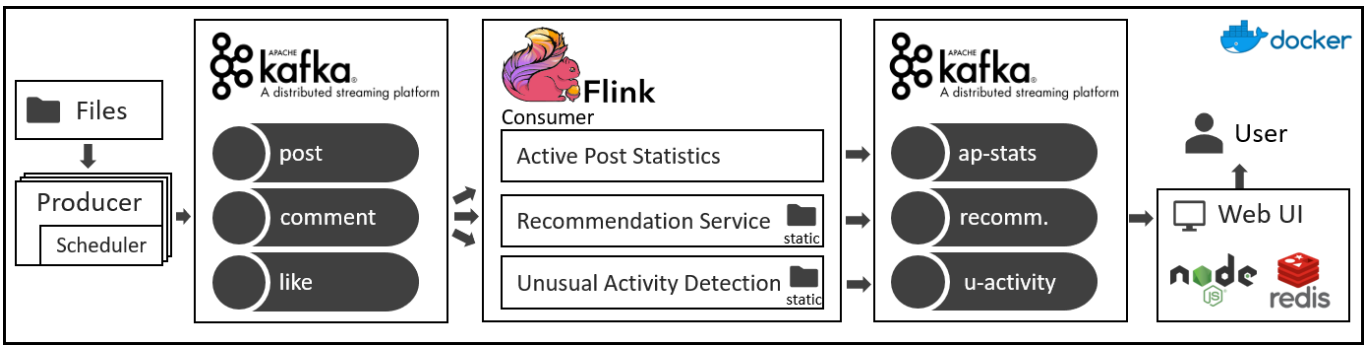
DSPA 2019 - Final Report

Nicolas Kuechler, 14-712-129

Architecture



The figure shows a high-level architectural overview of the application. Each stream has a producer reading the file from disk and writing the events to Kafka. The Flink application consumes the Kafka topics with the help of Avro schemas, processes the event streams and writes the output results in JSON back to Kafka. A web UI using Node JS and Redis consumes and displays the outputs for a certain time. All components run in separate Docker containers to simplify the coordination. In the following, the algorithms for the tasks are discussed along with `references` to the files where the functionality is implemented.

Data Preparation

The provided data has two issues. First, the streams do not adhere to the logical event time ordering (post before like/comment, reply before comment). This issue is resolved in a pre-processing step with a Python script `scripts/stream_cleaning.py`. A like or a comment/reply arriving before the parent is dropped. The comment subtree of a removed comment is also dropped. Another issue with the data is that event times are using a 12h clock without a.m. or p.m. period. This remains unchanged.

The functionality of serving events proportional to their event time including a speedup factor and a bounded random delay to serve events slightly out-of-order is implemented in the `stream-producer`.

Analytics Task - Shared Part

This section is about the parts that all three analytics tasks have in common. The first step for all input streams is to assign timestamps and watermarks using a `BoundedOutOfOrdernessTimestampExtractor` that takes into account the maximum random delay used in the producer.

In the comment stream, the next step is to resolve the post id for every reply `(CommentPostIdEnrichmentBroadcastProcessFunction)`. This is necessary because only root comments contain the link to the post but all three analytics tasks rely on the reference for all comments. (Note: the implementation changed again from the midterm report due to timestamp/watermark issues)

Analytics Task - Active Posts Statistics

The goal of this task is to aggregate statistics of active posts to find the posts which triggered the most discussions (comments, replies) among a large group of different people within the last 12 hours. This provides insight into what are currently the hottest topics in the social network.

Implementation

The input of the pipeline are the 3 streams with timestamps and watermarks and additionally, each comment/reply contains the id of the post. The heart of the implementation is in the file: `ActivePostsAnalyticsTask`.

The first step of the pipeline is to union all three inputs streams into one stream of events. For this, all streams are mapped to a common POJO `(PostActivity)` discarding all information except the type of the event (post, like, root comment, reply comment), the post id and the person id associated with the event. Afterwards, these post activities are keyed by post id.

For root- and reply-comment counts there is a separate sliding window of length 12h and slide 30 min with an incremental aggregation function `(TypeCountAggregateFunction)` only counting the events if they are of the respective type. Since all events including posts and likes are assigned to these 12-hour windows (but not counted), every post with interactions during the last 12 hours produces an output every 30 minutes.

The unique people count is implemented with a low level process function `(UniquePersonProcessFunction)` in order to reduce the amount of state. For every post and hour, there is a set of people who interacted with the post. A timer is registered such that it fires at the end of every active window, merges the person sets of the last 12 hours and outputs a count of unique people. Sets that are not required anymore are discarded to avoid unbounded growth of the state. In the end, the three streams of counts are combined and written to Kafka.

Analytics Task - Recommendations

The goal of the task is to find similar users for a set of selected users based on interactions of the last 4h and static data. The high-level idea is to characterize each user with a profile of tags, tag classes, forums, places, languages, and organizations with which he is somehow related (e.g. if a user has two posts in forum X, then the category forum X is assigned a count of two). This profile can be seen as a vector and hence two user profiles can be compared with various similarity measures. For simplicity, we implemented only the dot product but the application could easily be extended to others. For every selected user we end up with the similarity to every other user and can filter for the best matches to arrive at a good recommendation.

Implementation

The main part is implemented in the file: `RecommendationsAnalyticsTask`. As in the active post statistics, the first step of the pipeline is to union all three inputs streams into one stream of events. To achieve this all streams are mapped to a common POJO `(PersonActivity)` only preserving the information of activity type, person id, post id, and associated categories.

In an additional step the categories are enriched with static data (e.g. a post with a tag also counts for the tag class, a post in a forum also counts for the tags of the forum, ...). Additionally, comments and likes inherit certain categories from the post (e.g. a like of a post counts the tag of the post) `(CategoryEnrichmentProcessFunction)`.

Afterwards, the enriched events are keyed by person id and incrementally aggregated in a sliding window of length 4h and slide 1h `(PersonActivityAggregateFunction)` before they are combined with the static person activities `(StaticPersonActivityOutputProcessFunction)`. This results in a stream of person activities.

From this stream, two parts are created. One part contains the person activities of the selected people and the other part contains all person activities. These two streams are cross joined to calculate their similarity by broadcasting the person activities of the 10 selected people to all parallel operators `(PersonActivityBroadcastJoinProcessFunction)`.

In the resulting stream of person similarities, already existing friendships are filtered out (`FriendsFilterFunction`) and the largest 5 similarities per selected person and window (`TopKAggregateFunction`) are written to Kafka.

Analytics Task - Unusual Activity Detection

The goal of the task is to capture and continuously update "normal" user activity. Every incoming event both adjusts what is considered to be normal and is also compared against this normal behavior to check whether it matches. If the ratio of anomalous behavior is above a threshold within a certain window, the user is automatically flagged for a manual review.

Features

User activity in the form of incoming events is characterized by a set of features that were specifically designed to capture potential bot activity.

- timespan: the time between consecutive interactions of the same user
- contents short: the ratio of unique words for short contents
- contents medium: the ratio of unique words for contents of medium length
- contents long: the ratio of unique words for long contents
- tag count: number of tags of a post
- new user likes: users liking a post with a lot of likes from new users
- interactions ratio: the ratio between interactions involving content (post/comments) and likes

Implementation

The implementation in the file `AnomaliesAnalyticsTask` can be divided into multiple stages. In the first stage, the features listed above are extracted from the streams. The implementation of the features can be found in the folder: `anomalies/ops/features`.

In the second stage, a rolling mean and standard deviation is updated for every feature using Welford's Algorithm (`OnlineAverageProcessFunction`). The third stage consists of comparing the feature value with the rolling mean and standard deviation of the feature. In case the feature value deviates by too many standard deviations from the mean, the feature votes the event as anomalous. In the fourth stage, the votes of the different features are combined in an ensemble decision flagging suspicious events (`EnsembleProcessFunction`). The fifth and last stage collects the user activities in a 96-hour window and calculates the ratio of suspicious events. If this exceeds a threshold, the user is written to a Kafka topic for further manual inspection along with the events and features underlying the decision (`EventStatisticsWindowProcessFunction`).

Supportive Documentation

Usage Instructions

The application consists of multiple components that need to be started in the correct order. Since this is tedious we provide a dockerized environment. After installing all dependencies and cleaning the data, everything is controlled by a single script `scripts/_start.sh`. It takes arguments to run the application with different configurations. Details about how to run the application can be found in the `README`.

Verify Correctness

Every non-trivial user-defined function of every analytics task is tested with either a unit test or an integration test to verify the correct behavior. For the active post statistics and the recommendations task, there is an integration test verifying the complete pipeline. This is done by taking the first 6 days of the 1k-user streams and separately compute the expected results in batch form for every window. These expected results are then compared with the actual outputs of the Flink pipeline. All tests can be found in the folder `social-network-analysis/src/test/java/ch/ethz/infk/dspa` and follow the same structure as the application code. They can be run with `mvn test` from the `social-network-analysis` folder.

Work-Sharing

The approach for all analytics tasks was similar. First, we discussed the approach and outlined a skeleton of the dataflow graph together on paper. Following this, one implemented the skeleton with stubs for the user-defined functions. For each of them, an issue was created on GitLab and assigned to one of the team members. Completing such an issue involved both implementing the function and writing a test. This clear separation allowed us to work on the tasks in parallel and almost independently. Before merging functionality on the master branch we performed a code review to ensure both code quality and knowledge transfer.

For the data preparation and the shared part of the analytics tasks, I was responsible for the setup of the producer, the stream cleaning script and the comment to post id mapping. In the active post statistics task, I built the skeleton of the dataflow graph, the comment and reply count functionality and the integration test for the complete pipeline. In the recommendations task, I was responsible for the skeleton of the dataflow graph, the functionality to aggregate user activities in windows, properly inject static person activities into the stream, calculate similarities and select the top 5 per selected person. In the end, I wrote the integration test for the complete pipeline. In the unusual activity detection task, I implemented Welford's Algorithm to keep rolling statistics, the tag count feature, the new user likes feature and the ensemble decision combining the votes from different features.

Reflection

When starting all over again I would consider implementing most of the data preparation tasks in a Flink source instead of implementing them in a separate producer. Even though the initial design choice we made is a more realistic scenario in which events arrive in Kafka delayed and possibly already out-of-order, it complicates manually running the application because it requires to coordinate starting Kafka, starting three producers and starting the Flink application itself.

Apart from this, I'm pretty happy with how the project turned out, in particular, I think the clear separation of work in combination with the tests was a great choice which more than compensated for the time-consuming manual startup of the complete application.

Comments

As already mentioned in the midterm report, the debugging of the transparent handling of watermarks and timestamps for individual functions felt a bit annoying. I think it would be great for debugging if there would be a better way to track individual events in the pipeline and see when timers triggered. Additionally, the issues with the data were a bit troublesome.

For me, it was both surprising and nice to see that even though initially it took a bit to fully grasp the stream processing way of doing things, soon enough we were able to make a lot of progress. The lecture and the book helped me a lot in getting to this point. Additionally, it was great to see that our initial plan described in the design document was working nicely. All in all, I truly enjoyed the project, despite the fact that it was a lot of work. I think I learned a lot about Flink and stream processing in general while working on the project.