Grand Challenge: Real-time Analysis of Social Networks Leveraging the Flink Framework

Giacomo Marciani
giacomo.marciani@gmail.com

Marco Piu
pyumarco@gmail.com

Michele Porretta
micheleporretta@gmail.com

Matteo Nardelli
nardelli@ing.uniroma2.it

Valeria Cardellini
cardellini@ing.uniroma2.it

Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy

ABSTRACT

In this paper, we present a solution to the DEBS 2016 Grand Challenge that leverages Apache Flink, an open source platform for distributed stream and batch processing. We design the system architecture focusing on the exploitation of parallelism and memory efficiency so to enable an effective processing of high volume data streams on a distributed infrastructure. Our solution to the first query relies on a distributed and fine-grain approach for updating the post scores and determining partial ranks, which are then merged into a single final rank. Furthermore, changes in the final rank are identified so to update the output only if needed. The second query efficiently represents in-memory the evolving social graph and uses a customized Bron-Kerbosch algorithm to identify the largest communities active on a topic. We leverage on an in-memory caching system to keep the largest connected components which have been previously identified by the algorithm, thus saving computational time.

The experimental results show that, on a portion of the dataset large half that provided for the Grand Challenge, our system can process up to 400 tuples/s with an average latency of 2.5 ms for the first query, and up to 370 tuples/s with an average latency of 2.7 ms for the second query.

CCS Concepts

•Information systems → Data streams;

Keywords

Social-network graphs, Real-time data processing, Flink

1. INTRODUCTION

Social network analysis aims at discovering and investigating social phenomena and trends through the exploitation of methodologies and tools. With the spread diffusion of social networks, scientific communities from different fields and commercial companies have investigated how to effectively extract valuable information from the great amount of daily produced data. An emerging activity, which is gaining strategic value, is the real-time identification of communities within social networks that mostly stimulate the interest and interaction among their users. Indeed, companies often rely on these communities to conduct advertising campaigns with focused targets. The diffusion of on-demand computing resources (i.e., Cloud computing) and the consolidation of enabling technologies for real-time analytics (e.g., stream processing) have enriched decision-making processes that can now be supported by data, which, for example, can be gathered from communities, user reactions, and trending topics. Differently from other contexts, the real-time analysis of social networks is challenging, because it requires to handle an evolving social structure. Moreover, this structure is usually represented by graphs, which, per se, may require to apply algorithms with not negligible complexity. The sixth edition of the DEBS Grand Challenge [4] focuses on these problems and calls for applications that provide real-time analysis of an evolving social-network graph. Specifically, an application should collect streams of social events, such as friendships, posts, comments, likes, so to (1) determine the posts that currently trigger the most activity in the social network, and (2) identify large communities that are currently involved in a topic.

In this paper, we present our efficient and easily tunable solution to the Grand Challenge. It can run on a single node as requested for the Grand Challenge evaluation, but we design its architecture to be executed in a distributed environment. To this end, we rely on Apache Flink [3], an emerging open source and scalable data stream processing framework.

The paper is organized as follow. In Section 2 we present the design approach of our solution and the topologies that address the Grand Challenge queries. In Section 3 we present some performance results. Finally, we conclude in Section 4 identifying some further improvements for future work.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

DEBS '16 June 20-24, 2016, Irvine, CA, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4021-2/16/06.
DOI: http://dx.doi.org/10.1145/2933267.2933517

2. GRAND CHALLENGE SOLUTION

The DEBS 2016 Grand Challenge poses two queries that require to efficiently handle several entities interacting each other (i.e., posts and comments) with the goal of discovering some complex dynamics which strongly depend on the passing of time. The design approach of our application revolves around two principles: parallelization and memory ef-

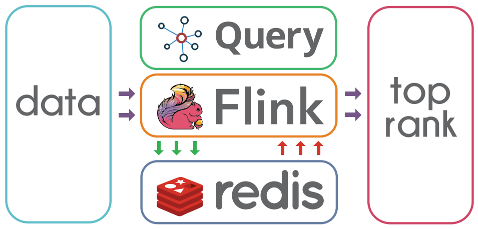

Figure 1: High-level architecture of our solution.

ficiency. The application incorporates two independent solutions that answer the two queries. Aside the specific details, both the solutions rely on a sequence of possibly parallel operators that apply stepwise transformations to the incoming events. Specifically, the application: (1) reads the events from the dataset stored on a file and pushes them within the system; (2) computes the score associated to each relevant entity (i.e., post, comments); (3) on the basis of this score, determines the top-k rankings with a two-step approach (being k a query-related parameter); and (4) emits the updated top-k rankings. The meaning of score, and the way it is computed, changes according to the query. For the first query, each post receives a score resulting from all its comments; the score provided by each comment and the score of the post itself slowly decay with the passing of time. For the second query, each comment receives a score depending on the size of the community that interacts with it within a given time window. Due to the great amount of data, computing the score represents the critical operation; therefore, we exploit parallelism to concurrently determine the score by working on independent partitions of the incoming events.

To focus our effort on the application logic, we need a high-throughput and scalable stream processing framework. After having examined several alternatives, such as Storm[1] and Spark[2], we have chosen Flink [3], an open source project maintained by the Apache Software Foundation, because it shows promising performance with respect to other well-known frameworks. Our application leverages some advanced features provided by Flink; the most relevant one is the feedback stream, i.e., a stream towards upstream operators, that we use to optimize the usage of memory by deleting expired posts which cannot compete for the top-k ranking ones. To answer the second query, the application also requires to efficiently store the social graph, which represents the users and their friendship relations, in order to periodically compute and retrieve the largest communities. We solve this problem through Redis [5], an in-memory data structure store, which avoids the bottleneck given by mass storage I/O. Figure 1 depicts a high-level overview of our solution.

2.1 Query 1

The goal of the first query is to determine the updates of the top-3 (i.e., $k = 3$) most influential posts in the social network, that is, those that maintain over time a high rate of interaction via comments. The scoring discipline encapsulate this concept, and is computed as follows. When a post/comment is created, its initial score is set to 10 and is then decreased by 1 once a day. The post score is the sum of its value plus the score of every direct and indirect comment rooted in it. Once the score reaches zero, the post expires and does not compete for any ranking. We now describe the operators involved in the first query, whose topology is shown in Figure 2.

The *Event Dispatcher* is a centralized operator that reads in parallel posts and comments from several data sources. It converts each entry in a tuple containing only the fields that are strictly necessary to the following operators. Taking advantage of the ascending events timestamp, it can efficiently carry out the interlacing of events. Afterwards, the *Comment Mapper* maps each comment to the related post. Observe that this operator also assigns indirect comments (i.e., comments of a post's comment) to the related post by maintaining in-memory a mapping table that links each post with its comments. Storing this mapping table is challenging because it can potentially grow unlimited, thus saturating the memory. To handle this situation, we exploit the feedback mechanism, which allows us to store only the mapping of not yet expired posts. The *Post Score Updater* is a parallel operator that receives the streams of posts and comments and updates the post score and number of commenters of the post. To preserve the application integrity with parallelism, the incoming streams are partitioned by the post identifier. To synchronize the parallel instances of this operator, so to properly update the scores, the upstream operator *Comment Mapper* broadcasts a *time sync* message when the time associated to posts and comments moves forward. The *Aggregator* is a parallel rolling counting operator that merges the single score updates to produce the total score for each post. Moreover, this operator is in charge of emitting the *Feedback Stream*. The *Post Rank* defines the partial top-3 ranking of posts handled by its upstreams operators; the ranking is sorted by the post score. Then, the *Post Rank Merger* merges all the partial rankings into a global one and identifies the top-3 posts that trigger the most activity in the social network. The latest two operators optimize the sorting operations by (1) reducing the elements to be sorted thanks to the parallelization; (2) pruning the expired posts; and (3) avoiding to sort the elements that cannot actually generate a ranking update (i.e., elements whose score is less than the lowest in the rankings). Finally, the *Post Rank Filter* produces an updated result every time the top-3 most active posts change.

2.2 Query 2

The second query focuses on identifying the k recent comments that are supported by the larger communities. The value of k is provided as parameter, whereas a community is defined as the set of users, friends each other, who have liked that comment. The topology of our solution is represented in Figure 3. The *Event Dispatcher* is a centralized operator that reads in parallel comments, like events, and friendship events from several data sources and, preserving their timestamp order, sends them downstream. The *Friendships Operator* receives the friendship events and updates the social graph, connecting the users that establish a new friend relation. The social graph is stored into Redis, where it is represented with adjacency lists indexed by the user identifier. This representation allows us to efficiently retrieve all the user's friends, that are needed to compute the largest community supporting a comment. The *Comment Score Updater* receives comments and like events, and computes the score associated to each comment as follows.

[1]http://storm.apache.org/
[2]http://spark.apache.org/

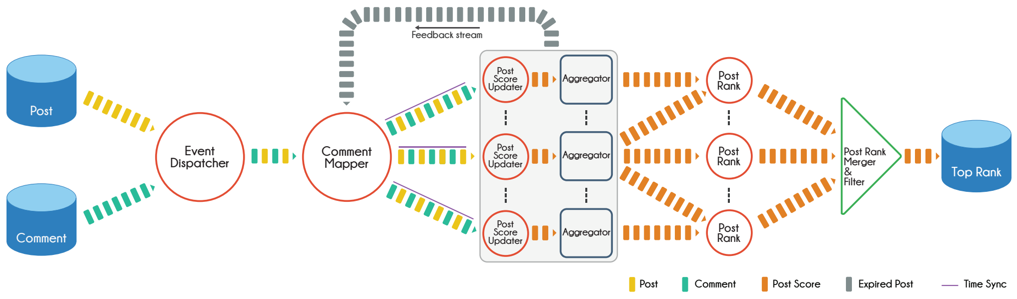

Figure 2: The topology for Query 1.

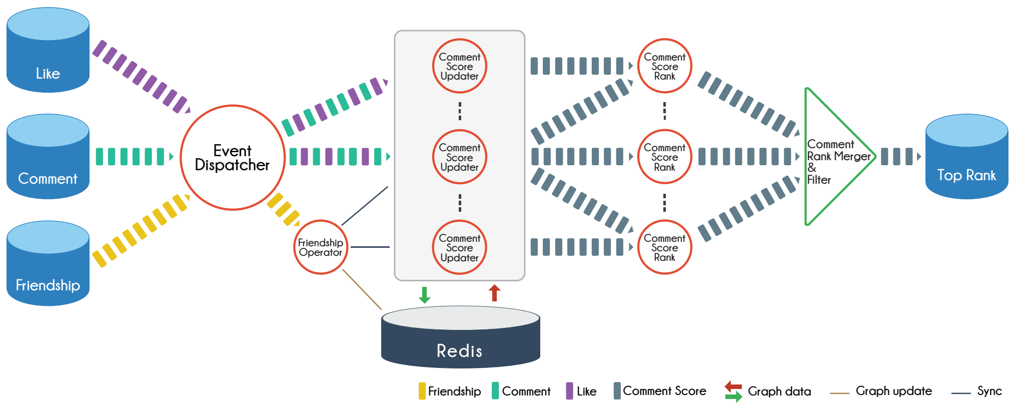

Figure 3: The topology for Query 2.

From each comment that was created not more than d seconds ago, where d is an input parameter, the operator extracts the set of users U that have liked it. For each user $u \in U$, the *Comment Score Updater* creates a user-based social graph G_u, containing the user u, his/her friends, and the friends of his/her friends; in G_u the users are interconnected with respect to their friendships. Afterwards, for each user-based social graph $G_u, \forall u \in U$, the operator runs a customized version of the well-known and widely used Bron-Kerbosch algorithm [2] to identify the largest clique C_u associated to each user. Observe that, at this point, the computed cliques depend only on the user-based social graph (i.e., his/her friendships) and also include users who have not liked the comment. Therefore, *Comment Score Updater* removes from each clique C_u the users who have not liked the comment, thus identifying the largest one as the community that supports the comment. Determining the cliques within a graph is an NP-hard problem, so we adopted a lazy approach that executes the Bron-Kerbosch algorithm (1) on subgraphs of the social network that, relying on friendships and being independent from like events, should change slowly; and (2) just on comments that have received a new like event, i.e., avoiding to recompute the clique if not needed. Nevertheless, when a new friendship relation is established, the *Comment Score Updater* invalidates and recomputes all the cliques. Since this operator performs critical operations, we deploy multiple instances of it; to preserve the application integrity while increasing the parallelism, the incoming streams are partitioned relying on the comment identifier. Similarly to the first query, i.e., using a step-wise approach, the *Comment Score Rank* and the *Comment Rank Merger* rank the comments and identify the top-k ones that are supported by the larger communities.

3. EVALUATION

We evaluate the performance of our solution on a single Amazon EC2 c4.xlarge instance, running Debian 8.3 (Jessie) and equipped with an Intel Xeon E5-2666 Haswell (2.6 GHz, 4 cores and 25 MB cache), 8 GB of RAM and SSD with 900 IOPS [1]. This experimental environment is very similar to that employed for evaluating the Grand Challenge solutions [4]. Our solution is implemented in Java 1.8 and relies on Apache Flink 1.0.0 [3] and Redis 3.0.7 [5].

The experimental analysis evaluates for both queries (1) the latency distribution on crucial operators, (2) the average latency per tuple, and (3) the average memory utilization. The latency is measured using the analysis tools provided by the Flink framework, whereas the memory utilization is recorded every second during the application execution.

Our solution aims at properly processing the dataset proposed by the Grand Challenge in a timely way. This dataset presents 55.9×10^6 events, distributed as: 44% comments, 39% likes, 15% posts, and 2% friendships. Preserving this event distribution, our experimental evaluation considers portions of the dataset ranging from 1% (55.9×10^3 events) to 50% (27.9×10^6 events) of the original dataset. We parametrized the second query to identify the top-3 comments supported by the larger communities with a sliding window of 1 hour (i.e., $k = 3$, $d = 3600$ s).

To inspect the latency distribution, we can partition the topology into the following meta-operators:

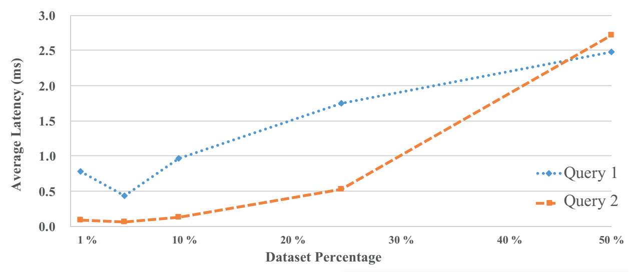

Figure 4: Query 1 and Query 2: average latency per tuple.

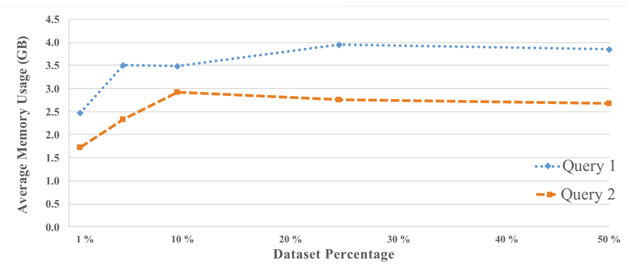

Figure 5: Query 1 and Query 2: average memory utilization.

1. *Src-Snk:* which encapsulates the logic involved in introducing and producing the events. It comprises latencies due to the I/O operations and the generation of tuples that allows to compute the scores (i.e., event ordering, time synchronization, comment to post mapping);
2. *Updater:* which encapsulates the logic involved in computing the post or comment score. It comprises the most memory-intensive data structures, the feedback stream (first query), and clique computation (second query).
3. *Ranker:* which encapsulates the logic involved in ranking the posts/comments and filtering the updates. It comprises latencies due to the step-wise sorting and ranking approach.

During all the experiments, approximately half of the latency is introduced by the *Updater* and about a third by the *Ranker*.

The average latency per tuple of both queries is represented in Figure 4. As the dataset increases in size, the trend of this metric becomes clearer. For the first query, the average latency increases almost linearly with the dataset size; this trend is due to the growing number of post scores that the system has to manage. For the second query, the average latency increases exponentially as the dataset grows, and, upto the 30% of the dataset, the system achieves latencies lower than 1 ms. The performance trend of this query is readily explained recalling that (1) the application has to maintain in memory the whole social-graph, and (2) identifying the cliques is an NP-hard problem. Moreover, we have obtained empirical evidences that the centralized usage of Redis for small and high-frequency updates shows poor performances.

As regards the memory utilization, Figure 5 shows a comparison of this metric for both the queries. The general tendency shows that the required memory reaches a steady state below 4 GB. In the first query, this result is achieved thanks to the feedback stream which allows to discard all the data (e.g., comments, score) concerning the expired posts. Observe that the advantage of exploiting the feedback stream is proportional to the amount of posts and to their expiration rate. In the second query, the memory utilization stabilizes around 3 GB when the application processes at least 10% of the whole dataset. Evaluating the memory utilization trend, we can suppose that most of the social graph structure is built quickly while it tends to evolve slowly. When the dataset grows over the 25%, an increasing computational effort is due to the management of Redis, slowing down the task of identifying the largest communities.

Summing up, the experimental results show that our solution provides *effective load balancing* because the workload is balanced among the operators and cores, thus making our solution not affected by back-pressure phenomena in any portion of the stream and in any stage of computation; there is also an *efficient memory usage*, because the physical memory is never close to saturation, thus avoiding the overhead introduced by the garbage collection and memory swapping. However, there are some *latency bottlenecks* that slow down the application execution, mainly due to the presence of centralized and complex operators.

4. CONCLUSIONS

We have presented the design, implementation, and evaluation of our Flink-based solution to the DEBS 2016 Grand Challenge. Despite its young age, Flink turned out to be a solid stream processing framework with some interesting features. An example is represented by the feedback stream, which has allowed us to minimize the wastage of memory, keeping its occupation low and steady, and to avoid back-pressure and performance degradation. The experimental results, conducted in the reference environment, show that our solution can process up to 400 tuples/s with an average latency of 2.5 ms for the first query, and up to 370 tuples/s with an average latency of 2.7 ms for the second query, considering 50% of the original Grand Challenge larger dataset.

Leveraging the experience gathered to answer the Grand Challenge, we identify the following improvements as future work. We will exploit a finer-grained parallelism by enhancing the application components with more sophisticated data structures and operations, which enable concurrent and efficient updates. To further reduce the average application latency, we will decouple the social events (e.g., post, comment, like) from their content (e.g., post message), so to reduce the streams transmission time. Finally, we could make the feedback stream self-adaptive, so to properly handle the back-pressure mechanism, even when posts expire with an unpredictable rate.

5. REFERENCES

[1] Amazon. EC2 Instance Types. aws.amazon.com/ec2.
[2] C. Bron and J. Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM*, 16(9):575–577, 1973.
[3] Apache Flink. flink.apache.org, 2016.
[4] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow. The DEBS 2016 Grand Challenge. In *Proc. of ACM DEBS '16*. ACM, 2016.
[5] Redis. redis.io, 2016.