Final Report

Data Stream Processing and Analytics
Semester Project - Spring 2019

oezbeko, patricst

1 System Overview

A summary of the designed system is given in Figure 1. Our system uses Apache Kafka in order to read input data and output the results. Streaming computations are done by using Apache Flink Library.

Input streams and tables are first preprocessed to make sure that the data is completely in order and integrity constraints are not violated. For example, a post's timestamp must be before any comment with reply_to_postId that points the post. Same also apply for comments. Additionally, timestamps are added to the events by parsing the event time which is included in the data. After the data is processed, it is copied to the working directory. The path for the working directory is configurable. The code for the preprocessing step can be found under the src/main/java/preparation directory.

After the data is copied to the working directory, a Kafka producer writes the data in the streams folder to the corresponding Kafka topics. The code for Kafka producers and consumers, serialization and deserialization schemes can be found under /src/main/java/kafka directory.

The next step after writing the data to Kafka is to read it as if it is a realistic stream source. Namely, this is the Task 0 which is mentioned in the semester project document. Since our data is saved to Kafka, we are able to perform Task 0 calculations when reading from the three Kafka topics. The core logic of Task 0 sits in the process function ReorderProcess. When we read the first entry we calculate the time difference between the timestamp and the current processing time which allows us to translate every further timestamp. So when we read a new entry we add this difference, a random offset and calculate the speedup to get the output time. To get the reordering of the entries we read enough from Kafka and put everything in a sorted buffer from which we output. After Task 0, the streams are watermarked using BoundedOutOfOrdernessTimestampExtractor class from Flink. Maximum out-of-orderness is configurable, and both ReorderProcess and watermark assigner will use the same maximum out-of-orderness.

The details about how we approached the analytical tasks are going to be discussed in the next section. Each analytical task is going to stream its results to a dedicated Kafka topic in a human readable format.

2 Analytical Tasks

2.1 Task 1

For this task, our approach was to first handle comments that do not directly correspond to any post (does not have a rely_to_post_id) and create a reliable comment stream that contains all comments with their postId. This is done with a process function called CommentResolutionProcess. Then, comments and likes are mapped to a common representation of a Tuple, keyed by postId (all events have a postId after the matching process) and another process function (Counter Process) counts the number of comments, replies and user engagement. If the post is active, the process will output the updated counts every 30 minutes. Comment and reply counts will be updated every 30 minutes whereas user engagement counts will be updated every hour. An outdated value will be added to the stream if it is not the time to update that count (This applies for user engagement counts).

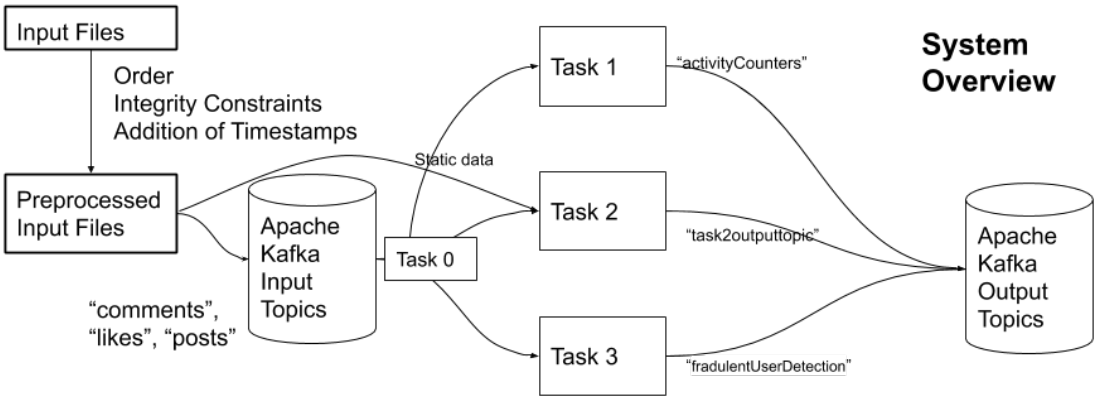

Figure 1: System Overview

As a final step, a window-all operator is used on the counts in order to group them into 30-minute windows that contain all keys, so a more structured output can be formed. The output is a Tuple2 (Long, Iterable<Tuple4>). Long corresponds to the end of the window and the iterable contains posts with post ids and 3 counts as described in the semester project document.

2.1.1 Comment Resolution Process

In this part, the Broadcast State Pattern as described in Apache Flink documentation is used. Comments with postIds are keyed by their postId and remaining Comments are broadcasted to downstream and cached using an operator state. Each key has a MapState (equivalence class) which works as an equivalence class of comments that reply to same postId (it includes indirect relationships). The cache is a map from a commentId to a list of comments. All comments in a commentId's list, contains comments with reply_to_comment_id which matches that commentId.

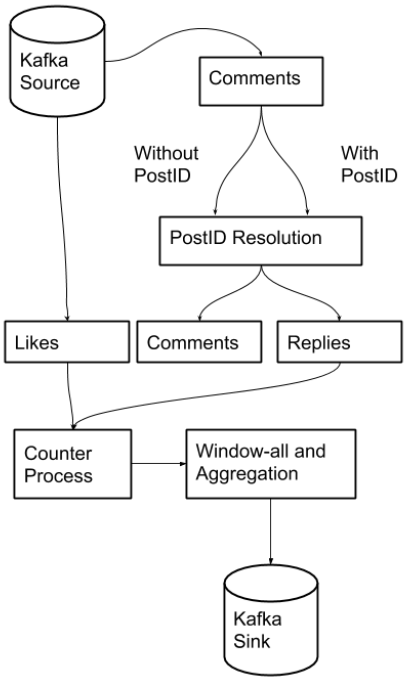

Figure 2: Task 1 Overview

Each time a new watermark is received, if the cache contains an entry with a commentId included in the equivalence class, these events are enriched with postId (because we matched them), in other words the currentKey in the context, added to the equivalence class and collected recursively. Consequently, the entry with that commentId is removed from the cache. The representation of this process is given in the figure below.

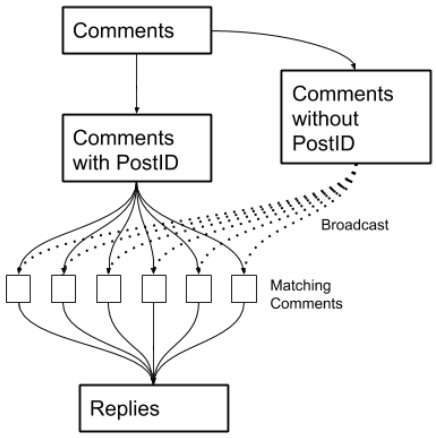

Figure 3: Task 1 Comment Resolution

2.1.2 Counter Process

CounterProcess contains the main logic for Task1. Since all comments have their reply_to_postIds after the Comment Resolution Process, we can partition these events by their postId. First, likes and comments are mapped into a single representation. A Tuple4 of (PostId, EventType, PersonId, Timestamp). EventType indicates whether the event is like, comment or reply. Comments are directly connected to their post whereas replies can also be connected in indirect ways (comment of comment). Since we need to update the counters every 30 minutes, for all incoming events, first the timestamp is checked. If the event belongs to the current window, then counters are updated directly. If the event belongs to a future window, then, the starting time of this future window is calculated, and a counter is created for that specific window. This is done through a MapState which maps a window's beginning timestamp to a count. Three of these MapStates exist, corresponding to three counters. A timer is registered for each window boundary. After the first timer is set, the next timer is also registered in onTimer method. After collecting the events (emitting the counts) and incrementing the window, MapState for each counter is checked for the beginning timestamp of the current window. If it exists, the counter which is accumulated in that key is added to the main counter. One important point to mention is how unique user engagement counts are calculated. A set of userIds is created for each key in the keyed stream (postId). If an event which belongs to the current timestamp is received and event's personId does not exist in the set, that personId is added to the set and userEngagement counter is incremented. If it belongs to a future window, its personId is added to the MapState. Here, a HashSet is used instead of a simple counter because an event might be unique in processing time but might not be unique in the event time because we can receive another event with earlier timestamp from the same user.

Feature ID	Calculated with	Description
0	Posts	Number of profanity filtered words over the number of posts
1	Posts	Number of unique words over the number of words in posts in average
2	Comments	Number of profanity filtered words over the number of posts
3	Comments	Number of unique words over the number of words in posts in average
4	Comments	Number of unique words over the number of all words
5	Likes	Number of people liked by a user over number of likes by a user

Table 1: Task 3 Features

So, before incrementing the userEngagementCounter in the onTimer method, userIds are checked in order to understand whether the event is unique or not.

2.1.3 Tests

For Task 1, two tests are implemented. All tests are self-contained, meaning that there is no need to make any setups (apart from setting up a Kafka cluster) prior to running the tests. Tests are going to copy the preprocessed input files to the working directory, create input Kafka topics, compute correct results in a single thread using batch processing, use the processes to compute the results from the streaming application, write the results to a Kafka topic and check whether the outputs match by reading from the Kafka topic which is just created.

The first test checks whether CommentResolutionProcess works properly or not. For all comments, parent postId is found. Then, this is compared against the commentId-postId pairs returned from the streaming application. The test checks three things: all commentId-postId pairs are correct, all comments are processed only once and all comments in the input directory is processed.

The second test checks whether Task 1 overall works as intended. For this test, a data structure which can give any type of count at any timestamp. This is simply dome by a HashMap which maps a postId to a linked list of sorted timestamps. given a key and a timestamp, the linked list corresponding to a key is traversed until we reach to an element which is greater than the given timestamp. The index gives us the count. For each window, we know that the count which is returned from the streaming application should be greater than the count from the last window (the real count calculated by monolithic batch processing) and less than the true current count. This constraint is checked for all posts for all updates.

2.2 Task 2

In Task 2 there exist two types of features that we consider: static which are table-based and dynamic which are stream-based. Throughout Task 2 we use ScoreHandlers in order to keep track of the scores one of the ten selected users has. A ScoreHandler holds a data structure that maps the userId to the score value. The static data is processed before we start the stream on all three topics that are registered to Kafka. For these streams we have custom process functions that calculate the ScoreHandlers and return a 4 hour window. After that we need to merge all dynamic ScoreHandlers together and add the static score on top. Finally we filter out friends (inactive users) and sort everything according to the scores. Top 5 people will be returned as result.

2.3 Task 3

Task 3 can be separated into two subtasks: calculation of the features and outlier detection. For each input stream, keyed by personId, features are calculated as described in Table 1. Then, OutlierDetectionProcess calculates the mean and variance of these features over different people every day (event time), and return users with a feature that is three standard deviations away from the mean.

For posts, there exists 2 features, the first is the number of profanity filtered words used by a person over the number of posts written by this person. This feature is used to detect abusive chat and report any person who excessively uses strong language. The second feature is the number of unique words over the number of words in posts in average. If a person uses the same words over and over in his/her posts, that might mean that this post is a spam or written carelessly. In that case, this feature is going to be significantly lower than other people. Furthermore, this feature is included as an example in the semester project document.

For comments, first two features are the same. the third feature is the number of unique words over the number of all words. If a person has a limited vocabulary size compared to his/her number of posts, it is likely that he/she writes same things over and over again. This is a suspicious behaviour since it might be an indication of a spam message which is randomly pasted all over the forum. In that case, this feature might be useful for detecting such behaviour.

For likes, the only feature which is calculated is the number of people liked by a person over number of likes by a person. Imagine a user who tries to increase the likes he/she gets by opening a fake account which likes his/her posts. This statistic will be particularly low for such a fraudulent account since it is going to probably like the person's main account excessively. One problem about the calculation of this statistic is the fact that we

do not have the liked person's id in the like event. Because of that, a BroadcastState is used in order to get postId-personId mapping from post steam.

In the end, all of these features are united to create a stream for all features, all of these features will be updated each watermark, sent to downstream and keyed by featureId. Then, OutlierDetectionProcess is going to detect outlier people for each feature. Once the mean and standard deviation is calculated for the first time in the first day, the outlier detection becomes functional. The mean and the variance is recalculated every day. This behaviour can be adjusted by changing a constant inside the source code of OutlierDetectionProcess class.

3 Individual Contribution

We divided the tasks into 2 equal parts, my partner Patrick did Task 0 and Task 2, and I implemented Task 1 and Task 3. Preprocessing part, event models and Kafka producers/consumers are implemented by both of us, we edited each others' work in order to come up with a good design overall. We regularly had meetings in order to talk about each others' tasks and share information. This collaboration led to significant improvements over the final design, and helped us to explore Apache Flink more effectively.

4 Comments

4.1 What could be done differently?

As a general observation for streaming applications, I believe that test driven development could be a very good development practice. The reason for that is, for streaming applications, there are many ways to do the same calculation and it is easy to switch between different dataflows. However, designing a dataflow for a streaming application is not as easy as designing a classical map-reduce job (or a Spark application); one needs to cope with out-of-orderness, and always be careful about the difference between processing time and event time. In my case, many changes that I have made for efficiency caused erroneous results. Since I did not produce my test cases before starting the development, it took too much time to understand that I was on the wrong path. Maybe this is because I did not have enough experience on stream processing before, but I believe that writing the tests first would increase my development speed and also the robustness of my solutions.

Throughout this project, I relied on the low-level Process API for my streaming applications. The reason was flexibility, I wanted to be on the safe side and to be able to implement a correct solution in any case. I could have spent more time by reading the documentation to find a more elegant solution with higher-level API with even less time. For example for Task 1, I implemented something similar to a window operator for the counting art which took a lot of time; maybe using a window operator with an aggregation might be also valid and I could have implemented that solution in less time.

The third thing which could be done differently is Task 3. As described in Section 2.3, mean and variance are regularly recalculated in batches. I could have come up with an online algorithm to calculate these statistics. For mean, the calculation is intuitive, the only thing to be careful is to be careful about whether we insert a new key or update the statistics for a key, so we should not blindly take the average of all incoming numbers. The standard deviation is not intuitive, but I believe that it should be possible by using a modified version of Welford's Online Algorithm, as the variance of the sum of independent random variables is the sum of each random variable's variance. So we can eliminate a key's contribution to variance if we need to update it. Nevertheless I could not find any material on how to make such a computation, and could not take the risk of not being able to implement it.

4.2 Other Comments

Generally, the project was fun and informative. I believe that it gave me a very good introduction on stream processing, and how it is different from batch processing. While doing the project, we had two problems about Apache Flink, and it was a challenge for us to find a workaround. The first challenge was the limitation of timer service to keyed streams, in Task 0, we need to calculate a time to fire each event according to a delay and speedup factor, and fire the event when the time comes. Since we could not use the timer service in order to send events, we needed to make a check in process element method. Of course it is unlikely to have such an operator (adding out-of-orderness) in real life, but I believe that there might be a case where using timer service in a regular data stream (not keyed) is necessary. Another feature that I would expect from Flink is a shared state among all workers in a keyed stream as a service. I also think that this is contradictory to the nature of Flink, but there are many cases that such a feature is necessary. An aggregator process with a queryable state combined with side outputs from workers, or patterns like Broadcast State can also be used in such cases.