Event Processing - Course Project

Group: auto_offset_earliest

Bäurle, Simon
Technical University Munich
simon.baeurle@tum.de

Neudeck, Nicolas
Technical University Munich
nicolas.neudeck@tum.de

ACM Reference format:
Bäurle, Simon and Neudeck, Nicolas. 2020. Event Processing - Course Project. In *Proceedings of Event Processing: Course Project, Munich, SS2020 (EP),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SYSTEM STRUCTURE

The infrastructure relies on the distributed streaming platform Kafka as Event Processing System. Paired with ksqlDB as event streaming database and producers/consumers written in Kotlin (using the Apache Kafka Java SDK [2]), several Queries were implemented.

1.1 Kafka and ksqlDB

Our setup containing the Kafka `broker`, `zookeper`, `ksqlDB` and the `ksql-cli` were run individually in Docker using the provided docker-compose setup. The setup (based on the images of Confluent, Inc.) exposes the application ports, allowing us to communicate using localhost. We used Conduktor as a graphical interface to monitor our local Kafka cluster.

1.2 Source and Sink

For every line in the `events.json` file, our Kotlin-based producer writes a ByteArray representation of the event to a configured Kafka topic. We've chosen a ByteArray representation simply because it was already provided by the SDK. (additionally this avoids costly (de-)serialization in the producer and optimizes for maximum throughput) Yet, to allow the identification of each event based on Kafka event keys (needed for 3.2) we extracted the `event_id` of each line and used it as event key.

On the sink/consumer site, we implemented a simple Kotlin-based consumer, storing the event key and timestamp in an in-memory map (also needed for 3.2). For more advanced consumers (see 2.2) we use kotlinx.serialization.json to serialize the events and extracted needed fields.

EP, SS2020, Munich

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.3 Streams

For the event streams we chose KSQL over Kotlin because of its simplicity. The individual queries were written and executed using the `ksqldb-cli` container. We defined the following streams:

- `meetup_events_stream`: Define a event schema for further queries/streams.
- `MEETUP_EVENTS_STREAM_DE`: Filter input to all events in Germany.
- `MEETUP_EVENTS_STREAM_DE_MUNICH`: Filter input to all events in Munich.

Here you can see our KSQL definition of the `MEETUP_EVENTS_STREAM_DE`, the stream for Munich is defined equivalently.

```
CREATE STREAM meetup_events_stream_de AS
SELECT * FROM meetup_events_stream
WHERE "GROUP"->country = 'de';
```

2 ADVANCED QUERIES

Since we're a team of two, we implemented some more advanced queries on the events.

2.1 Top-K cities

We decided to implement the suggested top 10 meet up cities in Europe as a query. In general our solution is completly based on KSQL and split into following subqueries/-streams:

- `meetup_events_europe`: Filter events to all countries in Europe (27 in total).
- `events_per_city_europe`: Aggregate the count of events per city in Europe.
- `user-query`: Output the result of the previous stream. (Unfortunately KSQL does not support the `ORDER BY` statement yet and thus it's not possible to output the top 10 cities using only KSQL)

In the code block below, you can see the aggregate stream, counting the events per city in Europe.

```
CREATE TABLE events_per_city_europe AS
    SELECT "GROUP"->city as city, COUNT(*) AS count
    FROM meetup_events_europe GROUP BY "GROUP"->city;
```

2.2 Heat-map visualization

In addition we implemented a heat map visualization of the ḧottestl̈ocations in munich. A new stream was created, reducing the `MEETUP_EVENTS_STREAM_DE_MU` stream to only contain the venue latitude, longitude and event id. To visualize the events from this stream, a custom consumer was implemented. With `"auto.offset.reset"` set to `"earliest"` all events of our stream will be consumed. The consumer polls these events and parses the json (again using `kotlinx.serialization.json`)

and creates a custom code block for each event with valid latitude and longitude. This custom code block is added into a HTML/JavaScript file containing the Google Maps based code to display the heat map. In the end, the results can be viewed from the browser interactively (see figure 1).

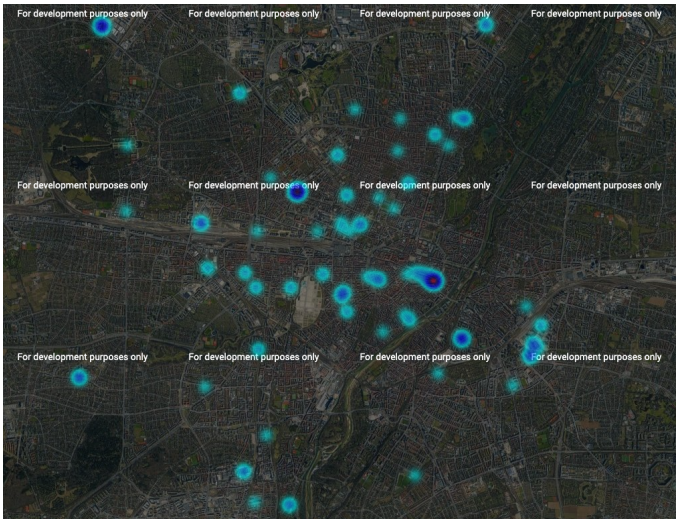

Figure 1: Heatmap

3 PERFORMANCE MEASUREMENTS

Based on the system described in section 1, we integrated measurements into the producer/consumer side.

3.1 Producer Measurements

Using the Java SDK for Kafka [2], it's possible to obtain producer side metrics directly within the code. These metrics include average latency and request size as well as incoming/outgoing byte rate (just a small subset of all available metrics). Sadly, these metrics were very inconsistent (with multiple outcomes for the same producer run), so they weren't suitable as a data source for measurements. Instead we went with a more general approach and measured the total runtime of our producer application and calculated the producer byte rate based on the file size and the measured runtime. We ran our producer multiple times using different machines (MacBook Pro 15" 2018 with Docker for Mac and Ryzen 2600x Tower PC with native Linux Docker). The runtime to process the whole json file (~335MB) was around 18s (MacBook) or 8.3s (Tower). This results in a total throughput of 19.75 MB/s (MacBook) or 42.84 MB/s (Tower).

3.2 Time-Stamp Based Latency

wann timestamps generiert werden! Additionally we chose an approach based on a blog post by Alexis Seigneurin. In his approach, he defined a stream consuming events from his initial topic and outputting them into a new topic [6]. He then defined two consumers (one per topic), consumed both topics and calculated the processing time based on the event timestamp assigned by the broker. We adapted this approach and defined two concurrent consumer (on `event_test` and `MEETUP_EVENT_STREAM_DE`). We consumed events from the broker and calculated the internal Kafka latency for all events that occurred in both streams. Interestingly, even when producing all events from the `events.json` file, the latency to process the incoming event and filter them further down using our streams was always below 1 ms. Whilst this seems unrealistic at first, this coincides with the results of Alexis Seigneurin, using his own setup [6].

4 CONCLUSION

To conclude our paper, we will highlight interesting results and learnings from the project. As seen, the performance gap between the MacBook Pro and the Tower PC is significant, which can't be explained solely by the better CPU of the tower PC. We suspect the difference between Docker for Mac (hosting Docker in a virtual machine) and native Docker support on Linux as the major source for this gap. It was interesting to gain further insights into the internal timestamp generation of Kafka with different timestamps per stream/topic. Especially during the implementation of the TopK Query, we encountered some major limitations of KSQL in comparison to the ANSI SQL standard (no implementation of `ORDER BY` or the `IN` operator for queries). Lastly, the elegant handling of concurrency in Kotlin using Co-routines was a valuable take-away.

REFERENCES

[1] Apache Software Foundation. Apache Kafka Documentation. https://kafka.apache.org/24/documentation.html.
[2] Apache Software Foundation. Javadoc - kafka 1.0.1 API. https://kafka.apache.org/10/javadoc/overview-summary.html.
[3] Confluent, Inc. Documentation — Confluent Platform. https://docs.confluent.io/current/.
[4] Confluent, Inc. ksqlDB: The event streaming database purpose-built for stream processing applications. https://docs.ksqldb.io/en/latest/.
[5] Alexis Seigneurin. Kafka tutorial #6 - Kafka Streams in Kotlin. https://aseigneurin.github.io/2018/08/03/kafka-tutorial-6-kafka-streams-in-kotlin.html.
[6] Alexis Seigneurin. Kafka tutorial #9 - Latency measures. https://aseigneurin.github.io/2018/08/20/kafka-tutorial-9-latency-measures.html.