Social network data stream processing and real-time analytics with Apache Flink, Storm and Kafka Streams

Data stream processing project for "Systems and Architectures for Big Data"
AY 2017/2018

Ovidiu Daniel Barba
ovi.daniel.b@gmail.com

Laura Trivelloni
laura.trivelloni@gmail.com

Emanuele Vannacci
emanuele.vannacci@gmail.com

ABSTRACT

This paper provides a description of the second project for the *Systems and Architectures for Big Data* course at the University of Rome Tor Vergata: the design and implementation of a scalable distributed system able to perform specific queries on the dataset provided by the *ACM DEBS Grand Challenge 2016*, comparing the performance of various frameworks for data stream processing like *Apache Flink*, *Apache Storm* and *Apache Kafka Streams*.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous—*data stream processing*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

distributed systems, big data, real-time analysis, Kafka Streams, Storm, Flink, Docker

1. INTRODUCTION

The goal of the application is to provide real-time statistics and rankings computed by three specific queries working on streaming data originating from a social network, like relationships among users, comments and posts. The input data is injected into a distributed publish/subscribe system and later consumed and processed by various data stream processing (DSP) frameworks. All the queries must output statistics related to event-time windows with different size. The final results are first published on an output publish/subscribe system and later retrieved and stored on a in-memory key-value NoSQL database.

2. SYSTEM ARCHITECTURE

The application is structured as a four-layer system, as shown in Figure 1, with their relative purpose:

University of Rome Tor Vergata '18 Rome, ITALY

- Data ingestion layer: filtering and redirecting incoming data into the processing layer
- Data processing layer: stream analytics of published data. Composed by three different sub-systems of data stream processing
- Data storage layer: final results persistence
- Decoupling layer: published results before persisting them

The overview of the adopted frameworks related to the high level architecture described are shown in Figure 2 and analyzed below.

2.1 Source system

The source system that provides data to the application is represented by a social network where users can write posts, comments and make friendship relationships among themselves. It is not a data producer with constant frequency of emission so data can arrive with high variance and different loads.

Each friendship between two users is described by:

- `timestamp`: when the friendship was estabilished
- `user_id_1`: first node of the relationship
- `user_id_2`: second node of the relationship

The same friendship information can be produced twice because the relation can be bidirectional.

Each post written in the social network is described by:

- `timestamp`: when the post was published
- `post_id`: unique post identifier
- `user_id`: unique user identifier

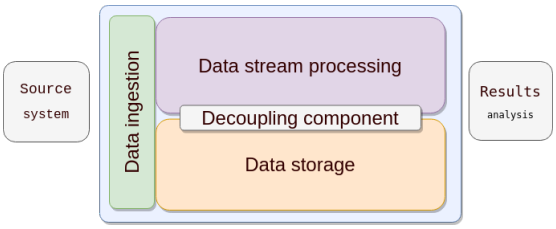

Figure 1: The System structure

- `post`: content
- `user`: author

Each comment written in the social network is described by:

- `timestamp`: when the comment was published
- `comment_id`: unique comment identifier
- `user_id`: unique user identifier
- `comment`: content
- `user`: author
- `comment_replied`: commented comment identifier (if any)
- `post_commented`: commented post identifier (if any)

2.2 Avro external data producer

We used a client application to ingest data into the System to simulate data arrival. We implement it with the possibility to vary frequency of data generation, mainly for testing needs. The frequency is proportional to the inter-arrival time among next tuples. The reference time is assumed to be the timestamp indicated in the `ts` field in the records. We choose Apache Avro as serialization format both for input and results data.

2.3 Data ingestion and decoupling layers

Both for publish and subscribe to streams of records in the system is used Apache Kafka framework:

- input topics: external data producer publish data on the related topic
- output topics: Kafka, Flink, Storm publish on those and make results available for storing or visualizing once consumed

2.4 Data storage layer

Redis is an in-memory data structure store, used as a database, cache and message broker. We used this framework to store results data published from the processing layer on the output Kafka queue.

2.5 Data processing layer

This layer has three different sub-systems, each one with a data stream processing framework dedicated to specific queries:

- *Apache Flink* for all three queries
- *Apache Storm* for Query 2
- *Apache Kafka Streams* for Query 1

3. FRIENDSHIP ANALYSIS

The aim of the first query is to analyze the friendship relations to derive statistics on the time slot in which these relations are created. Statistics are computed every 24 hours, 7 days of event time and since the start of the social network. The query is answered using both **Apache Flink** and Kafka Streams. We paid particular attention to bidirectional friendships in the dataset in order to do not account at duplicate relations.

3.1 Kafka Streams

Kafka Streams is a client library for Kafka servers to build real-time applications integrated perfectly with Kafka technology. This library offers Streams API and Processor API to make available, respectively, higher and lower level of programming. The main phases of implementation of the first query are described below:

- `Consuming` from Kafka topic where input data are published
- `Filtering` of incoming stream to discard those tuples that refer to the same relationship, avoiding to be counted more than once:
  - select a combination of the first user id read from record and the second user id as key for all tuple
  - grouping by key to have the same relationship on the same operator
  - reduce discarding all but once values represented by friendship confirmation timestamp
- **Daily statistics**:
  - select timestamp as key and counter initialized at 1
  - group tuple by key and then make tumbling windows of 24 hours to collect tuple referring to that slot of time; the reference time is imposed as the timestamp included in the record instead of the default tuple arrival time
  - count all tuple referring to the same window
  - composed results setting as key the initial statistics timestamp given by window start and as value an array of 24 elements, one per hour, to maintain the related counters
  - grouping by key and reduce summing all counters at the same index in the array to obtain total count per hour
- **Weekly statistics**:
  - starting from the results stream obtained from the sum for 24 hours, we group by the key to make new windows of size 7 days with sliding of 24 hours to compute statistics updated every 24 hours
  - reduce again summing same array indeces to count per hour
  - select as key the window start to compose the format results
- **From beginning statistics**: this requires the use of Processing API to allow the definition of a State Store to keep aggregated data from the beginning of processing time:
  - the results streams coming from 7-days-sums is composed as a new couple key-value, mapping the value in an array of 25 positions to make possible saving of the first timestamp and the 24 counters too

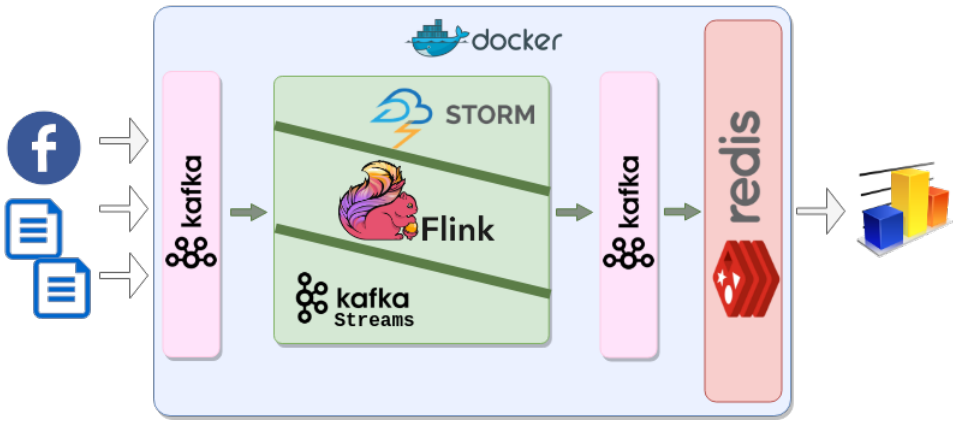

Figure 2: The System architecture overview

– transform operation implemented in the **FromBeginningCounterTransformer** class to change input stream in an output stream updating the state store at each incoming tuple: sum each counter value index-by-index and keep the minimum timestamp to get the statistics start time

- `Publishing` of results streams on output topics

3.2 Apache Flink

We exploited **Apache Flink DataStream API** to read the input from Apache Kafka, apply transformations and to consume the data stream into the output source. Flink windows are used to split the stream into buckets of finite size over which we derive time based statistics. Some stateful functions and operators are involved in order to store data across the processing of individual events. The main phases are described below:

- **Filtering**: The input stream is keyed with the `user_id_1` and `user_id_2` fields sorting them in descendent order. The filtering operator keep a state for each key and it is able to remove duplicates caused by bidirectional friendships not forwarding them in the output stream.

- **Event time assignment**: We assign event time based on the embedded field `timestamp` in order to make the progress of time dependent on the data.

- **Individual daily statistics**: The input is keyed on the `user_id_1` field before defining the window, splitting the infinite input stream into logical keyed streams. Having a keyed stream allows the windowed computation to be performed in parallel over multiple tasks. A window is evaluated on each key and an aggregation operator is in charge of compute statistics. Two functions are involved within the operator: An *AggregateFunction* exploits an accumulator to count the number of friendships in each hourly slot, incrementally aggregating input elements of a window as they arrive; Its output is forwaded to a *ProcessWindowFunction* that gets an iterable object containing all the elements in the window; *ProcessWindowFunction* is involved because it allows to access a context object providing useful information about the current window. In particular it is used to place the time at which the window start as the statistics starting point.

- **Aggregate daily statistics**: A *ReduceFunction* takes the stream and compute the global statistics as sum of the individual daily statistics.

- **Weekly statistics**: The statistics based on 7 days of event time are computed starting by the individual daily statistics. The input data stream structured as a timestamp and an array of counters is aggregated into a keyed sliding window that emits a tuple for each key. The output stream goes through a reduce operator that compute the final weekly count.

- **Global statistics**: Since the weekly statistics are computed on a sliding window with emission frequency of one day, we cannot compute global statistics by the weekly ones. Sliding windows overlap and so we would count the same events over and over again. Therefore the global statistics are computed from the daily one. A stateful operator keeps the counters of friendships per time slot explicitly managing checkpoint and the state initialization.

Then at each output streams is finally attached a sink to consume and forward them to a topic of Apache Kafka.

4. RANKING QUERIES

The goal of Queries 2 and 3 is the real-time top-10 ranking of posts and active users in the social network. In particular Query 2 ranks the posts by the number of received comments while Query 3 the most active users. Each user U has a score composed of the triple *(a,b,c)* where:

- a = the number of friendships created by U
- b = the number posts written by U
- c = the number of comments written by U

A user U_1 is more active than U_2 if $a_1 + b_1 + c_1 > a_2 + b_2 + c_2$. Both queries must provide rankings in hourly, daily and weekly event-time windows. Since they have the same goal, a general ranking query structure has been defined and later

applied to the specific query and related data stream(s). This has been made possible with the development of specific ranking data structures.

4.1 Ranking Data Structures

To be able to rank elements, one needs to provide a score of a specific element in order to compare it with others. Since the two queries work on different scores (a simple count in Query 2 and a triple of counts in Query 3), a more general *Score* has been defined to accomodate both needs. Score is a Scala `trait` (similar to Java's `interface`) that a concrete score class needs to implements. It defines two simple methods:

- `add(other:Score):Score` adds this score with other and returns a new one
- `score():Int` returns a "total" score used to compare it with others

The items to be ranked are of class `GenericRankElement[A](id:A, score:Score)` where `A` is the (generic) type of the element's *ID* (in our case IDs are *Long*).So the rank elements holds its ID and its relative previously defined *Score* .

4.1.1 Ranking Board

The ranking's core data structure is the `GenericRankingBoard[A](k:Int)` where `A` defines the rank element's ID type and k the maximum number of elements to be ranked (default is 10). The *Board* maintains the current score of ranked elements (in a *HashMap*) and caches and updates the top-K elements (using a *ListBuffer* of size K) so to retrieve it in constant time. It provides methods for updating an element's score, clearing the entire board, return the top-K and check whether the ranking has changed since its previous check. Every time the score s is incremented, the board checks if the top-K also needs to be updated. If s is less than the top-K's minimum score and the buffer is full, the rank is not changed.Otherwise the new element is inserted (after removing itself if already present), and a new minimum is computed and removed from the top-K until it reaches size K.

4.1.2 Ranking Result

Once the top-K has been returned from the *Board*, a rank must be assigned a timestamp from which the ranking computing has begun. This is done in the `GenericRankingResult[A]` with *elements* as the rank and k the maximum number of ranked elements. Since the ranking result r_1 can be a partial ranking, it has a method of merging it with another ranking r_2 resulting in another ranking r_3 with:

- $timestamp(r_3) = min(timestamp(r_1), timestamp(r_2))$
- $K(r_3) = min(K(r_1), K(r_2))$
- $rank(r_3) = (rank(r_1) + rank(r_2)).top(K(r_3))$

The merging will be useful when computing global ranking. It also exposed a way to *incrementally merge* ranking results from non-overlapping windows and return a new ranking with the characteristics mentions before except for the final ranking list. In this case, the scores of elements with same ID are summed together while in the previous case only the maximum was taken and the other ones discarded.

4.1.3 Global Ranking Holder

It is a data structure used for holding and updating global ranks. The ranks are kept in a *TreeMap* ordered by timestamp. The *TreeMap* can be cleared of rankings older than *delta* milliseconds (specified when the holder is created) from the timestamp passed as parameter. It also offers a method to compute global ranking by passing it a partial ranking. It does so by checking any existing partial ranking and merging it with the new one (also updating it in the map).

4.2 Ranking with Apache Flink

We first describe a general approach for solving the two ranking queries and later dive into more detail about each specific one.

4.2.1 General Approach

```
<keyed-stream>
.window(<Tumbling or Sliding Event Time Window>)
.aggregate(<ScoreAggregator>, PartialRanker)
.setParallelism(< gte 1 >)
.process(GlobalRanker)
.setParallelism(1)
```

The elements in <> are the ones that can change for a specific query or for a specific time windowed statistic. Given a keyed stream of tuples *(ID, Score)*, create a specific event time window (event timestamp are specified previously by taking the timestamp in the input data), incrementally aggregate (sum) scores of specific keys (post IDs or user IDs depending on the query) and pass them to the PartialRanker that keeps and updates (in a *RankingBoard*) the them. It is also responsible for emitting *RankingResults* only if the current partial ranking has changed. The *RankingResults* have as timestamp the start (saved and updated as the windows advance) of the current event-time window. When the window advances it also clears the *board*, enabling it to store data only of the new window.Note that the parallelism of the *PartialRanker* can be greater than 1. This is not true for the *GlobalRanker*, whose job is to merge partial ranks from previous (along the computation flow) parallelised operator. It manages the global ranking emission (in addition has a simple previously sent duplicate detection) and clearing as the *PartialRanker* but it keeps a *GlobalRankingHolder* that stores and computes global ranking (of current window) with older partial ones. The described method work if you only need to compute rankings for a specific window only, e.g. *SlidingEventTimeWindow.of(Time.hours(24),Time.hours(1)*. If you need to extend it and compute rankings of larger windows using results from previous smaller windows, see next sections.

4.2.2 Query 2

```
commentsStream
 .flatMap {parseComment()  filter isPostComment()}
 .assignTimestamp(<post timestamp>)
 .map( => (postID, SimpleScore(1) )
 .keyBy(postID)
 .<general approach ranking>
```

The query simply parses the comment, filters away the comments related to other comments, assings it a timestamp to get in the correct window(s), transform it into a tuple with

postID and score of 1, keys the stream by the postID, windows all event starts the ranking.

4.2.3 Query 3

This query uses 3 different data streams to assign each user a score and rank them. for every incoming data in every streams, extracts and assigns a timestamp, extracts the userID and depending on the stream type assign a score to the user.E.g., the comments stream is initially processed like this:

```
commentsStream.
.assignTimestamp(extractTimeStamp())
.map(=> (userIDFromComment(), UserScore(0,0,1)))
```

Note that every user is assigned a *Score* (continuously aggregated in the specific window) which will allow its ranking in *Ranking Board* and other ranking data structures. Later all three streams are united using the *union* operator and the previous general ranking approach is applied.

```
postsData.union(commentsData, friendshipData)
  .keyBy(userID)
  .window(<EventTimeWindow>)
  .aggregate( UserScoreAggregator, PartialRanker)
  .process( GlobalRanker)
```

4.2.4 Tumbling and Sliding Event-Time Windows

Both ranking queries have been implemented in two different ways,each using one of the two modes of event-time windows : *Tumbling* and *Sliding*. With the **Tumbling** ones, the query becomes as following:

```
hourlyRankings =
<keyed-stream>
.window(TumblingWindow.of(Time.hours(1)))
.aggregate(<ScoreAggregator>, PartialRanker)
.process(GlobalRanker)

dailyRankings =
hourlyRankings
.assignTimestamp(<partial ranking timestamp>)
.windowAll(TumblingWindows.of(Time.hours(24)))
.process(IncrementalRankMerger)

weeklyRankings =
dailyRankings
.assignTimestamp(<partial ranking timestamp>)
.windowAll(TumblingWindows.of(Time.days(7)))
.process(IncrementalRankMerger)
```

The hourly results are computed using the general approach, while the daily and weekly by windowingAll previous rankings and incrementally merging them to obtain the desired final ranking of the specific window size. It was useless using the general approach to compute statistics with large window sizes since they only depend on final results of previous windows so a less resource consuming approach was used. With this approach it was difficult to use sliding windows since the partial rankings arrived in overlapped windows, creating uncertainty and leading to less precision when computing global rankings on larger sized windows.

If one wants to use *Sliding* windows to have a more updated view of the ranking (by decreasing the window slide period), we implemented the ranking by redirecting the input stream into three sub-queries using the general approach. E.g., to compute the weekly rankings:

```
weeklyRankings =
<keyed-stream>
.window(SlidingWindow.of(Time.days(7),<sliding>))
.aggregate(<ScoreAggregator>, PartialRanker)
.setParallelism(< gte 1 >)
.process(GlobalRanker)
.setParallelism(1)
```

In this way you have a more "recent" view of the ranking but input data replication is very high and leads to computational inefficiency.

4.3 Ranking with Apache Storm

A Storm topology is defined in order to answer the Query two. The main components that have been involved and the way in which they are connected are described below. A source stream is created starting from a single *Spout* in charge of continuously reading record in Avro format from a Kafka topic and forwarding them into the next components. Through a *shuffle grouping* tuples are randomly distribuited across all tasks of the **Parser**. It receives tuples as an array of byte and deserializes them to extract the fields to include in a new value to send to the **Filtering** bolt. It emits only tuples relative to a comment to a post and discards those concerning a comment to a comment.

The **Metronome** is a single task bolt that receives all the tuples from the filtering step. Its task is to emit tuple with a fixed emission frequency based on the event time. It looks at the timestamp embedded in the received record and compares it with the current time which is kept in its state. If the difference between the extracted timestamp and its current time is large enough then it emits a tuple in a new special declared stream. These tuples are sent to the next bolts in order to communicate them that is time to emit output. Since the subsequent components must receive all the emitted tuples from the Metronome in order to concurrently produce the output and have the same notion of time, the Metronome is connected with the other bolts with an *all grouping* that replicates the stream across all the bolts's task.

The core components of the Storm topology is the **WindowCountBolt**. It receives tuple from the Filtering bolt through a *field grouping* on the `post_commented` field because it is replicated across multiple tasks able to perform parallel computation. The WindowCountBolt keeps for each tuple it receives a slotted window structure. The bolt is parameterized with two arguments: the slide interval and the window size. The first defines the output emission frequency that is triggered by the tuples received from the Metronome's stream. The second parameter indicates the time interval on which consider the statistics computed on the seen event. Therefore the bolt keeps a window structure for each post it sees and in the window compute the count of comment of the relative post. The WindowCountBolt performs parallel computation on tumbling window of one hour, one day and one week. The daily statistics are computed by aggregating the hourly ones and the weekly statistics by the daily ones. The output is structured with a statistics start

date given by the window start, a field that contains the `post_id` and a integer value counting the comments to the post.

The windowed bolts are connected to multiple **PartialRank** bolts to compute an intermediate ranking. This component exploits services made available by the *RankBoard* object. The output is a partial rank that is aggregate by a single task of the **GlobalRank** bolt. This bolt builds a *RankingResult* data structure keeping the top-K element and merge it with other intermediate results over the time to keep an always updated ranking. A **Collector** bolt is the final component that receives all the output data and collect them.

5. CONCLUSIONS

In conclusions, we implemented a distributed data stream processing system composed by three subsystems to use different DPS frameworks as experimental goal.

For a better means of comparison among data stream processing framework used, it will be interesting to implement all queries for each framework and compare performance.