KAFKA STREAMING PROJECT REPORT

Eshref Yozdemir

1. INTRODUCTION

The aim of the project is to be able to stream data from Kafka and count unique users per minute. The main steps we followed in order to come up with my current solution is below:

2. REQUIREMENTS

Initially I went through the requirements to get an idea about possible solution paths I might follow. It was at this stage that I decided on the overall architecture and the data structure to use. In particular for a first version of the project, reading the data from Kafka could be done using a consumer and outputting back with a producer object. Additionally, due to the streaming nature of the data and the additional requirement that it should be possible to ingest historical data, I decided on a dynamic data structure that we can update on-the-go.

3. OVERALL ARCHITECTURE

In this section I describe the architecture of the system in more detail.

3.1. Consumer-Producer Loop

We read the streamed data in record batches using a `KafkaConsumer` object. Subsequently, these records are immediately processed and discarded. After polling, every record is stored as a `String` object, and user id (`uid`) and timestamp (`ts`) fields are extracted using simple string operations.

Following this, we update our data structure using the minute window as the key and the user ID as the value. Specifically, our data structure has the following properties:

1. Key: integer storing the smallest timestamp belonging to a minute, i.e. all 60 different timestamps in a minute are stored using this same value.

2. Value: set of user IDs

Note that this data structure naturally supports ingesting historical data, and also works even if the data is not ordered.

Finally, after all the records are processed, we report back the unique user counts both to standard output and also send it to a Kafka topic using a `KafkaProducer` object. In order not to hinder performance too much, this step is not performed on every consumer poll.

3.2. System Parameters

In the current implementation there are multiple system parameters that can be optimized to a specific load and system architecture. I haven't found time to perform such an optimization yet; therefore, their current values are somewhat arbitrary.

1. **benchmark period**: This is the time between two benchmark measurement times. Decreasing this value might produce finer performance diagnostics, but it might slow down the system as well.

2. **report period**: This is the time between two `KafkaProducer` send operations to a Kafka topic. At the same time it is the time between two prints to STDOUT. Decreasing this value too much would decrease the performance significantly; however, setting it to a too high value might reduce the value of the process in real-time analysis settings.

In both cases we can find a fine balance between the considerations using certain optimization techniques.

3.3. Performance Analysis and Optimization

I implemented the initial version of the program using a JSON parser to read the whole JSON object and get the necassary fields. After implementing the minimum viable version of the project, I profiled it using intellij CPU profiler in order to see if there are any performance bottlenecks. A screenshot of the call frequencies are in `profiler_calltree.png` (I don't include it here since it gets too small). As you can see a significant portion of the program is spent parsing the JSON objects.

A way to circumvent this problem and speed up the program is disabling the JSON parsing and getting the fields we require using find and substring operations. This is the second and final version of my implementation. You can see the call frequencies in `profiler_optimized_parse.png`. In this case we can clearly see that the main bottleneck now becomes polling data from Kafka. Therefore, in order to get further significant speed-ups we will need to work on this part.

Finally, I compared these two versions using the provided dataset and also its shuffled version. The results are as follows:

3.3.1. JSON Parsing

Mean	24108.499654171002
Std	7596.687627510015
Median	22875.739644970414
5th percentile	12570.183678214367
95th percentile	43341.21756487026

Table 1. Frame per second values

3.3.2. JSON Parsing - Shuffled Data

Mean	23587.450734103557
Mean	23587.450734103557
Std	7609.646365545338
Median	21871.047226731367
5th percentile	7518.664752753692
95th percentile	38047.48487389717

Table 2. Frame per second values

3.3.3. Optimized Parsing

Mean	39734.17186735624
Std	10929.316439392576
Median	44030.39215686274
5th percentile	6934.740560087462
95th percentile	47984.44026145342

Table 3. Frame per second values

3.3.4. Optimized Parsing - Shuffled Data

Mean	39935.297644785314
Std	27371.75160791181
Median	29287.72139297008
5th percentile	4293.968413231643
95th percentile	78482.79920477136

Table 4. Frame per second values

3.4. Future Work

Naturally, the project can be extended in many ways.

The first possibility is instead of using separate producers and consumers, we might use Kafka Streaming API to make this communication faster. However, I haven't yet found a way to combine this with our current data structure.

The second possibility is the inefficiency of JSON in this context. After optimizing the field extraction we can see that the majority of the execution time is spent for data transferring from Kafka. However, we don't need to transfer all of the object in order to get only a few of the fields. In this context, we might be able to leverage some other data storage formats, maybe column-storage, in order to efficiently stream only the necessary fields.

A third possibility is to make this system distributed if we have gigantic amounts of data to benefit from scaling out in this manner. For example, we can stream only parts of the whole topic to different consumers and then merge the results using efficient communication protocols.