Lab3 Report

Group 4
Kyriakos Psarakis (4909437) — Rafail Skoulos (4847482)
ET4310 Supercomputing for Big Data

November 1, 2018

Outline of the code

StateStores:
At first, we create two StateStores, one to store the key-value pair (name, count) containing how many times a name has occurred in the stream the past hour. The second one to store the key-value (timestamp,name) in order to keep a log with the timestamp of every name occurrence, so that we can decrement the previous StateStore when an hour passes.

Processing before and after the transformer:
As in Lab1, we transform each row in order to send to the transformer a tuple containing the GDelt timestamp and a single name. After the transformer, we send the (name,count) tuples to the gdelt-histogram topic.

Transformer's context schedule:
In order to decrease the count of each name when the timestamp ages over an hour, we created a scheduled process. This process every 500msec goes through our (timestamp,name) StateStore, next it finds all the entries that were made more than an hour ago, removes the entry and decreases the name's count in our other StateStore (name,count).

Pseudo-unique Timestamp:
We observe that the stream context timestamp is not unique. Consequently, we cannot use it, as is, as a key in a key-value pair StateStore. That led to the implementation of two functions that create a pseudo-unique timestamp in the following string format (timestamp-i) where i = 1,2, 3, . . . , n (n being the times that this timestamp has occurred).

Increment the count:
Now, the main functionality of increasing the count in the (name,count) StateStore happens inside the transform function of the Histogram transformer. At first, we check if the count of the name that we process exist. If not, we initialize it with the count of 1 but, if it already exists we increment it by 1.

General Kafka questions

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

Spark is an open-source framework which is designed for distributed processing of a large volume of data. It is designed in such a way that it can perform batch processing and stream processing. However, it uses micro batching for streaming data, meaning that it collects data for some time and then process these micro batches, so it not a really streaming platform.
On the other hand, Kafka Streams, which utilizes Kafka for more than being a message broker, does not do mini batching. The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams[1]. As a result, this makes Kafka Stream a real streaming application. To achieve that, when it performs aggregations, filtering, etc, it writes back the data to Kafka.

2) What is the difference between replications and partitions?

Every topic in Apache Kafka is divided into some partitions. Every message in a partition has an unique identifier called offset and can be identified in a cluster by a tuple consisting of the messages topic, partition, and offset within the partition.
On the other hand, replications are copies of existing partitions. In specific, each partition is replicated across a configurable number of brokers in order to succeed fault tolerance, if any of the brokers fail. Each of the partitions has a leader server and some follower servers (or none). The leader handles all read and write requests for the partition. In case of leader failure, followers replicate leaders and take over. every write in a leader partition, is replicated to the followers.

3) What is Zookeepers role in the Kafka cluster? Why do we need a separate entity for this?

ZooKeeper is used to store a lot of shared information about Consumers and Brokers. Especially, it is responsible for the controller election, the configuration of topics, the access control lists and the membership of the cluster. We need a separate entity for this because it ensures the coordination of all brokers in the cluster.

4) Why does Kafka by default not guarantee exactly once delivery semantics on producers?

In its attempt to succeed exactly once delivery semantics, Kafka assigns a unique ID to the producer during initialization and a sequence number, which starts from zero and is monotonically increasing, in every message per topic partition. The message will be rejected by the broker, if its sequence number is not exactly one greater than the last committed message from that PID-TopicPartition pair. Messages with a lower sequence

[1] https://kafka.apache.org/intro

number result in a duplicate error, which can be ignored by the producer while those a higher number result in an out-of-sequence error, which indicates that some messages have been lost, and is fatal[2]. The problem is that since each new instance of a producer has a new, unique ID, the exactly once semantics can be guaranteed within a single producer session.

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.

One important advantage of being a binary protocol is the better network utilization. Since it does not use HTTP for ingestion, it avoids HTTP headers, which can add a lot of size to otherwise small messages.
Another asset is that it delivers greater parallelism. The client can open up TCP connections to multiple brokers in the cluster and send or fetch data in parallel across multiple partitions of the same topic.

Questions specific to the assignment

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

In order to start our calculations we took some measurements. At first, the average string size for the names is 18 bytes, the size of the Long type count is 8 bytes and finally the size of our string timestamp is 15 bytes. Afterwards, we counted the average amount of tuples that a transformer processes per second, and they are 72. So, the amount of bytes the transformer consumes per second is:
(sizeOfTimestamp+sizeOfName) * tuplesProcessed/sec = (15+18) * 72 = 2376 bytes per second. The amount of bytes it produces per second is:
(sizeOfName+sizeOfCount) * (tuplesProcessed/sec+tuplesDecremented/sec) = (18 + 8) * (72 * 2) = 3744 bytes per second. The tuples that we decrement per second are the same as the tuples that are being processed per second.

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?

We can not use a Java/Scala data structure to manage our processors/transformers state. The main reason is that processors/transformers work in a distributed fashion being in different threads or even machines inside a cluster. Thus, a distributed data structure that guarantees fault tolerance is needed such as the KafkaStateStore.

[2]https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messag
98-ExactlyOnceDeliveryandTransactionalMessaging-IdempotentProducerGuarantees

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

Due to the fact that a KafkaStateStore is distributed and in a key-value format, traditional sorting methods are hard to implement. However, a way to do this is to create a StateStore that contains only the top-100 and at each timestep have the knowledge of what is the minimum count. So, if the next tuple, that comes from the transformer, has a bigger count than that we can swap them. Additionally, this top-100 StateStore should be updated every time the decrement process gets called. This is not very efficient but, easy to implement in a distributed key-value StateStore.

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?

Many times, we want to connect our kafka output topic to an external destination. The default kafka connectors might not be suitable for all our needs. In our case, we need to stream the state of our histogram to a webserver. For that, we need to implement a custom kafka sink connector but, the community has created many connectors that work with existing No-SQL and SQL databases which in extent are easy to access from a webserver.

5) What are the two ways you can scale your Kafka implementation over multiple nodes?

The first way to do this is to start another instance of our stream processing application on another machine. Those two instances will become aware of each other and will start to share the processing work.
The second way to scale Kafka is to increase the number of producers and consumers that write to the same Kafka topic. This way we exploit the fact that topics can be easily partitioned, writing and reading, to and from, the same topic on multiple disks. Thus, avoiding the disk I/O bottleneck.

6) How could you use Kafkas partitioning to compute the histogram in parallel?

We can divide the histogram topic in multiple partitions and at the same time open up TCP connections to multiple brokers in the cluster, according to the nodes of our machine. Each broker will be the leader for one partition. In this way, each connection can work on different partition and as a result the histogram will be computed in parallel.