# Super Computing for Big Data - Lab 3

Chia-Lun Yeh (4718836) and Shikhar Dev (4773071)

November 6, 2018

## 1 Implementation

The implementation of the stream processor revolves around the two Key-Value state stores that are used. The first one, *topic-count* stores the topic names as the key and count as the value. The second one, *time-topic* stores the time as the key and the list of topic names as the value. Since there is no default serializer for List or Array, we store the list of topics as a String separated by commas.
Before calling the transformer, we filter out incomplete data, get the *allNames* field and *flatMap* it to key-value pairs where the key is the same as the original stream (the timestamp) and the value is a single topic. Then the following processes are done within the transform function.

- For every incoming record containing (time, topic), we check whether the topic exists in *topic-count*. If not, we add the topic with count 1. Otherwise we retrieve the current count and add 1 to it. In addition, we check whether the time exists in *time-topic*. If not we add it with the topic, else we append the topic to the value of the entry. The key in *time-topic* is the same key as the incoming record.
- For every second, we iterate over all entries in *time-topic* and delete all the entries with key before an hour relative to the current timestamp obtained in the *ProcessorContext*. We use the WALL_CLOCK_TIME as it doesn't depend on whether there are streams coming in. For each deletion, the topics are parsed out and the counts of them are also updated in *topic-count*.

## 2 Questions

### 2.1 General

1. **What is the difference, in terms of data processing, between Kafka and Spark?**

   Apache Spark is a general purpose distributed computing platform, which processes big data in batches. Apache Kafka on the other hand is a unified data pipeline, optimised for mediating data transaction between producers and consumers. Both of these platform support streamed data processing with fundamentally different principles, both optimised for different purposes. Spark stream supports many platforms for data ingestion, including Kafka, Flume, Kinesis, or TCP sockets. It divides and processes streams in micro-batches and provides near real time throughput with latency typically in order of seconds. Kafka stream on the other hand is used to process data stored in Kafka in an event driven fashion, with latency in order of milliseconds.

2. **What is the difference between replications and partitions?**

   Kafka applications pipeline data segments that are most often too large to fit in a single compute node. In most cases, target datasets need to be broken down into segments for storage across multiple nodes. This process of breaking data into segments is called partitioning, and each of the segments is referred to as a partition. Note that each segment is

different to one another here, and all the segment cumulatively form the entire dataset. The partitions also relates to the degree of parallelism that can be achieved for the processing because each consumer in the consumer group consumes a partition in parallel. In critical applications where loss of data is unacceptable, each of the partitions need to be copied in separate compute nodes for fault tolerance. This replication allows automatic failover to the replicated datasets in case of server failures. The number of replication and partition are configurable, and should be turned based on the requirement of an application.

3. **What is Zookeepers role in the Kafka cluster? Why do we need a separate entity for this? (max. 50 words)**

   The main role of Zookeeper is to coordinate different nodes and maintain all the metadata such as existing topics of the Kafka cluster. It has similar function as a master node or configuration server in other systems. This separate entity is needed because a distributed system needs a centralized coordinator.

4. **Why does Kafka by default not guarantee exactly once delivery semantics on producers? (max. 100 words)**

   When a producer sends a message, it is saved to the log and an acknowledgement is sent to the producer. Either to save the log or to send the acknowledgement first decide whether the delivery is at-least-once or at-most-once. Exactly-once can be achieved by adding a unique identifier to each message so that duplicates can be detected or using a transactional scheme that sync saving and confirming. However, these strategies are likely to add performance overhead to Kafka [1]. It also requires users to handle some of the complexities in their applications. Thus it is supported but not as a default.

5. **Kafka is a binary protocol (with a reference implementation in Java), whereas Spark is a framework. Name two (of the many) advantages of Kafka being a binary protocol in the context of Big Data. (max. 100 words)**

   Kafka implements a unique messaging technique via its binary protocol, primarily aimed at optimizing performance. Two prominent advantages of Kafka being a binary protocol are as mentioned below.

   - The ability to multiplex requests: Kafkas protocol batches messages together, reducing the number of produce or fetch requests. Also, the communication data is customized to eliminate overheads of standard protocols, enhancing network utility.
   - The ability to simultaneously poll many connections: Kafka allows parallel connections to multiple brokers simultaneously, enabling multiple data transactions across multiple partitions of the same topic parallelly.

## 2.2 Analysis

1. **On average, how many bytes per second does the stream transformer have to consume? How many does it produce?**

   Originally we wanted to estimate the value by hand and would guess around 34 KB per second given that each file is around 300 KB. But since there might be timestamp or other data appended to the records, it's really difficult to have a reasonable guess. In the end we used JConsole as mentioned in [2] and look at the metrics that are collected. In figures 1 and 2, we show the incoming and outgoing bytes per second as shown in JConsole. On average, around 45000 bytes/sec is consumed and around 1400 bytes/sec is produced. It's quite interesting to see that every 15 minutes, there is a drop in the input as shown in figure 3. This is probably when it is downloading the new file and we will ignore this artifact.

2. **Could you use a Java/Scala data structure instead of a Kafka State Store to manage your state in a processor/transformer? Why, or why not? (max. 50 words)**

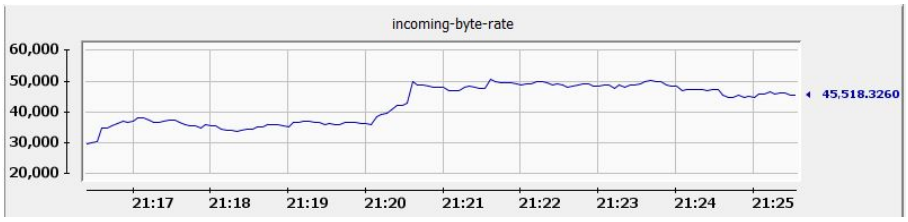


Figure 1: Incoming byte rate observed on JConsole.

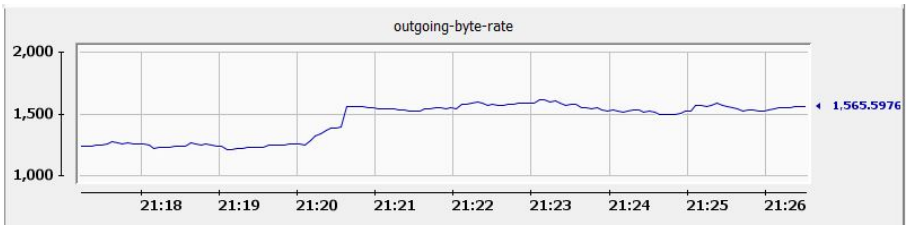


Figure 2: Outgoing byte rate observed on JConsole.

We could use Java/Scala data structures instead of a Kafka State Store but we would loose (or at least it'll be very difficult to handle) the persistence and fault-tolerance offered by state stores. It would also be very difficult to scale the application to multiple servers [3].

3. **Given that the histogram is stored in a Kafka StateStore, how would you extract the top 100 topics? Is this efficient? (max. 75 words)**

   An efficient method is to maintain a state store that saves always the top 100 topics and their counts. When a new (name, count) pair arrives, we can check whether the topic exists in the state store and whether the count is high enough to stay in the top 100 list. We can optimize the state store to sort fast so that we always compare a newly arrived record with the lowest count and not iterating through all the objects.

4. **The visualizer draws the histogram in your web browser. A Kafka consumer has to communicate the current state of the histogram to this visualizer. What do you think is an efficient way of streaming the state of the histogram to the webserver? (max. 75 words)**

   Since the state of the histogram is updated as stated in the last question, we can send the (name, count) pair to the server for each updates. If the web server takes messages less often than the updates of the records, we can use a map that takes name as the key to store the messages that are to be sent. This way if there are messages of the same name, only the latest one is sent.

5. **What are the two ways you can scale your Kafka implementation over multiple nodes? (max. 100 words)** Scaling in Kafka stream applications are primarily attained via distribution in storing data and via parallelism in application instances.

   (a) Partitioning topics enable applications to store and access data across multiple server nodes. It introduces scalability and fault-tolerance on Stream applications, allowing them to store increasing amounts of data by increasing the number of nodes.

   (b) Scaling in terms of processing speed can be achieved by breaking applications into multiple independent tasks, running in parallel across a processor topology.

6. **How could you use Kafkas partitioning to compute the histogram in parallel?(max. 100 words)**

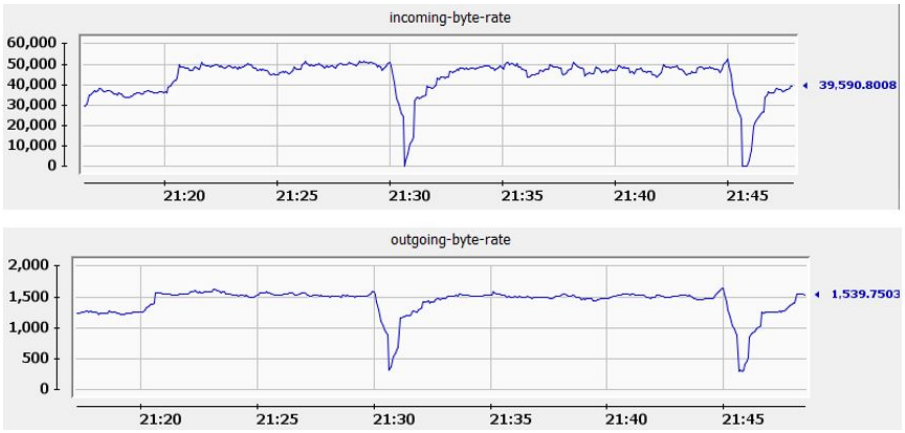


Figure 3: Monitoring longer in JConsole.

First, partition the *gdelt* topic so that the stream processors can filter the records and extract the topic names in parallel. Then, for each of the extracted name, we route the (time, name) pair to a specific partition of the *gdelt-name* topic. The partition is based on topic names so that the same name would be in the same partition. Since the same names are always in the same partition, the downstream processor can safely process them in parallel. Each of the processor maintains its *topic-count* and *time-topic* state stores, and performs what our current transformer does.

# References

[1] Apache kafka documentation. https://kafka.apache.org/documentation/theproducer.

[2] Datadog: Monitoring kafka performance metrics. https://www.datadoghq.com/blog/monitoring-kafka-performance-metrics/.

[3] Stackoverflow: Kafka streams local state stores. https://stackoverflow.com/questions/49381064/kafka-streams-local-state-stores.