263-3826-00 Data Stream Processing and Analytics **16 Mar 2019**
Semester Project - Design Document

Students: Jack Clark, Zhifei Yang Legi: 17-938-440, 17-941-998

We plan to build a streaming system for the backend of a social network. We plan to use a plain Java program to feed the provided data into Kafka, which will then be consumed by Flink. We chose Flink due to our familiarity with Java programming (and lack of familiarity with Rust programming), as well as the fact that we felt that the general streaming concepts we have learnt so far mapped more obviously to the Flink API than the Timely API. Furthermore, we felt the extensive documentation (including the draft version of a book) gave us more confidence that we could independently solve issues that arose compared to issues in Timely.

1 Reading and Sending Input

We have a source program which reads the csv files into memory and orders them based upon their timestamp. We shuffle the stream by moving each element by computing a new index for it, based upon its timestamp plus a bounded random amount of time (it can be a negative amount of time). We plan to serialize the messages using the Google protobuf library, but without modifying any fields.

The source program will send data to Kafka at a rate proportional to the timestamps in the messages. Obviously, if we were to leave the rate unchanged, we would have to wait a year to send all of the input. This can be resolved by introducing a speedup factor. In our implementation, we define 1 day of time in the timestamp domain to be 1 second of wall clock time in our source program. This should allow us to feed all the input into our system within roughly 6 minutes. Since we feed Kafka at a proportional rate, Flink can consume data from Kafka as fast or as slow as it can handle.

We plan to use combine the three streams into one Kafka topic, namely `Events`, as we see no advantage (at this stage) of having multiple topics, as we can increase the parallelism available to the Flink consumers by having a sufficiently large number of partitions. Deciding to send data to Kafka before Flink introduces extra complexity, as well as latency into the system. Our justification for doing this is that we can have deterministic replay of data from Kafka to Flink for testing purposes, and we are not limited by feeding data into Flink at a proportional rate, which would make a test take 6 minutes. Of course, we will still perform end to end testing with the proportional sending rate from the source program, however this will be done less frequently and can therefore afford to take longer.

2 Processing in Flink

2.1 Challenges

In this subsectoin we describe some fundamental challenges that we have identified that have shaped our design decisions and will motivate the next few subsections. The initial challenge we faced was to be able to associate replies to comments with the original post, so that we can count how many replies there are for each post. This is difficult with the schema provided, as replies do not store the id of the original post, but instead the id of the comment or reply it was replying to. We can follow this chain until we find the comment with the original post id. However, since there is no guarantee that we will see both the reply and the comment in the same time window, we must store this state across time windows, and also ensure it is stored durably, as we must always remember all mappings, as we can never know when we will get a reply to an old post.

We tried to solve this by keeping a global mapping of either comment_id or reply_id to post_id. However since we want to do this inside a KeySelector as a means to keyBy post_id, we are not operating on a keyed stream. This limits our ability to store state across parallel tasks in Flink. We can overcome this by storing the mapping in an external key-value store such as redis. This solves our immediate problem, however does introduce considerable latency into our computation compared to having KeyedState.

Unfortunately, the above will not work in the presence of reordering. Let's say we have two messages, a comment at t=1 and a reply at t=5. If they arrive in order, then everything works out, the comment_id to post_id mapping will exist in the k/v store. However should they be reordered, when we come to resolve the reply_id to post_id mapping, we must already have the comment to post mapping in the k/v store for this to work. We outline our plan to resolve this issue below, however it fundamentally comes down to buffering data for the duration of our bound on the possible delay before feeding it to the rest of our pipeline, which introduces significant latency into our system. Fortunately, as our system only needs to output results at most every 30 minutes, this isn't actually that much of a problem.

2.2 Flink Consumer

Our Flink program has one logical Kafka consumer, which will consume messages from the only kafka topic we use. Messages are deserialised using the method automatically generated by the protobuf compiler. We use event time in Flink as this is faithful to the original timestamps of the messages.

2.2.1 Task 1

For Task 1, we use an un-keyed sliding window operator to gather messages into a list of activities that happened in the past 12 hours, which outputs results every 30 mins. This gives us a buffer to deal with out of order messages. However, the elements of the window still have to be ordered according to event time, therefore it is necessary to sort the window by timestamp. The window has an allowed lateness that is no less than the bound of the delay we set for replaying. So, every 30 minutes, the operators outputs all events happened in the past 12 hours in order, and then, we use a keyBy operator to distribute them based on post_id. Because a `comment` or a `like` message does not necessarily carry the id of the post it belongs to (it only carries the id of whatever is its direct parent), we will implement the keyBy operator in such a way that if the message does include the post_id, it pushes the mapping information to an external key-value store, or if the message does not, it queries the key-value store for the mapping and pushes it to the key-value store as well. Since the messages has been ordered, when a mapping `(comment -> post)` is queried for, either it already exists in the key-value store, or the mapping of its parent (`(parent(comment) -> post)`) exists, so we can always figure out the mapping.

Now that we have all events happened in the past 12 hours ordered and keyed by post_id and updated every 30 minutes, the active posts are exactly all posts that any event happened in the past 12 hours belongs to. Then the analytics can be simply done by having a counting operator that outputs the number of replies, comments and unique persons for each post. To output the number of unique people every 1 hour, we can combine the results of the last two 30 minute windows using built in Flink functions.

2.2.2 Task 2

Similarly, a sliding window of 4 hours that outputs result every 1 hour will be used for this task. We plan to choose the top 10 people who has generated maximum number of activities in the past 4 hours, so that we can have more information to find similar persons for them.

2.2.3 Task 3

For this task, we will define some metrics and continuously compute them as new activities take place. The new activity will also be compared with the metrics, and if it matches our definition as *unusual*, it will be sent to the output for the moderators to do further processing. As we don't know a lot about fraud detection, we have chosen some statistics that seem reasonable to us. We will track the average number and standard deviation of likes for a post in order to detect gaming of the social network. We will also check the average number of words in a reply to try to determine whether or not a post with a lot of replies has legitimate replies, or is simply using a bot to generate short fake replies.

3 Output

We will output the results of each task to separate log files to facilitate easy testing. We will write an test to compare the output of the Flink job with the output of our test program which will calculuate the correct results in a batch process.