Event Processing Project SoSe 2022

Tung Nguyen

Abstract

This report is a part of the course project, which describes the project setup and summarizes the experiments on events streaming and processing

1 Task 1

In the following, we describe the general setup of the project:

- **Technologies**: The project is divided into 2 parts: the event-processing system and the event-driven application. For the first part, we use *Kafka* as the core framework and *KSQL* for efficient event stream processing. For minimalism, we consider *Docker* as the virtual hosting machine for our *Kafka* and *KSQL* servers. For the second part, we write a *Java* application which runs the scripts for producer and consumers on parallel threads

- **Kafka setup**: Our event-processing system includes 3 main parts: broker servers, *KSQL* server and control center. The overall setup can be found in the configuration file *docker-compose.yml*

  - **Brokers setup**: In order to avoid single-point-of-failure in our event-processing system, we setup 2 broker servers; one runs port 29092 and the other runs on port 29093. Both broker servers export to outside on ports 9092 and 9093 respectively. We also create a topic named *meetup-events*, which has 2 partitions

  - **KSQL setup**: The *KSQL* server runs on the port 8088

  - **Control center setup**: Our control center connects with 2 broker servers on ports 29092, 29093 and connects with *KSQL* server on port 8088

- **Application setup**: Our event-driven application includes 3 main parts: events source, producer and consumer. The detail implementation can be found in *src/main/java/tum/de*

  - **Source**: We use the file *events.json.gz* as source for events streaming

  - **Producer**: A script written in *Java*, which iterates through the source file, reads and sends events to 2 broker servers on port 9092, 9093. The events are published to the topic *meetup-events* every 10 ms

  - **Consumer**: A script written in *Java*, which listens to broker servers on port 9092, 9093 and subscribes the topic *meetup-events* and 3 event streams: *GERMANY_MEETUP_EVENTS_STREAM*, *GERMANY_MUNICH_MEETUP_EVENTS_STREAM* (discussed in task 2) and *MUNICH_MEETUP_EVENTS_STREAM* (discussed in the experiments)

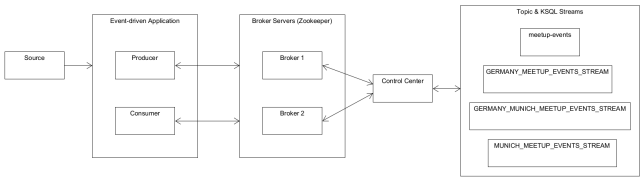

Figure 1. The general project flow

2 Task 2

In order to have the filter processors, we need to setup the following 3 event streams:

- *MEETUP_EVENTS_STREAM*: Event stream for events from *meetup-events* topic. In *KSQL*, the stream can be created as fig. 2

```
CREATE STREAM MEETUP_EVENTS_STREAM (
  key VARCHAR KEY, utc_offset BIGINT,
  venue STRUCT<country VARCHAR, city VARCHAR>
) WITH (KAFKA_TOPIC='meetup-events', VALUE_FORMAT='JSON');
```

Figure 2. Query for stream of meetup events

- *GERMANY_MEETUP_EVENTS_STREAM*: Event stream which selects the events held in Germany. Particularly, the event stream is derived from the *MEETUP_EVENTS_STREAM*. In *KSQL*, the stream can be created as fig. 3

```
CREATE STREAM GERMANY_MEETUP_EVENTS_STREAM
AS SELECT * FROM MEETUP_EVENTS_STREAM WHERE venue->country = 'de'
EMIT CHANGES;
```

Figure 3. Query for stream of meetup events in Germany

- *GERMANY_MUNICH_MEETUP_EVENTS_STREAM*: Event stream which selects the events held in Munich. Particularly, the event stream is derived from the *GERMANY_MEETUP_EVENTS_STREAM*. In *KSQL*, the stream is created as fig. 4

```
CREATE STREAM GERMANY_MUNICH_MEETUP_EVENTS_STREAM
AS SELECT * FROM GERMANY_MEETUP_EVENTS_STREAM
WHERE venue->city LIKE '%Munich%' or venue->city LIKE '%München%'
EMIT CHANGES;
```

Figure 4. Query for stream of meetup events in Germany-Munich

3 Experiments

We do 2 experiments in the project. In the first experiment, we estimate the selectivity of each filter processors. In the second experiments, we add another filter processor, which filters all events held in Munich directly, and measure the average latency on each event. In both experiments, we let the event-driven application run in 3 different time intervals: 10, 15 and 20 minutes

3.1 Experiment 1

The tab. 5 shows the selectivity of each filter processors with respect to different experimental time intervals. Note that the filter 1 is the first filter processor GERMANY_MEETUP_EVENTS_STREAM which filters meetup events held in Germany, whereas the filter 2 is the second filter processor GERMANY_MUNICH_MEETUP_EVENTS_STREAM which filters meetup events held in Munich from GERMANY_MEETUP_EVENTS_STREAM

Interval	Selectivity of Filter 1	Selectivity of Filter 2
10	0.012604	0.089770
15	0.011594	0.107362
20	0.010164	0.098039

Figure 5. Selectivity of the filter processors

From the 1.experiment, we can see that both filter processors have high selectivity, i.e., every 100 meetup events, there might be an event held in Germany and every 10 "German" meetup events, there might be one which is held in Munich.

3.2 Experiment 2

Based on the fact that Munich/München is a city in Germany, we can optimize the system by directly filtering the meetup events in Munich. The fig. 6 shows the stream flow topology

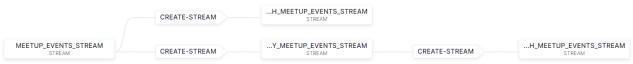

Figure 6. Overview of stream topology

The tab 7 shows the average latency (*AvgLat*) between the non-optimized and optimized filter processors. Note that the non-optimized includes 2 filter processors GERMANY_MEETUP_EVENTS_STREAM and GERMANY_MUNICH_MEETUP_EVENTS_STREAM, whereas the optimized includes single filter processor MUNICH_MEETUP_EVENTS_STREAM

Interval	AvgLat of non-optimized (ms)	AvgLat of optimized (ms)
10	198.069	101.441
15	199.243	104.714
20	204.293	103.907

Figure 7. AvgLat between non-optimized and optimized

From the 2.experiment, we can see that the latency is twice improved after the optimization. Besides, it also takes up less storage to have one filter processor in our event-processing system.