Systems and Architectures for Big Data project 2: Report

Simone D'Aniello
Data Science and Engineering
University of Rome Tor Vergata
daniello.simone.uni@gmail.com

Marius Dragos Ionita
Data Science and Engineering
University of Rome Tor Vergata
ionita.maryus@gmail.com

***Abstract*—Batch processing, despite being widely used in use cases such as financial services and life sciences, can't be the only way to analyze big data anymore. Today every application or sensor sends a continuous flux of data that has to be parsed in real time. This project report isn't intended to describe the importance of stream data processing but only with this awareness the reader can understand how much can be useful to know how to make framework cooperate. This paper has the aim to suggest general solutions to common problems.**

I. Introduction

For the occasion of DEBS 2016 Grand Challenge [4] a big dataset about a social network dynamic graph was provided. This data were intended to be used to calculate the next three queries:

1) Analyze the friendship relationships created in the social network in order to compute statistics about the time they are created. The output shows the number of relationships created each hour. Information must be aggregated to compute the same statistic per day, week and from the beginning of the gathering of data. The friendship relation can be unidirectional (the relation between A and B is sent once) or bidirectional (the same relation is sent twice). This must be considered during the implementation.
2) Compute in real time the ranking of the ten posts which have received more direct comments[1]. The output must be computed every hour and there must be a result for daily and weekly aggregated information.
3) Compute in real time the ranking of the ten more active users in the social network based on a score which must be computed from the sum of the friendships created by the user, the posts and the comments written by the user. Ranking must be computed every hour and there must be an output for the daily and weekly aggregated information.

In the second section of this paper architectural choices are discussed. In *Data Injection* is shown how data is taken from a file in order to simulate a real data stream situation. In *Queries* the implementation of each query is explained in depth while in data analysis the R [1] code is explained in order to validate the output. Performances are discussed in the section *Results*

[1]A direct comment is a comment written directly under a post. Comments which represent a reply to other comments are not considered

II. Choices

It is always hard to test a real time application. The dataset in input covers more than two years of information. Each query is developed in two different versions. The first version represents the fastest way to produce an output letting the processing engine reads data directly from the file. The second version simulates a real data stream processing situation.

Let's analyze this second solution. The data flow can be divided in four different steps. Data is read by Apache Kafka [2] from a file in the first part of the execution. This is probably the most common and used framework when talking about building real-time streaming data pipelines that reliably get data between systems or applications. Lines read are written into a Kafka topic. Now the second processing phase begins. Apache Flink [3] is used as processing engine to calculate real time computations. The output of this computation phase, that will be discussed in the section *Queries*, is written in a Kafka topic. The third phase is where the queries are really computed. In this computation no particular framework was used. The output is written in a csv file. In a first moment the project architecture comprehended the use of the Time Series database InlufxDB [5] but it is absent in the final version of the project due to the loss of performances. It's very interesting to see the section *Results* to analyze in depth the results comparison. For the communication between Kafka and Flink the Flink-Kafka connector was used.

For the first version of each query Apache Flink reads and processes data directly from the file. The data injection phase, that in the version previously analyzed was made using Apache Kafka, is no more needed. The analysis phase is done by Flink.

III. Data Injection

In this section the data injection phase is analyzed. This phase is intended to simulate the transmission of data from users.

Data in input is taken reading from three different files and each file has a different format. Apache Kafka is used to read data and send them into a topic. Flink instances, that can be deployed on different machines and receive data, listen to different topics, one for each query.

IV. Queries

In this section the implementation of the three queries is analyzed in depth.

The first query requests to analyze friendship relationships created in the social network in order to compute statistics every hour. This query is based on messages sent by users with the next format:

- timestamp
- user_1
- user_2

The first thing Flink does is to analyze the timestamp in order to extract information about hour of the day, day of the week, week of the year and year of the message. Number of occurrences are counted and every "virtual" hour Flink sends the result on a Kafka topic. This message contains the timestamp of the older message of the hour analyzed and the number of occurrences, the count. A *monitor* runs on a different machine and reads from the Kafka topic where results are written. This monitor keeps a static structure composed by the next fields:

- hour value
- day value
- week value
- lifetime value

The oldest timestamp arrived is stored for each hour (overwriting the older value). The monitor keeps the value of the hour, day, week and year related to the last message received.

When a new result arrives the value related to each field is increased by the count. When the message contains a timestamp with hour greater then the current one, the hour field is reset before the sum. When the message contains a timestamp with day greater the current one, both hour and day fields are reset before the sum, and so on. In the next table is shown this simple mechanism.

	hour	**day**	**week**	**lifetime**
2012-02-03T17:35:50	1	1	1	1
2012-02-03T17:50:12	2	2	2	2
:				
:				
2012-02-03T18:05:49	1	5	5	5
:				
:				
2012-02-04T07:33:33	1	1	30	30
:				
:				
2012-02-10T13:57:02	1	1	1	76

After the field reset, its old value is saved in a csv file together with the timestamp stored. If an incoming message presents a timestamp lower then the current one, the message is simply discarded.

There is no need to dwell too much on the description of the other version of this implementation. The only difference is that Apache Flink reads data from the file and computes the monitor operation by itself. This implementation reduces significantly execution time but hardly can simulate a real data stream processing application.

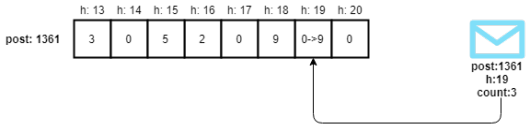

Fig. 1. Query 2: message into the window

The second query asks to find the ten posts which presents more direct comments. This posts needs to be ordered in order to create a ranking that has to be recomputed every hour. In the first part of this analysis the distributed version of the implementation is shown.

Flink receives message containing the next six fields:

- timestamp
- comment_id
- user_id
- comment
- user
- comment_replied
- post_commented

Computing the ranking of posts with more direct comments, messages containing a non-null *comment_replied* field are discarded, being *comment_replied* and *post_commented* alternative fields. Messages are grouped by post id (written in the field post_commented) and occurrences are counted. This count is written on a Kafka topic every "virtual hour".

During a simulation like this, Apache Kafka can send years of data in few minutes. Flink instances work in parallel and can compute data at different rates, sending results out of order. Being in a more complex situation then the one analyzed in the first query, the monitor has to be more complex too. Computing a ranking can seem an easy job for a software but when talking about big data some problems may occur. In order to analyze what is the post that has received more comments, all ids must be banally stored. Keeping constant the number of fields needed for the computation, as was done for the first query, is impossible.

The monitor maintains a dynamic list of posts encountered. Each item of this list has a sliding window initially filled with zero values. The size of this window is decided by the programmer (the default value is 24). When the first packet arrives, the left boundary of the sliding window is set to the packet timestamp value. Next positions represents different hours in sequence. As you can see from rig.2, when a message arrives from Flink, the value is positioned in the correct position. Now it is clear that the sliding window is needed to manage out of order packets, where the maximum out of order timestamp depends on the sliding window size But there is a reason why is is called sliding window. Every time a message arrives it is only stored in the right position. When a message exceeds the right boundary, the timestamp of

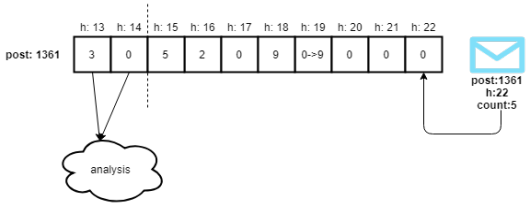

Fig. 2. Query 2: message out of the window

this message become the new right boundary and the window slides to right. The fields of the sliding window removed for each post, form a column. Before being deleted, the ranking is computed on this columns and the ten posts with more direct comments are written on the file (each column becomes a row on the csv file).

when reading from file, a single Flink instance will send ordered results to monitor. This means that in a non distributed situation, the sliding window size can be reduced to one. In this case each post maintains a similar structure to the one examined for the first query The third query asks to find the ranking of the ten more active users on the platform. this statistic has to be computed every hour and results must be aggregated per day and week. The first thing that has to be done is to call a function that merge the three files in input and write a single csv with lines ordered per timestamp. This is important to simulate the message arrival from users in a social network. Every line in the file is parsed to recognize if Flink is reading a friendship relationship, a comment or a post. Every message (or file line) is mapped in a tuple containing the *user_id*, a *timestamp* and the *username*. Flink groups data in input by the id value and sends the aggregate value to a topic. Monitor reads results from this topic and starts a computation phase similar to the one described in the second query. The only difference between first query monitor and second query monitor implementations is that the first computes the ranking based on the post id while the second uses user id (but the logic is the same).

V. Data Analysis

Data are available in three different text files *friendships.dat, posts.dat* and *comments.dat*. In order to guarantee input data integrity and consistency every dataset file was pre-processed with R language, in particular using Rstudio open-source framework.

During pre-processing phase it was checked if fields number of each row was the same of the one specified in the assignment, if event order was consistent and if every tuple type was of the right type. Talking about the first query, there is a strong assumption we've made: friendship relationships can't be count twice for definition, because a relationship is created when a user accepts other user's request. This assumption is the result of an analysis done on the rows of the file *friendship.dat*.

Planning choice for query 3 was to consider a unique source file unifying all available sources and ordering if for timestamp. The reason is to mostly simulate real data stream where tuple events about friendship, post or comment could be received to the system in every moment with partially ordered timestamps.

Rstudio was used at the end of the computation in order to control results correctness, particularly to check number of read tuples, comments or posts for every query.

VI. Testing

In this section is analyzed every test relative to each section of this application.

During the pre-processing phase data the three input files are merged and the result of this computation becomes the input for third query. This phase is not considered during the benchmark analysis because it is not closely related to the application. Its computation is also very fast and must be computed once.

During data injection phase Apache Kafka is used to read data from a file and send them into a Kafka topic.The following is the size of each file:

- friendship.dat - 63409 packets - 3.41 MB
- comments.dat - 742178 packets - 85.5 MB
- query3.txt - 1240712 packets - 126.2 MB

Next table shows the time spent in order to read all this data and send them to a Kafka topic. During this benchmark no consumer is reading this data.

	Query 1	**Query 2**	**Query 3**
Kafka Time	15s	6m 10s	5m 35s
Kafka Throughput (pk/s)	4227	2005	3703

In processing computation Flink reads data from a Kafka topic, computes the count as previously described in the section *Queries* and produces the output. In a real situation, output is written every hour. In this simulation the write rate depends on the speed of Flink and Kafka processing. The following is a table containing Flink processing time. This time contains the sum of time spent by Kafka in order to write packets on this topic and the time spent by Flink to process data and send them into another topic.

	Query 1	**Query 2**	**Query 3**
Flink Time	1m 10s	9m 35s	12m 55s
Kafka Time (s)	15s	6m 10s	5m 35s
Flink Throughput (pkts/s)	745	1290	1600
Flink overhead (ms/pkts)	0.8	0.27	0.35

When letting Flink read from a file, avoiding data injection phase, times significantly decrease. In the following table is written the time spent by Flink in order to read data from the file, process them and compute the query.

	Query 1	**Query 2**	**Query 3**
Flink Time (s)	10	100	30
Flink Throughput (pkts/s)	0	1290	1600

In a first moment it was considered to store results in a time series database as InfluxDB. In the last version of the implementation it was removed due its overhead. For example, looking at the computation of the first query, the simple processing with Flink reading from the file, takes around fifteen seconds. Using InfluxDB the time spent is around thirty five seconds.

Each query has been executed around fifty times through days. The results written represent an average of this values.

VII. Conclusions

Some architecture aspects and structures used in this project may seem too complex to reply to simple requested queries. Complexity is due to two different factors: first one is represented by project robustness. This system is able to manage until 24 hours out of order messages , which is a difficult event in real time stream processing situation. Second factor is due to the fact that data are read from an existing dataset represented by a text file and not from real users. For this fact the computation depends on event time and not on processing time as it should happen in a real context. Solutions shown in this project report are not intended to be the fastest but have the aim to simulate a real environment. As a matter of fact despite project implementation contains functions that can produce a result in a faster way, the core of application is of course the interaction between Apache Kafka and Apache Flink components which together represent a comun architecture in a real time stream processing context.

References

[1] RStudio. 2016. [ONLINE] Available at: https://www.rstudio.com/.

[2] Apache Kafka. 2011. [ONLINE] Available at: http://kafka.apache.org/ .

[3] Apache Flink [ONLINE] Available at: https://flink.apache.org/ .

[4] DEBS 2016 Grand Challenge: Small sample set. [2018]. https://www.dropbox.com/s/ vo83ohrgcgfqq27/data.tar.bz2 .

[5] InfluxDB. 2013. [ONLINE] Available at: https://www.influxdata.com/.