Data Stream Processing and Analytics

Mentor: Prof. Dr. Vasiliki Kalavri
Author: Svilen Stefanov 18-950-907

Final Report

31th May 2019

#1 High-level architecture and functionality of application

In this section I will first focus on the design decisions we made during the development process. Then I will discuss the functionality of the application and finally I will briefly talk about the structure of the code and the reasons for it.

Design decisions

As mentioned in the Midterm progress report, we mainly followed the design choices Dimitar Dimitrov and I introduced in our initial design document. I will therefore only talk about the changes and new design decisions that we made along the way:

We decided to decouple the 3 tasks completely so that each tasks consumes its data independently from the other tasks by specifying a unique group id in the properties for each tasks' KafkaConsumer. This was necessary because otherwise the data was being consumed in parallel from the streams of the different tasks. Thus, we couldn't compute accurate statistics since the streams did not see the whole data but only parts of it. Furthermore, we further decoupled task 1 and task 2 in three and two subtasks respectively. The three subtasks of task 1 were predefined in the semester project description. We introduced the subtasks of task 2 to be our two approaches to compute recommendations for users (see subsection Functionality of application).

Since we didn't have any experience with streaming applications before starting with the project most of the design decisions related to them were discussed and created quite late in the implementation phase where we could see arising problems first hand. This sometimes resulted in overwriting or partly changing code because some issues were well hidden and came up later in the testing phase.

Our team decided to use a MySQL database to retrieve information from the static data in our streaming application. This decision was mainly based on our previous experience with MySQL and the fact that it was the most straightforward way to extract the static data we use in our streams to compute dynamic similarity between users.

We did not end up using any Machine learning libraries for the unusual activity detection. Instead we compare the location of a user from the database with the location stamp of their post or comment. We consider an activity to be unusual if these originate from different continents.

Functionality of application

In this section, I will describe the functionality of our application with focus on the second and third tasks. The reasons for that are that task zero was already explained in the Midterm progress report and the functionality of task one was very well specified in the project description. More information about functionality of command line arguments and other parameters that we set up for convenience could be found in the readme of our project on gitlab.

Recommendations

Since we are the only team of 3 people, we decided to implement and compare two different approaches to recommend friends to our initially picked users (we selected them at random before starting with the implementation to avoid bias). The first approach is completely based on the similarity from the static data, whereas the second approach adds an additional dynamic similarity factor on top of the similarity from the static data. Both approaches entail the dynamic component of user activity in the last 4 hours (only active users from the last 4 hours are considered in the similarity computation). In both approaches we make sure that friends are not recommended to a given user by filling the similarity table with value of –1 for all his/her friends (including him-/herself).

Static similarity approach:

We compute this similarity as a linear combination of 5 factors – common interests, study place, work place, spoken languages and location. All of them are equally weighted and are used to compute the static similarity only once. After its initial computation, this static similarity is stored in a table in the database with similarities for each pair of users. To skip this computation, we have exported our databases for both the 1k and 10k datasets and they can be loaded using the provided import_database.sh script in the Database folder of our repository.

Dynamic similarity approach:

Initially, our plan was to compute the dynamic similarity based on the mutual interest of people to one and the same post. It turned out that a lot of these people are already friends and thus the dynamic similarity was very sparse. Therefore, as suggested by Vasia, we adopted a friends of friends approach that in addition to users that directly interacted with the same post also suggests their active friends. That means that for example if users A and B comment on the same post, and user C is a friend of B, user C would be a valid recommendation to user A, although it might have never interacted with a post where user A was active.

Unusual activity

For this task we compare the location timestamp of a given activity (comment or post) with the location from the static data for the user that initiated that activity. We use the table place_isPartOf_place to find the continent from which the activity was initiated and compare it with the continent where the user is located in according to the static data. If these do not match, we report this activity as unusual and store it in the final output file for this task.

Code structure – explanation and reasons

In our repository, we provide a readme with instructions on how to setup and run our application. Below, you will find a brief description of the packages and their use:

- **Config** – responsible for parsing the config.xml, as well as the static data and storing it in the database.
- **Database** – setup of the database, all SQL queries and a helper Graph class that enables us to compute a similarity measure based on common interests, are provided in this package. The Graph class is situated in this package since it converts information of 3 SQL tables in a different representation.
- **Model** – contains class representations of Likes, Posts, Comments and People from the forum. The EventInterface class is the parent class of the LikeEvent, PostEvent and CommentEvent classes (models the fields they share).
- **Schemas** – the classes in this package are used to serialize and deserialize the model objects (Likes, Posts and Comments).

- **Stream** – entails the classes we used to feed the data to the streams, define operations on the streams and store the results to .csv files.
- **Tasks** – provides classes for each of the subtasks and corresponding helper classes.

We tested our setup and the whole application on Linux, in particular the provided descriptions were tested on Ubuntu 16.04, Ubuntu 18.04 and Mint Sylvia.

#2 Individual contribution to the project

After creating the basic structure of the project together (the **config** and **model** packages – necessary for all the other parts of the project), we distributed the rest of the work. Throughout the course of the project Stevan and I worked on most of the things together since he was having some problems with his setup that he couldn't solve till the end. Initially the whole team was not very confident with Flink which is why we implemented task 0 multiple times – first Stevan and I implemented a first version and then Dimitar modified it slightly for the final design. From then Dimitar decided to implement task 1 and Stevan and I continued with task 2 and 3. Since we decided to implement two versions of task 2 and our second implementation incorporated concepts from both task 1 and our first implementation of task 2, this task was a team effort where Dimitar contributed with his experience from task 1 and Stevan and I with what we had developed for task 2. All important design decisions were taken together between all of us.

Below I will summarize my main contribution to the team and to the project:

In our team, I was mainly responsible for tasks 2 and 3. I was also the person that set up the SQL database and wrote the setup shell scripts. Apparently, I was the most experienced person from our team who has worked with git and was the person the other team members will turn to if problems came across in this regard.

From organizational point of view, I was also the person that organized the meetings in our team, was responsible for the distribution of the tasks and coordinated the work of our team.

#3 Reflection on the project

Despite the abrupt team change in the beginning of the semester, we managed to organize our work properly. In particular, in the second half of the semester we were meeting on a weekly basis to discuss current progress and each and every one of the team contributed with ideas and previous experience. We followed the timeline we proposed in the Midterm progress report and managed to finish everything in time and polish the code.

All in all, I am satisfied with our teamwork, with the gained knowledge and with the outcome of our project.

Improvement possibilities

In the following subsection I summarized a couple of ideas that I would change if we were to build the application again:

- **Design decisions**
  In general, I believe our design decisions make sense and fulfill the given tasks. However, there were some points we could not account for beforehand given our lack of experience with streaming applications and that slowed us down during the implementation of the project. This often influenced other parts of the code in a bad way and we ended up having a lot of static methods and variables that could be reduced. The way we query the database and the connections to it can also profit from a new design that accounts for all needs of our application.

- **Testing & code structure**
  I would probably suggest a test driven development approach in the very beginning of the project. We lost quite some time figuring out bugs that could have been easily detected by tests at the beginning. Leaving the tests for the end required some non-trivial last minute changes in task 1 (counting the replies) that could have been spotted earlier if we wrote the tests at the beginning. Appart from that we currently do not have additional topics for testing but use our original ones, which makes testing much more difficult than it could have been if we had designated test topics. Due to the time constaints and our different programming styles, our code structure ended up not as clean as I would want it to be but all in all, we managed to unify it and polish it at the end. Here, I would probably suggest to always have at least one code reviewer that ensures some predefined coding styles.
- **Optimizations - execution time**
  While doing performance tests for our application, we identified a bottleneck in task 2 when executing it for the first time. Currently the static similarities in task 2 are computed once for all users and then stored in the database for faster retrieval. We tried to optimize the computation as much as we could but the time to initialize the whole database with these similarities still takes quite some time the first time the database is created. Our present solution involves exporting our database and importing it before running the project as part of our setup.sh. Since this computation is executed only once, I believe this is not a major problem. However, a possible improvement would be to optimize or reduce the amount of SQL queries used for this computation.
- **Time management**
  Fortunately, we managed to finish everything in time and test our implementation but I wish we started working on the project a little bit earlier. The reason we did not start earlier or sometimes needed more time than planned was the learning schedule of fundamental concepts that we covered later in the course of the semester and that were needed for the project.

#4 Additional comments

In this section I present a short overview of some of the interesting findings, struggles that we had along the way, as well as a short feedback on the project:

Our initial assumption that task 3 will be the most difficult task, turned out to be incorrect since our current implementation is very simple while still realistic – this approach seems to be often used in practice. In comparison, task 1 was more troublesome than expected and required more testing.

We discussed and compared many possible implementations for the streaming computations like aggregations vs. windowProcessFunctions, single sliding windows vs. a combination of tumbling windows and sliding windows (our current approach) vs. ProcessFunction and many more.

Some of these discussions and findings might have sometimes been annoying or difficult to go around but we managed to have fun, learn a lot from them and accomplish our final goal. I am satisfied with the results, with our teams' effort and I would like to thank you for the opportunity to learn and grow personally and academically.