Event Processing: Streaming with Kafka & KSQL

Alberto Beyersdorff Filho

Abstract

Big Data and the necessity to transfer data through different applications, reinforce the importance of initiatives to process data in real-time. There are already several event stream frameworks, dealing with the data flow to perform stream analysis. In this project, a small event processing system was developed using Apache Kafka and KSQL to describe the usefulness of some tools.

ACM Reference Format:
Alberto Beyersdorff Filho. 2022. Event Processing: Streaming with Kafka & KSQL. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

The high volume of data constantly produced and the necessity to pass these data through different applications, reinforce the importance of initiatives to process data in real-time. Event stream processing (ESP) has the proposal to deal with an information flow to perform stream analysis, which consists of an input and output of data streams [4]. There are already several event stream frameworks like Apache Kafka, Apache Flink, Apache Flume and Apache Spark, specifically created for the process of distributing data, which offer a scalable, fault-tolerant stream system [8]. In this project on the subject Event Processing at the Technical University Munich (TUM), a small message stream process was developed using Apache Kafka and KSQL to comprehend these tools, briefly explaining the features and showing a few outcomes.

1 Apache Kafka and KSQL

Apache Kafka, developed by LinkedIn in 2011 [7], is a high-scalable framework built on a publisher-subscriber messaging system for the process of data streams. A publisher by Kafka is called a *Producer*, that is responsible for writing messages into a Kafka server (*Broker*), which organizes these messages in Topics. A subscriber, in Kafka called *Consumer*, reads these messages from the servers [5]. Zookeeper is responsible for the brokers' coordination (see Figure 1).

Conference'17, July 2017, Washington, DC, USA

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

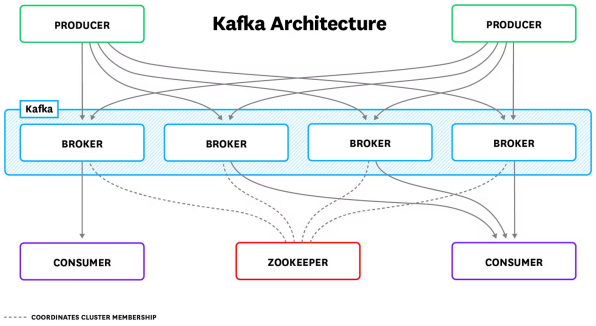

Figure 1. Architecture of the Kafka components [3]

KSQL is a streaming engine for Kafka. With KSQL it is possible to create streams, work with tables and define queries, like a database for streaming applications [2].

2 Guarantees and Faults

To guarantee the message delivery, most stream frameworks work with the exactly-once concept. In this concept, the messages are handled once and don't get lost. Kafka handles multiple messages per second and assures high availability and fault-tolerance by a mechanism that replicates partitions through brokers and maintains copies of these partitions [8]. Shree et al. even asserts that Kafka handles messages without any fault.

3 Latency & Throughput

When talking about performance, there are two important measurements, which play a major role also by event streaming applications. Latency is the amount of time a resource needs between input and output, in other words, the delay. Throughput is the number of resources divided by the unit time, resulting in the rate of resources [6]. In the context of streaming, these two metrics are relevant for the performance control system sending messages between producers and consumers.

4 Development

The project consists of streaming and filtering messages. The messages, in this case, are meetup events from a JSON file. The environment was based on the guidance from [1] and implemented with Python and Docker.

The file producer.py (see Listing 1) reads the entire JSON file, deserializes the events to a Python object, creates a Kafka Producer and finally produces the events to the respective topic, named MEETUP_EVENTS. Because the entire events

need to be transferred in a JSON format, they need to be serialized with the JSON encoder (dumps). The file consumer.py, on the other hand, creates a Kafka Consumer, subscribes to the topic and reads all the events.

```
    topic = "meetup_events"

    for event in events:
        producer.produce(
            topic, key=event['id'],
            value=json.dumps(event, indent=4,
    sort_keys=True),
            callback=delivery_callback
        )
        producer.poll(0.0)
```

Listing 1. Kafka Producer produces events to the topic

Docker was used for Kafka as well as KSQL services. The environment of KSQL worked as a filtering model, using streams (see Listing 2) to filter the output down to Germany and Munich. Furthermore, Prometheus and Grafana also run in a Docker Container for the performance analysis. In short, Prometheus is an event monitoring and alerting tool, which combined with Grafana, a visualisation and analytics platform, provides a meaningful performance system. Monitoring Kafka means measuring the performance of three components: Brokers, Producers and Consumers [3].

```
CREATE STREAM MEETUP_EVENTS_GERMANY AS
    SELECT *
      FROM MEETUP_EVENTS
      WHERE `group`->`country`='de';

CREATE STREAM MEETUP_EVENTS_MUNICH AS
    SELECT *
      FROM MEETUP_EVENTS
      WHERE `group`->`city`='Munich'
        OR `group`->`city`='München';
```

Listing 2. KSQL statement for stream creation

5 Findings & Outcomes

The Grafana Dashboard used to visualise the performance of the Kafka components could show some relevant data about the performance plus memory consumption when streaming the meetup events. The JSON file supplies the stream with a total of 142502 messages and consumes 450mb of memory (see Figure 2). The KSQL streams for Germany and Munich events consume naturally much less messages and memory, because of the small number of respective messages.

The KSQL streams are not persistent. So when the operator reaches the storage limit, then he goes back to zero as soon as no more messages can be read by the consumer. The MEETUP_EVENTS_GERMANY stream reached 3.84mb memory consumption, while MEETUP_EVENTS_MUNICH only 545kb. The event streaming process was started twice and had a mean latency of 524ms, so an event object could

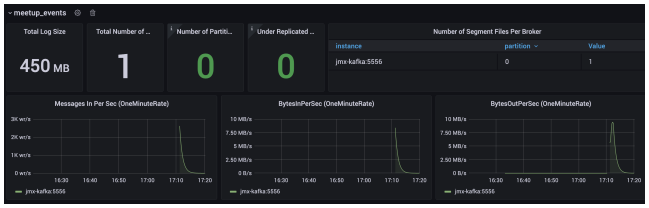

Figure 2. Storage consumption of topic

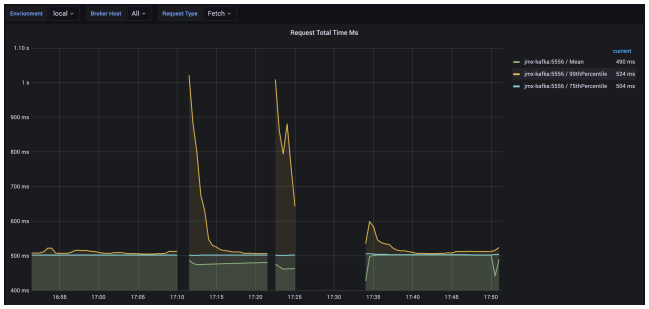

Figure 3. Request total time in ms

be on average consumed under 1s. In addition, it was possible to see that the streaming process behaved differently between the two attempts (see Figure 3). Both took about the same amount of time for the whole stream but at the first, the behaviour was monotonically decreasing, while in the second one, the stream acted linear decreasing with a short movement into the other direction at half-time. More in-depth studies of the event stream would be needed to determine the exact cause of the different behaviours.

It was possible to develop the event processing system without any prior knowledge in the field and it serves as a very instructive project for beginners. Kafka fits excellent with KSQL and has proven to be a powerful framework for data streaming processes.

References

[1] Inc Confluent. 2022. *Getting Started with Apache Kafka and Python.* https://developer.confluent.io/get-started/python/#create-project

[2] Inc Confluent. 2022. *ksqlDB: The database purpose-built for stream processing applications.* https://ksqldb.io/

[3] David M. Lentz Evan Mouzakitis. 2020. *Monitoring Kafka performance metrics.* https://www.datadoghq.com/blog/monitoring-kafka-performance-metrics/

[4] André Leon Sampaio Gradvohl, Hermes Senger, Luciana Arantes, and Pierre Sens. 2014. Comparing distributed online stream processing systems considering fault tolerance issues. *Journal of Emerging Technologies in Web Intelligence* 6, 2 (2014), 174–179.

[5] Matthias J Sax. 2019. Apache Kafka.

[6] T Sharvari and K Sowmya Nag. 2019. A study on modern messaging systems-kafka, rabbitmq and nats streaming. *CoRR abs/1912.03715* (2019).

[7] Rishika Shree, Tanupriya Choudhury, Subhash Chand Gupta, and Praveen Kumar. 2017. KAFKA: The modern platform for data management and analysis in big data domain. In *2017 2nd international conference on telecommunication and networks (TEL-NET)*. IEEE, 1–5.

[8] Giselle van Dongen and Dirk Van Den Poel. 2021. A performance analysis of fault recovery in stream processing frameworks. *IEEE Access* 9 (2021), 93745–93763.