Lab: Merge many streams into one stream

Problem Statement:

If you have many Kafka topics with events, how do you merge them all into a single topic?

Example use case:

Suppose that you have a set of Kafka topics representing songs of a particular genre being played. You might have a topic for rock songs, another for classical songs, and so forth. In this lab, we'll write a program that merges all of the song play events into a single topic.

Hands-on code example:

Run it

1. Prerequisites
2. Initialize the project
3. Get Confluent Platform
4. Write the program interactively using the CLI
5. Write your statements to a file

Test it

1. Create the test data
2. Invoke the tests

Run it

Prerequisites

This lab installs Confluent Platform using Docker. Before proceeding:

- Connect with lab environment VM using SSH:

  `ssh USERNAME@YOUR_VM_DNS.courseware.io`

  - **Username:** Will be provided by Instructor.
  - **Password:** Will be provided by Instructor.

- Verify that Docker is set up properly by ensuring no errors are output when you run `docker info` and `docker compose version` on the command line.

Initialize the project

To get started, make a new directory anywhere you'd like for this project:

```
mkdir merge-streams && cd merge-streams
```

Then make the following directories to set up its structure:

```
mkdir src test
```

Get Confluent Platform

Next, create the following `docker-compose.yml` file to obtain Confluent Platform:

```
---
version: '2'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.3.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-kafka:7.3.0
    hostname: broker
    container_name: broker
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://broker:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0

  schema-registry:
    image: confluentinc/cp-schema-registry:7.3.0
    hostname: schema-registry
    container_name: schema-registry
    depends_on:
      - broker
    ports:
      - "8081:8081"
    environment:
      SCHEMA_REGISTRY_HOST_NAME: schema-registry
      SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: 'broker:9092'

  ksqldb-server:
    image: confluentinc/ksqldb-server:0.28.2
```

```
    hostname: ksqldb-server
    container_name: ksqldb-server
    depends_on:
      - broker
      - schema-registry
    ports:
      - "8088:8088"
    environment:
      KSQL_CONFIG_DIR: "/etc/ksqldb"
      KSQL_LOG4J_OPTS: "-Dlog4j.configuration=file:/etc/ksqldb/log4j.properties"
      KSQL_BOOTSTRAP_SERVERS: "broker:9092"
      KSQL_HOST_NAME: ksqldb-server
      KSQL_LISTENERS: "http://0.0.0.0:8088"
      KSQL_CACHE_MAX_BYTES_BUFFERING: 0
      KSQL_KSQL_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"

  ksqldb-cli:
    image: confluentinc/ksqldb-cli:0.28.2
    container_name: ksqldb-cli
    depends_on:
      - broker
      - ksqldb-server
    entrypoint: /bin/sh
    environment:
      KSQL_CONFIG_DIR: "/etc/ksqldb"
    tty: true
    volumes:
      - ./src:/opt/app/src
      - ./test:/opt/app/test
```

And launch it by running:

```
docker compose up -d
```

Write the program interactively using the CLI

To begin developing interactively, open up the ksqlDB CLI:

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

First, you'll need to create a series of Kafka topics and streams to represent the different genres of music:

```
CREATE STREAM rock_songs (artist VARCHAR, title VARCHAR)
    WITH (kafka_topic='rock_songs', partitions=1, value_format='avro');

CREATE STREAM classical_songs (artist VARCHAR, title VARCHAR)
    WITH (kafka_topic='classical_songs', partitions=1, value_format='avro');

CREATE STREAM all_songs (artist VARCHAR, title VARCHAR, genre VARCHAR)
    WITH (kafka_topic='all_songs', partitions=1, value_format='avro');
```

Let's produce some events for rock songs:

```
INSERT INTO rock_songs (artist, title) VALUES ('Metallica', 'Fade to Black');
INSERT INTO rock_songs (artist, title) VALUES ('Smashing Pumpkins', 'Today');
INSERT INTO rock_songs (artist, title) VALUES ('Pink Floyd', 'Another Brick in the
Wall');
INSERT INTO rock_songs (artist, title) VALUES ('Van Halen', 'Jump');
INSERT INTO rock_songs (artist, title) VALUES ('Led Zeppelin', 'Kashmir');
```

And do the same classical music:

```
INSERT INTO classical_songs (artist, title) VALUES ('Wolfgang Amadeus Mozart', 'The
Magic Flute');
INSERT INTO classical_songs (artist, title) VALUES ('Johann Pachelbel', 'Canon');
INSERT INTO classical_songs (artist, title) VALUES ('Ludwig van Beethoven', 'Symphony
No. 5');
INSERT INTO classical_songs (artist, title) VALUES ('Edward Elgar', 'Pomp and
Circumstance');
```

Now that the streams are populated with events, let's start to merge the genres back together. The first thing to do is set the following properties to ensure that you're reading from the beginning of the stream in your queries:

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

Time to merge the individual streams into one big one. To do that, we'll use `insert into`. This bit of syntax takes the contents of one stream and pours them into another. We do this with all of the declared genres. You'll notice that we select not only the title and artist, but also a string literal representing the genre. This allows us to track the lineage of which stream each event is derived from. Note that the order of the individual streams is retained in the larger stream, but the individual elements of each stream will likely be woven together depending on timing:

```
INSERT INTO all_songs SELECT artist, title, 'rock' AS genre FROM rock_songs;
INSERT INTO all_songs SELECT artist, title, 'classical' AS genre FROM classical_songs;
```

To verify that our streams are connecting together as we hope they are, we can describe the stream that contains all the songs:

```
DESCRIBE ALL_SONGS EXTENDED;
```

This should yield roughly the following output. Notice that our insert statements appear as writers to this stream:

```
Name                 : ALL_SONGS
Type                 : STREAM
Timestamp field      : Not set - using <ROWTIME>
Key format           : KAFKA
Value format         : AVRO
Kafka topic          : all_songs (partitions: 1, replication: 1)
Statement            : CREATE STREAM ALL_SONGS (ARTIST STRING, TITLE STRING, GENRE
STRING) WITH (KAFKA_TOPIC='all_songs', KEY_FORMAT='KAFKA', PARTITIONS=1,
VALUE_FORMAT='AVRO');

 Field  | Type

 ARTIST | VARCHAR(STRING)
 TITLE  | VARCHAR(STRING)
 GENRE  | VARCHAR(STRING)
```

```
Queries that write from this STREAM
-----------------------------------
INSERTQUERY_5 (RUNNING) : INSERT INTO all_songs SELECT artist, title, 'rock' AS genre
FROM rock_songs;
INSERTQUERY_7 (RUNNING) : INSERT INTO all_songs SELECT artist, title, 'classical' AS
genre FROM classical_songs;

For query topology and execution plan please run: EXPLAIN <QueryId>

Local runtime statistics
------------------------


(Statistics of the local KSQL server interaction with the Kafka topic all_songs)

Consumer Groups summary:

Consumer Group       : _confluent-ksql-default_query_INSERTQUERY_7
<no offsets committed by this group yet>

Consumer Group       : _confluent-ksql-default_query_INSERTQUERY_5
<no offsets committed by this group yet>
```

Let's quickly check the contents of the stream to see that records of all genres are present. Issue the following transient push query. This will block and continue to return results until its limit is reached or you tell it to stop.

```
SELECT artist, title, genre FROM all_songs EMIT CHANGES LIMIT 9;
```

This should yield the following output:

```
+------------------------------+------------------------------+-----------------------
-------+
|ARTIST                        |TITLE                         |GENRE
|
+------------------------------+------------------------------+-----------------------
-------+
|Wolfgang Amadeus Mozart       |The Magic Flute               |classical
|
|Johann Pachelbel              |Canon                         |classical
|
|Ludwig van Beethoven          |Symphony No. 5                |classical
|
|Edward Elgar                  |Pomp and Circumstance         |classical
|
|Metallica                     |Fade to Black                 |rock
|
|Smashing Pumpkins             |Today                         |rock
|
|Pink Floyd                    |Another Brick in the Wall     |rock
|
|Van Halen                     |Jump                          |rock
```

```
|
|Led Zeppelin                  |Kashmir                        |rock
|
Limit Reached
Query terminated
```

Finally, we can check the underlying Kafka topic by printing its contents:

```
PRINT all_songs FROM BEGINNING LIMIT 9;
```

Which should yield:

```
Key format: ¯\_(ツ)_/¯ - no data processed
Value format: AVRO or KAFKA_STRING
rowtime: 2020/05/04 22:36:27.150 Z, key: <null>, value: {"ARTIST": "Metallica",
"TITLE": "Fade to Black", "GENRE": "rock"}, partition: 0
rowtime: 2020/05/04 22:36:27.705 Z, key: <null>, value: {"ARTIST": "Wolfgang Amadeus
Mozart", "TITLE": "The Magic Flute", "GENRE": "classical"}, partition: 0
rowtime: 2020/05/04 22:36:27.789 Z, key: <null>, value: {"ARTIST": "Johann Pachelbel",
"TITLE": "Canon", "GENRE": "classical"}, partition: 0
rowtime: 2020/05/04 22:36:27.912 Z, key: <null>, value: {"ARTIST": "Ludwig van
Beethoven", "TITLE": "Symphony No. 5", "GENRE": "classical"}, partition: 0
rowtime: 2020/05/04 22:36:28.139 Z, key: <null>, value: {"ARTIST": "Edward Elgar",
"TITLE": "Pomp and Circumstance", "GENRE": "classical"}, partition: 0
rowtime: 2020/05/04 22:36:27.263 Z, key: <null>, value: {"ARTIST": "Smashing
Pumpkins", "TITLE": "Today", "GENRE": "rock"}, partition: 0
rowtime: 2020/05/04 22:36:27.370 Z, key: <null>, value: {"ARTIST": "Pink Floyd",
"TITLE": "Another Brick in the Wall", "GENRE": "rock"}, partition: 0
rowtime: 2020/05/04 22:36:27.488 Z, key: <null>, value: {"ARTIST": "Van Halen",
"TITLE": "Jump", "GENRE": "rock"}, partition: 0
rowtime: 2020/05/04 22:36:27.601 Z, key: <null>, value: {"ARTIST": "Led Zeppelin",
"TITLE": "Kashmir", "GENRE": "rock"}, partition: 0
Topic printing ceased
```

Write your statements to a file

Now that you have a series of statements that's doing the right thing, the last step is to put them into a file so that they can be used outside the CLI session. Create a file at `src/statements.sql` with the following content:

```
CREATE STREAM rock_songs (artist VARCHAR, title VARCHAR)
    WITH (kafka_topic='rock_songs', partitions=1, value_format='avro');

CREATE STREAM classical_songs (artist VARCHAR, title VARCHAR)
    WITH (kafka_topic='classical_songs', partitions=1, value_format='avro');

CREATE STREAM all_songs (artist VARCHAR, title VARCHAR, genre VARCHAR)
    WITH (kafka_topic='all_songs', partitions=1, value_format='avro');

INSERT INTO all_songs SELECT artist, title, 'rock' AS genre FROM rock_songs;

INSERT INTO all_songs SELECT artist, title, 'classical' AS genre FROM classical_songs;
```

Test it

Create the test data

Create a file at `test/input.json` with the inputs for testing:

```
{
  "inputs": [
    {
      "topic": "rock_songs",
      "value": {
        "artist": "Metallica",
        "title": "Fade to Black"
      }
    },
    {
      "topic": "rock_songs",
      "value": {
        "artist": "Smashing Pumpkins",
        "title": "Today"
      }
    },
    {
      "topic": "rock_songs",
      "value": {
        "artist": "Pink Floyd",
        "title": "Another Brick in the Wall"
      }
    },
    {
      "topic": "rock_songs",
      "value": {
        "artist": "Van Halen",
        "title": "Jump"
      }
    },
    {
      "topic": "rock_songs",
      "value": {
        "artist": "Led Zeppelin",
        "title": "Kashmir"
      }
    },
    {
      "topic": "classical_songs",
      "value": {
        "artist": "Wolfgang Amadeus Mozart",
        "title": "The Magic Flute"
      }
    },
    {
      "topic": "classical_songs",
```

```
      "value": {
        "artist": "Johann Pachelbel",
        "title": "Canon"
      }
    },
    {
      "topic": "classical_songs",
      "value": {
        "artist": "Ludwig van Beethoven",
        "title": "Symphony No. 5"
      }
    },
    {
      "topic": "classical_songs",
      "value": {
        "artist": "Edward Elgar",
        "title": "Pomp and Circumstance"
      }
    }
  ]
}
```

Similarly, create a file at `test/output.json` with the expected outputs. Note that we're expecting events in the order that we issued the insert statements. The test runner will determine its output order based on the order of the statements.

```
{
  "outputs": [
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Metallica",
        "TITLE": "Fade to Black",
        "GENRE": "rock"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Smashing Pumpkins",
        "TITLE": "Today",
        "GENRE": "rock"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Pink Floyd",
        "TITLE": "Another Brick in the Wall",
        "GENRE": "rock"
      }
    },
```

```
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Van Halen",
        "TITLE": "Jump",
        "GENRE": "rock"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Led Zeppelin",
        "TITLE": "Kashmir",
        "GENRE": "rock"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Wolfgang Amadeus Mozart",
        "TITLE": "The Magic Flute",
        "GENRE": "classical"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Johann Pachelbel",
        "TITLE": "Canon",
        "GENRE": "classical"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Ludwig van Beethoven",
        "TITLE": "Symphony No. 5",
        "GENRE": "classical"
      }
    },
    {
      "topic": "all_songs",
      "value": {
        "ARTIST": "Edward Elgar",
        "TITLE": "Pomp and Circumstance",
        "GENRE": "classical"
      }
    }
  ]
}
```

Invoke the tests

Lastly, invoke the tests using the test runner and the statements file that you created earlier:

```
docker exec ksqldb-cli ksql-test-runner -i /opt/app/test/input.json -s
/opt/app/src/statements.sql -o /opt/app/test/output.json
```

Which should pass:

```
  >>> Test passed!
```

Cleanup Resources

Delete all the resources by running following command in the `docker-compose.yml` file directory from the terminal:

```
docker compose down
```

Note: If you get above error while running above command. Manually stop the containers and run `docker compose down` again. **Do not delete kafkanew container**.