Grand Challenge: Dynamic Graph Management for Streaming Social Media Analytics

Christos Vlassopoulos[1], Ioannis Kontopoulos[2], Michail Apostolou[3], Alexander Artikis[4,1] and Dimitrios Vogiatzis[1,5]
[1]Institute of Informatics and Telecommunications, N.C.S.R. "Demokritos", Greece
[2]Department of Informatics and Telematics, Harokopio University of Athens, Greece
[3]Department of Electrical and Computer Engineering, University of Thessaly, Greece
[4]Department of Maritime Studies, University of Piraeus, Greece
[5]The American College of Greece, Deree
{cvlas, a.artikis, dimitrv}@iit.demokritos.gr, it21215@hua.gr, miaposto@uth.inf.gr

ABSTRACT

We present a system for analytics on streaming social media that computes the most active posts, based on the age and the amount of comments for each post, and tracks the largest communities that comprise friends that are fond of the same content. To deal with high velocity data streams, we implemented an algorithm for incrementally updating graphs expressing social networks. The evaluation of our system is based on the datasets of the DEBS 2016 challenge.

CCS Concepts

•**Information systems** → **Data streaming; Data stream mining;** •**Computer systems organization** → *Real-time systems;*

Keywords

Social Network Activity; Graph Models; Community Tracking

1. INTRODUCTION

The continuously rising popularity and functionality of the social networks, makes the efficacious and effective recognition and tracking of their activity a sought-after property. To that effect, the DEBS 2016 Grand Challenge [2] requires the implementation of quick and accurate methods of calculating social network analytics, such as ranking popular posts and retrieving communities that are interested in the same content, based on a stream of timestamped events. These events consist of posts, comments, likes and friendships of a hypothetical social network. The goal of the first query is to compute the top three active posts and output the result every time the ordered list of posts has changed. As far as the second query is concerned, the goal is to find the k comments with the largest range not more than d seconds ago. The range of a comment is defined as the size of the largest connected component in the graph defined by persons who have liked that comment and form a community between them (they are friends).

DEBS '16 June 20-24, 2016, Irvine, CA, USA

ACM ISBN 978-1-4503-4021-2/16/06.

DOI: http://dx.doi.org/10.1145/2933267.2933515

2. CHALLENGE SOLUTION

In the following, we will describe the implementation used to solve both queries of the challenge. Our approach was focused on two main directions. Firstly, minimizing the need for storing information and increasing the amount of on-the-fly computations as much as possible. Secondly, incrementally updating large data structures and graphs.

2.1 Query 1

The purpose of Query 1 is to compute the top-3 ranking posts, based on an event stream comprising temporally sorted tuples representing the occurrence of posts and comments in our hypothetical social network. The ranking is decided through a scoring strategy that promotes posts that i) receive many comments and ii) are relatively recent. Each post or comment is given a score of 10 upon its creation. This score decrements by 1 every day. The total score of a post is the sum of its individual score plus the respective scores of all its comments.

The input data are processed in a streaming fashion. Upon receiving an event, our system decides whether it is a comment or a post, so that it can store the corresponding information appropriately. Each post or comment has its own unique id, as well as a set of features, and can be considered as a unique ($key \rightarrow value$) pair. Hence, a convenient way to store them is by using hash maps. Statistics on the sample dataset have shown that the vast majority of posts do not receive any comments. This observation implies that we could avoid some unnecessary calculations, by dividing the collection of posts into two smaller groups, based on whether they have received comments or not, which are treated separately. Thus, we use three hash maps for our data: two for the posts ("Commented" and "Commentless"), and one for the comments.

Figure 1 illustrates the first actions of our system, upon receiving a new event. If it is a new post, it is directly stored in the "Commentless" hash map, whereas if it is a comment, it is stored in the "Comments" hash map. However, in case of a comment, since comments are related to posts, our method

must also link this comment to its respective post. To that effect, it checks whether this comment is a direct reply to a post or a reply to another comment. In the first case, if the post has never been commented before, it is moved to the "Commented" hash map, and then the comment and its author are added to the post's list of comments and set of commenters, respectively. In the second case, the corresponding post is retrieved via the comment that is being replied, by looking it up on the "Comments" hash map.

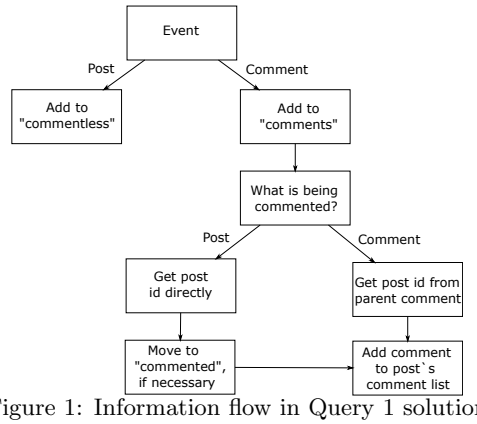


Figure 1: Information flow in Query 1 solution

Now that the new event has been identified and stored appropriately, our method begins calculating the top-3 scoring active posts. The pseudocode of this process is presented at Algorithm 1.We first get the timestamp of the event and we update the clock. Then, we initialize a data structure that will eventually contain the resulting top-3 posts (see lines 1 and 2 of Algorithm 1). Subsequently, we start calculating post scores and filling the top-3 array. We start by calculating scores only for the posts in the "Commented" collection, because since they have been replied, they obtain bigger scores in general and thus are much more likely to be in the top-3 than posts that have not yet received any comments. As lines 3-8 of the Algorithm show, only if there are still berths available in the top-3 slot or it contains scores that are not high enough to ensure that no commentless post can make it to the top-3 do we examine the collection of commentless posts. In fact, as new comments rush into our system, the scores of the commented posts rise to a point that the top-3 is comprised solely by commented posts. Therefore, rarely do we need to check the "Commentless" hash map, thus avoiding a significant amount of redundant computations.

The procedure FILLTOP3 that appears in lines 4, 6, and 8 of Algorithm 1, is presented in lines 9-21. This procedure incrementally builds the highest scoring triplet of posts, by using the previously computed top-3, if it exists. Experiments on the sample dataset have indicated that quite often posts that appear in the top-3 tend to remain in the top 3 for quite a long time. Therefore, we make use of this fact in order to avoid building the top-3 from scratch every time and all the unnecessary computations that this procedure entails. In lines 10-12 we take the previously calculated top-3 and update their scores. If any post has become inactive (i.e. its score has dropped to 0) we remove it completely. Then we iterate over the collection of posts ("Commented" or "Commentless", as mentioned earlier) and calculate the score of each post (line 14). If a post has become inactive then it must be deleted (line 19). Otherwise, our method checks if this post is suitable to enter the top-3. A post can be appended to the array if the top-3 is not full (i.e. the array contains less than three posts), or if its score exceeds that of a post that is already in the top-3. Ultimately, the resulting array is sorted and trimmed so that it contains only three elements.

Algorithm 1 Query 1

```
 1: now ← getEventTimestamp()
 2: top3 ← ∅
 3: if commented ≠ ∅ then
 4:     FILLTOP3(commented, now)
 5:     if top3 not full ∨ top3 contains score ≤ 10 then
 6:         FILLTOP3(commentless, now)
 7: else
 8:     FILLTOP3(commentless, now)
 9: procedure FILLTOP3(collection, now)
10:     top3 ← updateScores(top3)
11:     top3 ← sort(top3)
12:     top3 ← removeZeroScores(top3)
13:     for each post ∈ collection do
14:         t ← CALCULATESCORE(post, now)
15:         if t.score > 0 then
16:             if (top3 not full ∨ t.score ≥ top3.lowest) ∧
    post ∉ top3 then
17:                 top3 ← top3.append(post)
18:         else
19:             remove post from collection
20:     top3 ← sort(top3)
21:     top3 ← trim(top3)
22: function CALCULATESCORE(post, now)
23:     postDecay ← post's age (in days)
24:     newPostScore ← (10 − postDecay)+
25:     totalCommentScore ← 0
26:     if post ∈ commented then
27:         for each comment ∈ post.comments do
28:             commentDecay ← comment's age (in days)
29:             newCommentScore ← (10 − commentDecay)+
30:             if newCommentScore = 0 then
31:                 remove comment from post.comments
32:                 update post.commentAuthorSet
33:             totalCommentScore ← totalCommentScore +
    newCommentScore
34:     totalScore ← newPostScore + totalCommentScore
35: return (post, totalScore)
```

The function calculating post scores is shown in lines 22-35 of Algorithm 1. This function first computes the age of the post in days and then subtracts its age from its initial score, which is 10, as described above (see lines 23,24). Then, if the post is a commentless one, no further action is required. However, if the post is a commented one, then for each of its comments, our function computes its age and subtracts it from its initial score, and makes sure that if a comment's score drops to 0, then it removes it from the list of this post's comments (lines 30-32). Ultimately, CALCULATESCORE takes the sum of all the comments' scores for this post and adds it to the individual score of the post.

2.2 Query 2

This query concerns three event types, namely friendships, likes and comments that, like in Query 1, arrive in a streaming fashion. The goal is to compute the k most popular comments, in terms of user communities, that have been created within a d-second window. Each comment corresponds to a graph. This graph's vertices represent the users that have liked this comment and its edges represent the friendships between these users. Each graph may contain more than one connected component, the size of the largest of which defines the range of the graph. Therefore, the range of the comment is the size of the largest community of users that have liked this comment, and Query 2 asks for the comments with the k highest ranges.

In order to perform our calculations, we need to keep track of the comments and their respective graphs. Thus, we have two main data structures: a map of comments and a map of their graphs. Apart from these, we also need to store the friendships and likes that occur in two other, auxiliary maps. Like in Query 1, when a new event enters the system, it is classified according to its type. In case of it being a comment, we just store the comment and initiate its graph with no vertices or edges, and a range of 0. In case of a like, we add a vertex to the graph of the comment that was liked, along with all the necessary edges that match the friendships that this user adds to the community. Finally, if we have a friendship as input, we isolate the comments that have been liked by both users that participate in the friendship and add an edge between the respective vertices to each of them.

To efficiently calculate the range of a graph we need to keep track of all its connected components. To do so, we used a hash map which holds the connected components of the graph (each connected component is represented as a hash set of vertices), a different hash set which contains all of the vertices of the graph, and an integer denoting the range. hash maps and hash sets are used because of their $O(1)$ time complexity (constant time) in finding an element.

The algorithm of incrementally calculating the range consists of two parts: i) what happens when a vertex is added (Algorithm 2) and ii) what computations take place when an edge is added (Algorithm 3). When a vertex is added to the graph, we check if the integer expressing the range equals to zero. If this is the case, then the range is incremented by 1. Subsequently, we add the vertex id to the hash set that holds all of the vertices of the graph, and then add the new vertex as a component to the hash map of components (a vertex initially is a component by itself).

Algorithm 2 Add a vertex to the graph

```
1: procedure ADDVERTEX(vertexId)
2:     if currentRange = 0 then
3:         currentRange ← 1
4:     vertices.add(vertexId)
5:     components.add(componentId, vertexId)
6:     componentId ← componentId+1
```

To insert an edge (friendship) to a graph, we iterate the hash map of components and locate the component(s) that contain the vertices that are to be linked (see lines 12, 14 and 17 of Algorithm 3). Once we find them, we concatenate them and in case these vertices are located in the same component

Algorithm 3 Add an edge to the graph

```
1: procedure ADDEDGE(vertexId1,vertexId2)
2:     flag ← 0
3:     ckey ← 0
4:     for each component ∈ components do
5:         if flag ≠ 0 then
6:             if flag ∈ component then
7:                 concatenate components
8:                 if component.size > currentRange then
9:                     currentRange ← component.size
10:                components.remove(component.key)
11:                break
12:        if vertexId1 ∈ component ∧ vertexId2 ∈ component then
13:            break
14:        else if vertexId1 ∈ component then
15:            ckey ← component.key
16:            flag ← vertexId2
17:        else if vertexId2 ∈ component then
18:            ckey ← component.key
19:            flag ← vertexId1
```

we just break the iteration. The reason is that a new edge inside a connected component does not change the range of the graph.

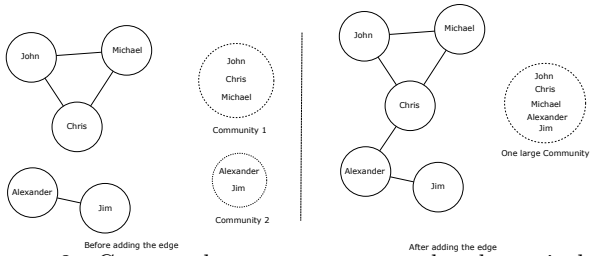


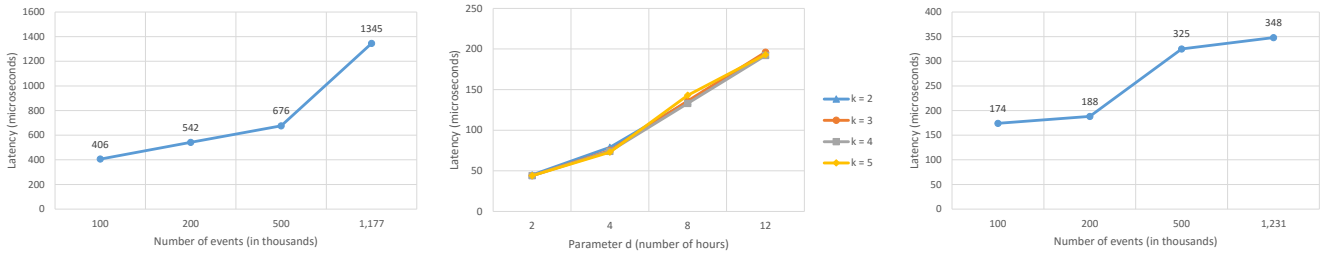
Figure 2: Connected components example; the arrival of friendship connects two components

In the left side of Figure 2 we can see that the graph comprises two communities in the form of two connected components. The dashed circles in each case denote the communities. The range in the first example is 3. When a new friendship arrives (Chris - Alexander), the two subgraphs connect with each other (see the right side of Figure 2) and as a result the two communities merge into one (one dashed circle). The new range of the graph is 5. Even if another friendship arrives, for instance (Jim - Chris), it will not affect the size of the community, and therefore the range of the comment.

The output of Query 2 is also affected by the time decay. If a comment is in the top-k, but its age surpasses the d-second threshold, then this comment is considered too old and we must remove it completely from our system. To avoid unnecessary calculations and permutations we always keep the top-k list sorted and when a change occurs the sorting algorithm only moves that particular comment.

3. EVALUATION

We describe the experiments conducted in order to evaluate both queries. The evaluation has taken place on a Virtual Machine (VM) with 4 cores (Intel(R) Xeon(R) CPU



(a) Scalability of Query 1 solution (b) Average latency of Query 2 solution (c) Scalability of Query 2 solution

Figure 3: Empirical analysis of Query 1 and Query 2 solutions

E5-2630 v2 @ 2.60GHz) and 8 GB of RAM. The dataset used for the evaluation is the sample dataset provided in the DEBS 2016 Grand Challenge website[1]. This dataset has a total size of 152 MB and, for Query 1 the posts and comments streams combine to give a total of 1,177,303 input events, and for Query 2 the comments, likes, and friendships streams produce a total of 1,231,531 events. The source code of our solution is available on GitHub, under the BSD-3 license[2].

3.1 Query 1

Testing for Query 1 has shown that our method was able to achieve average latencies as low as 1.3 milliseconds (ms). To test how our method scales over different amounts of input events, we created 3 extra, smaller datasets, by taking the first 100,000, 200,000, and 500,000 events from the sample dataset. Figure 3a shows the results. This figure shows that the average latency is proportional to the amount of input events. This behavior can be explained if we consider the size of the data structures we use. When more posts and comments rush into our system, then all hash maps (commented posts, commentless posts and comments) grow and so does the time needed to search for, or iterate over their elements.

3.2 Query 2

For Query 2 we used different values for parameters k and d – the decay factor. We used constant k equal to 2 and tuned parameter d to values from 2 to 12 hours. Then, we repeated the experiments for k equal to 3, 4 and 5 (Figure 3b). In addition, we conducted experiments with the 3 reduced versions of the sample dataset, like we did with Query 1 (Figure 3c). The figures above illustrate the scalability of our method for different values of the parameters k, d, as well as the number of input events. We observe that our method is tolerant towards changes in the value of k, and proportional to d and the amount of input events. The reason behind this behavior is that as events enter our system, we need to take into account more elements while building and sorting the list of top-k comments.

4. FUTURE WORK

The ideas presented here may be further exploited in the Reveal project[3], where the aim is to quickly assess the trustworthiness of information in streaming news media. In particular the first query could be used to detect a trending post in Twitter. The comments in this case would be the mentions a Tweet receives, whereas the score could be related to the number of followers of the user that created a Tweet. Additionally, the rate of change of a score could be used to predict trending or dying topics.

Detecting information cascades entails tracking the spread of information in a network. Studying cascades at a mesoscale level, such as the community structures, usually entails segmenting a time series of communities into predefined intervals and then tracking the evolution (i.e. growth, continuation, dissolution) of communities across them. This can be very slow on large data sets [1]. Even though the communities we are usually interested in are not connected components but rather dense structures that may or may not be connected, the ideas developed in the current work could be extended to produce dynamic community detection algorithms.

[1]http://debs2016.org/
[2]https://github.com/cvlas/DEBS2016GC
[3]http://revealproject.eu/

5. ACKNOWLEDGMENTS

The authors would like to acknowledge partial support of this work from the European Community Seventh Framework Programme, as part of the FP7 610928 REVEAL (REVEALing hidden concepts in Social Media).

6. REFERENCES

[1] G. Diakidis, D. Karna, D. Fasarakis-Hilliard, D. Vogiatzis, and G. Paliouras. Predicting the evolution of communities in social networks. In *Proceedings of the 5th International Conference on Web Intelligence, Mining and Semantics*, page 1. ACM, 2015.

[2] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow. The debs 2016 grand challenge. *DEBS 2016 proceedings*, June 2016.