A Novel Technique and Infrastructure for Online Analytics of Social Networks

By

Lian Liu

Thesis

Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

Master of Science

in

Computer Science

August, 2016

Nashville, Tennessee

Approved:

Aniruddha Gokhale, Ph.D

Abhishek Dubey, Ph.D

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. Aniruddha Gokhale for the continuous support of my master study and research, for his patience, motivation, enthusiasm, and immense knowledge. This thesis would not have been possible without his guidance.

Besides my advisor, I would like to thank my second thesis reader: Dr. Abhishek Dubey for his encouragement, insightful comments, and hard questions.

My sincere thanks also goes to Shweta Khare, for the stimulating discussion, and in particular, for enlightening me the first glance of research.

I also would like to thank the EECS department at Vanderbilt. As an international student, you gave me the chance to study here, meet new friends, gain more knowledge, and broaden my horizons, I really appreciate this experience.

Last but not the least, I would like to thank my family: my parents for giving birth to me in the first place and supporting me spiritually throughout my life.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

I. INTRODUCTION

In the last decade, the online social networks have become one of the most important platforms for social life, where people all over the world come together and get connected. At the same time, the popularity of online social networks have exploded at an enormous scale. The number of Internet users visiting one online social network at least once a month has grown from 41% in 2008 to over 65% in 2014 [1]. For instance, at the end of its first year, Facebook recorded only 1 million monthly active users on its platform [2]. Since then a tremendous increase has been witnessed. In the first quarter of 2016, this number has reached over 1.6 billion [3]. In the meanwhile, social activities on social networks have become the online activity that people spend the most time on, which accounts for 28% of time among all internet activities in 2015 [4]. These social platforms produce hundreds of millions of Gigabytes of data everyday. We can benefit from this huge amount of data being generated by users at incredibly rapid rates for a variety of applications, such as in homeland security or understanding the demographics patterns and their likes and dislikes.

Up until now, social network analysis has been implemented mainly in an offline mode[5]. Such methods cannot discover the large wealth of real-world events, reflect the undergoing trends in the social networks, and will be easily overwhelmed by the pace of generating data. Second, because of the pace of data generation, it is possible to lose track of prior history and the size of data may also make it infeasible to accumulate so much information. Taking into account the rapid growth of online social networks which generates datasets with a strong temporal and structural aspect, new methods and systems for analysis are needed.

In this thesis, we use a representative example from the real-world to demonstrate our approach. We focus on the first query of the ACM DEBS 2016 Grand Challenge problem [6]. The DEBS 2016 Grand Challenge is on processing of data streams originating from the dataset provided along with the LDBC Social Network Benchmark [7]. The data provides information of posts, comments, friendships, and likes. The goal of the DEBS 2016 Grand Challenge is to provide the analysis metrics for an evolving social-network graph. Specifically, the first query of 2016 Grand Challenge targets the problem of identifying of the posts that currently trigger the most activity in the social network. The query requires continuous analysis of a dynamic graph under the consideration of two streams (of post and comment) that reflect updates to the graph.

In this thesis, we set out to come up with an efficient algorithm to solve the first query of the DEBS 2016 Grand Challenge. In order to solve this challenge, we used Redis, which is an in-memory key-value storage system rather than contemporary graph data processing systems and graph-based storage systems to hold the graph data we analyzed. As part of this thesis, we discuss why we chose Redis, the in-memory key-value storage system, over the graph data processing systems and graph-based storage systems in this specific graph analysis problem.

The remainder of this thesis is organized as follow: Section II describes our research motivation and background material, including the database technology we will analyze for a social graph analysis in this paper; Section III covers how we implemented our algorithms, how we chose among key-value store and graph database for our implementation, and the challenge we met to solve the query 1; Section IV presents the experiment settings, results, and analysis; Section V presents related works; Section VI presents concluding remarks.

II. BACKGROUND

Although the world has only witnessed such a rapid expansion of online social networks in the last decade, social networks were not an invention of recent years. People always have needs to communicate with other people. Throughout the history, people used to visit the people they wanted to see, sent letters through posts; later, people sent messages through telegraph and telegraph. Now, the 21st century brought us the Internet.

At the end of 2015, nearly two-thirds of American adults (65%) [9] have an account on at least one social network site, with 78% actively using one[4]. People are most likely to use social network in order to keep up with friends (44%) or to fill time (39%). Moreover, the average internet user spends 1 hr 49 mins per day on social networks. Without question, online social networks have become one of the most important aspects of our daily life. Because of their importance and popularity, social networks are the most efficient way to obtain valuable information in marketing, crime detection, detection of epidemics, social relationships, and etc[4].

Such a gigantic and active community produces massive amount of data every minute. Facebook, which is the most active of social networks, generates user likes over 4 million posts every minute. Instagram comes in second with 1,736,111 likes on photos each minute. Twitter users generate 347,222 Tweets each minute. Similarly, Snapchat users share 284,722 snaps per minute[8].

To analyze such enormous amount of data generated in social networks, the use of powerful computing machines is required. To represent social networks, graph theories are usually applied[10]. Graphs are mathematical structures used to model pair-wise relations

between objects from a certain collection. A graph can be defined as a set of vertices V and a set of edges. Vertices can be any abstract data types and can be presented with the points in the plane. A line connecting these vertices is called an edge. An edge can also be an abstract data type that shows the relation between the nodes (which again can be an abstract data types). For instance, if an edge {a,b} exists, then we can say that nodes a and b are related to each other [11].

Graph theory, which is a branch of mathematics concerned with graphs, dates back to 1735, when the Swiss mathematician Leonhard Euler solved the Königsberg bridge problem[11]. In the age of the Internet, graph theory is employed to this new realm. The online social network can be viewed as a very large graph[12]. Vertices are individuals (or groups of individuals) and edges are the links between different individuals (or groups of individuals). In this thesis, we focus on a specific social network graph problem from the DEBS 2016 Grand Challenge, and propose an efficient algorithm to solve this challenge.

The first query of DEBS 2016 Grand Challenge requires the output of the top-3 most popular posts in a social network. The input data of this query is organized in two separate streams, which are posts and comments. As the data comes into the system, the outputs should reflect these updates to the graph every time the ranking of the top-3 posts change.

Overview of Graph Data Processing Systems

Since the problem is amenable to be solved as a graph problem, it is important to survey existing graph processing systems to determine their suitability in our context. The graph data processing systems, such as Google Pregel [13], Apache Giraph [14], and Apache Spark

GraphX[15], manage the graph in main memory but also comprise an underlying storage layer. Such an integration mechanism can speed up data processing operations.

Pregel [13] is a large-scale graph data processing system. It adopts Bulk Synchronous Parallel (BSP) model for synchronization. A BSP computer consists of a set of compute nodes, a global communications network, and a mechanism for the efficient barrier synchronization of the processors. A computation consists of a sequence of parallel super steps, where each super step consists of a sequence of steps, followed by a barrier synchronization at which point all data communications will be completed. However, Pregel is a batch-oriented system, which is not suitable for the online stream processing needs for the problems we focus on.

Giraph[14], which is inspired by Pregel, is an iterative graph processing system built for high scalability. It also uses the BSP model for synchronization. Like Pregel, it is a batch-oriented system as well.

GraphX[15] is Apache Spark's API for graphs and graph-parallel computation. It unifies ETL (Extract, Transform and Load), exploratory analysis, and iterative graph computation within a single system. At the time we were studying, GraphX was still evolving and the APIs were not finalized. So we did not consider it for our work.

In summary, despite their desirable features for graph analysis, all these three popular graph data processing systems are offline systems, which cannot provide real-time output and online processing for an evolving graph required by the DEBS challenge. Thus, we needed to explore other alternatives.

Overview of In-Memory Key-Value Storage Systems and Graph Based Storage Systems

Given the pace of generation of social data, the structure of social data, and the nature of online analysis, storing and querying data in near real-time presents a challenge to storage systems. Also the graph structure of this social data has to be handled by these systems. For this research, we investigated two storage system types: in-memory key-value based and graph based, and finally decided to use in-memory key-value based storage system as the storage system for our algorithm to solve the challenge problem. We provide a rationale for why the in-memory key-value storage system is better than the graph based one in the case of this particular problem.

Key-value based storage system can be considered the most basic and backbone implementation of NoSQL. Similar to a dictionary, these kind of storage system works by matching keys with values. There is no structure nor relation. After connecting to the database server, an application can provide a key and retrieve the matching value. An in-memory key-value database is a storage system that primarily relies on main memory for computer data storage. Thanks to this feature, they are extremely performant, efficient and usually easily scalable.

Figure 1 shows a basic model of a Key-Value Data Storage System.

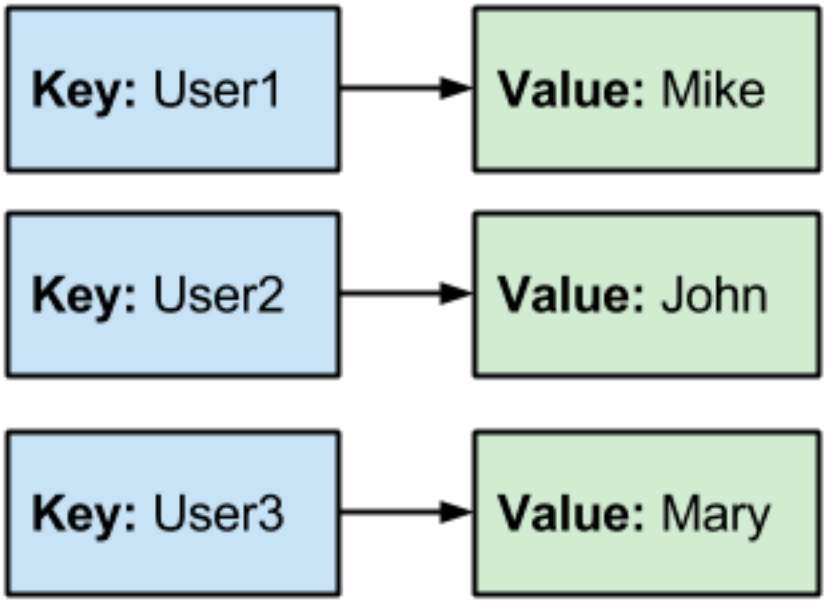

Fig 1. Key-Value Data Storage System

On the other hand, the graph based storage system represents the data in a more complicated way. It uses the graph structures to model the data. Similarly to mathematics, certain operations, such like graph traversal and graph merging, are much simpler to perform using these graph structures, thanks to their nature of linking and grouping related pieces of information. With the graph structures, graph based databases are commonly used by applications where clear boundaries for connections are necessary to be established.

Figure 2 shows the basic structure of Graph Based Data Storage System.

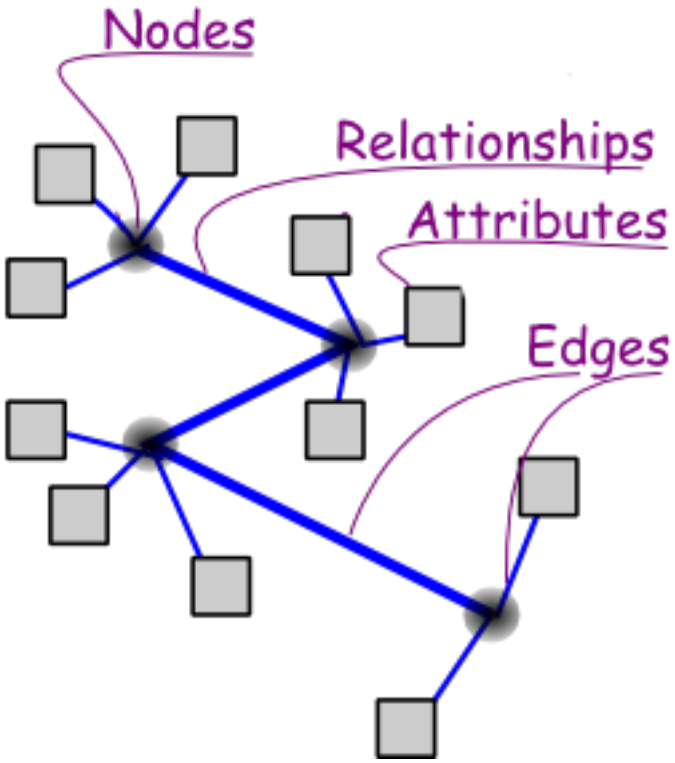

Fig 2. Graph Based Data Storage System

However, for cases such as the DEBS 2016 Grand Challenge the graph is constantly evolving as new data comes into the system. Thus, every query to the database needs to get access to stored data in the disk, which significantly slows down the performance. Second, the data has a time limit, which does not require a long term persistent storage. These limitations call for a faster but ephemeral type of storage systems. Third, in the case of the ACM DEBS challenge, a requirement was to solve the challenge in a single machine of 8 GB memory. Thus, for situations where resources are relatively constrained and where we cannot exploit massive parallelism, we need an alternative to graph databases.

To overcome this performance bottleneck and fulfill the challenge requirements, the in-memory key-value storage system Redis was deemed to be a promising alternative. Redis is an

open source in-memory data structure store, which supports data structures such as strings, hashes, lists, sets, and sorted sets with range queries. Redis has built-in replication, Lua scripting, Least Recently Used (LRU) eviction, transactions and different levels of on-disk persistence, and provides other services for high availability [16].

Figure 3 shows Redis built-in data structures and services.

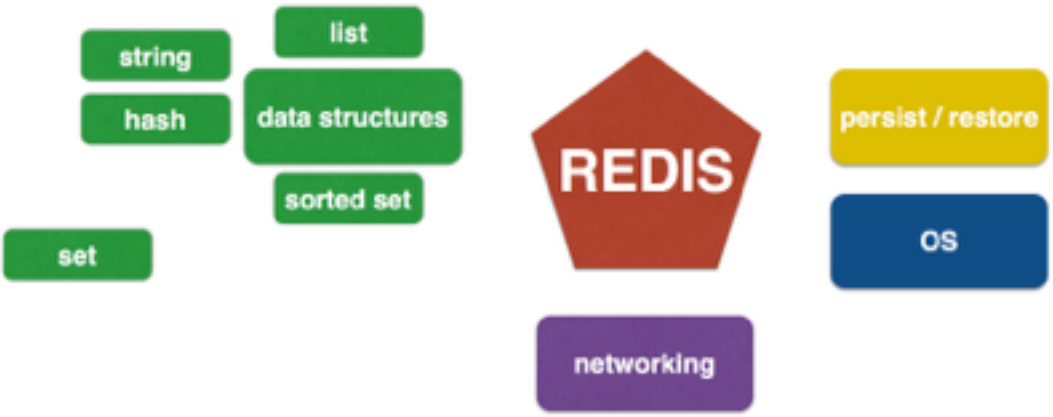

Fig 3. Redis

III. ALGORITHM AND IMPLEMENTATION

Overview of the ACM DEBS 2016 Grand Challenge

The ACM DEBS 2016 Grand Challenge is the sixth in a series of challenges which seek to provide a common ground and uniform evaluation criteria for a competition aimed at both research and industrial event-based systems. The focus of the DEBS 2016 Grand Challenge is on

processing of data streams originating from the dataset provided with the LDBC Social Network Benchmark. The data provides information of posts, comments, friendships, and likes.

The goal of the DEBS 2016 Grand Challenge is to provide the analysis metrics for a dynamic (evolving) social-network graph. Specifically, the 2016 Grand Challenge targets the following problems: (1) identification of the posts that currently trigger the most activity in the social network, and (2) identification of large communities that are currently involved in a topic. The corresponding queries require continuous analysis of a dynamic graph under the consideration of multiple streams that reflect updates to the graph[15]. In this thesis, we provide an efficient algorithm for the first query only because of the similarities between the two queries.

Data

The input data is organized in several separate streams, each provided as a text file. Specifically, the Challenge Problem provides the following input data files(only list data files related to query 1 are shown) in Table 1 and Table 2.

Table 1 shows data format of post data, which is <ts, post_id, user_id, post, user>. Table 2 shows data format of comment data, which is <ts, comment_id, user_id, comment, user, comment_replied, post_commented>.

ts	is the post's timestamp
post_id	is the unique id of the post
user_id	is the unique id of the user
post	is a string containing the actual post content
user	is a string containing the actual user name

Tab 1. Posts Data Format

ts	is the comment's timestamp
comment_id	is the unique id of the comment
user_id	is the unique id of the user
comment	is a string containing the actual comment
user	is a string containing the actual user name
comment_replied	is the id of the comment being replied to (-1 if the tuple is a reply to a post)
post_commented	is the id of the post being commented (-1 if the tuple is a reply to a comment)

Tab 2. Comments Data Format

Each file is sorted based on its respective timestamp field.

Query 1

The goal of query 1 is to compute the top-3 scoring active posts, producing an updated result every time they change.

The total score TS of an active post P is computed as the sum of its own score plus the score of all its related comments. A comment C is related to a post P if it is a direct reply to P or if the chain of C's preceding messages links back to P.

Each new post has an initial own score PostScore (PS) of 10 which decreases by 1 each time an interval of 24 hours elapses since the post's creation. Each new comment's score CommentScore(CS) is also initially set to 10 and decreases by 1 in the same way since the comment's creation. Both PS and CS are non-negative numbers. A post is considered no longer active (that is, no longer part of the present and future analysis) as soon as its total score reaches zero, even if it receives additional comments in the future [6].

Output specifications are shown in Figure 4.

Input Streams: *posts, comments*

Output specification:

```
<ts,top1_post_id,top1_post_user,top1_post_score,top1_post_commenters,
top2_post_id,top2_post_user,top2_post_score,top2_post_commenters,
top3_post_id,top3_post_user,top3_post_score,top3_post_commenters>
```

Fig 4. Query 1 Output Specifications

ts is the timestamp of the tuple event that triggers a change in the top-3 scoring active posts appearing in the rest of the tuple.

topX_post_id is the unique id of the top-X post.

topX_post_user is the user author of top-X post.

topX_post_commenters is the number of commenters (excluding the post author) for the top-X post

Results should be sorted by their timestamp field. The character "-" (a minus sign without the quotations) should be used for each of the fields (post id, post user, post commenters) of any of the top-3 positions that has not been defined. The logical time of the query advances based on the timestamps of the input tuples, not the system clock [6].

The sample output tuples for the query is shown in Figure 5.

Sample output tuples for the query

```
2010-09-19 12:33:01.923+0000,25769805561,Karl Fischer,115,10,25769805933,Chong
Liu,83,4,-,-,-,-
2010-10-09 21:55:24.943+0000,34359739095,Karl Fischer,58,7,34359740594,Paul
Becker,40,2,34359740220,Chong Zhang,10,0
2010-12-27 22:11:54.953+0000,42949673675,Anson Chen,127,12,42949673684,Yahya
Abdallahi,69,8,42949674571,Alim Guliyev,10,0
```

Fig 5. Query 1 Output Sample

Leveraging the Publish-Subscribe Model

The raw data is organized in different text files by event type. In order to simulate the streaming semantics of real world online social networks, we needed a model to sort in time order the data in these text files and output the sorted data to our ranking system. This is where our Publish-Subscribe Model comes into play.

Basically, the Publish-Subscribe Model reads all these files in parallel. Specifically, it reads one line of data from each file at a time, and only emits to the publisher the data of the earliest timestamp among new data and other pending data, while storing all other new data into the list for later comparison. With this model, the data to be consumed is always in time order.

Figure 6 shows the structure of the Publish-Subscribe Model.

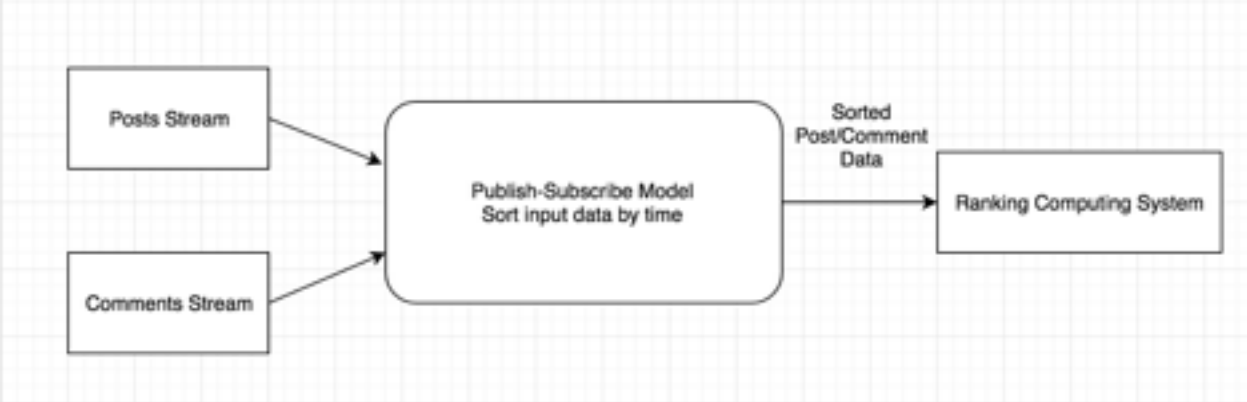

Fig 6. Publish-Subscribe Model

Algorithm Design Challenges and Data Structure

There are several challenges when we designed the algorithm to handle the query.

The first challenge was how to only output the new and different ranking from the last output(ranking with different posts; same posts as the last time but in different order can be regarded as different ranking; same posts as the last time but any of them with different scores cannot be regarded as different ranking). We created a set as a store for last top-3 ranking. After a new top-3 ranking is computed, this newly computed ranking is compared to the last top-3 ranking.

The second challenge was how to maintain substantial number of states in the post and comment data. The third challenge was how to only keep track of active posts and comments in the online social network. The data structure to solve these three challenges are discussed below.

1. If the event type of data received is post

    1. First create a Hashmap to store the info of the post. The info we need are timestamp, post ID, user ID, user name, last modified timestamp. Upon creation, the last

modified timestamp is set to its timestamp. **(post_id): { timestamp, user_id, user name, last modified timestamp }**

2. Initialize a sorted set if it does not already exist (call it RANKING). This sorted set is to store all active posts with their scores as item scores. Obviously, if the score of one post drops below 0, it will be removed from RANKING. **RANKING: { (post_id): score }**

2. If the event type of data received is a comment

1. First create a Hashmap to store the info of the post. The info we need are timestamp, comment ID, post ID, last modified timestamp. Upon creation, last modified timestamp is set to its timestamp. From the data schema we know that if a comment is a reply to another comment, the field of post_commented is set to -1. For this nature, I first fetch the comment ID from comment_replied, query for this comment, and then get the post ID from it. Since the data is pre-processed and emitted in time order, we always can find the preceding comment from which we get post ID. **(comment_id): { timestamp, user_id, post_id, last modified timestamp }**

2. Initialize a set for the post of post ID if it does not already exist. This set is for checking the size of comments of a post. Then store the comment ID into this set. **(P:(post_id)): { comment_id }**

3. Initialize a set for the post of post ID if not existed. This set is for checking the size of unique commenters of a post. Then store the commenter ID (user ID) into this set. **(Commenters:(post ID)): { user_id }**

To store and keep track of information of posts and comments, we now have two hashes, posts and comments. Comments hashes are linked directly to their replied post. Each post with its replied comments forms a tree with only one-level depth of children.

A tree is an undirected graph in which any two vertices are connected by exactly one path. Solving Query 1 boiled down to solving a tree problem using Redis-installed data structures. In this section, we first decompose the social graph into several sets and sorted sets, which are data structures natively supported in Redis. In the rest of this section, we design an algorithm to solving this query using the data structures mentioned earlier.

Algorithm Design

After processing the incoming data and storing each of it into Redis, this triggers the computing ranking algorithm. This algorithm is triggered for each data.

The pseudocode for the algorithm is shown in Figure 7.

After getting all posts from Redis, we check the contributions of each post (lines 2-14). Namely, for the timestamp of the current processed data as CT, if 24 hours (multiple 24 hours) have elapsed since last modification of that post, decrement score(s) accordingly (for every 24 hours, decrease 1 point). If the score becomes 0 or less, remove Post:(post ID), and post ID from RANKING. If score is still greater than 0, check if the post's own score is still greater than 0. For this matter, we get the timestamp from Post:(post ID) as PostTime (PT), check the time difference of PT and CT is equal to or greater than 10 days.

After checking activeness of posts, we check contributions of comments in the remaining posts (lines 15-26). If the difference between the timestamp of the comment and the timestamp

of the current processed data is equal to or greater than 10 days, remove Comment:(comment ID), and comment ID from Post:(post ID), decrement the score of the post comment replied in RANKING. Otherwise, decrement the score of the post comment replied in RANKING.

Using the build-in method of Redis sorted set called ZRANK, we can get the top-3 posts stored in formate of <top_1_post_id, top_2_post_id, top_3_post_id >. Insert the new computed rank into RANKING_STORE. If successful, we get a new top-3 ranking, output it in the required format, and remove the last ranking. Thereafter, begin processing the next data.

Algorithm 1: First Query of DEBS 2016 Grand Challenge

```
1  posts ←— RANKING
2  for post in posts do
3      check post contributions
4      if 24 hours elapse then
5          decrement post score
6      end
7      if score < 0 then
8          remove post related data
9      else
10         if post own score > 0 then
11             decrement score
12         end
13     end
14 end
15 posts ←— RANKING
16 for post in posts do
17     comments ←— P : (post,d)
18     for comment in comments do
19         check comment contributions
20         if current ts − comment ts >= 0 then
21             remove Comment related data
22         else
23             decrement post score
24         end
25     end
26 end
27 new ranking ←— RANKING
28 compare new ranking with last ranking:
29     if they are different then
30         output ranking
31     end
```

Fig 7. Query 1 Algorithm

IV. EXPERIMENTAL EVALUATION

In this section, we present the performance analysis results of our solution using Redis as the storage system. In our experiments, we use a the dataset provided with the LDBC Social Network Benchmark.

Experiment configurations

Experiments are conducted on a single machine as stipulated by the challenge problem. We installed the in-memory key-value storage system Redis[14] on the machine, and installed Python 3.5 and redis-py 2.4.9 as the scripting language for the Redis APIs. The detailed environmental settings are listed in Table 3.

Type	Model or performance metrics
CPU number	Intel(R) Core(TM)i7-4710MQ CPU @2.5 GHz
Memory size	8 G
Disk size	100 G
Operation system	Ubuntu 14.04.3
Software	Python 3.5, Redis 3.2.0, redis-py 2.4.9, pyzmq 15.3.0

Tab 3. The Experimental Environmental Settings

Performance metrics

In Table 4, we list three most important metrics in our experiment.

Memory usage is a critical component of our algorithm performance. If the used_memory exceeds the total available system memory (8 GB in our experiment), the OS will start swapping old or unused sections of memory. Every swapped section is written to disk, severely impacting performance. Writing or reading from disk is up to 5 orders of magnitude slower than writing or reading from memory (0.1 μs for memory vs. 10 ms for disk) [16].

Latency is the measurement of the time it takes between when input data triggers the algorithm and when the system finishes computing the new ranking. Tracking latency is the most direct way to detect changes in performance of our algorithm.

Tracking the throughput of post and comment data states processed is another critical metrics for the performance of our algorithm. We investigated this aspect by measuring the number of post and comment data states processed per second.

Name	Description
used_memory	Total number of bytes allocated by Redis
instantaneous_ops_per_sec	Total number of states processed per second
latency	Average time for the Redis server to compute a new ranking

Tab 4. The Performance Metrics Settings

Experiment Results

Memory:

This experiment that we ran tested the memory usage of our systems when the system is processing the data. The results are shown in Figure 8, where the x-axis represents the memory usage in MB, and y-axis is the number of nodes stored in Redis.

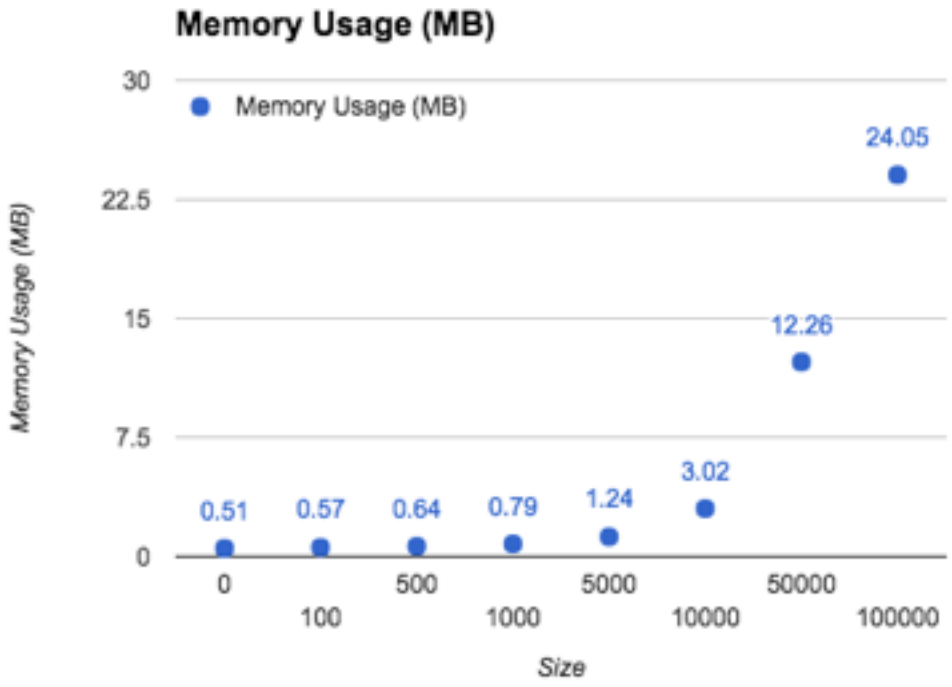

Fig 8. Memory Usage of Using Query 1 Algorithm

In this experiment, we used data of size up to 1 million nodes (with posts and comments). The initial memory usage for Redis is 0.57 MB. As the size of nodes in Redis increases, memory usage of Redis per node does not increase.

Latency:

n this experiment, we experimented how increasing the size of the nodes stored in Redis impacts the performance of our system.

Figure 9 show the results of latency of our system when Redis stored different number of nodes. In the figure, the x-axis represents the number of nodes stored at Redis, and y-axis is the latency in second.

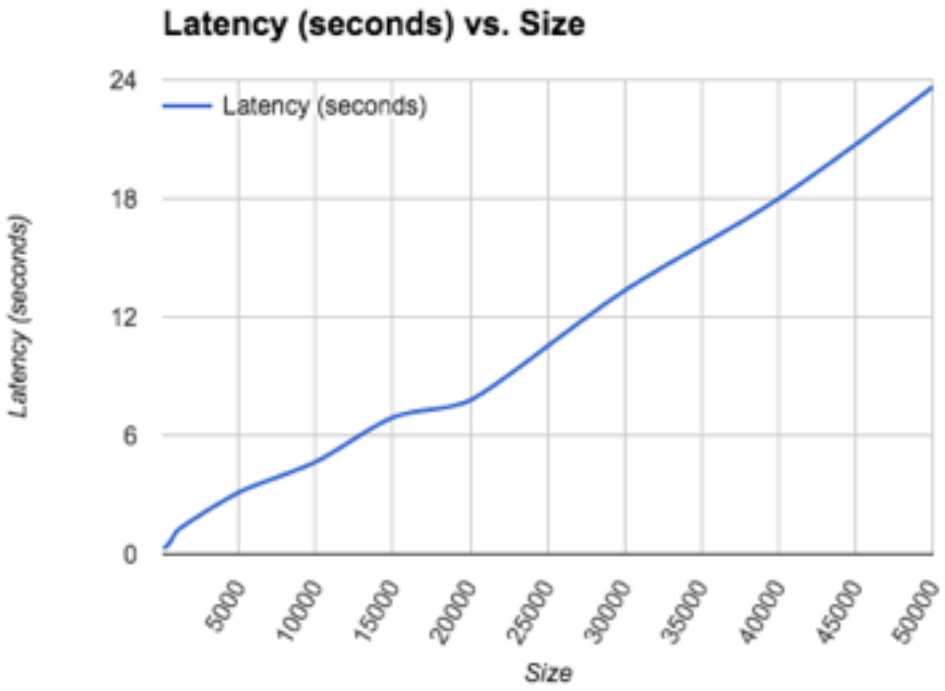

Fig 9. Latency of Output Using Query 1 Algorithm

In this experiment, we used data of size up to 50000 nodes (with posts and comments). As the size of nodes in Redis increases, the latency of each output increases linearly. In a constrained environment, our system can solve problems up to size 20,000 without any noticeable degradation but beyond that the system will begin to slow down.

Throughput:

In this experiment, we determined the throughput of our system.

Figure 10 show the results of throughput of our system when Redis stored different number of nodes. In the figure, the x-axis represents the number of nodes stored at Redis, and y-axis is the number of data states processed by Redis per second.

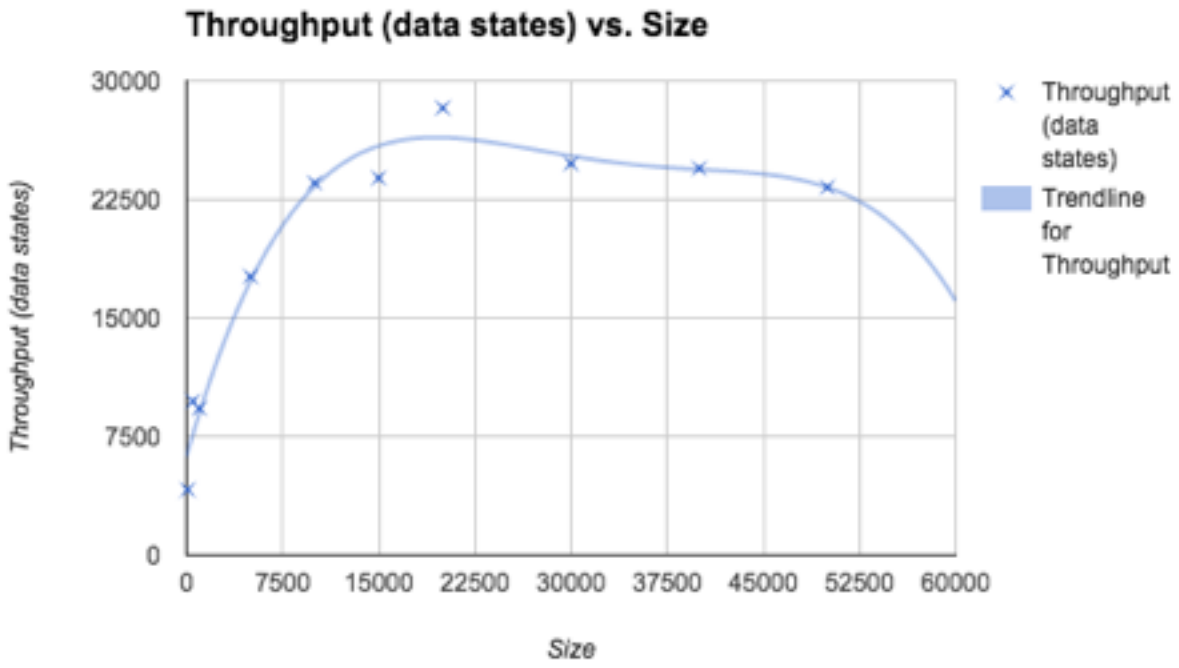

Fig 10. Throughput of Output Using Query 1 Algorithm

The surprising result shown in Figure 10 is that the throughput of our system first goes up from around 4,000 states per second to 28,000 states per second, as the number of nodes stored in Redis increases from 0 to 20,000. We find out that this result is due to Publish-Subscribe model we designed for sorting and managing the input stream data, which becomes the bottleneck when the processing speed of Redis is faster than that of the Publish-Subscribe model. It is therefore important to design a new publish-subscribe model with higher throughput.

In this diagram, we can clearly see that throughput goes up when the number of nodes increases. According to the results shown in the figure, we find that throughput does go down when the number of nodes stored at Redis exceeds 20,000.

Analysis of results

A key conclusion that we can draw from these results is that our algorithm is a feasible solution and it works efficiently when the size of community is smaller than 20,000. However, as shown in the results:

1) The memory usage will increase linearly with increases in size of nodes in a graph

2) The latency will increase with increases in size of the community

3) The throughput will reach its peak and decrease when the size of the nodes stored in Redis exceeds 20,000.

A shortcoming of the experiments is that the Publish-Subscribe model is still not fast enough to draw strong conclusions and more research is still needed. However, the initial results for the efficiency of our system are promising. In future work, we need to design a new Publish-Subscribe model with higher throughput.

Future work

The primary area for improvement is more robust experimentation and a more efficient model for data sorting. In future work, not only do we need to design a new Publish-Subscribe model with higher throughput, but also experiment it on single-board computer such like

beaglebone black and raspberry pi with contrained CPU and memory resources, which can impact Redis throughput, latency and persistency.

V. RELATED WORK

Designing a graph processing systems for analyzing a social network is a subject, which can be investigated from many different angles, and researchers in social network companies have produced a variety of interesting ideas in this space. In this thesis, we only provide an online solution that works with a stand-alone machine (requirement of DEBS 2016 Grand Challenge). But a number of works have looked at this problem with a lager scale of data set in both online and offline fashion.

Pinterest, one of the largest photo sharing websites, had 53.3 million uniques in March of 2015 [17]. On Pinterest, every screen of the user interface performs a query to see if a board or user is already followed. Abhi Khune, Engineering Manager at Pinterest, used Redis to scale up Pinterest and help analyze its giant social graph [18] .

Bronson et al. introduced a simple data model and API tailored for serving the social graph, and its implementation TAO, a geographically distributed data store, to meet the demand of a billion reads and millions of writes each second from Facebook [19].

Twitter introduced their distributed, fault-tolerant graph database, FlockDB to support goals of a high rate of add/update/remove operations, potentially complex set of arithmetic queries, paging through query result sets containing millions of entries, horizontal scaling including replication, online data migration, and etc [20].

During the time of designing and implementing our system, other DEBS 2016 Grand Challenge participants were preparing and creating their own solutions to this challenge. In June 2016, the ACM DEBS committee published the best papers of this year's competition. We compare our solution to two of these solutions.

In the first paper, authors use WSO2 Complex Event Processor (WSO2 CEP) to solve the challenge. WSO2 CEP is a light weight, easy-to-use, Complex Event Processor (CEP) which is available as an open source software under Apache Software License v2.0. To compute the top-3 ranking, the architecture has a data loader that first reads data from posts and comments files. The data loaders act as a producer while the event-ordering thread are consumers for the emitted data. This thread gets the data and order them based on their time stamp. The sorted data emitted by event-ordering system goes to the processing threads that are responsible for executing the query logic and writing the final output into files. With this architecture they can process 90,000 events per second with a mean latency of 6 ms for query 1 [21].

However, there are two noticeable limitations. It uses more memory. In the paper, authors employ the time window to determine the active posts and comments so that all stale and irrelevant posts and comments remains in the systems even if they become inactive. Even if no deletion operations on the graph improves performance, but as the size of graph increases the memory usage by the system will exceed the limit. Second, the capacity of containers for posts or comments needs to be predefined due to the in-efficiency of re-sizing maps in Java. In real life cases, the number of posts and comments generated would be large and unpredictable, which leads to expensive map re-sizing operation and impacts the performance in unexpected way.

In the second paper, authors propose an innovative framework, GraphCEP, to solve the challenge by combining graph processing frameworks and Complex Event Processor.

GraphCEP consists of a split–process–merge architecture. The splitter is in charge of finding consistent splitting points in the incoming event streams. The operator instances run in parallel and use an interface to a full-fledged distributed graph processing system. Finally, a merger reorders the concurrently detected events from the operator instance into a deterministic order [22].

This solution shows interesting performance traits. The event latency for query 1 is not increasing over time. And throughput is very amazingly high (more than 150, 000 events per second). However, to fully utilize this framework, a company needs to hire a set of experts from domain, graph engines and event processing area. Moreover, they also need to invest a lot of money and time to study, tune and make adaptions to this framework, which is not an economic and easy to use solution to most of the clients. Further, even though it is a good try to bring together the graph processing frameworks and CEP, to what degree the compatibility between them is not stated in the paper and more work needs to be done to understand it.

In contrast, our solution using Redis as storage system empowers the analysis with simple structure and less memory usage though it has limitations for larger community sizes.

VI. CONCLUSIONS

A variety of frameworks exist to support analytics for social networks. However, when these analytics must be performed online and in relatively resource-constrained environments, the choice of the framework and the strategy used make significant difference to the outcomes and the timeliness at which results are obtained. This thesis provides such insights and validates the claims in the context of a real-world scenario drawn from the ACM DEBS 2016 Challenge Problem.

To that end, this thesis first summarizes the development of online social networks and graph theory. Then, it introduces the DEBS 2016 Grand Challenge and the requirements of its first query. The thesis proposes an efficient solution to address this challenge. To fully utilize Redis and simplify the complexity of the problem, our approach decomposes the graph structure by employing Redis built-in data structures set, sorted set, and hash. The evaluation results show clearly that our solution can solve the challenge efficiently. However, we found that when the number of nodes stored in Redis exceeds 20,000, the latency increases dramatically and throughput begins to drop.

Overall, the results warrant future research into social graph based problem and its effectiveness as an online graph analysis framework.

REFERENCES

1. D.A. Williamson, Social Network Demographics and Usage, 2010. Online available: http://www.emarketer.com/Reports/All/Emarketer_2000644.aspx

2. Facebook: 10 years of social networking, in numbers. Online available: http://www.theguardian.com/news/datablog/2014/feb/04/facebook-in-numbers-statistics

3. Number of monthly active Facebook users worldwide as of 1st quarter 2016 (in millions). Online available: http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

4. GlobalWebIndex's quarterly report on the latest trends in social networking

5. F. Alvanaki et al. See what's enblogue: real-time emergent topic identification in social media. In EDBT, pages 336–347, 2012.

6. DEBS 2016 The 10th ACM International Conference on Distributed and Event-Based Systems. Online available: http://www.ics.uci.edu/~debs2016/call-grand-challenge.html

7. LDBC. Online available: http://www.ics.uci.edu/~debs2016/call-grand-challenge.html

8. How Much Data Is Generated Every Minute On Social Media? Online available: http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/

9. Americans' Internet Access: 2000-2015 As internet use nears saturation for some groups, a look at patterns of adoption

10. Otte, Evelien; Rousseau, Ronald (2002). "Social network analysis: a powerful strategy, also for the information sciences". Journal of Information Science 28: 441–453. doi: 10.1177/016555150202800601. Retrieved 2015-03-23.

11. Graph Theory, Encyclopedia Britannica

12. Graph structure in the web. Online available: http://www9.org/w9cdrom/160/160.html

13. Pregel: A System for Large-Scale Graph Processing. Online available: http://www.dcs.bbk.ac.uk/~dell/teaching/cc/paper/sigmod10/p135-malewicz.pdf

14. Scaling Apache Giraph to a trillion edges. Online available: https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

15. The GraphX Graph Processing System. Online available: http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project21_report_ver2.pdf

16. Redis. Online available: http://redis.io

17. 50 Shades of Graph: How Graph Databases Are Transforming Online Dating. Online available: http://www.forbes.com/forbes/welcome/

18. Building a follower model from scratch. Online available: https://engineering.pinterest.com/blog/building-follower-model-scratch

19. TAO: Facebook's Distributed Data Store for the Social Graph. Online available: https://cs.uwaterloo.ca/~brecht/courses/854-Emerging-2014/readings/data-store/tao-facebook-distributed-datastore-atc-2013.pdf

20. Introducing FlockDB. Online available: https://blog.twitter.com/2010/introducing-flockdb

21. Malith Jayasinghe, Anoukh Jayawardena, Bhagya Rupasinghe, Miyuru Dayarathna, Srinath Perera, Sriskandarajah Suhothayan, Isuru Perera, Grand Challenge: Continuous Analytics on Graph Data Streams using WSO2 Complex Event Processor

22. Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, Kurt Rothermel, Grand Challenge: GraphCEP - Real-time Data Analytics Using Parallel Complex Event and Graph Processing