \
  docker volume prune -f
```

Note, the second and third commands completely remove the instance and its data, and thus simulates a complete loss…

6. Check who is the leader now:

```
$ for N in {1..3}; do
  echo mntr | nc zk-${N} ${N}2181 | \
    grep zk_server_state | xargs -I '{}' echo "zk-$N {}";
  done
zk-1 zk_server_state    follower
zk-2 zk_server_state    leader
```

As you can see, `zk-2` is the new leader now in my case.

7. Create an environment variable to represent the new leader:

```
$ export NEW_LEADER=<new leader number>
```

8. Check that the ensemble is in-sync:

```
$ echo mntr | nc zk-${NEW_LEADER} ${NEW_LEADER}2181 | grep -E
'zk_(synced_)?followers'
zk_followers    1
zk_synced_followers     1
```

Evidently everything is in sync and we're good to continue...

9. Do a backup of the new leader.
We can skip this here since this is only an exercise!

10. Add a new ZooKeeper instance to the ensemble:

```
$ docker-compose up -d
...
Creating zk-3 ... done
```

Where `<container-name>` is the name of the ZK instance that you previously stopped and removed (`zk-3` in my case).

11. Check that the ensemble is in-sync again:

```
$ echo mntr | nc zk-${NEW_LEADER} ${NEW_LEADER}2181 | grep -E
'zk_(synced_)?followers'
zk_followers    2
zk_synced_followers     2
```

12. Double check in Control Center that all messages produced during the down period of the ZK instance have been consumed orderly and that:
    - no failed production requests
    - no failed consumption requests
    - total messages produced/consumed per minute remains the same
    - Average and total latency remains the same

Cleanup

1. Press `Ctrl+C` to stop the producer.
2. Press `Ctrl+C` to stop the consumer.
3. Execute the following command to cleanup the system:

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

Summary

We have used the **four letter words** to investigate the status and health of our ZooKeeper ensemble. We have then simulated a complete ZK server loss and replaced it with a new ZK instance. The operation of Kafka has not been compromised during that event.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 08 Troubleshooting & Tuning Brokers

a. Troubleshooting Brokers

When troubleshooting components of the event streaming platform such as a broker, we often don't find enough information in the logs since the log level of the relevant logger might be `INFO` and we really need it to be more verbose such as `DEBUG`.
In this exercise we're going to troubleshoot a broker and will change the log level of one of its loggers without having to stop and restart the broker. This is what you would do on a production system that allows no outages or disruptions.

> When you are going to troubleshoot a production system then security is an important topic. It is important to know that access to JMX metrics of a component running on a production cluster can be secured with authentication and/or TLS encryption. But this is beyond the scope of this lab.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-brokers
```

2. Run the Confluent Platform:

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

Procedure

In this section we lay out the general approach needed to change the log level of a logger. The step by step instruction is provided in the subsequent section.

In order to set the log level and **avoid a restart** to the broker service the following is needed:

1. Given the (production) issue at hand, determine the logger that requires the change.
2. Decide on which new log level you need for debugging (`INFO`, `DEBUG`, `TRACE`).
3. Alter the Logger to the new log level using the `kafka-configs` command.
4. **Example:** kafka-configs --bootstrap-server kafka:9092 --alter --add-config "kafka.server.ReplicaManager=WARN,kafka.server.KafkaApis=DEBUG" --entity-type broker-logger --entity-name 0 // set some log levels for broker 0

Changing the Log Level

1. List your Loggers:

```
$ kafka-configs --bootstrap-server localhost:9092 --describe --entity
-type broker-loggers --entity-name 101
Dynamic configs for broker-logger 101 are:
  org.apache.zookeeper.CreateMode=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.producer.internals.Sender=INFO
sensitive=false synonyms={}
  org.apache.kafka.common.utils.AppInfoParser=INFO sensitive=false
synonyms={}
  kafka.utils.KafkaScheduler=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.ClientUtils=INFO sensitive=false
synonyms={}
  org.apache.http.impl.execchain.ProtocolExec=INFO sensitive=false
synonyms={}
  org.apache.http.client.protocol.RequestClientConnControl=INFO
sensitive=false synonyms={}
  io.confluent.serializers.ProtoSerde=INFO sensitive=false
synonyms={}
  org.apache.zookeeper.ClientCnxnSocketNIO=INFO sensitive=false
synonyms={}
  state.change.logger=TRACE sensitive=false synonyms={}
  io.confluent.metrics.reporter.VolumeMetricsProvider=INFO
sensitive=false synonyms={}
  org.apache.zookeeper.ZooKeeper=INFO sensitive=false synonyms={}
  org.apache.http.impl.conn.DefaultHttpClientConnectionOperator=INFO
sensitive=false synonyms={}
  org.apache.zookeeper.ClientCnxnSocket=INFO sensitive=false
synonyms={}
  org.apache.kafka.clients.CommonClientConfigs=INFO sensitive=false
synonyms={}
  io.confluent.metrics.reporter.ConfluentMetricsReporter=INFO
sensitive=false synonyms={}
  kafka.server.ReplicaManager$LogDirFailureHandler=INFO
sensitive=false synonyms={}
  kafka.log.Log=INFO sensitive=false synonyms={}
  org.apache.http.impl.conn.DefaultManagedHttpClientConnection=INFO
sensitive=false synonyms={}
```

```
  org.apache.zookeeper.common.X509Util=INFO sensitive=false
synonyms={}
  org.apache.zookeeper.common.ZKConfig=INFO sensitive=false
synonyms={}
  kafka.server.DelayedProduce=INFO sensitive=false synonyms={}
  kafka.utils.FileLock=INFO sensitive=false synonyms={}
  org.apache.kafka.common.record.MemoryRecords=INFO sensitive=false
synonyms={}
  org.apache.http.conn.ssl.AllowAllHostnameVerifier=INFO
sensitive=false synonyms={}
  org.apache.kafka.common.metrics.Metrics=INFO sensitive=false
synonyms={}
  io.confluent.support.metrics.submitters.KafkaSubmitter=INFO
sensitive=false synonyms={}
  kafka=INFO sensitive=false synonyms={}
  org.apache.kafka.common.protocol.Errors=INFO sensitive=false
synonyms={}
  org.apache.kafka.common.utils.LoggingSignalHandler=INFO
sensitive=false synonyms={}
  kafka.log.OffsetIndex=INFO sensitive=false synonyms={}
  org.apache.zookeeper.client.ZooKeeperSaslClient=INFO
sensitive=false synonyms={}
  org.apache.kafka.common.network.Selector=INFO sensitive=false
synonyms={}
  kafka.server.DelayedOperationPurgatory=INFO sensitive=false
synonyms={}
  org.apache.zookeeper.client.StaticHostProvider=INFO sensitive=false
synonyms={}
  org.apache.zookeeper.SaslServerPrincipal=INFO sensitive=false
synonyms={}
  kafka.log.LogConfig=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.producer.internals.ProducerBatch=INFO
sensitive=false synonyms={}
  com.yammer.metrics.reporting.JmxReporter=INFO sensitive=false
synonyms={}
  org.apache.http.impl.auth.HttpAuthenticator=INFO sensitive=false
synonyms={}
  org.apache.http.headers=INFO sensitive=false synonyms={}
  org.apache.avro.LogicalTypes=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.NetworkClient=INFO sensitive=false
synonyms={}
  kafka.network.Acceptor=INFO sensitive=false synonyms={}
  kafka.server.KafkaServer=INFO sensitive=false synonyms={}
  kafka.server.AdminManager=INFO sensitive=false synonyms={}
  kafka.network.Processor=INFO sensitive=false synonyms={}
  kafka.server.ReplicaFetcherManager=INFO sensitive=false synonyms={}

org.apache.kafka.clients.producer.internals.ProducerInterceptors=INFO
sensitive=false synonyms={}
  org.apache.http.impl.execchain.RedirectExec=INFO sensitive=false
synonyms={}
  kafka.producer.async.DefaultEventHandler=DEBUG sensitive=false
synonyms={}
  kafka.utils.CoreUtils$=INFO sensitive=false synonyms={}
  kafka.server.ClientQuotaManager$ThrottledChannelReaper=INFO
sensitive=false synonyms={}
  kafka.coordinator.group.GroupCoordinator=INFO sensitive=false
synonyms={}
  org.apache.kafka.clients.ClusterConnectionStates=INFO
sensitive=false synonyms={}
  kafka.controller.ZkPartitionStateMachine=INFO sensitive=false
synonyms={}
  org.apache.kafka.common.record.MultiRecordsSend=INFO
sensitive=false synonyms={}
  org.apache.http.impl.conn.CPool=INFO sensitive=false synonyms={}
  org.apache.kafka.common.network.NetworkReceive=INFO sensitive=false
synonyms={}
  kafka.coordinator.transaction.TransactionCoordinator=INFO
sensitive=false synonyms={}
  io.confluent.support.metrics.utils.WebClient=INFO sensitive=false
synonyms={}
  org.apache.kafka.common.config.AbstractConfig=INFO sensitive=false
synonyms={}
  org.apache.kafka.clients.producer.internals.RecordAccumulator=INFO
sensitive=false synonyms={}
  kafka.server.KafkaApis=INFO sensitive=false synonyms={}
  kafka.controller.TopicDeletionManager=INFO sensitive=false
synonyms={}
  org.apache.kafka.common.requests.FetchMetadata=INFO sensitive=false
synonyms={}
  kafka.server.FetchSessionCache=INFO sensitive=false synonyms={}
  kafka.coordinator.transaction.ProducerIdManager=INFO
sensitive=false synonyms={}
  io.confluent.metrics.reporter.KafkaMetricsHelper=INFO
sensitive=false synonyms={}
  org.apache.http.client.protocol.RequestAuthCache=INFO
sensitive=false synonyms={}
  kafka.server.KafkaRequestHandler=INFO sensitive=false synonyms={}
  io.confluent.support.metrics.BaseMetricsReporter=INFO
sensitive=false synonyms={}
  org.apache.http.impl.client.ProxyAuthenticationStrategy=INFO
sensitive=false synonyms={}
  org.apache.http.wire=INFO sensitive=false synonyms={}
  org.apache.http.conn.ssl.StrictHostnameVerifier=INFO
sensitive=false synonyms={}
  kafka.utils.Log4jControllerRegistration$=INFO sensitive=false
synonyms={}
  org.apache.http.impl.execchain.MainClientExec=INFO sensitive=false
synonyms={}
  kafka.server.IncrementalFetchContext=INFO sensitive=false
synonyms={}
  kafka.log.LogManager=INFO sensitive=false synonyms={}
  org.apache.http.impl.client.DefaultRedirectStrategy=INFO
sensitive=false synonyms={}
  org.apache.http.impl.conn.PoolingHttpClientConnectionManager=INFO
sensitive=false synonyms={}
  kafka.controller.ControllerEventManager$ControllerEventThread=INFO
sensitive=false synonyms={}
```

```
  org.apache.avro.file.CodecFactory=INFO sensitive=false synonyms={}
  kafka.controller.KafkaController=INFO sensitive=false synonyms={}
  kafka.server.KafkaConfig=INFO sensitive=false synonyms={}
  org.apache.http.conn.ssl.SSLConnectionSocketFactory=INFO
sensitive=false synonyms={}
  kafka.cluster.Partition=INFO sensitive=false synonyms={}
  kafka.server.DelayedCreatePartitions=INFO sensitive=false
synonyms={}
  kafka.coordinator.group.GroupMetadataManager=INFO sensitive=false
synonyms={}
  org.apache.zookeeper.server.ZooKeeperThread=INFO sensitive=false
synonyms={}
  kafka.log.AbstractIndex=INFO sensitive=false synonyms={}
  kafka.zk.AdminZkClient=INFO sensitive=false synonyms={}
  org.apache.http.conn.ssl.BrowserCompatHostnameVerifier=INFO
sensitive=false synonyms={}
  kafka.server.BrokerMetadataCheckpoint=INFO sensitive=false
synonyms={}
  kafka.network.RequestChannel=INFO sensitive=false synonyms={}
  kafka.request.logger=WARN sensitive=false synonyms={}
  io.confluent.support.metrics.BaseSupportConfig=INFO sensitive=false
synonyms={}
  org.apache.http.conn.ssl.DefaultHostnameVerifier=INFO
sensitive=false synonyms={}

kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThrea
d=INFO sensitive=false synonyms={}
  io.confluent.support.metrics.common.kafka.KafkaUtilities=INFO
sensitive=false synonyms={}
  kafka.log.LogCleanerManager$=INFO sensitive=false synonyms={}
  org.apache.http.client.protocol.RequestAddCookies=INFO
sensitive=false synonyms={}
  kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper=INFO
sensitive=false synonyms={}
  kafka.network.SocketServer=INFO sensitive=false synonyms={}
  io.confluent.support.metrics.SupportedServerStartable=INFO
sensitive=false synonyms={}
  kafka.coordinator.transaction.TransactionMarkerChannelManager=INFO
sensitive=false synonyms={}
  kafka.controller=TRACE sensitive=false synonyms={}
  org.apache.zookeeper.ClientCnxn=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.Metadata=INFO sensitive=false synonyms={}
  kafka.network.RequestChannel$=WARN sensitive=false synonyms={}
  kafka.log.LogSegment=INFO sensitive=false synonyms={}
  org.apache.http.impl.client.TargetAuthenticationStrategy=INFO
sensitive=false synonyms={}
  org.apache.kafka.common.security.JaasUtils=INFO sensitive=false
synonyms={}
  kafka.controller.RequestSendThread=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.producer.internals.ProducerMetadata=INFO
sensitive=false synonyms={}
  io.confluent.support.metrics.SupportedKafka=INFO sensitive=false
synonyms={}
  org.apache.kafka.clients.producer.KafkaProducer=INFO
```

```
sensitive=false synonyms={}
  root=INFO sensitive=false synonyms={}
  kafka.server.ReplicaManager=INFO sensitive=false synonyms={}
  kafka.server.epoch.LeaderEpochFileCache=INFO sensitive=false
synonyms={}
  kafka.server.FullFetchContext=INFO sensitive=false synonyms={}
  kafka.zookeeper.ZooKeeperClient=INFO sensitive=false synonyms={}
  org.apache.kafka.clients.producer.ProducerConfig=INFO
sensitive=false synonyms={}
  org.apache.kafka.common.metrics.JmxReporter=INFO sensitive=false
synonyms={}
  org.apache.kafka.common.utils.Utils=INFO sensitive=false
synonyms={}
  io.confluent.metrics.reporter.ConfluentMetricsReporterConfig=INFO
sensitive=false synonyms={}
  io.confluent.support.metrics.submitters.ConfluentSubmitter=INFO
sensitive=false synonyms={}
  org.apache.http.impl.conn.DefaultHttpResponseParser=INFO
sensitive=false synonyms={}
  kafka.controller.ZkReplicaStateMachine=INFO sensitive=false
synonyms={}
  org.apache.http.client.protocol.ResponseProcessCookies=INFO
sensitive=false synonyms={}
  kafka.authorizer.logger=WARN sensitive=false synonyms={}
  org.apache.kafka.common.requests.OffsetFetchRequest=INFO
sensitive=false synonyms={}
  io.confluent.support.metrics.KafkaSupportConfig=INFO
sensitive=false synonyms={}
  org.apache.http.impl.client.InternalHttpClient=INFO sensitive=false
synonyms={}
  kafka.server.FetchManager=INFO sensitive=false synonyms={}
  org.apache.kafka.common.utils.KafkaThread=INFO sensitive=false
synonyms={}
  kafka.log.LogCleaner=INFO sensitive=false synonyms={}
  org.apache.http.impl.execchain.RetryExec=INFO sensitive=false
synonyms={}
  org.apache.kafka.common.network.PlaintextChannelBuilder=INFO
sensitive=false synonyms={}
  kafka.zk.KafkaZkClient=INFO sensitive=false synonyms={}
  kafka.controller.ControllerChannelManager=INFO sensitive=false
synonyms={}
  kafka.log.TimeIndex=INFO sensitive=false synonyms={}
```

Specifically have a look for the debug levels defined for the various loggers, e.g. `root=INFO` meaning that the `root` logger has log level `INFO`.

2. Set the log level of the logger `kafka` to `DEBUG`:

```
$ kafka-configs --bootstrap-server localhost:9092 --alter --add
-config "kafka=DEBUG" --entity-type broker-loggers --entity-name 101
Completed updating config for broker-logger 101.
```

3. List the loggers again and double check that the log level of the logger `kafka` has indeed been set to `DEBUG`:

```
$ kafka-configs --bootstrap-server localhost:9092 --describe --entity
-type broker-loggers --entity-name 101 | grep kafka=
  kafka=DEBUG sensitive=false synonyms={}
```

4. Double check the log of the broker and assert that there are now log entries of type `DEBUG`:

```
$ docker-compose logs kafka | grep DEBUG
```

You should see quite a few entries that look similar to this:

```
...
kafka                | [2020-10-16 19:07:28,524] DEBUG Request key
TopicPartitionOperationKey(_confluent-metrics,11) unblocked 0 Produce
operations (kafka.server.DelayedOperationPurgatory)
kafka                | [2020-10-16 19:07:28,524] DEBUG Request key
TopicPartitionOperationKey(_confluent-metrics,11) unblocked 0
DeleteRecords operations (kafka.server.DelayedOperationPurgatory)
kafka                | [2020-10-16 19:07:28,524] DEBUG [Partition
_confluent-metrics-7 broker=101] High watermark updated from
(offset=68 segment=[0:97597]) to (offset=69 segment=[0:98653])
(kafka.cluster.Partition)
kafka                | [2020-10-16 19:07:28,524] DEBUG Request key
TopicPartitionOperationKey(_confluent-metrics,7) unblocked 0 Fetch
operations (kafka.server.DelayedOperationPurgatory)
kafka                | [2020-10-16 19:07:28,524] DEBUG Request key
TopicPartitionOperationKey(_confluent-metrics,7) unblocked 0 Produce
operations (kafka.server.DelayedOperationPurgatory)
kafka                | [2020-10-16 19:07:28,524] DEBUG Request key
TopicPartitionOperationKey(_confluent-metrics,7) unblocked 0
DeleteRecords operations (kafka.server.DelayedOperationPurgatory)
kafka                | [2020-10-16 19:07:28,524] DEBUG [Partition
_confluent-controlcenter-6-0-0-1-MetricsAggregateStore-repartition-0
broker=101] High watermark updated from (offset=11083
segment=[0:255904]) to (offset=11466 segment=[0:264235])
(kafka.cluster.Partition)
kafka                | [2020-10-16 19:07:28,524] DEBUG [ReplicaManager
broker=101] Produce to local log in 1 ms
(kafka.server.ReplicaManager)
...
```

5. Before you leave, never forget to reset the log level of the logger `kafka` to its original value `INFO`:

```
$> kafka-configs --bootstrap-server localhost:9092 --alter --add
-config "kafka=INFO" --entity-type broker-loggers --entity-name 101
Completed updating config for broker-logger 101.
```

Cleanup

1. Clean up your environment by running the following command:

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

Summary

In this exercise you have learned how to dynamically change the log level of log appenders of a broker that cannot be stopped and restarted, since it might be part of a production system. It is important to reset the log level to its original value once our debugging session has ended.

⚠	Remember what we said in the introduction to the lab, when you are going to troubleshoot a production system then security is an important topic. It is important to know that access to JMX metrics of a component running on a production cluster can be secured with authentication and/or TLS encryption. But this is beyond the scope of this lab.

b. Not enough ISRs

In this exercise we're going to troubleshoot the situation where we do not have enough in-sync replicas for normal operation.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-isr
```

2. Run the Confluent Platform:

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

3. Create a topic `sample-topic`:

```
$ kafka-topics --bootstrap-server kafka-1:19092 --create \
    --topic sample-topic \
    --partitions 10 \
    --replication-factor 3
Created topic "sample-topic".
```

Producing loads of data

1. To produce a lot of data we will use the Kafka performance test tool:

```
$ kafka-producer-perf-test \
    --record-size 500 \
    --throughput 1000 \
    --num-records 10000000 \
    --topic sample-topic \
    --producer.config ~/confluent-cao/solution/ts-
isr/config/producer.properties \
    --print-metrics
5002 records sent, 999.8 records/sec (0.48 MB/sec), 24.7 ms avg
latency, 499.0 ms max latency.
5004 records sent, 996.6 records/sec (0.48 MB/sec), 14.6 ms avg
latency, 125.0 ms max latency.
5021 records sent, 1003.4 records/sec (0.48 MB/sec), 6.5 ms avg
latency, 113.0 ms max latency.
5007 records sent, 1001.0 records/sec (0.48 MB/sec), 9.2 ms avg
latency, 132.0 ms max latency.
...
...
```

2. Now we also need to consume the data to have a complete data pipeline. Open another terminal and run the Kafka console consumer:

```
$ NS=interceptor.classes=io.confluent.monitoring.clients.interceptor
&& \
  kafka-console-consumer \
    --group test-consumer \
    --bootstrap-server kafka-1:19092 \
    --consumer-property ${NS}.MonitoringConsumerInterceptor \
    --from-beginning \
    --topic sample-topic
SSXVNJHPDQDXVCRASTVYBCWVMGNYKRXVZXKGXTSPSJDGYLUEGQFLAQLOCFLJBEPOWFNSO
MYARHAOPUFOJHHDXEHXJBHWGSMZJGNLONJVXZXZOZITKXJBOZWDJMCBOSYQQKCPRRDCZW
MRLFXBLGQPRPGRNTAQOOSVXPKJPJLAVSQCCRXFRROLLHWHOHFGCFWPNDLMWCSSHWXQQYK
ALAAWCMXYLMZALGDESKKTEESEMPRHROVKUMPSXHELIDQEOOHOIHEGJOAZBVPUMCHSHGXZ
YXXQRUICRIJGQEBBWAXABQRIRUGZJUUVFYQOVCDEDXYFPRLGSGZXSNIAVODTJKSQWHNWV
PSAMZKOUDTWHIORJSCZIQYPCZMBYWKDIKOKYNGWPXZWMKRDCMBXKFUILWDHBFXRFAOPRU
GDFLPDLHXXCXCUPLWGDPPHEMJGMTVMFQQFVCUPOFYWLDUEBICKPZKHKVMCJVWVKTXBKAP
WAPENUEZNWNWDCACD
SSXVNJHPDQDXVCRASTVYBCWVMGNYKRXVZXKGXTSPSJDGYLUEGQFLAQLOCFLJBEPOWFNSO
MYARHAOPUFOJHHDXEHXJBHWGSMZJGNLONJVXZXZOZITKXJBOZWDJMCBOSYQQKCPRRDCZW
MRLFXBLGQPRPGRNTAQOOSVXPKJPJLAVSQCCRXFRROLLHWHOHFGCFWPNDLMWCSSHWXQQYK
ALAAWCMXYLMZALGDESKKTEESEMPRHROVKUMPSXHELIDQEOOHOIHEGJOAZBVPUMCHSHGXZ
YXXQRUICRIJGQEBBWAXABQRIRUGZJUUVFYQOVCDEDXYFPRLGSGZXSNIAVODTJKSQWHNWV
PSAMZKOUDTWHIORJSCZIQYPCZMBYWKDIKOKYNGWPXZWMKRDCMBXKFUILWDHBFXRFAOPRU
GDFLPDLHXXCXCUPLWGDPPHEMJGMTVMFQQFVCUPOFYWLDUEBICKPZKHKVMCJVWVKTXBKAP
WAPENUEZNWNWDCACD
SSXVNJHPDQDXVCRASTVYBCWVMGNYKRXVZXKGXTSPSJDGYLUEGQFLAQLOCFLJBEPOWFNSO
MYARHAOPUFOJHHDXEHXJBHWGSMZJGNLONJVXZXZOZITKXJBOZWDJMCBOSYQQKCPRRDCZW
MRLFXBLGQPRPGRNTAQOOSVXPKJPJLAVSQCCRXFRROLLHWHOHFGCFWPNDLMWCSSHWXQQYK
ALAAWCMXYLMZALGDESKKTEESEMPRHROVKUMPSXHELIDQEOOHOIHEGJOAZBVPUMCHSHGXZ
YXXQRUICRIJGQEBBWAXABQRIRUGZJUUVFYQOVCDEDXYFPRLGSGZXSNIAVODTJKSQWHNWV
PSAMZKOUDTWHIORJSCZIQYPCZMBYWKDIKOKYNGWPXZWMKRDCMBXKFUILWDHBFXRFAOPRU
GDFLPDLHXXCXCUPLWGDPPHEMJGMTVMFQQFVCUPOFYWLDUEBICKPZKHKVMCJVWVKTXBKAP
WAPENUEZNWNWDCACD
...
...
```

Introducing network latency for broker 103

We now want to simulate a bad network between broker 103 and the rest of the Kafka cluster.

1. Open another terminal and navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-isr
```

2. Install the `libmnl` package on `kafka-3` so the `tc` tool is available:

```
$ docker-compose exec -u root kafka-3 /config/install-libmnl.sh
```

Ignore the `NOKEY` Warning message if it appears.

3. Use the `tc` tool to simulate a bad network connection on `kafka-3`:

```
$ docker-compose exec -u root kafka-3 tc qdisc add dev eth0 root tbf
rate 1Mbit burst 32Kb latency 400ms
```

Here is what each option means:

qdisc: modify the scheduler (aka queuing discipline)
add: add a new rule
dev eth0: rules will be applied on device eth0
root: modify the outbound traffic scheduler (aka known as the egress qdisc)
tbf: use the token buffer filter to manipulate traffic rates
rate: sustained maximum rate
burst: maximum allowed burst
latency: packets with higher latency get dropped

4. Verify the outbound traffic scheduler rule is present:

```
$ docker-compose exec -u root kafka-3 tc qdisc show dev eth0
qdisc tbf 8002: root refcnt 2 rate 1Mbit burst 32Kb lat 400.0ms
```

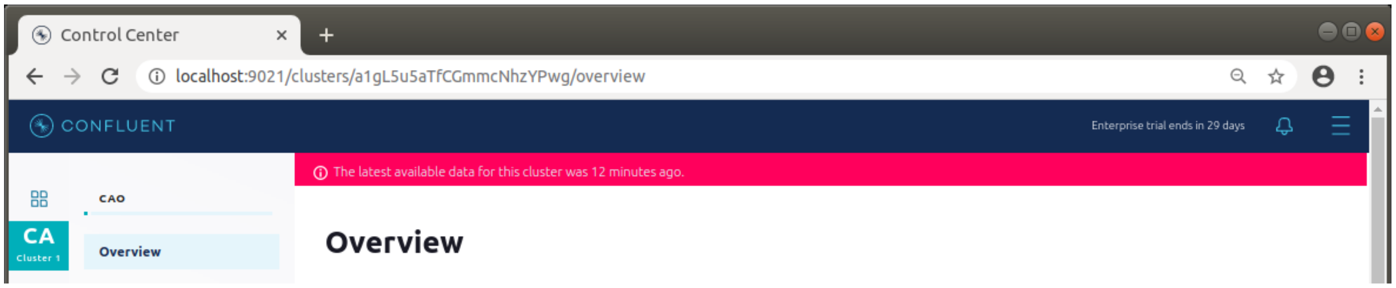
5. In the terminal where the producer is running you should see warnings like this:

```
[2020-10-16 19:27:53,601] WARN failed to produce for
client_id=producer-1 client_type=PRODUCER session=
cluster=0dtzDgeFSQeGGR9OBfcXfw
(io.confluent.monitoring.clients.interceptor.MonitoringInterceptor)
org.apache.kafka.common.errors.NotEnoughReplicasException: Messages
are rejected since there are fewer in-sync replicas than required.
[2020-10-16 19:27:53,601] WARN failed to produce for
client_id=producer-1 client_type=PRODUCER session=
cluster=0dtzDgeFSQeGGR9OBfcXfw
(io.confluent.monitoring.clients.interceptor.MonitoringInterceptor)
org.apache.kafka.common.errors.NotEnoughReplicasException: Messages
are rejected since there are fewer in-sync replicas than required.
```

which indicates that our system is troubled and has too little ISRs.

It may take a moment before the warnings begin to appear.

6. If we were to examine Control Center, we would see it has become ineffective since it cannot update its own topics anymore correctly.


Control Center
localhost:9021/clusters/a1gL5u5aTfCGmmcNhzYPwg/overview
CONFLUENT
Enterprise trial ends in 29 days
CAO
CA
Cluster 1
Overview

The latest available data for this cluster was 12 minutes ago.

Overview

This is one example of why Confluent recommends letting Control Center run on its own Kafka cluster!

7. Double check that all components of the platform are still running:

```
$ docker-compose ps
```

All components should be up and running...

8. Let's then fix the network problem of `kafka-3`. Remove the throughput limit from this container's network endpoint:

```
$ docker-compose exec -u root kafka-3 tc qdisc del dev eth0 root
```

9. Observe that
   - immediately the producer starts producing again.
   - immediately the consumer starts consuming again.
   - Confluent Control Center is recovering and taking up normal operations

Cleanup

1. Stop the producer by pressing `Ctrl+C`.
2. Stop the consumer by pressing `Ctrl+C`.
3. Clean up your environment by running the following command:

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

Summary

In this exercise you have learned how to identify issues with the Kafka producer causes by a suboptimal network connection. We have been using the `kafka-producer-perf-test` tool to simulate a producer, the `kafka-console-consumer` to simulate a consumer and the `tc` Linux tool to cripple the network connection of broker 103.

Confluent recommends that in a production environment you run the Confluent Control Center on its own Kafka cluster to decouple it from any malfunction in the production cluster itself!

c. Tuning Brokers

In this exercise you will use a Prometheus and Grafana based dashboard to monitor the single broker in your Kafka cluster. The primary tasks of this exercise are:

1. Run your Kafka based event streaming platform
2. Create a sample topic
3. Use the Kafka producer perf tool to write (a lot of) data into the sample topic
4. Use the Kafka console consumer to consume messages from the sample topic
5. Use a given dashboard in Grafana that contains charts and counters for important broker metrics
6. Observe the trends

7. Experiment with the producer and consumer and observe the influence on your dashboard

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/tu-brokers
```

2. Run the Confluent Platform:

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

Analyze the Prometheus & Grafana Setup

1. Open the file `docker-compose.yml` in the project folder and note the following:
   a. In addition to a ZooKeeper and a Broker instance, we are using Prometheus `2.17.0` and Grafana `6.7.2` containers.
   b. The broker has the `JMX_PORT` and `JMX_HOSTNAME` environment variables set and maps port 9999 to the host so that JMX metrics are generated and available to inspect e.g. from the host with **JConsole**.
   c. The broker has the `KAFKA_OPTS` environment variable set with instructions to start the JMX to Prometheus exporter. The exporter uses the `agents/kafka.yml` configuration file. Have a look into that file. It contains the definition of which JMX MBeans to export and how to call them in Prometheus.
2. The Prometheus service uses the configuration file found in `prometheus/prometheus.yml`. Also have a look into that file. It contains the job descriptions for the Prometheus server on which sources to poll metrics from and how often
3. The Grafana service uses a volume called `grafana-storage` to persist its data. It also gets provisioned upon start with the files available in the folder `grafana/provisioning`. There are a data source and a dashboard provider YAML file. The latter points to the predefined dashboard `broker-tuning.json` which is located in the folder `grafana/dashboards`.

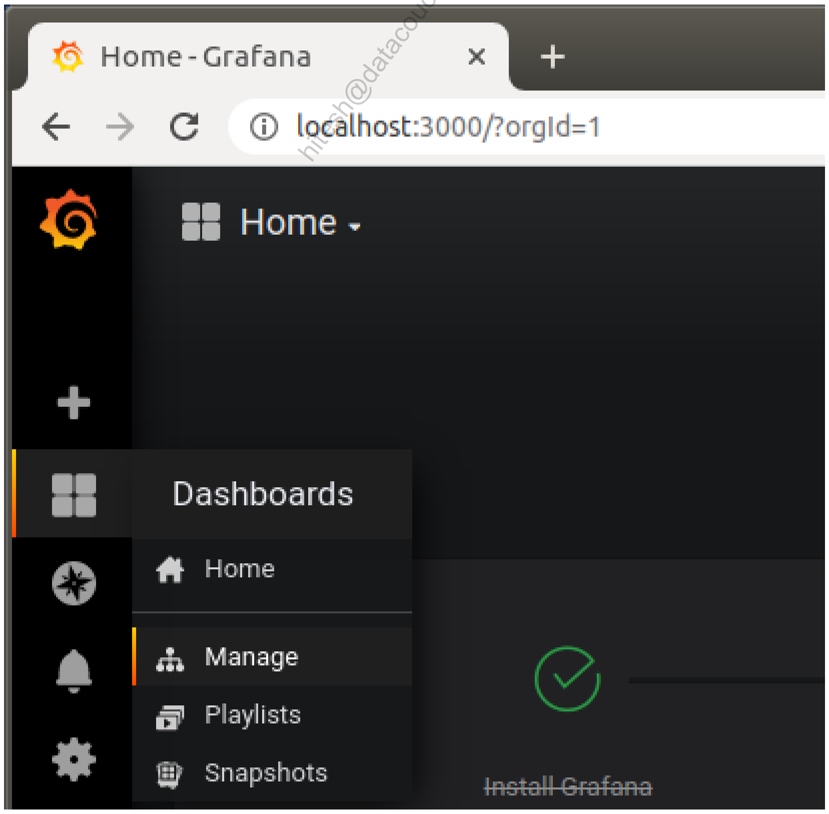
Use the Prometheus UI

To experiment with available metrics you can use the simple Prometheus UI.

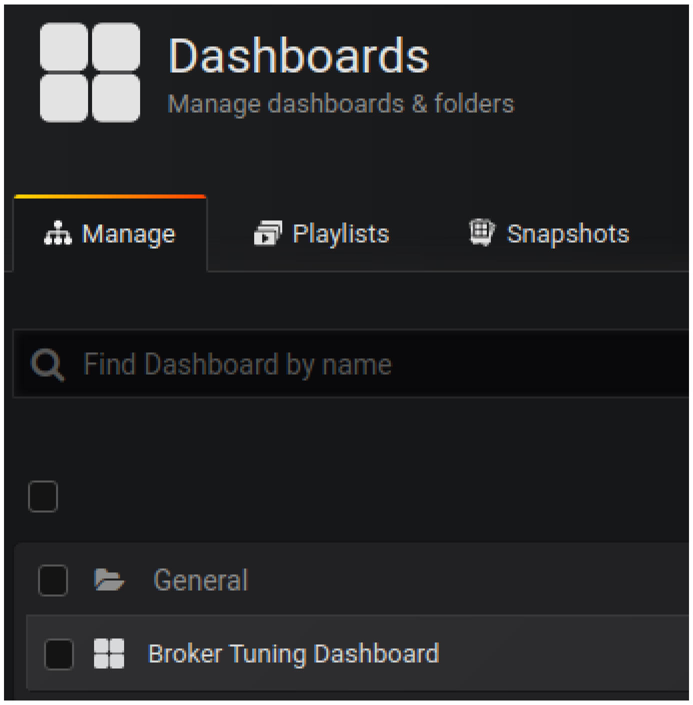
1. Navigate to http://localhost:9090
2. Explore and experiment with the available metrics.
3. Find the metrics that are most probable to show relevant information while tuning the broker

Open the Grafana Dashboard

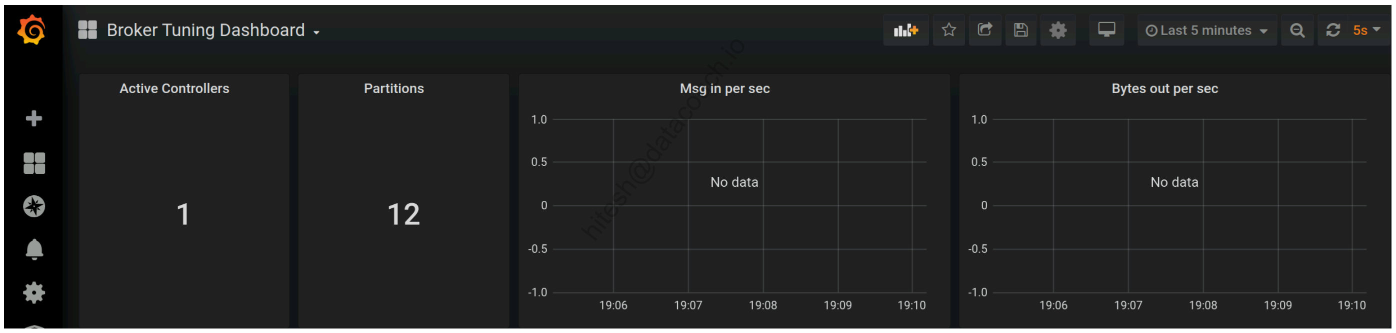
1. Navigate to http://localhost:3000
2. The first time you login with **username** `admin` and **password** `admin`. When prompted, you can change the default password or select **Skip**.
3. On the home page, mouse over the dashboard icon and click **Manage**:



4. In the dashboard list, click **Broker Tuning Dashboard**:



You should see something like this:



We can see that our Kafka cluster has one single active controller and 12 partitions. The charts don't show any data yet since we have not generated or consumed any.

5. While you continue with the exercise keep the dashboard visible to see changes.

Tune the Broker

1. Back in the terminal, create a new topic:

```
$ kafka-topics --create \
    --bootstrap-server kafka:9092 \
    --topic test-topic \
    --replication-factor 1 \
    --partitions 1
Created topic "test-topic".
```

Observe how the number of partitions on the dashboard after some time reaches `2`. The value we are seeing is the average number of partions over the `Last 5 minutes`. The time range can be seen at the upper right toolbar. Our cluster has 2 topics, each with a single partition.

2. Let's create some data:

```
$ kafka-producer-perf-test \
    --topic test-topic \
    --record-size 200 \
    --throughput 1000 \
    --num-records 10000000 \
    --producer-props \
        bootstrap.servers=kafka:9092
```

   Observe on the dashboard how the **Msg in per sec** chart changes. The **Bytes out per sec** chart remains at zero since we're not consuming yet.

3. Wait until the production rate reaches a stable plateau. How high is it? Note it down for further reference.

4. Stop the producer with `Ctrl+C` and observe how the line in the **Msg in per sec** chart approaches zero.

5. Open a new terminal and run a consumer to produce some movement in the **Bytes out per sec** chart:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --group test-consumer-group \
    --topic test-topic \
    --from-beginning
```

   Observe the dashboard.

6. Now play with the broker settings that we have discussed in the theory, that influence the throughput and latency of the broker. Use the Grafana dashboard as a feedback loop.

Cleanup

1. Press `Ctrl+C` to stop the consumer.

2. Execute the following command to cleanup the system:

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

Summary

In this lab you have examined a Prometheus and Grafana based dashboard for your broker. You have then experimented with various producer and consumer settings and observed their effect on the dashboard.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 09 Troubleshooting the Schema Registry

a. Troubleshooting the Schema Registry

In this exercise we're going to troubleshoot the Confluent Schema Registry.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-sr
```

2. Run the Confluent Platform:

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

Generating Schemas

1. Have a look into the subfolder `schemas` of the project folder and look into the three files `test.json`, `test-v2.json` and `test-v2-default.json` that are located there. You will be using those in the following steps.

2. Create a schema named `test` with two simple fields `string field1` and `int field2`:

```
$ curl -X POST \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    -H "Accept: application/vnd.schemaregistry.v1+json" \
    -d "@schemas/test.json" \
    schema-registry:8081/subjects/test/versions
```

The response should be:

```
{"id":1}
```

If the response is different than `{"id":1}`, Schema Registry may still be starting. Wait a bit and retry the command.

3. List all registered schemas:

```
$ curl -s schema-registry:8081/subjects | jq .
[
  "test"
]
```

4. Let's see what versions of the schema `test` are installed:

```
$ curl -s schema-registry:8081/subjects/test/versions | jq .
[
  1
]
```

5. Get version 1 of the schema `test`:

```
$ curl -s schema-registry:8081/subjects/test/versions/1 | jq .
{
  "subject": "test",
  "version": 1,
  "id": 21,
  "schema":
"{\"type\":\"record\",\"name\":\"test\",\"namespace\":\"app\",\"field
s\":[{\"name\":\"field1\",\"type\":\"string\"},{\"name\":\"field2\",\
"type\":\"int\"}]}"
}
```

6. To get the unescaped **schema** use this command:

```
$ curl -s schema-registry:8081/subjects/test/versions/1/schema | jq .
{
  "type": "record",
  "name": "test",
  "namespace": "app",
  "fields": [
    {
      "name": "field1",
      "type": "string"
    },
    {
      "name": "field2",
      "type": "int"
    }
  ]
}
```

which is much nicer to use...

7. Let's see what the **compatibility** setting of our Schema Registry is:

```
$ curl -s schema-registry:8081/config | jq .
{
  "compatibilityLevel": "BACKWARD"
}
```

OK, so it's `BACKWARD` as expected. For a detailed description of the possible compatibility settings and their meaning refer to the documentation:

- https://docs.confluent.io/current/schema-registry/docs/api.html#compatibility
- https://docs.confluent.io/current/schema-registry/docs/avro.html#compatibility-types

  In the case of `BACKWARD` compatibility the following changes to the schema are allowed:

- Delete fields
- Add optional fields

8. Let's now see if our `v2` of the `test` schema fulfills the compatibility test:

```
$ curl -X POST -s \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    -H "Accept: application/vnd.schemaregistry.v1+json" \
    -d "@schemas/test-v2.json" \
    schema-registry:8081/compatibility/subjects/test/versions/latest
| jq .
{
  "is_compatible": false
}
```

OK, so, adding a field to a new version of the schema is not compatible with the latest version! The reason is, that we did not specify a default value for the new field!

9. Let's do it again but this time use a `v2` schema that includes a default value for the new field:

```
$ curl -X POST -s \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    -H "Accept: application/vnd.schemaregistry.v1+json" \
    -d "@schemas/test-v2-default.json" \
    schema-registry:8081/compatibility/subjects/test/versions/latest
| jq .
{
  "is_compatible": true
}
```

and this time it works (compare the two files `test-v2.json` and `test-v2-default.json`).

10. So, let's then store this new version of the schema:

```
$ curl -X POST -s \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    -H "Accept: application/vnd.schemaregistry.v1+json" \
    -d "@schemas/test-v2-default.json" \
    schema-registry:8081/subjects/test/versions | jq .
{
  "id": 2
}
```

11. Listing all versions gives us now:

```
$ curl -s schema-registry:8081/subjects/test/versions | jq .
[
  1,
  2
]
```

Using the Maven plugin

There is a Maven plugin called `kafka-schema-registry-maven-plugin` that can be used during development to work with the Schema Registry.

1. Open a new terminal window
2. Execute a Maven container interactively and attach it to the same network as the Kafka cluster:

   ```
   $ docker run --rm -it \
       --net ts-sr_confluent \
       -v "${HOME}/.m2-docker":/root/.m2 \
       -v "${HOME}/confluent-cao/solution/ts-sr/maven":/maven \
       -w /maven \
       maven:3.5.3-jdk-8-alpine /bin/sh
   /maven #
   ```

   Note how we are mounting the subfolder `maven` of the project folder into the container.

3. From the `maven` directory within the docker container, verify that it contains an `avro` directory which contains three schema files `user.avsc`, `user.v2.avsc` and `user.v3.avsc`.

   ```
   /maven # ls avro
   user.avsc      user.v2.avsc  user.v3.avsc
   ```

4. First we want to use the plugin's goal `schema-registry:register` to register the AVRO schema `avro/user.avsc`:

   a. On the host, with your favorite editor open the `maven/pom.xml` file and notice the snippet needed to use the `schema-registry:register` goal:

```
<configuration>
    <schemaRegistryUrls>
        <param>http://schema-registry:8081</param>
    </schemaRegistryUrls>
    ...
    <subjects>
        <users-value>avro/user.avsc</users-value>
    </subjects>
</configuration>
```

We need to provide the info where to locate the schema registry and then a list of schemas to register under `<subjects>`. The tag names correspond to the subject name under which the respective schema will be registered in the registry. In our case the subject name will be `users-value`

We call the subject `users-value` since later on we will discuss the integration of Schema Registry with Confluent Control Center. There we will create a topic called `users` and create some messages using the AVRO schema that we're defining here. By default the AVRO serializers use a default policy to create the subject name which is `<topic name>-key` for the AVRO schema used for the **key** of the topic records, and `<topic name>-value` for the AVRO schema used for the **value** of the topic records.

b. Execute the Maven goal to register the schema:

```
/maven # mvn schema-registry:register
```

The output generated should end in:

```
...
[INFO] --- kafka-schema-registry-maven-plugin:6.0.0:register
(default-cli) @ sample-app ---
...
Downloading ...
Downloading ...
...
[INFO] Registered subject(users-value) with id 3 version 1
[INFO]
------------------------------------------------------------------
------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------
------
[INFO] Total time: 19.061 s
[INFO] Finished at: 2020-10-16T21:21:09Z
[INFO]
------------------------------------------------------------------
------
```

5. Now we can use the goal `schema-registry:test-compatibility` to make sure a set of given schemas is compatible with the content of the schema registry:

   a. To the node `<subjects>` in the file `pom.xml` add another subject referencing the schema `users.v2.avsc` such as that the result looks like this:

   ```
   <subjects>
       <users-value>avro/user.avsc</users-value>
       <users-value>avro/user.v2.avsc</users-value>
   </subjects>
   ```

   b. Execute the goal:

   ```
   /maven # mvn schema-registry:test-compatibility
   ```

   The output should look similar to this:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< cio.confluent.test:sample-app
>--------------------
[INFO] Building sample-app 1
[INFO] --------------------------------[ jar
]---------------------------------
[INFO]
[INFO] --- kafka-schema-registry-maven-plugin:6.0.0:test-
compatibility (default-cli) @ sample-app ---
[ERROR] Schema /maven/avro/user.v2.avsc is not compatible with
subject(users-value)
[INFO]
-------------------------------------------------------------------
------
[INFO] BUILD FAILURE
[INFO]
-------------------------------------------------------------------
------
[INFO] Total time: 1.262 s
[INFO] Finished at: 2020-10-16T21:23:10Z
[INFO]
-------------------------------------------------------------------
------
[ERROR] Failed to execute goal io.confluent:kafka-schema-registry-
maven-plugin:6.0.0:test-compatibility (default-cli) on project
sample-app: Execution default-cli of goal io.confluent:kafka-
schema-registry-maven-plugin:6.0.0:test-compatibility failed: One
or more schemas found to be incompatible with the current version.
-> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven
with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug
logging.
[ERROR]
[ERROR] For more information about the errors and possible
solutions, please read the following articles:
[ERROR] [Help 1]
http://cwiki.apache.org/confluence/display/MAVEN/PluginExecutionEx
ception
```

Which indicates that the schema `user.v2.avsc` is not compatible with subject `users-value` that exists in our schema registry.

6. Remembering that one can only add optional fields with the compatibility mode that our registry is working in, let's try with `avro/user.v3.avsc`:

   a. First update the `pom.xml` to list the `avro/user.v3.avsc` instead of `avro/user.v2.avsc` under `<subjects>`. It should now look like this:

```
<subjects>
    <users-value>avro/user.avsc</users-value>
    <users-value>avro/user.v3.avsc</users-value>
</subjects>
```

b. Then run the goal `schema-registry:test-compatibility` again:

```
/maven # mvn schema-registry:test-compatibility
```

This time the output looks like this:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< cio.confluent.test:sample-app
>--------------------
[INFO] Building sample-app 1
[INFO] --------------------------------[ jar
]---------------------------------
[INFO]
[INFO] --- kafka-schema-registry-maven-plugin:6.0.0:test-
compatibility (default-cli) @ sample-app ---
[INFO] Schema /maven/avro/user.v3.avsc is compatible with
subject(users-value)
[INFO]
------------------------------------------------------------------
------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------
------
[INFO] Total time: 1.325 s
[INFO] Finished at: 2020-10-16T21:24:36Z
[INFO]
------------------------------------------------------------------
------
```

7. Now that we know that it is compatible with the previous version of the schema, let's register `v3` of this schema:

```
/maven # mvn schema-registry:register
```

which generates this output:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< cio.confluent.test:sample-app
>--------------------
[INFO] Building sample-app 1
[INFO] --------------------------------[ jar
]---------------------------------
[INFO]
[INFO] --- kafka-schema-registry-maven-plugin:6.0.0:register
(default-cli) @ sample-app ---
[INFO] Registered subject(users-value) with id 4 version 2
[INFO]
---------------------------------------------------------------------
---
[INFO] BUILD SUCCESS
[INFO]
---------------------------------------------------------------------
---
[INFO] Total time: 1.453 s
[INFO] Finished at: 2020-10-16T21:25:12Z
[INFO]
---------------------------------------------------------------------
---
```

confirming that the new version has been registered successfully with `id=4` and `version=2`.

8. We can also download schemas from our Schema Registry using the goal `schema-registry:download`:

   a. First, inspect the node `<plugin><configuration>` of the `pom.xml` file and make sure you find the following snippet:

   ```
   <outputDirectory>src/main/avro</outputDirectory>
   <subjectPatterns>
       <param>^users-value$</param>
   </subjectPatterns>
   ```

   the `<subjectPatterns>` node contains a list of **regex patterns** of subjects to download from the schema registry. In our case we just want to download subject `users-value`.

   b. run the goal

```
/maven # mvn schema-registry:download
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< cio.confluent.test:sample-app
>--------------------
[INFO] Building sample-app 1
[INFO] --------------------------------[ jar
]---------------------------------
[INFO]
[INFO] --- kafka-schema-registry-maven-plugin:6.0.0:download
(default-cli) @ sample-app ---
[INFO] Getting all subjects on schema registry...
[INFO] Schema Registry has 2 subject(s).
[INFO] Downloading latest metadata for users-value.
[INFO] Writing schema for Subject(users-value) to
/maven/src/main/avro/users-value.avsc.
[INFO]
------------------------------------------------------------------
------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------
------
[INFO] Total time: 1.357 s
[INFO] Finished at: 2020-10-16T21:27:02Z
[INFO]
------------------------------------------------------------------
------
```

c. Assert that a file `/maven/src/main/avro/users-value.avsc` has been downloaded and verify that it contains `v3` of the schema.

```
/maven # ls src/main/avro
users-value.avsc
```

9. Exit the `maven` container with `Ctrl+D`.

Integration with Confluent Control Center

1. Create a `users` topic:

```
$ kafka-topics --bootstrap-server kafka:9092 \
     --create \
     --partitions 1 \
     --replication-factor 1 \
     --topic users
Created topic "users".
```

2. Open the Confluent Control Center at http://localhost:9021.
3. Wait until Control Center is initialized and the **Clusters** view displays the **cao** cluster with 1 broker running.
4. Click the **cao** cluster and navigate to **Topics**.
5. In the list of topics, click **users**.
6. Click the **Messages** tab. There will be nothing displayed at this time...
7. Produce some messages to Topic `users`.

   a. Execute `kafka-avro-console-producer`:

```
$ schema='{
    "type":"record",
    "namespace": "example.avro",
    "name":"user",
    "fields":[{"name":"name","type":"string"},
              {"name":"favorite_number","type":"int"}]
  }' && \
  kafka-avro-console-producer --bootstrap-server kafka:9092 \
    --topic users \
    --property schema.registry.url=http://schema-registry:8081 \
    --property value.schema="${schema}"
```

After starting `kafka-avro-console-consumer`, it is waiting for records to be entered. If a record is entered that does not match the expected schema, an exception will occur and `kafka-avro-console-consumer` will terminate with an exception. Pressing **Enter** without first providing a record will result in an exception as well.

```
org.apache.kafka.common.errors.SerializationException:
Error deserializing json  to Avro of schema
{"type":"record","name":"user","namespace":"example.avro
","fields":[{"name":"name","type":"string"},{"name":"fav
orite_number","type":"int"}]}
Caused by: java.io.EOFException
...
```

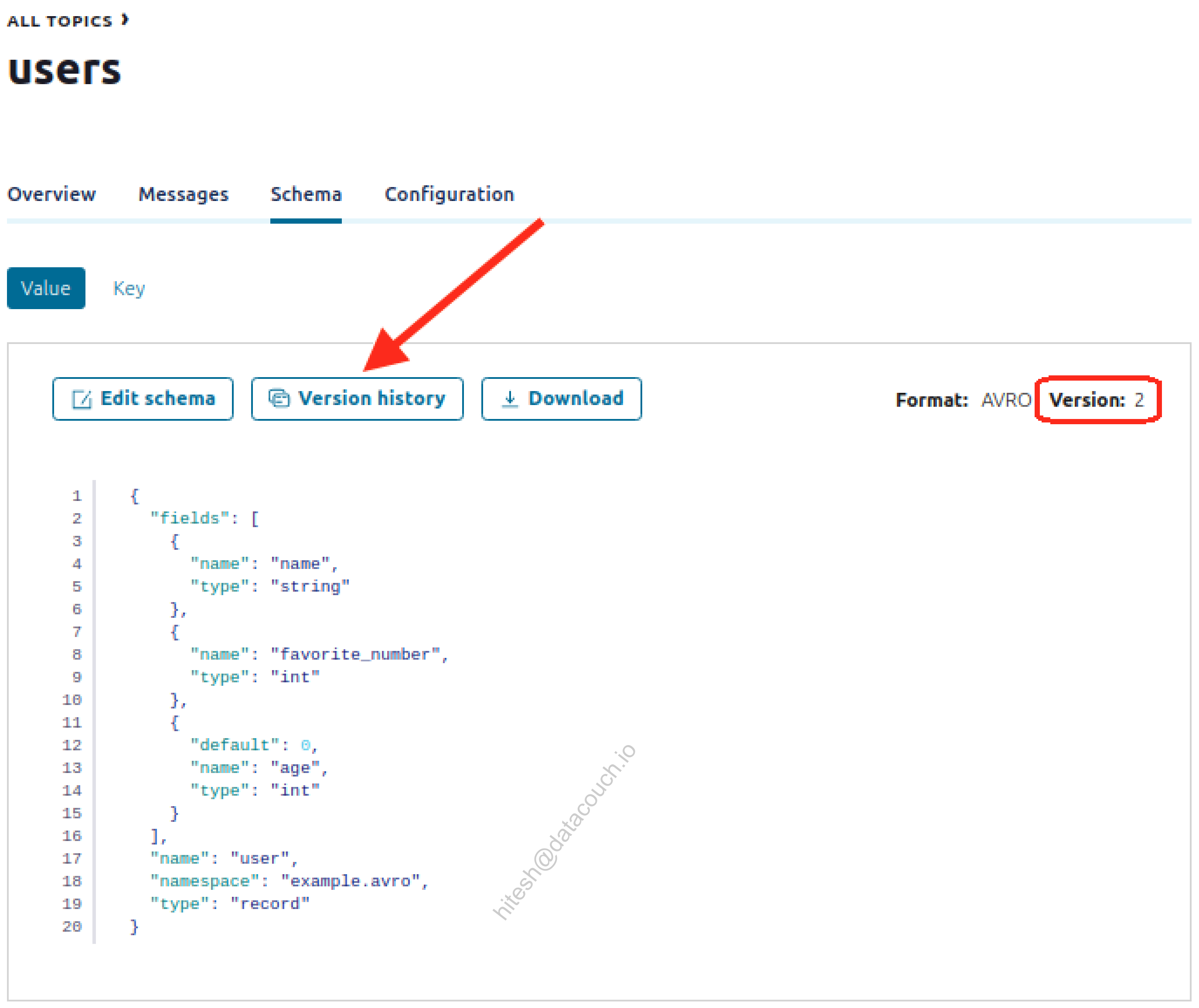
b. Produce a message to the topic by entering the following text and pressing **Enter**:

```
{"name":"Betty","favorite_number":3}
```

c. Produce two additional messages using the following text:

```
{"name":"Joe","favorite_number":21}
{"name":"Adele","favorite_number":13}
```

8. Exit the producer with `Ctrl+C`.
9. Make sure you see the records appearing in Control Center on the **Messages** tab.
10. Switch to the **Schema** tab and verify that you see the `user` schema that we had registered with Schema Registry before. Also verify that the version of the schema is indicated as `2`:



```
{
  "fields": [
    {
      "name": "name",
      "type": "string"
    },
    {
      "name": "favorite_number",
      "type": "int"
    },
    {
      "default": 0,
      "name": "age",
      "type": "int"
    }
  ],
  "name": "user",
  "namespace": "example.avro",
  "type": "record"
}
```

11. Click on **Version history** and verify that so far you have versions 1 and 2 available, where version2 is marked as **current**.
12. Click the checkbox **Turn on version diff** and observe how you see the two versions side by side. In this view you should be able to notice the differences quickly.
13. Run the producer with schema version`v3`:

```
$ schema='{
    "type":"record",
    "namespace": "example.avro",
    "name":"user",
    "fields":[{"name":"name","type":"string"},
              {"name":"favorite_number","type":"int"},
              {"name":"age","type":"int","default":0}]
  }' && \
  kafka-avro-console-producer --bootstrap-server kafka:9092 \
    --topic users \
    --property schema.registry.url=http://schema-registry:8081 \
    --property value.schema="${schema}"
```

14. Add some records with using this new version of the schema:

```
{"name":"David","favorite_number":33,"age":11}
{"name":"Luana","favorite_number":5,"age":21}
{"name":"Sarah","favorite_number":2,"age":55}
```

15. Exit the producer with `Ctrl+C`.

16. Return to Control Center and observe new messages.

    a. On the **Messages** tab, set the `Offset` value to `3`.

    b. Observe the new message that include the `age` field.

Cleanup

1. Clean up your environment by running the following command:

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

Summary

In this exercise you have learned how to explicitly register a new schema, check the compatibility of a new version of the schema with the last version in the Schema Registry and then store the new version, IF it is compatible.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 10 Troubleshooting & Tuning Producers

a. Troubleshooting the Producer

In this exercise we're going to troubleshoot a Kafka Producer.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-prod
```

2. Run the Confluent Platform:

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

In the `docker-compose.yml` file the `base` service has been added the capability `NET_ADMIN` so we can use tools to artificially throttle the network throughput.

3. Execute into the `base` container

```
$ docker-compose exec -u root base bash
[root@base appuser]#
```

4. Install the `libmnl` package so the `tc` tool is available:

```
[root@base appuser]# /config/install-libmnl.sh
```

Ignore the `NOKEY` Warning message if it appears.

5. Now introduce a constant delay of 200ms to the `eth0` interface. This is a good way to simulate a bad network between producer and Kafka cluster. Execute:

```
[root@base appuser]# tc qdisc add dev eth0 root netem delay 200ms
```

Here is what each option means:

qdisc: modify the scheduler (aka queuing discipline)
add: add a new rule
dev eth0: rules will be applied on device eth0
root: modify the outbound traffic scheduler (aka known as the egress qdisc)
netem: use the network emulator to emulate a WAN property
delay: the network property that is modified
200ms: introduce delay of 200 ms

6. Test it by pinging `kafka`:

```
[root@base appuser]# ping -c 3 kafka
PING kafka (172.22.0.6) 56(84) bytes of data.
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=1 ttl=64
time=436 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=2 ttl=64
time=201 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=3 ttl=64
time=200 ms

--- kafka ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 5ms
rtt min/avg/max/mdev = 200.351/279.148/436.303/111.126 ms
```

Note the `time=200 ms` for each ping after the initial ping! Normally a ping is in the sub-millisecond range.

Testing the performance

In this section of the lab you are going to measure how different network conditions affect the overall performance of a producer. We are using artificial delays (could simulate a big geographical distance between Kafka client [producer] and the Kafka cluster) as well as artificial package loss (could be a bad network connection).

1. Create the topic `stress-topic`:

```
[root@base appuser]# kafka-topics --bootstrap-server kafka:9092 \
    --create \
    --topic stress-topic \
    --partitions 10 \
    --replication-factor 1
```

2. Run a performance test using the `kafka-producer-perf-test` tool:

```
[root@base appuser]# kafka-producer-perf-test \
    --record-size 200 \
    --throughput 1000 \
    --num-records 20000 \
    --topic stress-topic \
    --producer.config /config/producer.properties \
    --print-metrics
```

The `--print-metrics` parameter in the above command will make the tool print out a list of performance relevant metrics at the end of its operation. With the knowledge that you acquired in this module and the modules before, try to identify the most important metrics.

3. Note down the values for:
   - Outgoing byte rate
   - Request Rate
   - Request latency avg
   - Response Rate

4. Now change the network delay to `20ms`:

```
[root@base appuser]# tc qdisc change dev eth0 root netem delay 20ms
```

5. Run the stress test again and write up the same metrics as in the previous attempt and compare them.

   Here are values from my two tests:

	Throttle	**Throttle**

Metric Name	delay 200ms	delay 20ms
producer-metrics:outgoing-byte-rate	85264.806	95692.881
producer-metrics:request-latency-avg	212.702	20.451
producer-metrics:request-rate	8.684	95.841
producer-metrics:response-rate	8.719	95.882

6. Now delete the network rule:

```
[root@base appuser]# tc qdisc del dev eth0 root
```

7. Run the stress test again and write up the same metrics as in the previous 2 attempts and compare them.

8. Now we want to simulate a very unreliable network with an average packet loss of about 25%:

```
[root@base appuser]# tc qdisc add dev eth0 root netem loss 25%
```

9. Check with ping that indeed the loss is approx. 25%:

```
[root@base appuser]# ping -c 10 kafka
PING kafka (172.19.0.2) 56(84) bytes of data.
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=3 ttl=64
time=0.150 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=4 ttl=64
time=0.145 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=5 ttl=64
time=0.116 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=6 ttl=64
time=0.203 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=7 ttl=64
time=0.044 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=9 ttl=64
time=0.111 ms
64 bytes from kafka.ts-prod_confluent (172.19.0.2): icmp_seq=10
ttl=64 time=0.116 ms

--- kafka ping statistics ---
10 packets transmitted, 7 received, 30% packet loss, time 180ms
rtt min/avg/max/mdev = 0.044/0.126/0.203/0.045 ms
```

10. Run the stress test again and write up the same metrics as in the previous attempts and

compare them.

Here are values from my tests:

	Throttle			
Metric Name	**delay 200ms**	**delay 20ms**	**None**	**loss 25%**
producer-metrics:outgoing-byte-rate	85264.806	95692.881	107199.249	41726.879
producer-metrics:request-latency-avg	212.702	20.451	0.245	6851.500
producer-metrics:request-rate	8.684	95.841	189.369	0.387
producer-metrics:response-rate	8.719	95.882	189.373	0.416

11. Before proceeding delete the network rule:

```
[root@base appuser]# tc qdisc del dev eth0 root
```

Exceeding max message size

1. Create a topic named `test-topic`:

```
[root@base appuser]# kafka-topics --create \
    --bootstrap-server kafka:9092 \
    --topic test-topic \
    --replication-factor 1 \
    --partitions 1
```

2. Run:

```
[root@base appuser]# kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --producer-property max.request.size=100 \
    --topic test-topic
```

Note that we defined the `max.request.size` to be `100` bytes only. This is unrealistically small, but we use it to demonstrate problems that occur if the message size is exceeded.

3. Enter a small record value such as:

```
test
```

Apparently the resulting serialized message size is smaller than 100 bytes and everything works fine.

4. Now enter a record value that causes the serialized record to be greater than 100 characters, such as:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nec
lacus eu tellus pulvinar aliquam. Nam in sodales enim. Suspendisse
potenti.
```

Observe the error messages such as:

```
[2020-10-17 01:54:34,165] ERROR Error when sending message to topic
test-topic with key: null, value: 140 bytes with error:
(org.apache.kafka.clients.producer.internals.ErrorLoggingCallback)
org.apache.kafka.common.errors.RecordTooLargeException: The message
is 226 bytes when serialized which is larger than 100, which is the
value of the max.request.size configuration.
```

This is the kind of error you would be looking for in the log file(s) generated by your producers, if you expect problems with the message size.

Cleanup

1. Stop the producer with `Ctrl+C` and exit the `base` container by pressing `Ctrl+D`
2. Clean up your environment by running the following command:

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

Summary

In this exercise you have learned how to identify issues with the Kafka producer causes by a suboptimal network connection. We have been using the `kafka-producer-perf-test` tool to simulate a producer.

b. Tuning Producers

In this exercise we're going to tune a sample producer.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/tu-prod
```

2. Run the Confluent Platform:

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

3. Install the `libmnl` package on the broker containers so the `tc` tool is available:

```
$ docker-compose exec -u root kafka-1 /config/install-libmnl.sh && \
docker-compose exec -u root kafka-2 /config/install-libmnl.sh && \
docker-compose exec -u root kafka-3 /config/install-libmnl.sh
```

   Ignore the `NOKEY` Warning message if it appears.

4. To simulate a more realistic scenario where producer and brokers are not on the same physical computer we're going to introduce some artificial latency (5 ms) on the brokers using the `tc` tool:

```
$ docker-compose exec -u root kafka-1 tc qdisc add dev eth0 root
netem delay 5ms && \
  docker-compose exec -u root kafka-2 tc qdisc add dev eth0 root
netem delay 5ms && \
  docker-compose exec -u root kafka-3 tc qdisc add dev eth0 root
netem delay 5ms
```

Troubleshooting High Latency

1. On the host open the file `producer.properties` in the `config` subfolder of the project folder and make sure the following settings are present:

```
batch.size=1
acks=0
linger.ms=200
```

2. Create the topic `test-topic`:

```
$ kafka-topics --bootstrap-server kafka-1:19092,kafka-2:29092 \
    --create \
    --topic test-topic \
    --partitions 10 \
    --replication-factor 3
Created topic "test-topic".
```

3. Run a performance test using the `kafka-producer-perf-test` tool:

```
$ kafka-producer-perf-test \
    --record-size 200 \
    --throughput 10000 \
    --num-records 200000 \
    --topic test-topic \
    --producer.config config/producer.properties \
    --print-metrics
```

This application will run for a moment and once done will output a bunch of metrics. Please be patient.

4. In the output note down the following metrics and their values:
    - outgoing byte rate (in MB/sec)
    - outgoing bytes total (in MB)
    - record send rate
    - records per request avg
    - request latency avg
    - IO wait ratio
    - IO time ns avg
5. In the file `producer.properties` change the value `acks` to `1`.
6. Run the same performance test again as before and note the values and compare them with the ones from the previous run. Notice that there is a slight drop in the **outgoing**

byte rate and a significant increase in the **average latency**.

Here are the values from the first two tests.

Metric Name	acks=0	acks=1
producer-metrics:outgoing-byte-total	58855974.000	58744486.000
producer-metrics:outgoing-byte-rate	1036799.091	1372630.932
producer-metrics:record-send-rate	3534.068	4692.192
producer-metrics:records-per-request-avg	3.194	3.308
producer-metrics:request-latency-avg	NaN	10.219
producer-metrics:io-wait-ratio	0.265	0.594
producer-metrics:io-time-ns-avg	82109.227	91105.659

7. In the file `producer.properties` change the value `acks` to `all`.
8. Run the same performance test again as before and note the values and compare them with the ones from the previous run. Notice that there is a significant drop in the **outgoing byte rate** and a significant increase in the average latency. Also the IO wait ratio increases significantly due to the fact that the producer has to wait for all 3 brokers to ack each send.

If you see a lot of errors from the producer when using `acks=all` then try to figure out what's happening. You might use Confluent Control Center (at http://localhost:9021) to see if the replicas are out of sync, etc. Another option is to reduce the amount of (artificial) network delay that we introduced. This is a good troubleshooting exercise.

9. To reduce the delay on the brokers to 3ms, run the following command:

```
$ docker-compose exec -u root kafka-1 tc qdisc change dev eth0 root
netem delay 3ms && \
  docker-compose exec -u root kafka-2 tc qdisc change dev eth0 root
netem delay 3ms && \
  docker-compose exec -u root kafka-3 tc qdisc change dev eth0 root
netem delay 3ms
```

10. Now increase the `batch.size` in the file `producer.properties` to `16384` (16kB) and repeat the above runs again for all 3 possible settings of `acks`. Note down the numbers and compare to those where we used no batching.

Working with Compression

1. Set the value `compression.type` in the file `producer.properties` to `lz4` and repeat all the above measurements with and without batching and with all possible values of `acks`.
2. Compare the values to those without compression. Specifically observe the outgoing bytes total (that is the necessary network bandwidth between producer and brokers)!

Optional: Max In-Flight Requests

1. Set the value `max.in.flight.requests.per.connection` in the file `producer.properties` to `1` (default is `5`) and repeat the measurements.
2. Compare the values with those gained in the previous runs.

Cleanup

1. Exit the `base` container with `Ctrl+D`
2. Clean up your environment by running the following command:

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

Summary

In this lab you have been tuning various parameters of a producer and measured the effect of doing so, using the tool `kafka-producer-perf-test`.

c. Selecting the best partition strategy

To achieve the highest throughput with the lowest possible latency on a given streaming platform, it is essential that data is evenly distributed among partitions of a topic. This allows to spread the workload evenly across consumers of the topic in a consumer group and thus achieve maximum parallelism.

Use Case

As a consultant you are confronted with the following scenario:

- IoT app (weather data) produces records with high frequency into Kafka
- Records have `Region ID` as key; this cannot be changed
- This leads to uneven distribution of orders in partitions
- Minimum and maximum numbers of records / partition / day differ by factor of up to 10
- Mission critical downstream real-time analytics has high latency due to this fact
- Downstream processing includes plenty of stateful operations such as aggregation
- A data record has (among others) the following fields: probe ID, station ID, region ID, temperature, wind speed, humidity, etc.

Task

Draft a custom partitioner (pseudo code) that evenly distributes the data across all partitions. Justify you choice. Discuss potential consequences of your choice.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 11 Troubleshooting & Tuning Consumers

a. Troubleshooting Consumers

In this lab we're going to reset the consumer group offset of an existing consumer group. This is e.g. useful if the consumer has been updated after a bug has been fixed in its source code and now we want to reprocess data from a given point in time.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-consumers
```

2. Run the Confluent Platform:

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

3. Create the topic pageviews:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic pageviews \
    --partitions 6 \
    --replication-factor 1
Created topic "pageviews".
```

We will use the ksql-datagen tool to create data for us. This tool has a number of predefined data types - you see us requesting one of them here with the parameter `quickstart=pageviews` in the command below.

For more info on this tool, see: https://docs.ksqldb.io/en/latest/developer-guide/test-and-debug/generate-custom-test-data/.

1. Open a new terminal and run the ksql-datagen command:

```
$ ksql-datagen quickstart=pageviews format=delimited \
    topic=pageviews bootstrap-server=kafka:9092
```

2. Minimize the ksql-datagen terminal window and return to previous terminal window.

Working with the Consumer Offset

1. Run the `kafka-console-consumer` as consumer in a consumer group named `sample-consumer`:

```
$ kafka-console-consumer \
    --group sample-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic pageviews
...
1603066765444,User_9,Page_31
1603066765444,User_8,Page_47
1603066765444,User_1,Page_75
1603066765444,User_3,Page_58
...
```

Let the consumer run for a few seconds and then stop it by pressing `Ctrl+C`.

2. Describe the consumer offsets:

```
$ kafka-consumer-groups --describe \
    --group sample-consumer \
    --bootstrap-server kafka:9092

Consumer group 'sample-consumer' has no active members.

GROUP           TOPIC       PARTITION  CURRENT-OFFSET  LOG-END-OFFSET
LAG      CONSUMER-ID   HOST    CLIENT-ID
sample-consumer pageviews   3          71053           294730
223677   -             -       -
sample-consumer pageviews   4          55265           295108
239843   -             -       -
sample-consumer pageviews   1          47412           295299
247887   -             -       -
sample-consumer pageviews   2          47391           295213
247822   -             -       -
sample-consumer pageviews   5          47546           294673
247127   -             -       -
sample-consumer pageviews   0          47333           294405
247072   -             -       -
```

Note that at the time of the above snapshot I already had a significant consumer lag on each partition. If you run the same command again, the `LOG-END-OFFSET` and `LAG` will have increased since the datagen connector is continuing to produce data into the topic and the consumer is no longer consuming records.

3. Now let's run the consumer again:

```
$ kafka-console-consumer \
    --group sample-consumer \
    --bootstrap-server kafka:9092 \
    --topic pageviews
```

Let the consumer run for a few seconds and then stop it by pressing `Ctrl+C`.

4. Read the consumer offset using the `kafka-console-consumer`:

```
$ kafka-console-consumer \
    --from-beginning \
    --topic __consumer_offsets \
    --bootstrap-server kafka:9092 \
    --formatter \

"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatte
r" \
    | grep sample-consumer
```

In the output you should find entries similar to these:

```
...
[sample-consumer,pageviews,0]::OffsetAndMetadata(offset=72782,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067633808,
expireTimestamp=None)
[sample-consumer,pageviews,4]::OffsetAndMetadata(offset=102673,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067633808,
expireTimestamp=None)
[sample-consumer,pageviews,5]::OffsetAndMetadata(offset=71354,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067633808,
expireTimestamp=None)
[sample-consumer,pageviews,1]::OffsetAndMetadata(offset=95028,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067633808,
expireTimestamp=None)
[sample-consumer,pageviews,2]::OffsetAndMetadata(offset=95005,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067633808,
expireTimestamp=None)
[sample-consumer,pageviews,3]::OffsetAndMetadata(offset=118658,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067633808,
expireTimestamp=None)
[sample-consumer,pageviews,0]::OffsetAndMetadata(offset=94949,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067636106,
expireTimestamp=None)
[sample-consumer,pageviews,4]::OffsetAndMetadata(offset=126478,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067636106,
expireTimestamp=None)
[sample-consumer,pageviews,5]::OffsetAndMetadata(offset=95162,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067636106,
expireTimestamp=None)
[sample-consumer,pageviews,1]::OffsetAndMetadata(offset=118836,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067636106,
expireTimestamp=None)
[sample-consumer,pageviews,2]::OffsetAndMetadata(offset=110116,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067636106,
expireTimestamp=None)
[sample-consumer,pageviews,3]::OffsetAndMetadata(offset=142459,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1603067636106,
expireTimestamp=None)
```

Note how we have two entries for each partition of the topic `pageviews`. This is due to the fact that I ran the consumer twice. Each time the consumer committed its offsets.

5. Stop the consumer by pressing `Ctrl+C`.

The `__consumer_offsets` topic partition to which a consumer group writes its offset records is determined by the following formula:

`Partition = hash(consumer group name) % Number of _consumer_offsets` `partions`

Given the default `50` partitions and consumer group name `sample-consumer`, the offset records are written to partition `35`.

6. Run the following command to display the offsets directly from the logs (for the topic `__consumer_offsets`) of the broker:

```
$ docker-compose exec -u root kafka kafka-run-class \
    kafka.tools.DumpLogSegments \
    --files /var/lib/kafka/data/__consumer_offsets-
35/00000000000000000000.log \
    --offsets-decoder
```

```
Dumping /var/lib/kafka/data/__consumer_offsets-
35/00000000000000000000.log
Starting offset: 0
...
...
| offset: 21 CreateTime: 1603067636106 keysize: 34 valuesize: 24
sequence: 0 headerKeys: [] key: offset_commit::group=sample-
consumer,partition=pageviews-0 payload: offset=94949
| offset: 22 CreateTime: 1603067636106 keysize: 34 valuesize: 24
sequence: 1 headerKeys: [] key: offset_commit::group=sample-
consumer,partition=pageviews-4 payload: offset=126478
| offset: 23 CreateTime: 1603067636106 keysize: 34 valuesize: 24
sequence: 2 headerKeys: [] key: offset_commit::group=sample-
consumer,partition=pageviews-5 payload: offset=95162
| offset: 24 CreateTime: 1603067636106 keysize: 34 valuesize: 24
sequence: 3 headerKeys: [] key: offset_commit::group=sample-
consumer,partition=pageviews-1 payload: offset=118836
| offset: 25 CreateTime: 1603067636106 keysize: 34 valuesize: 24
sequence: 4 headerKeys: [] key: offset_commit::group=sample-
consumer,partition=pageviews-2 payload: offset=110116
| offset: 26 CreateTime: 1603067636106 keysize: 34 valuesize: 24
sequence: 5 headerKeys: [] key: offset_commit::group=sample-
consumer,partition=pageviews-3 payload: offset=142459
baseOffset: 27 lastOffset: 27 count: 1 baseSequence: -1 lastSequence:
-1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 2721 CreateTime:
1603067636118 size: 119 magic: 2 compresscodec: NONE crc: 6331625
isvalid: true
| offset: 27 CreateTime: 1603067636118 keysize: 19 valuesize: 32
sequence: -1 headerKeys: [] key: group_metadata::group=sample-
consumer payload:
{"protocolType":"consumer","protocol":null,"generationId":4,"assignme
nt":"{}"}
```

We can find the current consumer group offsets for `sample-consumer` as payload of the records with offset 21 to 26. The partition number is as suffix of the key `offset_commit::group=sample-consumer,partition=pageviews-<X>`, where `<X>` is the partition number.

7. Let's now reset the offset to beginning, but only as a `--dry-run` first:

```
$ kafka-consumer-groups --reset-offsets \
    --group sample-consumer \
    --bootstrap-server kafka:9092 \
    --topic pageviews \
    --to-earliest \
    --dry-run

GROUP                   TOPIC            PARTITION  NEW-OFFSET
sample-consumer         pageviews        0          0
sample-consumer         pageviews        0          0
sample-consumer         pageviews        4          0
sample-consumer         pageviews        5          0
sample-consumer         pageviews        1          0
sample-consumer         pageviews        2          0
sample-consumer         pageviews        3          0
```

8. If we're satisfied with the result we can run the command for good:

```
$ kafka-consumer-groups --reset-offsets \
    --group sample-consumer \
    --bootstrap-server kafka:9092 \
    --topic pageviews \
    --to-earliest \
    --execute

GROUP                   TOPIC            PARTITION  NEW-OFFSET
sample-consumer         pageviews        0          0
sample-consumer         pageviews        0          0
sample-consumer         pageviews        4          0
sample-consumer         pageviews        5          0
sample-consumer         pageviews        1          0
sample-consumer         pageviews        2          0
sample-consumer         pageviews        3          0
```

9. Let's describe the consumer group to double check the effect:

```
$ kafka-consumer-groups --describe \
    --group sample-consumer \
    --bootstrap-server kafka:9092

GROUP           TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-
OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
sample-consumer pageviews       3          0               11879032
11879032        -               -               -
sample-consumer pageviews       4          0               11879273
11879273        -               -               -
sample-consumer pageviews       1          0               11879342
11879342        -               -               -
sample-consumer pageviews       2          0               11879665
11879665        -               -               -
sample-consumer pageviews       5          0               11878979
11878979        -               -               -
sample-consumer pageviews       0          0               11878687
11878687        -               -               -
```

And indeed, all offsets have been reset to zero!

10. Run the consumer group again:

```
$ kafka-console-consumer \
    --group sample-consumer \
    --bootstrap-server kafka:9092 \
    --topic pageviews
```

11. While the consumer is running open another terminal window, execute the following command and note down the exact time (in UTC):

```
$ date -u +"%Y-%m-%dT%H:%M:%S.%3N"
2020-11-05T00:39:43.862
```

12. Stop the consumer by pressing `Ctrl+C`

13. Describe the consumer group to see the current offsets:

```
$ kafka-consumer-groups --describe \
    --group sample-consumer \
    --bootstrap-server kafka:9092
```

14. Reset the consumer group offset. Replace `<time value>` in the command with the exact time value you noted from the previous step.

```
$ kafka-consumer-groups --reset-offsets \
    --group sample-consumer \
    --bootstrap-server kafka:9092 \
    --topic pageviews \
    --to-datetime "<time value>" \
    --dry-run
GROUP                          TOPIC
PARTITION  NEW-OFFSET
sample-consumer                pageviews                      0
14091275
sample-consumer                pageviews                      4
14091854
sample-consumer                pageviews                      5
14091626
sample-consumer                pageviews                      1
14091933
sample-consumer                pageviews                      2
14092271
sample-consumer                pageviews                      3
14091875
```

The time is in UTC

Cleanup

1. Execute the following command to cleanup the system:

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

Summary

In this lab we have used a consumer group to consume data from a topic. After some time we discovered that the consumer code had a flaw and was producing wrong results. To be able to use a fixed version of the consumer and reproduce the results from a given time in the past we used the consumer group reset tool to reset the offsets to the desired point in time. Subsequently we ran the updated consumer and verified that it indeed consumed from the expected offset in the past.

b. Tuning Consumers

In this exercise we're going to tune a sample consumer.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/tu-cons
```

2. Run the Confluent Platform:

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

No Parallelism

1. Create the topic `test-topic-1` with a single partition:

```
$ kafka-topics --bootstrap-server kafka-1:19092,kafka-2:29092
--create \
    --topic test-topic-1 \
    --replication-factor 3 \
    --partitions 1
Created topic "test-topic-1".
```

2. Run the tool `kafka-producer-perf-test` to generate a lot of data in as high frequency as possible:

```
$ kafka-producer-perf-test \
    --record-size 200 \
    --throughput 10000 \
    --num-records 10000000 \
    --topic test-topic-1 \
    --producer.config config/producer.properties \
    --print-metrics
```

3. Open another terminal window and use the tool `kafka-console-consumer` to represent the consumer we want to tune:

```
$ docker container run --rm -d \
    --name "consumer-client-1" \
    --net tu-cons_confluent \
    -v ~/confluent-cao/solution/tu-cons/config:/config \
    -v /usr/share/java/monitoring-interceptors:/monitoring-
interceptors \
    --cap-add NET_ADMIN \
    -u root \
    -e CLASSPATH=/monitoring-interceptors/monitoring-interceptors-
6.0.0.jar \
    confluentinc/cp-enterprise-kafka:6.0.0-1-ubi8 \
    /bin/bash -c "/config/script-1.sh 1"
```

Note how we're using the script `config/script-1.sh` as a startup command for the container. Please open the script for reference. In this script, on line 4 we're defining a network delay of 50ms to simulate a slow consumer. When running the `kafka-console-consumer` on line 5, we're calling the consumer group `group-with-1-partition` to distinguish it from other coming consumer groups that work on topics with more partitions.

4. Open Confluent Control Center at http://localhost:9021

5. Navigate to the **Consumer lag** view for consumer group `group-with-1-partition` and examine consumer lag and partition assignment

6. Change to the **Consumption** view, set the time range to **Last 30 minutes**, and observe the consumption

> There may be a short lag before data appears in the graph

7. Try to add a second instance to the consumer group called `consumer-client-2`:

```
$ docker container run --rm -d \
    --name "consumer-client-2" \
    --net tu-cons_confluent \
    -v ~/confluent-cao/solution/tu-cons/config:/config \
    -v /usr/share/java/monitoring-interceptors:/monitoring-
interceptors \
    --cap-add NET_ADMIN \
    -u root \
    -e CLASSPATH=/monitoring-interceptors/monitoring-interceptors-
6.0.0.jar \
    confluentinc/cp-enterprise-kafka:6.0.0-1-ubi8 \
    /bin/bash -c "/config/script-1.sh 2"
```

8. Note that when this consumer instance starts, it will request to join the consumer group. A consumer group rebalance will take place and since the topic has only 1 partition, only one of the two comsumer instances will be assigned a partition and the second will be idle. You can verify this by:

   ◦ using the `docker stats` command

   ◦ viewing the consumer lag for this consumer group in Control Center (only one instance should listed)

9. When done, cleanup:

   a. Stop the producer by pressing `Ctrl+C`

   b. Remove containers `consumer-client-1` and `consumer-client-2`:

   ```
   $ docker rm -f consumer-client-1 consumer-client-2
   ```

With Parallelism

1. Create the topic `test-topic-6` with a six partitions

```
$ kafka-topics --bootstrap-server kafka-1:19092,kafka-2:29092 \
    --create \
    --topic test-topic-6 \
    --replication-factor 3 \
    --partitions 6
Created topic "test-topic-6".
```

2. Run the tool `kafka-producer-perf-test` to generate a lot of data in as high frequency as possible into the topic `test-topic-6`

```
$ kafka-producer-perf-test \
    --record-size 200 \
    --throughput 10000 \
    --num-records 10000000 \
    --topic test-topic-6 \
    --producer.config config/producer.properties \
    --print-metrics
```

3. Open a new terminal window and run the following script to generate six consumer instances forming a consumer group called `group-with-6-partitions`:

```
for N in {1..6}; do
  docker container run --rm -d \
    --name "client-${N}" \
    --net tu-cons_confluent \
    -v ~/confluent-cao/solution/tu-cons/config:/config \
    -v /usr/share/java/monitoring-interceptors:/monitoring-
interceptors \
    --cap-add NET_ADMIN \
    -u root \
    -e CLASSPATH=/monitoring-interceptors/monitoring-interceptors-
6.0.0.jar \
    confluentinc/cp-enterprise-kafka:6.0.0-1-ubi8 \
    /bin/bash -c "/config/script-6.sh ${N}"
done
```

4. In Control Center, navigate to the **Consumer lag** view for consumer group `group-with-6-partitions` and examine consumer lag and partition assignment
5. Change to the **Consumption** view and observe the consumption
6. Compare the consumer lag and consumption metrics seen with this case **with** parallelism with the previous case **without** parallelism
7. When done remove consumer containers:

```
$ for N in {1..6}; do docker rm -f client-$N; done
```

8. Also stop the `kafka-producer-perf-test` by pressing `Ctrl+C`.

Tuning Config Settings

1. Create the topic `tuning-topic` with a single partition:

```
$ kafka-topics --bootstrap-server kafka-1:19092,kafka-2:29092 \
    --create \
    --topic tuning-topic \
    --replication-factor 3 \
    --partitions 1
Created topic "tuning-topic".
```

2. Run a consumer which uses the default settings for `fetch.max.wait.ms` and `fetch.min.bytes`:

```
$ docker container run --rm -d \
    --name "client-with-defaults" \
    --net tu-cons_confluent \
    -v ~/confluent-cao/solution/tu-cons/config:/config \
    -v /usr/share/java/monitoring-interceptors:/monitoring-
interceptors \
    --cap-add NET_ADMIN \
    -u root \
    -e CLASSPATH=/monitoring-interceptors/monitoring-interceptors-
6.0.0.jar \
    confluentinc/cp-enterprise-kafka:6.0.0-1-ubi8 \
    /bin/bash -c "/config/use-default-settings.sh"
```

3. In another terminal window run the producer:

```
$ kafka-producer-perf-test \
    --record-size 200 \
    --throughput 1000 \
    --num-records 10000000 \
    --topic tuning-topic \
    --producer.config config/producer.properties \
    --print-metrics
```

4. In Control Center, navigate to the **Consumer lag** view for consumer group `group-with-default-settings`. Observe how the consumer lag develops over time.

5. Change to the **Consumption** view and observe the consumption and specifically the latency. What is the minimum we achieve? Observe for a few minutes until the system is stabilized.

6. Stop the producer with `Ctrl+C`

7. Let the consumer catch up until consumer lag is zero and then stop it with:

```
$ docker rm -f client-with-defaults
```

8. Now run the consumer with modified values of `fetch.max.wait.ms` and `fetch.min.bytes`. Specifically we're setting the value of `fetch.min.bytes` so (unreasonably) high that the criteria `fetch.max.wait.ms` will apply for the consumer:

```
$ docker container run --rm -d \
    --name "client-with-custom-values" \
    --net tu-cons_confluent \
    -v ~/confluent-cao/solution/tu-cons/config:/config \
    -v /usr/share/java/monitoring-interceptors:/monitoring-
interceptors \
    --cap-add NET_ADMIN \
    -u root \
    -e CLASSPATH=/monitoring-interceptors/monitoring-interceptors-
6.0.0.jar \
    confluentinc/cp-enterprise-kafka:6.0.0-1-ubi8 \
    /bin/bash -c "/config/use-custom-settings.sh"
```

9. Run the producer:

```
$ kafka-producer-perf-test \
    --record-size 200 \
    --throughput 1000 \
    --num-records 10000000 \
    --topic tuning-topic \
    --producer.config config/producer.properties \
    --print-metrics
```

10. In Control Center, observe the average and maximum latency of the consumer group `group-with-custom-settings` with the custom values set. What do you observe? Discuss with your peers.

11. Also observe the consumer lag of the consumer group `group-with-custom-settings` and how it develops over time.

Cleanup

1. Exit the producer with `Ctrl+C`.

2. Exit the consumer with:

```
$ docker rm -f client-with-custom-values
```

3. Clean up your environment by running the following command:

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

Summary

You have investigated how the number of partitions is a direct measure for the achievable parallelism that can be achieved when consuming data from Kafka. Furthermore you have discovered the influence of the settings `fetch.max.wait.ms` and `fetch.min.bytes` on the latency experienced by the messages handled by the consumer.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 12 Troubleshooting & Tuning Kafka Streams and ksqlDB Apps

a. Troubleshooting ksqlDB Apps

In this exercise we're going to troubleshoot a ksqlDB application.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-streams
```

2. Run the Confluent Platform:

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

Analyzing Consumer Lag

1. Create the topic `users`:

```
$ kafka-topics \
    --bootstrap-server kafka-1:19092,kafka-2:29092 \
    --create \
    --topic users \
    --partitions 6 \
    --replication-factor 1
Created topic "users".
```

2. Create the topic `pageviews`:

```
$ kafka-topics \
    --bootstrap-server kafka-1:19092,kafka-2:29092 \
    --create \
    --topic pageviews \
    --partitions 6 \
    --replication-factor 1
Created topic "pageviews".
```

3. Open a new terminal and run the ksql-datagen command to produce sample data to the `user` topic:

```
$ ksql-datagen quickstart=users format=json topic=users \
    bootstrap-server=kafka-1:19092 \
    propertiesFile=~/confluent-cao/solution/ts-
streams/config/datagen.properties
```

4. Open a new terminal and run the ksql-datagen command to produce sample data to the `pageviews` topic:

```
$ ksql-datagen quickstart=pageviews format=json \
    topic=pageviews bootstrap-server=kafka-1:19092 \
    propertiesFile=~/confluent-cao/solution/ts-
streams/config/datagen.properties
```

5. Minimize the two ksql-datagen terminal windows and return to previous terminal window.
6. Open **Control Center** at http://localhost:9021
7. When Control Center is initialized and the **Clusters** view displays the **cao** cluster with 3 brokers running, click the cluster.
8. On the left hand side click the **Topics** view option and verify the two topics `pageviews` and `users` are there and are populated.
9. Run the ksqlDB CLI:

```
$ ksql http://ksqldb-server:8088
```

10. Create a stream from topic `pageviews`:

```
ksql> CREATE STREAM PAGEVIEWS (
    viewtime string,
    userid string,
    pageid string
) WITH(KAFKA_TOPIC='pageviews',VALUE_FORMAT='JSON');
```

11. Create a table from topic `users`:

```
ksql> CREATE TABLE USERS (
    registertime string,
    userid string PRIMARY KEY,
    regionid string,
    gender string
) WITH(KAFKA_TOPIC='users',VALUE_FORMAT='JSON');
```

12. Create a new stream `PAGEVIEWS_ENRICHED` as follows:

```
ksql> CREATE STREAM PAGEVIEWS_ENRICHED AS
    SELECT p.userid AS userid, viewtime, pageid,
        regionid, gender
    FROM PAGEVIEWS p
    LEFT JOIN USERS u ON p.userid=u.userid
    EMIT CHANGES;
```

13. In the Control Center, navigate to **Consumers** and select **_confluent-ksql-default_query_CSAS_PAGEVIEWS_ENRICHED_<X>** which corresponds to the stored query for stream `PAGEVIEWS_ENRICHED`. Note, `<X>` is a number and depends on your setup.

14. In the Control Center **Consumer lag** view, observe the lag of the consumer group **_confluent-ksql-default_query_CSAS_PAGEVIEWS_ENRICHED_<X>**. As you can see, we have the lag for the two input streams the query is using: `users` and `pageviews`. Is the lag growing or is the query up to date?

15. Now define a heavier, stateful operation. We are going to aggregate (count) the page views by region, and will define a time window of 1 minute:

```
ksql> CREATE TABLE PAGEVIEWS_PER_REGION AS
    SELECT regionid, COUNT( * ) count
    FROM PAGEVIEWS_ENRICHED
    WINDOW TUMBLING (size 60 seconds)
    GROUP BY regionid;
```

16. In the Control Center, navigate to the **Consumer lag** view for consumer group **_confluent-ksql-default_query_CTAS_PAGEVIEWS_PER_REGION_<X>** which corresponds to the stored query for table `PAGEVIEWS_PER_REGION` and observe the lag.
17. Finally we will consume the `PAGEVIEW_PER_REGION` topic that was created by the previous query. We use our console consumer for that.

    Open a new terminal window and run the consumer:

```
$ kafka-console-consumer \
    --group region-consumer-group \
    --bootstrap-server kafka-1:19092 \
    --from-beginning \
    --topic PAGEVIEWS_PER_REGION \
    --consumer.config ~/confluent-cao/solution/ts-
streams/config/console-consumer.properties
```

18. In the Control Center, navigate to the **Consumer lag** view for consumer group `region-consumer-group` and observe the lag.
19. Navigate to the **Consumption** view and observe the **% messages consumed** and **End-to-end latency**.

Troubleshooting Deserialization Errors

In this section of the lab we're using a somewhat contrived example just to show you how you can discover deserialization errors occurring on the ksqlDB server. Such errors happen when the source and the target serialization format do not match, or if the schema of the message input from the Kafka topic has an error or contains unsupported elements.

1. In the terminal window running the ksqlDB CLI, create a stream from an internal topic called `_confluent-metrics`:

```
ksql> CREATE STREAM METRICS (col1 int, col2 int, col3 varchar)
  WITH (KAFKA_TOPIC='_confluent-metrics', VALUE_FORMAT='JSON');
```

2. Let's make sure that we're getting data from the beginning:

```
ksql> SET 'auto.offset.reset'='earliest';
```

3. Now query the stream:

```
ksql> SELECT * FROM METRICS EMIT CHANGES;
```

OK, no results are coming back...

4. Press `Ctrl+C` to terminate the query.
5. Investigate what topic we are querying:

```
ksql> DESCRIBE EXTENDED METRICS;
```

Which should give something like this (shortened for readability):

```
Name                    : METRICS
Type                    : STREAM
Timestamp field         : Not set - using <ROWTIME>
Key format              : KAFKA
Value format            : JSON
Kafka topic             : _confluent-metrics (partitions: 12,
replication: 3)
Statement               : CREATE STREAM METRICS (col1 int, col2 int,
col3 varchar)
  WITH (KAFKA_TOPIC='_confluent-metrics', VALUE_FORMAT='JSON');
...
```

This looks as expected.

6. Let's use `kafkacat` to see if there is any data in the topic:

   a. Open another terminal window and navigate to the project folder:

   ```
   $ cd ~/confluent-cao/solution/ts-streams
   ```

   b. Run `kafkacat`:

   ```
   $ kafkacat -b kafka-1:19092 \
       -C -t _confluent-metrics \
       -o beginning -c 1
   ```

   You should see an output similar to this (shortened):

```
    kafka.log⊠ogStartOffset⊠og"Npartition.2.topic._confluent-
controlcenter-6-0-0-1-AlertHistoryStore-
changelog*ukafka.log:type=Log,name=LogStartOffset,topic=_confluent
-controlcenter-6-0-0-1-AlertHistoryStore
-changelog,partition=2⊠⊠   �⊠
�⊠
    kafka.log⊠ogStartOffset⊠og"Npartition.3.topic._confluent-
controlcenter-6-0-0-1-AlertHistoryStore-
changelog*ukafka.log:type=Log,name=LogStartOffset,topic=_confluent
-controlcenter-6-0-0-1-AlertHistoryStore
-changelog,partition=3⊠⊠   �⊠
�⊠

kafka.server⊠otalFetchRequestsPerSec⊠rokerTopicMetrics"Btopic.
_confluent-controlcenter-6-0-0-1-AlertHistoryStore-changelog*
�⊠afka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerS
ec,topic=_confluent-controlcenter-6-0-0-1-AlertHistoryStore-
changelog⊠  ��b1� @B⊠fdI60fQSkauvoKwvaq9ZwHgj 6.0.0-
ccsr⊠7b744c31e00868b
```

Apparently there is data in this topic, but it is certainly not formatted as JSON. Yet we have defined the value format of our stream to be JSON...

7. Let's look at the log generated by ksqlDB Server (at `usr/logs/ksqldb-streams.log`) and watch out for deserialization errors:

```
$ docker-compose exec -u root ksqldb-server tail -f /usr/logs/ksql-
streams.log \
  | grep -a "deserialization error"
```

We should see errors similar to the following. It may take few minutes for errors to appear based upon how much data is being produced to the `_confluent-metrics` topic.

```
[2020-10-20 20:39:04,973] WARN stream-thread [_confluent-ksql-
default_transient_5129243792365034831_1603225855967-57551a9a-dffc-
4114-b021-d79cffda56c0-StreamThread-2] task [0_9] Skipping record due
to deserialization error. topic=[_confluent-metrics] partition=[9]
offset=[1]
(org.apache.kafka.streams.processor.internals.RecordDeserializer:86)
```

Cleanup

1. Exit the `tail` command by pressing `Ctrl+C`.
2. Exit the console consumer by pressing `Ctrl+C`.
3. Exit the two `ksql-datagen` producers by pressing `Ctrl+C`.
4. Exit the ksqlDB CLI by pressing `Ctrl+D`.
5. Execute the following command to cleanup the system:

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

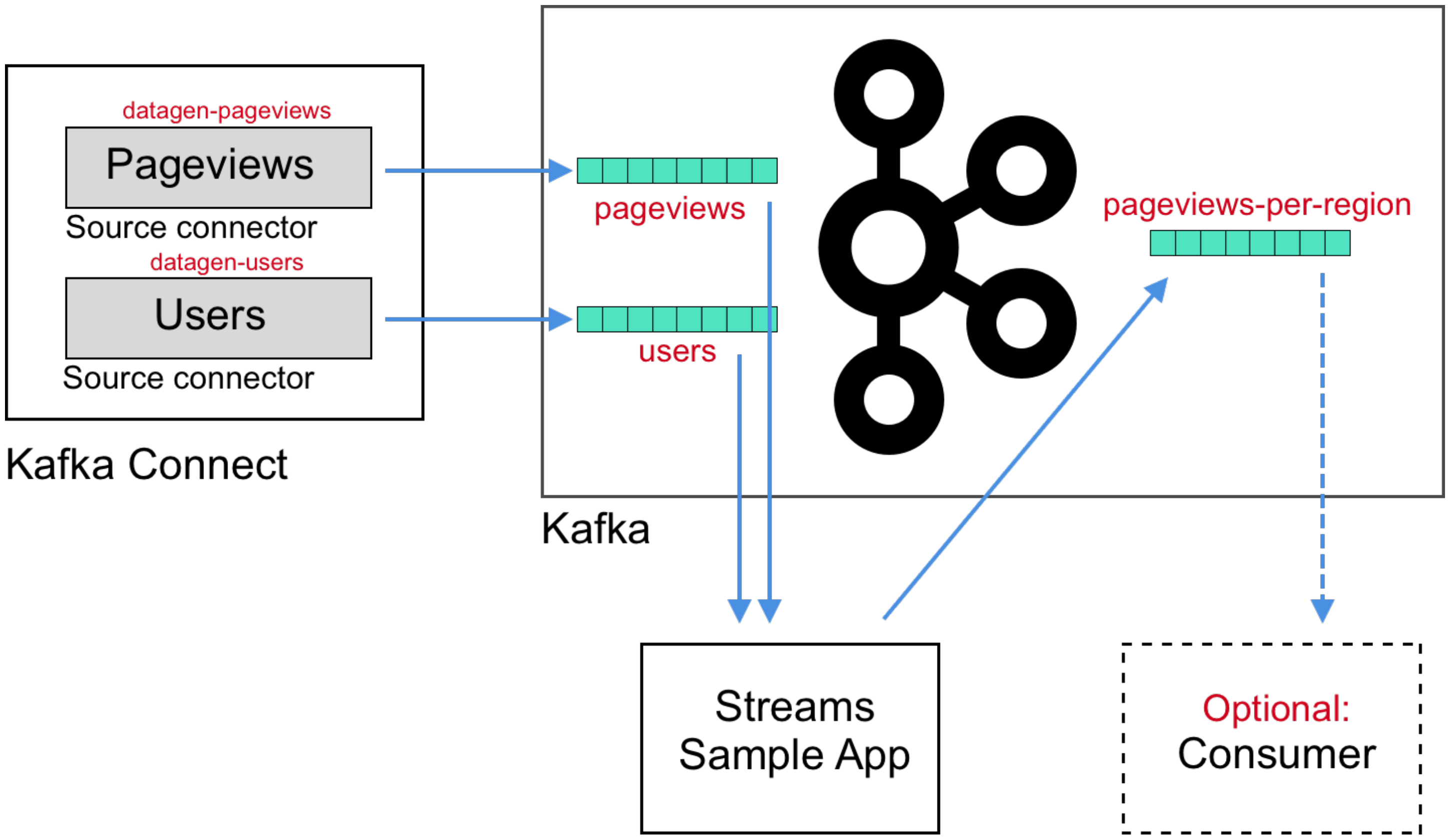
Summary

We have generated loads of data by using the `ksql-datagen` tool. This data served as input data for the ksqlDB queries we defined. We defined a simple join query and also an aggregate query whose throughput and consumer lag we then observed in Confluent Control Center. We have also defined a query in ksqlDB in a way that some deserialization errors will be caused. Those errors we have then found in the logs of the ksqlDB server using `grep`.

b. Tuning Kafka Streams Apps

In this example we are using standby replicas for a Kafka Streams application to accelerate the failover or rebalancing time when forcefully taking one of the Kafka Stream application instances down.

Here is a schema of the app we're going to use:



Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/tu-streams
```

2. Build the container for the Kafka Streams sample app:

```
$ docker-compose build streams-app-1
```

   Please be patient since this takes a moment.

   You can find the full code of this application in the subfolder `streams-sample` of the project folder. It is a good sample on how to write a non-trivial Kafka Streams application.

3. Run the Confluent Platform:

```
$ docker-compose up -d zookeeper kafka ksqldb-server schema-registry
control-center
```

Please give the app a couple of minutes to startup and initialize.

4. Create the topic `pageviews`:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic pageviews \
    --partitions 6 \
    --replication-factor 1
Created topic "pageviews".
```

5. Create the topic `users`:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic users \
    --partitions 6 \
    --replication-factor 1
Created topic "users".
```

6. Create the topic `pageviews-per-region`:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic pageviews-per-region \
    --partitions 6 \
    --replication-factor 1
Created topic "pageviews-per-region".
```

7. Open a new terminal and run the ksql-datagen command to produce sample data to the `user` topic:

```
$ ksql-datagen quickstart=users format=json topic=users \
     bootstrap-server=kafka:9092
```

8. Open a new terminal and run the ksql-datagen command to produce sample data to the `pageviews` topic:

```
$ ksql-datagen quickstart=pageviews format=json \
    topic=pageviews bootstrap-server=kafka:9092
```

9. Minimize the two ksql-datagen terminal windows and return to previous terminal window.
10. Double check with Confluent Control Center (http://localhost:9021) that data is generated. For this you navigate to **Topics** and select the respective topic `pageviews` or `users` and click the **Messages** tab.

Both topics `pageviews` and `users` contain data in JSON format.

Working without Standby Instances

The streams application runs with 2 instances called `streams-app-1` and `streams-app-2` (have a look in the `docker-compose.yml` file).

1. Run the streams applications:

```
$ docker-compose up -d streams-app-1 streams-app-2
```

2. Let the streams application run for a while.
3. In Control Center navigate to **Consumers** and click the **StreamsSampleApp** consumer group which represents our Kafka Streams application.

   Observe the consumer lag for the consumer group `StreamsSampleApp`. Notice in the list on the bottom of the view that you have two consumer instances (since we have scaled to 2 app instances).

   The idea is mainly that you see all the consumer groups that have been created and their respective lag. It should give you a feeling how this simple sample already provides plenty of information in the area of consumer performance.

4. Scale down the `streams-app`:

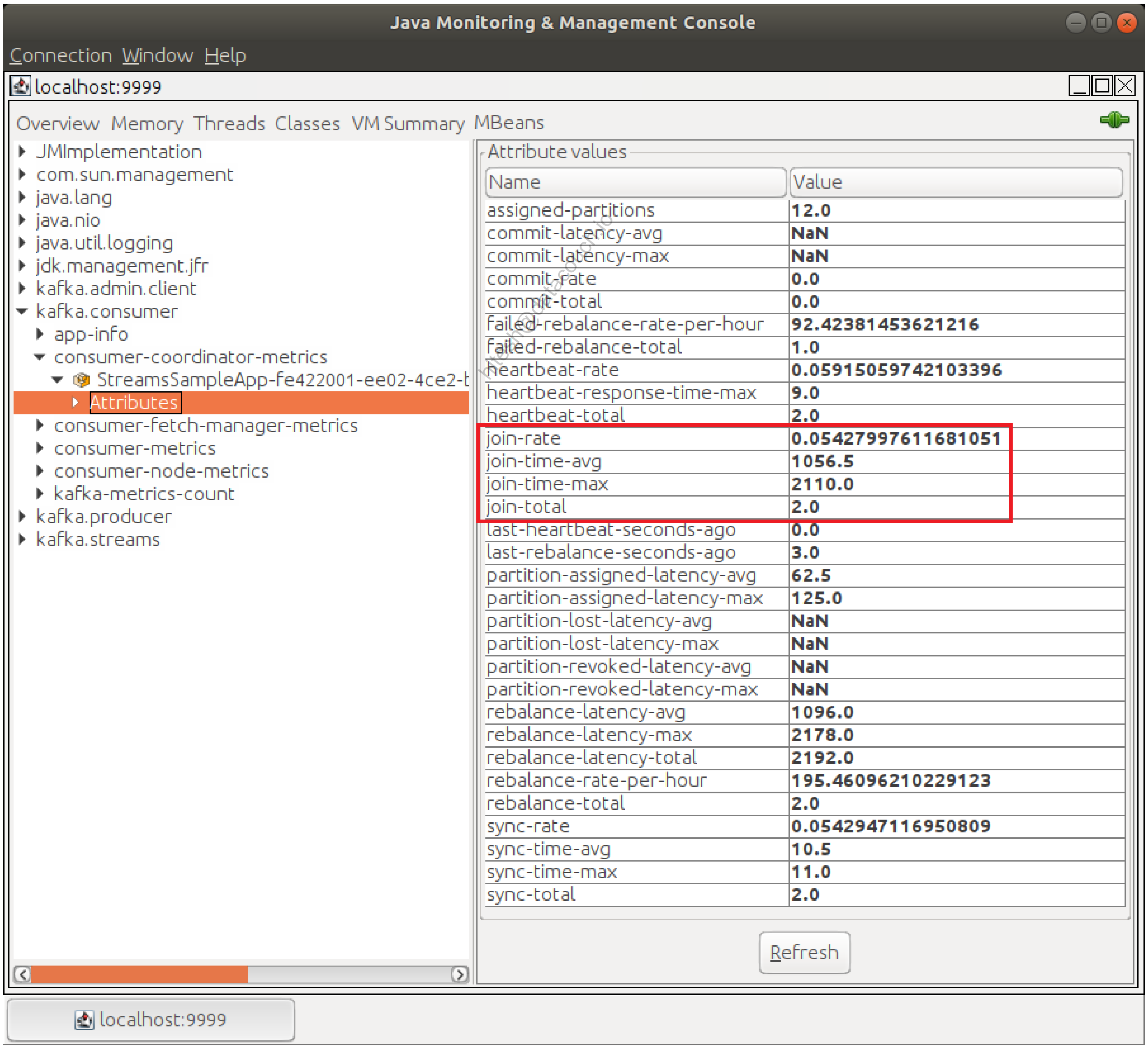
```
$ docker-compose stop streams-app-1 && \
  docker-compose rm streams-app-1
```

Notice that the consumer lag increases with only one streams app instance running.

5. Start the stopped and removed instance again:

```
$ docker-compose up -d streams-app-1
```

6. Use JConsole (use connection URL localhost:9999) to measure the time needed to fail over. Look at in the consumer for the attributes `join-time-avg` and `join-time-max`:



Name	Value
assigned-partitions	12.0
commit-latency-avg	NaN
commit-latency-max	NaN
commit-rate	0.0
commit-total	0.0
failed-rebalance-rate-per-hour	92.42381453621216
failed-rebalance-total	1.0
heartbeat-rate	0.0591505974210339б
heartbeat-response-time-max	9.0
heartbeat-total	2.0
join-rate	0.05427997611681051
join-time-avg	1056.5
join-time-max	2110.0
join-total	2.0
last-heartbeat-seconds-ago	0.0
last-rebalance-seconds-ago	3.0
partition-assigned-latency-avg	62.5
partition-assigned-latency-max	125.0
partition-lost-latency-avg	NaN
partition-lost-latency-max	NaN
partition-revoked-latency-avg	NaN
partition-revoked-latency-max	NaN
rebalance-latency-avg	1096.0
rebalance-latency-max	2178.0
rebalance-latency-total	2192.0
rebalance-rate-per-hour	195.46096210229123
rebalance-total	2.0
sync-rate	0.0542947116950809
sync-time-avg	10.5
sync-time-max	11.0
sync-total	2.0

You will have to view these consumer group join metrics fairly quickly (within a minute or so) before the measurement window passes and the values reset to 0.

Working with Standby Instances

1. Stop the Kafka Streams application:

```
$ docker-compose stop streams-app-1 streams-app-2
```

2. Open the `docker-compose.yml` file and in the sections where the `streams-app-1` and `streams-app-2` services are defined change the value of the environment variable `NUM_STANDBY_REPLICAS: 0` to `NUM_STANDBY_REPLICAS: 1`
3. Start both streams app instances:

```
$ docker-compose up -d streams-app-1 streams-app-2
```

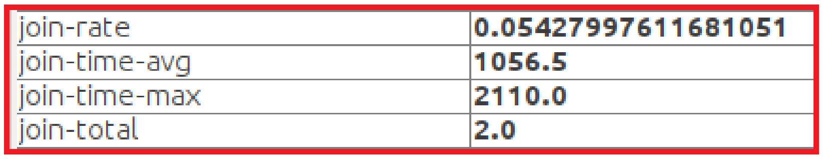
   Let the instances run for a while after which each instance should be the standby replica for the other instance.

4. Use JConsole (use connection localhost:9999) to verify that the attributes `join_time_avg|max` are equal to zero for instance 1 of the streams app.
5. Stop the stream application instance 2:

```
$ docker-compose stop streams-app-2
```

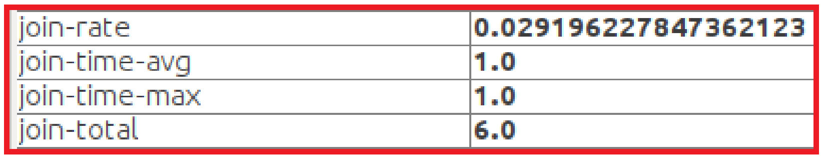
   Stream application instance 1 should take over.

6. Use JConsole to measure the time needed to fail over and compare the value to the previous situation without standby instances (you may have to click **Refresh** in JConsole to get the updated values). Discuss your result with your peers.



join-rate	0.05427997611681051
join-time-avg	1056.5
join-time-max	2110.0
join-total	2.0

NUM_STANDBY_REPLICAS: 0



join-rate	0.029196227847362123
join-time-avg	1.0
join-time-max	1.0
join-total	6.0

NUM_STANDBY_REPLICAS: 1

You will have to view these consumer group join metrics fairly quickly (within a minute or so) before the measurement window passes and the values reset to 0.

7. Stop the stream application instance 1:

```
$ docker-compose stop streams-app-1
```

OPTIONAL

If you want to tinker with the source code and build and test it then these are some commands you will need:

1. Navigate to the kafka streams sample folder:

```
$ cd ~/confluent-cao/solution/tu-streams/streams-sample
```

2. Install gradle and build the source code

```
$ sudo apt install gradle
$ gradle build
```

3. Run the streams app for a test:

```
$ export BOOTSTRAP_SERVERS=kafka:9092 && \
  export NUM_STANDBY_REPLICAS=0 && \
  export JMX_PORT=9999 && \
  export JMX_HOSTNAME=127.0.0.1 && \
  tar -xvf build/distributions/streams-sample.tar && \
  java \
        -Dcom.sun.management.jmxremote \
        -Dcom.sun.management.jmxremote.authenticate=false \
        -Dcom.sun.management.jmxremote.ssl=false \
        -Djava.rmi.server.hostname=${JMX_HOSTNAME} \
        -Dcom.sun.management.jmxremote.rmi.port=${JMX_PORT} \
        -Dcom.sun.management.jmxremote.port=${JMX_PORT} \
        -classpath "streams-sample/lib/*" \
        streams.SampleStreamsApp
```

4. Exit the streams app with `Ctrl+C`

Cleanup

1. Stop the two datagen producers with `Ctrl+C`.
2. Execute the following command to cleanup the system:

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

Summary

In this lab we have investigated the difference in fail-over when running a Kafka Streams application with and without standby instances.

Appendix

If you want to tinker with the source code and build and test it then these are some commands you will need:

1. Navigate to the kafka streams sample folder:

```
$ cd ~/confluent-cao/solution/tu-streams/streams-sample
```

2. Install gradle and build the source code

```
$ sudo apt install gradle
$ gradle build
```

3. Run the streams app for a test:

```
$ export BOOTSTRAP_SERVERS=kafka:9092 && \
  export NUM_STANDBY_REPLICAS=0 && \
  export JMX_PORT=9999 && \
  export JMX_HOSTNAME=127.0.0.1 && \
  tar -xvf build/distributions/streams-sample.tar && \
  java \
        -Dcom.sun.management.jmxremote \
        -Dcom.sun.management.jmxremote.authenticate=false \
        -Dcom.sun.management.jmxremote.ssl=false \
        -Djava.rmi.server.hostname=${JMX_HOSTNAME} \
        -Dcom.sun.management.jmxremote.rmi.port=${JMX_PORT} \
        -Dcom.sun.management.jmxremote.port=${JMX_PORT} \
        -classpath "streams-sample/lib/*" \
        streams.SampleStreamsApp
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 13 Tuning Kafka Connect

a. Tuning Kafka Connect

In this exercise we're going to tune the Confluent JDBC source connector. We are using Postgres as data source. It contains a table `credits_and_grants` which has approximately half a million of entries originating from public data of the World Bank (https://finances.worldbank.org/Loans-and-Credits/IDA-Statement-Of-Credits-and-Grants-Historical-Dat/tdwh-3krx).

You can use a tool such a **pgAdmin 4** to play with the Postgres DB we're using. Just connect to `localhost:5432`, user `postgres` (no password).

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/tu-connect
```

2. Run the Confluent Platform:

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

3. We use the Kafka Connect JDBC connector in this exercise. If you did not complete **Lab 08 Where are my System Log Files?** which includes the steps to install the Kafka Connect JDBC connector on the worker, you will need to complete that section now before continuing with the next step. You can navigate to the required section by clicking this link:

   → Install the Kafka Connect JDBC Connector

Import data from CSV into Postgres

1. Unzip the file `source/IDA_Credits_and_Grants.csv.zip` to

source/IDA_Credits_and_Grants.csv

```
$ cd source && unzip IDA_Credits_and_Grants.csv.zip && cd ..
```

2. Exec into the postgres container where you run psql:

```
$ docker-compose exec -u root postgres psql -U postgres
psql (11.2)
Type "help" for help.

postgres=#
```

3. Create the table credits_and_grants:

```
postgres=# CREATE TABLE credits_and_grants(
    End_of_Period timestamp,
    Credit_Number varchar(10),
    Region varchar(50),
    Country_Code varchar(2),
    Country varchar(50),
    Borrower varchar(50),
    Credit_Status varchar(20),
    Service_Charge_Rate numeric(15,2),
    Currency_of_Commitment  varchar(3),
    Project_ID  varchar(10),
    Project_Name  varchar(50),
    Original_Principal_Amount numeric(15,2),
    Cancelled_Amount numeric(15,2),
    Undisbursed_Amount numeric(15,2),
    Disbursed_Amount numeric(15,2),
    Repaid_to_IDA numeric(15,2),
    Due_to_IDA numeric(15,2),
    Exchange_Adjustment numeric(15,2),
    Borrowers_Obligation numeric(15,2),
    Sold_3rd_Party numeric(15,2),
    Repaid_3rd_Party numeric(15,2),
    Due_3rd_Party numeric(15,2),
    Credits_Held numeric(15,2),
    First_Repayment_Date timestamp,
    Last_Repayment_Date timestamp,
    Agreement_Signing_Date timestamp,
    Board_Approval_Date timestamp,
    Effective_Date timestamp,
    Closed_Date timestamp,
    Last_Disbursement_Date timestamp
);
CREATE TABLE
```

4. Import the data from the CSV file:

```
postgres=# COPY Credits_and_Grants FROM
'/source/IDA_Credits_and_Grants.csv' DELIMITER ',' CSV HEADER;
COPY 501070
```

5. Add a primary key to the table:

```
postgres=# ALTER TABLE credits_and_grants ADD COLUMN id SERIAL
PRIMARY KEY;
ALTER TABLE
```

6. Exit `psql` and the container `postgres` by pressing `Ctrl+D`.

If you prefer a graphical tool to edit your Postgres database then I recommend **PgAdmin4**. You can run this tool as a container as follows:

```
$ docker container run --rm -d \
    -p 8080:80 \
    --net tu-connect_confluent \
    -e PGADMIN_DEFAULT_EMAIL=student@confluent.io \
    -e PGADMIN_DEFAULT_PASSWORD=TopSecret \
    dpage/pgadmin4
```

Then in your browser navigate to localhost:8080 and login with `student@confluent.io` and password `TopSecret`.

Configure the source connector

1. Create a new Topic called `ida_credits_and_grants` with six partitions and one replica:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic ida_credits_and_grants \
    --partitions 6 \
    --replication-factor 1

WARNING: Due to limitations in metric names, topics with a period
('.') or underscore ('_') could collide. To avoid issues it is best
to use either, but not both.
Created topic "ida_credits_and_grants".
```

2. Add a JDBC source connector via command line and **REST API** of Connect:

```
$ curl -s -X POST \
        -H "Content-Type: application/json" \
        --data '{
            "name": "Credits-and-Grants-Connector",
            "config": {
                "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
                "connection.url":
"jdbc:postgresql://postgres:5432/postgres",
                "connection.user": "postgres",
                "table.whitelist": "credits_and_grants",
                "mode":"incrementing",
                "incrementing.column.name": "id",
                "table.types": "TABLE",
                "topic.prefix": "ida_",
                "numeric.mapping": "best_fit",
                "transforms": "createKey,extractInt",
                "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createKey.fields": "id",
                "transforms.extractInt.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractInt.field": "id"
            }
        }' http://connect:8083/connectors | jq
```

should give this output:

```
{
  "name": "Credits-and-Grants-Connector",
  "config": {
    "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "jdbc:postgresql://postgres:5432/postgres",
    "connection.user": "postgres",
    "table.whitelist": "credits_and_grants",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "table.types": "TABLE",
    "topic.prefix": "ida_",
    "name": "JDBC-Source-Connector",
    "transforms": "createKey,extractInt",
    "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields": "id",
    "transforms.extractInt.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
    "transforms.extractInt.field": "id"
  },
  "tasks": [],
  "type": "source"
}
```

3. Let's see what we get:

```
$ kafka-avro-console-consumer \
      --bootstrap-server kafka:9092 \
      --property schema.registry.url=http://schema-registry:8081 \
      --topic ida_credits_and_grants \
      --from-beginning \
      --max-messages 10 \
      --property print.key=true
```

and we should see something like this:

```
2
{"end_of_period":{"long":1380499200000},"credit_number":{"string":"ID
A00020"},"region":{"string":"AFRICA"},"country_code":{"string":"SD"},
"country":{"string":"Sudan"},"borrower":{"string":"Ministry of
Finance and National
Economy"},"credit_status":{"string":"Disbursed"},"service_charge_rate
":{"double":0.75},"currency_of_commitment":{"string":"USD"},"project_
id":{"string":"P002553"},"project_name":{"string":"ROSEIRES
IRRIGATION"},"original_principal_amount":{"double":1.550969687E7},"ca
ncelled_amount":{"double":633197.93},"undisbursed_amount":{"double":0
.0},"disbursed_amount":{"double":1.487649894E7},"repaid_to_ida":{"dou
ble":7502114.12},"due_to_ida":{"double":7374384.82},"exchange_adjustm
ent":{"double":0.0},"borrowers_obligation":{"double":7374384.82},"sol
d_3rd_party":{"double":0.0},"repaid_3rd_party":{"double":0.0},"due_3r
d_party":{"double":0.0},"credits_held":{"double":7374384.82},"first_r
epayment_date":{"long":48384000000},"last_repayment_date":{"long":129
5049600000},"agreement_signing_date":{"long":-
269827200000},"board_approval_date":{"long":-
269913600000},"effective_date":{"long":-
258854400000},"closed_date":{"long":62985600000},"last_disbursement_d
ate":null,"id":2}
3
{"end_of_period":{"long":1380499200000},"credit_number":{"string":"ID
A00030"},"region":{"string":"SOUTH
ASIA"},"country_code":{"string":"IN"},"country":{"string":"India"},"b
orrower":{"string":"CONTROLLER OF AID ACCOUNTS &
AUDIT"},"credit_status":{"string":"Repaid"},"service_charge_rate":{"d
ouble":0.75},"currency_of_commitment":{"string":"USD"},"project_id":{
"string":"P009610"},"project_name":{"string":"HIGHWAYS"},"original_pr
incipal_amount":{"double":7.211271347E7},"cancelled_amount":{"double"
:530000.0},"undisbursed_amount":{"double":0.0},"disbursed_amount":{"d
ouble":7.158271347E7},"repaid_to_ida":{"double":7.158271347E7},"due_t
o_ida":{"double":0.0},"exchange_adjustment":{"double":0.0},"borrowers
_obligation":{"double":0.0},"sold_3rd_party":{"double":0.0},"repaid_3
rd_party":{"double":0.0},"due_3rd_party":{"double":0.0},"credits_held
":{"double":0.0},"first_repayment_date":{"long":51062400000},"last_re
payment_date":{"long":1297728000000},"agreement_signing_date":{"long"
:-269222400000},"board_approval_date":{"long":-
269308800000},"effective_date":{"long":-
263606400000},"closed_date":{"long":-
79142400000},"last_disbursement_date":null,"id":3}
...
```

4. Use the ksqlDB CLI:

```
$ ksql http://ksqldb-server:8088
```

you can safely ignore the warning `OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will likely be removed in a future release.`

5. Declare that all queries shall start from beginning of the topic:

```
ksql> set 'auto.offset.reset'='earliest';
```

6. Create a stream from the topic `ida_credits_and_grants`:

```
ksql> CREATE STREAM credits_and_grants
    WITH(VALUE_FORMAT='AVRO', KAFKA_TOPIC='ida_credits_and_grants');
```

7. List the first 10 items of the stream:

```
ksql> SELECT ID, BORROWER, ORIGINAL_PRINCIPAL_AMOUNT,
             CURRENCY_OF_COMMITMENT, REGION, COUNTRY_CODE
      FROM credits_and_grants
      EMIT CHANGES
      LIMIT 10;
1 | MINISTERIO DE HACIENDA Y CREDITO PUBLICO | 9000000.0 | USD |
LATIN AMERICA AND CARIBBEAN | HN
7 | MINISTRY OF FINANCE | 4115989.22 | USD | EAST ASIA AND PACIFIC |
TW
8 | CONTROLLER OF AID ACCOUNTS & AUDIT | 7222066.3 | USD | SOUTH ASIA
| IN
22 | CONTROLLER OF AID ACCOUNTS & AUDIT | 1.808611732E7 | USD | SOUTH
ASIA | IN
25 | CONTROLLER OF AID ACCOUNTS & AUDIT | 2.110047023E7 | USD | SOUTH
ASIA | IN
28 | CONTROLLER OF AID ACCOUNTS & AUDIT | 2.108549249E7 | USD | SOUTH
ASIA | IN
30 | Ministere du Plan Et du Dev. Regional | 6000486.61 | USD |
MIDDLE EAST AND NORTH AFRICA | TN
33 | MINISTERE DE L'ECONOMIE ET DES FINANCES | 421997.01 | USD |
LATIN AMERICA AND CARIBBEAN | HT
44 | MINISTRY OF PLANNING & INTNL COOPERATION | 4019059.72 | USD |
MIDDLE EAST AND NORTH AFRICA | JO
59 | MINISTRY OF FINANCE AND ECONOMIC AFFAIRS | 3.002056359E7 | USD |
SOUTH ASIA | PK
Limit Reached
Query terminated
```

8. Exit the ksqlDB CLI by pressing `Ctrl+D`.

Filtering Data with Custom Query

1. Create a new Topic called `ida_credits_and_grants_query` with six partitions and one replica. This topic will contain the reduced dataset with only a subset of all fields and regions:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic query_credits_and_grants \
    --partitions 6 \
    --replication-factor 1
```

2. Have a look into the file `source/custom-query.json`, which contains the details on how we define a source connector that uses a custom query to limit the import.

3. Create a source connector using this query based ingestion method:

```
$ curl -s -X POST \
        -H "Content-Type: application/json" \
        --data @source/custom-query.json \
        http://connect:8083/connectors | jq
```

4. Analyze the content of the topic `query_credits_and_grants` using `kafka-avro-console-consumer` and assert that only the desired fields have been imported:

```
$ kafka-avro-console-consumer \
      --bootstrap-server kafka:9092 \
      --property schema.registry.url=http://schema-registry:8081 \
      --topic query_credits_and_grants \
      --from-beginning \
      --max-messages 10 \
      --property print.key=true
```

you should get something like this (shortened for readability):

```
...
4       {"id":4,"borrower":{"string":"MINISTERIO DE OBRAS
PUBLICAS"},"region":{"string":"LATIN AMERICA AND
CARIBBEAN"},"country_code":{"string":"CL"},"original_principal_amount
":{"double":2.287891907E7},"currency_of_commitment":{"string":"USD"}}
8       {"id":8,"borrower":{"string":"CONTROLLER OF AID ACCOUNTS &
AUDIT"},"region":{"string":"SOUTH
ASIA"},"country_code":{"string":"IN"},"original_principal_amount":{"d
ouble":7222066.3},"currency_of_commitment":{"string":"USD"}}
9       {"id":9,"borrower":{"string":"MINISTRY OF
FINANCE"},"region":{"string":"EAST ASIA AND
PACIFIC"},"country_code":{"string":"TW"},"original_principal_amount":
{"double":5215362.64},"currency_of_commitment":{"string":"USD"}}
...
```

and indeed we only get the desired fields in the output.

Filtering Data on the Source

Here we are going to reduce the amount of exported data by limiting it at the source with a view.

1. Create a new Topic called `ida_red_v_credits_and_grants` with six partitions and one replica. This topic will contain the reduced dataset from a postgres view with only a subset of all fields and regions:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic ida_red_v_credits_and_grants \
    --partitions 6 \
    --replication-factor 1
```

2. Exec into the `postgres` container and run `psql` inside it:

```
$ docker-compose exec -u root postgres psql -U postgres
psql (11.2)
Type "help" for help.

postgres=#
```

3. Create a view in Postgres that only uses a subset of fields of the source table and further more filters the data by region:

```
postgres=# CREATE VIEW v_credits_and_grants AS
    SELECT ID, BORROWER, REGION, COUNTRY_CODE,
ORIGINAL_PRINCIPAL_AMOUNT, CURRENCY_OF_COMMITMENT
    FROM credits_and_grants
    WHERE REGION='AFRICA';
```

4. Exit `psql` and the `postgres` container by pressing `Ctrl+D`.

5. Add another JDBC source connector via command line and **REST API** of Connect, which will move data to the topic `ida_red_v_credits_and_grants`:

```
$ curl -s -X POST \
        -H "Content-Type: application/json" \
        --data '{
            "name": "Whitelisted-Connector",
            "config": {
                "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
                "connection.url":
"jdbc:postgresql://postgres:5432/postgres",
                "connection.user": "postgres",
                "table.whitelist": "v_credits_and_grants",
        "table.types": "VIEW",
        "validate.non.null": false,
                "mode":"incrementing",
                "incrementing.column.name": "id",
                "topic.prefix": "ida_red_",
                "numeric.mapping": "best_fit",
                "transforms": "createKey,extractInt",
                "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createKey.fields": "id",
                "transforms.extractInt.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractInt.field": "id"
            }
        }' http://connect:8083/connectors | jq
```

6. Analyze the content of the topic `ida_red_v_credits_and_grants` e.g. using `kafka-avro-console-consumer` and assert that only the desired fields and regions have been imported:

```
$ kafka-avro-console-consumer \
      --bootstrap-server kafka:9092 \
      --property schema.registry.url=http://schema-registry:8081 \
      --topic ida_red_v_credits_and_grants \
      --from-beginning \
      --max-messages 10 \
      --property print.key=true
```

Cleanup

1. Clean up your environment by running the following command:

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

Summary

In this lab we have used Kafka Connect and have configured a JDBC connector, used to import a massive amount of data from a table in a Postgres database. We have then shown a technique that can be used to reduce the amount of imported data.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab Extra Troubleshooting Resource Problems

a. Troubleshooting Resource Problems

In this exercise we're going to extract various crucial information from the logs that the Confluent Platform produces.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-cao/solution/ts-res
```

2. Run the Confluent Platform:

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

Troubleshooting a containerized environment

If you are running your event streaming platform in containers then your first point of investigation should be the container host (Docker in this case).

1. Use `docker stats` to get an overview on what's happening on your Docker host in regards to CPU, memory and I/O:

```
$ docker stats
CONTAINER ID        NAME                          CPU %
MEM USAGE / LIMIT     MEM %                 NET I/O             BLOCK
I/O           PIDS
afd457e664dd        ts-res_ksqldb-cli_1           0.00%
1.094MiB / 9.758GiB   0.01%                 6.93kB / 0B         0B / 0B
1
81bff5fe3de2        ts-res_connect_1              0.88%
1.504GiB / 9.758GiB   15.42%                755kB / 670kB       481MB /
36.9kB        37
89206ad4aa91        ts-res_kafka-3_1              2.61%
456.3MiB / 9.758GiB   4.57%                 15MB / 15.1MB       0B /
1.75MB           75
286c11a7b249        ts-res_kafka-2_1              2.68%
474.1MiB / 9.758GiB   4.74%                 15.6MB / 16.7MB     131kB /
1.74MB        79
75caf22b3a60        ts-res_schema-registry_1   0.45%
212.9MiB / 9.758GiB   2.13%                 261kB / 196kB       0B /
8.19kB           33
f20ad2076bac        ts-res_kafka-1_1              2.98%
457.9MiB / 9.758GiB   4.58%                 15.1MB / 18.9MB     86kB /
1.74MB         75
0dfe5105fab0        ts-res_base_1                 0.00%
1.84MiB / 9.758GiB    0.02%                 6.93kB / 0B         0B / 0B
1
9db593dd7ab6        ts-res_zk-2_1                 0.13%
92.46MiB / 9.758GiB   0.93%                 604kB / 906kB       188kB /
2.34MB        55
604031a29ba0        ts-res_ksqldb-server_1     29.20%
256.8MiB / 9.758GiB   2.57%                 958kB / 199kB       0B /
65.5kB           35
68f0cc47fc38        ts-res_zk-3_1                 0.09%
82.67MiB / 9.758GiB   0.83%                 345kB / 118kB       0B /
2.22MB           47
88de7020a870        ts-res_zk-1_1                 0.10%
91.37MiB / 9.758GiB   0.91%                 702kB / 819kB       4.1kB /
2.24MB        48
061cbba9c000        ts-res_control-center_1    2.41%
474.7MiB / 9.758GiB   4.75%                 14MB / 9.65MB       0B /
15.5MB           131
```

Your output may look a bit different. The cluster needs some time to initialize.
In the above output we can see that the brokers use from about 455MB to 475MB of memory. We also note that ksqlDB currently consumes about 29% of CPU and the highest users of I/O are the kafka brokers and Control Center.
Quit Docker stats by pressing `Ctrl+C`.

2. Use the `ps` tool to analyze the resource consumption of the Java process in the container `kafka-1` (which corresponds to broker 101):

```
$ docker-compose exec -u root kafka-1 ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
appuser        1  9.5  4.8 8211288 498132 ?      Ssl  14:53   1:12 java
-Xmx1G -Xms1G -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35 -XX:+ExplicitG
root         165 31.0  0.0  45844  3664 pts/0    Rs+  15:06   0:00 ps
aux
```

We can see that the broker (represented by the java process) is pretty idle in any regard at the moment. Note that I have run the above command **after** the broker has completely initialized itself. In your case the CPU utilization may be higher if your broker is still initializing.

In a native installation of Kafka you would still use the `ps` tool, but run it directly on the host.

Investigating Out of Memory

Note, the following commands are executed directly on the host. Also note that they only work on a Linux host and not on Mac or Windows if you're using Docker for Desktop.

1. Check the memory usage with the `free` tool:

```
$ free -m
total        used         free        shared  buff/cache   available
Mem:          15718         5561         7689         121          2467
9728
Swap:             0            0            0
```

Here we want to make sure the **free** memory is sufficiently big. As we have learned in the module **sufficiently big** means at least 40% free memory.

2. The next way to check memory usage is to read the /proc/meminfo file:

```
$ cat /proc/meminfo
```

Which gives (shortened):

```
MemTotal:        16095524 kB
MemFree:          7847984 kB
MemAvailable:     9940104 kB
Buffers:           133336 kB
Cached:           2272840 kB
SwapCached:             0 kB
Active:           6840484 kB
Inactive:         1034224 kB
Active(anon):     5469740 kB
Inactive(anon):    122936 kB
Active(file):     1370744 kB
Inactive(file):    911288 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:              0 kB
SwapFree:               0 kB
```

Here again, the two numbers `MemFree` and `MemAvailable` matter most.

3. Similar to the above command is the `vmstat -s` command:

```
$ vmstat -s
16095524 K total memory
 5751140 K used memory
 6875336 K active memory
 1038536 K inactive memory
 7806976 K free memory
  133596 K buffer memory
 2403812 K swap cache
       0 K total swap
       0 K used swap
       0 K free swap
  313681 non-nice user cpu ticks
      12 nice user cpu ticks
   58334 system cpu ticks
  301676 idle cpu ticks
    1711 IO-wait cpu ticks
       0 IRQ cpu ticks
    8172 softirq cpu ticks
      46 stolen cpu ticks
 2205044 pages paged in
 2246528 pages paged out
       0 pages swapped in
       0 pages swapped out
29386844 interrupts
75652584 CPU context switches
1597221548 boot time
   41575 forks
```

Make sure you have enough `free memory`.

4. We can also use `top` or better `htop` to get information about memory.

> You can install `htop` with the command `sudo apt update && sudo apt install -y htop`

5. Open a second terminal and run a `cnfltraining/stress:1.0` container to generate memory pressure:

```
$ docker run --rm -it cnfltraining/stress:1.0 --vm 2 --vm-bytes 512M
--timeout 60s
```

6. Execute the commands discussed in point 1 to 4 again and observe how the numbers have changed.

Analyzing I/O Bottlenecks

In this section we are going to use the `netshoot` container. This container contains all the tools engineers commonly use to troubleshoot a system. The advantage is that one does not have to pollute the host machine and install all those tools first. This of course only makes sense if you troubleshoot a host on which Docker is installed, e.g. in a Kubernetes cluster.

We are going to simulate low and high I/O operations using the `iostat` tool. High values in disk I/O may indicate that the throughput of your Kafka cluster is limited by the disk subsystem.

1. Run the `netshoot` container:

```
$ docker container run -it --rm --privileged
cnfltraining/netshoot:1.0
```

2. Inside the container run a report for all devices every 2 seconds:

```
netshoot$ iostat 2
```

Observe how the numbers of block read/write per second are relatively low. The output might look similar to this:

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          11.50    0.00    1.75    0.12    0.00   86.62

Device:             tps    Blk_read/s    Blk_wrtn/s    Blk_read
Blk_wrtn
loop0              0.00          0.00          0.00           0
0
loop1              0.00          0.00          0.00           0
0
loop2              0.00          0.00          0.00           0
0
loop3              0.00          0.00          0.00           0
0
loop4              0.00          0.00          0.00           0
0
nvme0n1            3.00          0.00        292.00           0
584
nvme0n1p1          3.00          0.00        292.00           0
584
```

3. Now we want to produce some I/O with our stress container and observe how the numbers change. Open another terminal window and run the following command:

   ```
   $ docker container run --rm -it cnfltraining/stress:1.0 --hdd 5
   ```

   It spawns 5 workers spinning on write()/unlink().
   You should see a significant increase in the numbers, e.g. devices `sda`:

   ```
   avg-cpu:  %user   %nice %system %iowait  %steal   %idle
             33.68    0.00    5.20   56.05    0.00    5.07

   Device:             tps    Blk_read/s    Blk_wrtn/s    Blk_read
   Blk_wrtn
   loop0              0.00          0.00          0.00           0
   0
   loop1              0.00          0.00          0.00           0
   0
   loop2              0.00          0.00          0.00           0
   0
   loop3              0.00          0.00          0.00           0
   0
   loop4              0.00          0.00          0.00           0
   0
   nvme0n1          554.50          0.00     262240.00           0
   524480
   nvme0n1p1        554.50          0.00     262240.00           0
   524480
   ```

4. Terminate `iostat` container by pressing `Ctrl+C`, and then `Ctrl+D`.

5. Terminate `stress` container by pressing `Ctrl+C`

Troubleshooting Low Throughput

The following exercise shows how one can measure the network throughput and bandwidth between two hosts that each run a component of the Confluent Streaming Platform. The exercise uses the `netshoot` container by Nicolas Karbar.

1. Create a new Docker network called `perf-test`:

   ```
   $ docker network create perf-test
   ```

2. Create a container `perf-test-server`:

```
$ docker container run --rm -it --name perf-test-server \
    --net perf-test cnfltraining/netshoot:1.0 iperf -s -p 9999
```

The relevant command here is `iperf` that we are running inside the `netshoot` container. The parameter `-s` indicates that this container's role is that of a **server**. `-p 9999` indicates the port that is used for communication.

3. Open a new terminal window and from within it create a second container `perf-test-client`:

```
$ docker container run --rm -it --name perf-test-client \
    --net perf-test cnfltraining/netshoot:1.0 \
    iperf -c perf-test-server -p 9999
```

This container's role is that of a **client** (parameter `-c`) and it tries to access `perf-test-server` on port 9999 and will measure the network throughput. After a moment the container shows this output:

```
------------------------------------------------------------
Client connecting to perf-test-server, TCP port 9999
TCP window size:  680 KByte (default)
------------------------------------------------------------
[  3] local 172.19.0.3 port 44766 connected with 172.19.0.2 port 9999
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  29.4 GBytes  25.3 Gbits/sec
```

Apparently we have a throughput of 29.4 GBytes in 10 sec., which leads to a bandwidth of 25.3 Gbits/sec. between the two containers. If that seems high, it is likely because the two containers are on the same host (VM). A more realistic measurement would e.g. run the server container on the machine hosting say Broker 101 and the second (client) container on the machine hosting Broker 102.

4. Quit the server container by pressing `Ctrl+C`.

Cleanup

1. Clean up your environment by running the following commands:

```
$ docker network rm perf-test
$ cd ~/confluent-cao/solution/ts-res
$ docker-compose down -v
```

Summary

In this exercise you have learned how to use tools such as `ps`, `free`, `vmstat` and others as well as the `netshoot` container to troubleshoot resource problems by which components of the Confluent Streaming platform may be affected.

ps, top, htop	Displays Linux processes
free	Displays the amount of free memory in the system
vmstat	Reports virtual memory statistics
iostat	Report CPU and I/O statistics for devices and partitions
iperf	Measures network bandwidth between a server and a client - this is ideal to check the network between two brokers to make sure the connection between them is working as expected

STOP HERE. THIS IS THE END OF THE EXERCISE.