REPLETE: A Realtime Personalized Search Engine for Tweets

by

Akshay Vivek Choche

(Under the direction of Lakshmish Ramaswamy)

Abstract

Powerful search capabilities are fundamentally important for popular micro-blog platforms such as Twitter. While recently there has been some work aimed at enhancing the scalability of search, very few existing techniques incorporate personalization into their search and ranking process. However, personalization also raises new scalability challenges because it adversely impacts the effectiveness of caching and other performance enhancement techniques. We present REPLETE: a scalable, real-time search framework for micro-blogs that incorporates personalization by analyzing the follower relationships in Twitter. Three unique aspects characterize the REPLETE framework. First, our technique takes into account both the search parameters as well as the distances of the follower relationships when ranking the search results. Second, we design an in-memory temporal index that preserves the temporal significance of tweets and helps speed up query evaluation. Third, to ensure overall quality and timeliness of search results, we identify important tweets and prioritize their indexing.

Index words: Personalization, Real-time Search, Indexing Tweets

REPLETE: A Realtime Personalized Search Engine for Tweets

by

Akshay Vivek Choche

B.E., University of Mumbai, 2008

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2013

REPLETE: A Realtime Personalized Search Engine for Tweets

by

Akshay Vivek Choche

Approved:

Major Professor: Lakshmish Ramaswamy

Committee: John A. Miller
Eileen Kraemer

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2013

REPLETE: A Realtime Personalized Search Engine for Tweets

Akshay Vivek Choche

April 26, 2013

Dedications

This work is dedicated to my loving and caring parents.

Acknowledgments

I would cherish the past three years I spent here at UGA. It was a big learning curve for me as I joined the Masters program in the Computer Science Department, but, my advisor Dr. Lakshmish Ramaswamy has always helped me, not only in my professional advancement but also in my personal life. I would like to take this opportunity to thank Dr. Lakshmish Ramaswamy for continuously motivating me through these years. I would also like to thank Dr. Eileen Kraemer and Dr. John A. Miller for providing constant inputs, guidance and motivation through out my work.

Finally I would also like to thank Dr. Jessica C. Kissinger for giving me an opportunity to work as a Graduate Assistant in her lab.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Recent years have seen a tremendous growth in popularity of micro-blogs. Twitter – a social networking (SN) and microblogging service – has emerged as a popular medium for users all around the world to disseminate and receive updates on live events and current issues. Recently Twitter reported it has more than 500 million active users and processes more than 55 million tweets each day [24][10], which stands as a testimony to its enormous popularity.

Powerful search techniques are indispensable to micro-blogging platforms such as Twitter as they seek to establish themselves within the highly competitive online social media space [11]. However, searching on SN-based microblog platforms is radically different than searching content on the World Wide Web, and it poses several unique challenges. First and foremost, because micro-blogs (henceforth referred to as *tweets*) are very short pieces of text (maximum of 140 characters), heavyweight text analytic techniques such as Term Frequency-Inverse Document Frequency(TF-IDF) algorithms are not very effective. Second, as Twitter is most commonly used for sharing updates about current events and issues, freshness and timeliness of search results becomes critically important. In other words, the results returned to a user should include most recent matching tweets. Third, since Twitter

also incorporates SN features, it is important to take SN-relationships into account during the search and ranking processes. Fourth, the indexing, retrieval and ranking mechanisms needs to be extremely scalable considering that Twitter currently has to ingest tweets at a rate of 4000 tweets per second.

Unfortunately, very few existing works comprehensively address the above challenges. Twitter recently published its real-time search engine called Earlybird [2]. Earlybird considers two major factors in producing its search results, namely, syntactic match between the query and the tweets and the temporal recency of matching tweets. Works such as Tweet Index (TI) [1] focus mainly on the efficiency of indexing and searching and not on the quality of search results. To the best of our knowledge, none of the existing works incorporate *personalization* into their search and ranking strategies. In other words, the existing search and ranking techniques do not take into account the relationships between the user issuing the query and the users posting the tweets. Thus, the social network aspect of Twitter is completely ignored during the search and ranking processes.

In this thesis, we argue that it is important to personalize search results based on SN relationships. The follower relationship in Twitter is often an indicator of commonality of interests and opinions. Thus, it is natural for a user issuing a query to expect higher rankings to matching tweets from users that she is transitively following even when those tweets are not the most recent ones. However, personalization also raises new scalability and performance issues. First, personalization requires additional analytics which are often computationally intensive. Second, personalization reduces reusability of search results thereby adversely impacting the efficacy of proven efficiency enhancement techniques such as caching.

This thesis presents *REPLETE* – a scalable, real-time search framework for micro-blogs

that incorporates personalization by analyzing the follower relationships on Twitter. The REPLETE framework incorporates three novel features.

- First, we combine three major tweet relevance factors, namely, degree of syntactic match, distance of the follower relationships, and the temporal recency of tweets into a unique tweet ranking metric. This metric is used for generating personalized search results for each query.

- Second, we design an in-memory temporal index that preserves the temporal significance of tweets. This index enhances the performance of query evaluation thereby effectively ameliorating the scalability challenge posed by personalization.

- Third, we present a priority-based indexing scheme that identifies tweets that are likely to have significant impact on the overall quality of search results, and accords these significant tweets higher indexing priorities.

We have performed a number of experiments to study the costs and benefits of the REPLETE framework. The results demonstrate that our techniques incorporate personalization into the tweet search and ranking processes at very reasonable costs.

The rest of the thesis is organized as follows. Chapter 2 is a literature review, Chapter 3 motivates the problem of personalized search of micro-blogs. Chapter 4 provides an overview of the architecture of the REPLETE framework. Chapter 5 and 6 describe the personalized query evaluation engine and scalable indexing respectively. Chapter 7 discusses the experimental evaluation. We discuss related work in Chapter 8, and conclude our work in Chapter 9.

Chapter 2

Literature Review

The popularity of micro-blogging websites such as Twitter continues to grow at an exponential rate. Users are turning to it for information on current events. Twitter is being used for keeping up with news [28] [29], finding quick answers for random questions, getting reviews about new movies, and keeping up with how a sports team is performing in current season, etc. The list is endless. A user simply issues a query and he will receive the latest tweets on that query. Recent reports indicate that Twitter has more than 500 million active users and processes more than 55 million tweets each day [24]. Powerful real-time search capabilities are fundamentally important for such micro-blog platforms. While recently there has been some work aimed at enhancing the scalability of micro-blog search, we found that very few existing techniques that incorporate personalization into their search and ranking processes. We argue that because Twitter is a social network, it is essential to personalize search results to each user. However, personalization also raises new challenges because it reduces the reusability of search results and hence server side caching for search results becomes less effective. Personalization also raises scalability issues since it requires additional analytics, which are often computationally intensive.

Before discussing personalization, we first discuss real-time search. For real-time search a key requirement is the ability to consume new content rapidly and make it searchable immediately, and at the same time to support a query evaluation engine that has low latency and high throughput. This implies the duration between content creation and its availability to be searchable should be minimized. Now, looking at this from a database perspective we can think about creating up-to-date indices for each content and inserting these pieces of content into the database. We can then measure the search quality by the time gap between insertion time and availability of index. However we believe it is not feasible to implement real-time searching over micro-blogs using this approach since it poses too many new challenges for micro-blogging systems, where large number of users are uploading their micro-blogs or tweets simultaneously. Such a system would either fail to index this large number of tweets or fall short of providing scalable indexing service.

Let us look at some salient requirements of real-time search, that are highly desired by users [2].

- **Low-latency, high-throughput query evaluation.** For any search engine to be successful it must be able to cope with large query volumes. Users are generally impatient when it comes to web search; they want the results to be made available as quickly as possible. Hence this feature is one of the most important features that a search engine should provide.

- **Data availability with least amount of latency.** For a real-time search engine one of the prime requirements is that the data be available as soon as they are created. This requirement is not so stringent when it comes to web search engines, where indexing is considered a batch operation. Typically, web search engines make use of web crawlers to get new content and index this content in batch cycles. Modern web-crawlers do achieve

high throughput; however, it is not expected that crawled content be available for search immediately. A delay of a few minutes, hours or a day is acceptable depending on the content being indexed. However, when it comes to real-time search, content usually arrives at a very high rate, and often in sudden spikes. One of the example would be the *"passing away of Steve Jobs"*. A real-time search engine should be designed in such a way that it is immune to a sudden increases in the arrival rate of content and performs efficiently under heavy loads.

- **Temporal significance of content.** For a web search engine, temporal significance of content sometimes has very little or no significance when answering a query submitted by a user. A web search engine like Google [12] typically pays more attention on the number of times the content was viewed (page hits, in Google terminology) when ranking a query result. In contrast, in a real-time search engine more significance is given to the timestamp carried by the content. It is highly desirable that the most recent content be ranked at a higher position when compared to content that was submitted several days back. One way to perform ranking would be to arrange the content in the search result in reverse chronological order.

For a search engine to be successful it should not only be able to answer queries with low latency but it should have a solid ranking mechanism in order to deliver quality results. A search engine must take into account as many factors as possible when ranking results. This could include factors ranging from the reliability of the author submitting the tweet *(was the content published by a reputed resource such as CNN or BBC when it comes to news reports)*, relevance of the content with respect to the query, to one of the most important factors for a real time search engine, temporal significance of the content. This brings up a new topic of discussion, personalization. We will discuss personalization in a later section of this chapter. First we present literature reviews that were performed as part of this research.

This includes papers on ranking approaches used for Microblogs, and two search engines targeted towards Twitter, Twitter's own search engine Earlybird [2] and a search engine called Tweet Index (TI) [1].

First some terminology. Twitter is a micro-blogging website that allows registered users to post short (140 character) messages called tweets. Twitter also allows registered users to follow other users. When user U_A follows user U_B all the tweets posted by user U_B are available to user U_A. Further, we can visualize this follower relationship as a directed graph, in which users act as nodes and edges represent the follower relationship. In our example above the directed graph would have an edge from U_A to U_B. Nagmoti et al. [15] mention that ranking tweets as a search result for queries is a challenging task because of the sheer number of tweets generated everyday. Further, the short length make this task even more challenging because it is difficult to use traditional content-based relevance ranking algorithms such as Term Frequency-Inverse Document Frequency (TF-IDF) [30]. Apart from that authors tend to avoid posting links to articles since that would consume most of the space, hence we cannot use traditional link based ranking algorithms such as PageRank [7]. Presently Twitter only provides keyword matching based search results, where tweets are ranked in reverse chronological order. Such a ranking strategy does not guarantee that the most interesting tweets appear on the top, given the sheer number of tweets that are generated. Hence in order to over come these drawbacks the authors present a new ranking system that incorporates social networking properties of the authors of microblogs in addition to properties of the microblog itself.

Nagmoti et al. use a two stage ranking strategy [8], in which they use the existing Twitter search engine to get the top-K tweets related to a query ranked in reverse chronological order and then re-rank this result based on various factors such as TweetRank (TR), FollowerRank

(FR), LengthRank, and URLRank, defined below.

- **TweetRank:** The tweet rank for an author a is defined as:

$$TR(a) = N(a) \tag{2.1}$$

  where *N(a)* the total number of tweets posted by a so far. The authors believe that tweets from active publishers are more valuable information sources than those from inactive publishers.

- **FollwerRank:** The FollowerRank of an author a is defined as:

$$FR(a) = \frac{i(a)}{i(a) + o(a)} \tag{2.2}$$

  with $i(a)$ being the indegree of a i.e., the number of followers a has and $o(a)$ being the outdegree. It is believed that an author is more influential if he has more followers, in which case users are interested in knowing what this particular author has to say.

- **LengthRank:** The LengthRank of a tweet t with respect to a query q is defined as:

$$f_{LR}(t, q) = \frac{l(t)}{\max_{s \in T_q^k} l(s)} \tag{2.3}$$

  with T_q^k the set of the top-K tweets returned for query q and $l(t)$ and $l(s)$ the length of tweet t and s respectively. Nagmoti et al. believe if the tweet contains more information then it should be ranked at a higher position.

- **URLRank:** : Let t be a tweet and q a query, then the URLRank of t with respect to q is defined as:

$$f_{UR}(t,q) = \begin{cases} c & \textbf{if t contains URL} \\ 0 & \textbf{otherwise} \end{cases} \tag{2.4}$$

The Final rank for the tweet is computed by summing up the above-mentioned factors. One of the drawbacks of such a ranking system is that it gives equal weight to each of the factors, we believe that URLRank should not be given equal emphasis. The idea behind URLRank is that if a tweet does not have URL embedded in it, then it should be given lower priority, which is not always the case. We believe instead of just adding these factor the authors should compute a weighted sum, giving more weight to factors such as TweetRank and FollowerRank as compared to LengthRank and giving the lowest amount of weight to the URLRank.

In existing real time search engines such as Earlybird [2] and TI [1], one of the major drawbacks is that they do not offer any personalization to users. Personalization allows the result set for a query to be tailored to preferences of individual users. To better explain the concept of personalization with respect to Twitter, consider the following scenario in which we have a user U_1 who is a book enthusiast and follows a number of authors on Twitter, and we have user U_2 who is a sports enthusiast who likes watching soccer and follows a number of soccer players. If both the users issue a query "Stephen King" it is highly possible that Twitter would return tweets related to Stephen King the author due to his enormous popularity. This result will not please U_2 who was actually searching for Stephen King the soccer star. This example illustrates that there is a need to implement personalization on Twitter. Earlybird, the current real time search engine used by Twitter considers only two major factors in producing search results: the syntactic match between the query and the tweets and the temporal significance of the matching tweets.

Scholars from the National University of Singapore proposed a system called Tweet Index (TI) [1], which implements real time search functionality over tweets. TI uses a partial indexing strategy in which it classifies incoming tweets as Significant or Insignificant based on different criteria. The core idea behind identifying a Significant tweet is the probability of the tweets appearing in future search results. These tweets are indexed immediately while Insignificant tweets are added into a log file where they are indexed offline, in batch processing fashion. By performing partial indexing TI is able to save some portion of its computation power, since it does not have to index each and every incoming tweet immediately. This saved computation power can be used by the query evaluation engine to provide high throughput and low latency. However, a drawback of this system is that the use of a disk based indexing system requires time consuming I/O operations while indexing as well as during query evaluation. A more efficient solution to this would be to use an in-memory index, which would help reduce these I/O operations. Although maintaining an in-memory index requires a bigger memory, the speed gain should outweigh the cost of extra memory. The ranking system used by TI is more sophisticated than Earlybird, since it not only takes into account the temporal significance of a tweet, but it also takes into consideration the author issuing the tweet and whether the tweet is part of a current trending topic on Twitter. However TI does not offer any personalization, which as mentioned earlier is a highly desired functionality. Further, this paper does not provide enough details about the evaluation performed on the system. Although they speak about the query evaluation time they fail to test the system under heavy loads, so we have little information about the scalability of the system.

Chapter 3

Background and Motivation

In this section, we motivate our work by highlighting the need for a personalized real-time search engine for tweets. We point out the difference between searching over the World Wide Web (WWW) and searching over a Microblogs, and why we cannot use searching and ranking methods used for WWW when searching and ranking in a Microblog. We first point out the peculiarity of a Microblogging system such as Twitter. Twitter is a Microblogging website which is currently leading the highly competitive online social media race [26]. Twitter has evolved into a medium where users share information about current events and issues, and other users can get these updates [27]. Twitter allows registered users to publish microblogs called tweets. These tweets can be up to 140 characters long. Twitter allows registered users to follow other users whom they find interesting. This phenomenon leads to formation of follower relationships in Twitter, which can be represented by a directed graph. Nodes represent users and edges represent the *"follower relationships"*.

Searching over Microblogs differs from searching over the WWW in many ways. Information retrieval over the web uses Term Frequency-Inverse Document Frequency(TF-IDF) extensively, even though TF-IDF is a relatively old weighting scheme, being simple and ef-

fective has made it a popular beginning point for many sophisticated new algorithms [31]. However, when it comes to the Microblogging domain, the shortness of tweets (140 characters) makes it difficult to use this proven text analytics technique. Another common technique used for web search and ranking is a family of linked based algorithms, in which importance is given to content having more direct links to it [36] [37]. It is not possible to use such techniques for tweets, since inherently tweets are not linked to other tweets. Further, most tweets do not contain any links in them, due to the shortness of tweets. Another important factor for a real-time search is that the arrival rate of content is not fixed or constant. Most of the time new content arrives in bursts, quite analagous to *flash crowds* [32] [33]. A real-time search engine should be able to handle these spikes and make content searchable by indexing it with minimum delay. The ability to handle flash crowds[1] is one of the properties that makes real-time search engines different from traditional web search engines. In a traditional web search engine the indexing operation is considered a batch operation, done in an offline fashion. New content is discovered using web crawlers [34], which typically crawl the web for new content after a fixed duration of time. It is not necessary that this new content be indexed immediately. A delay of a few hours, days or even a week is generally acceptable depending upon the type of content being indexed. However, this technique cannot be applied to a real-time search engine over a microblog, where tweets are generated at a very high rate, and where it is desirable that every tweet be indexed and be available for searching as soon as possible.

For a traditional web search engine an index used for speeding up query evaluation is generally a static read-only index. This index is updated in batch operations when web crawlers discover new content. However, when considering real-time search over microblogs, the index used should be able to support concurrent reads and writes, in order to handle

[1]Swarms of users on a computer network that appear, then disappear, in a flash.

the large number of tweets published by users, and the large number of queries submitted to Twitter. Finally, traditional web search engines give minimum significance to the temporal significance of content. Web search engines like Google place more emphasis on the number of times some content is visited, or the number of links that point to particular content, instead of the temporal significance of the content (content creation time) [35]. However, for a real-time search engine over microblogs, significant importance needs to be given to the content's timestamp when answering queries. The paragraph below highlights the need for *personalization*, another feature that is desired from a real-time search engine over microblog. To the best of our knowledge, none of the existing work incorporates *personalization* into their search and ranking strategy.

Here we highlight the need for personalization of search results by taking into account the social networking (SN) relationships. The shortness of tweets makes it difficult to come up with meaningful search and ranking mechanisms. Thus, it is necessary to use additional contextual information when ranking search results on Twitter. As mentioned in the introduction, follower relationships in Twitter often embody commonalities in interests and opinions. When a user U_1 chooses to follow another user U_2, it inherently signifies that U_1 is interested in the updates and opinions being posted by U_2. Transitively, U_1 is also likely to be interested in the tweets from users that U_2 is following thus, U_1 will be much more satisfied with the search results if matching tweets (tweets containing one or more query words) from users that she is directly or indirectly following are ranked high. A second benefit of using SN-relationships is in terms of search query disambiguation. Search queries are often ambiguous – they can have multiple meanings or they can refer to multiple entities or personalities. Consider the search query "Stephen King". It can either refer to the famous author or the famous soccer player. It is hard to discern which Stephen King the query is referring to. Now suppose the user who issued the query is following several authors on Twitter, it is

very likely that she is referring to the author Stephen King. On the other hand, if the user is following sports commentators, it is likely that he is referring to the soccer player. Incorporating SN-based personalization provides a natural way for query term disambiguation.

While personalization provides significant benefits, it also introduces new performance bottlenecks. First, the follower relationship graph in Twitter is massive. Running analytics on such a scale in real-time is extremely challenging. Second, personalization reduces the reusability of search results. Thus, server-side caching of search results becomes less effective. Hence, there is a need for techniques that carefully balance the tradeoff between search result personalization and system performance.

Chapter 4

System Architecture

In this chapter we outline the high-level architecture of REPLETE, then provide a detailed description of major components of the system. In the previous chapters we talked about the various issues and challenges involved in building a real-time search engine that supports personalization. In order to overcome these challenges and to build a scalable system we need to index tweets selectively rather than indexing each and every tweet that is posted. By doing this the system is able to reduce the overhead of indexing and thus becomes immune to the ever increasing number of tweets that are generated. This allows us to use the rest of the computation power to perform other relevant tasks such as personalizing the result for each user.

As illustrated in Figure 4.1, the REPLETE framework is composed of two major components – an *index engine (IE)* and a *personalized query evaluation engine (PQE)*. The personalized query evaluation engine is responsible for accepting user queries and returning personalized results to each user. The query evaluation engine is made up of six components 1) the Personalization engine, 2) Query Engine API, 3) Query Processor, and 4) Query Log are the dedicated components. The remaining two components, 5) Temporal index and 6)

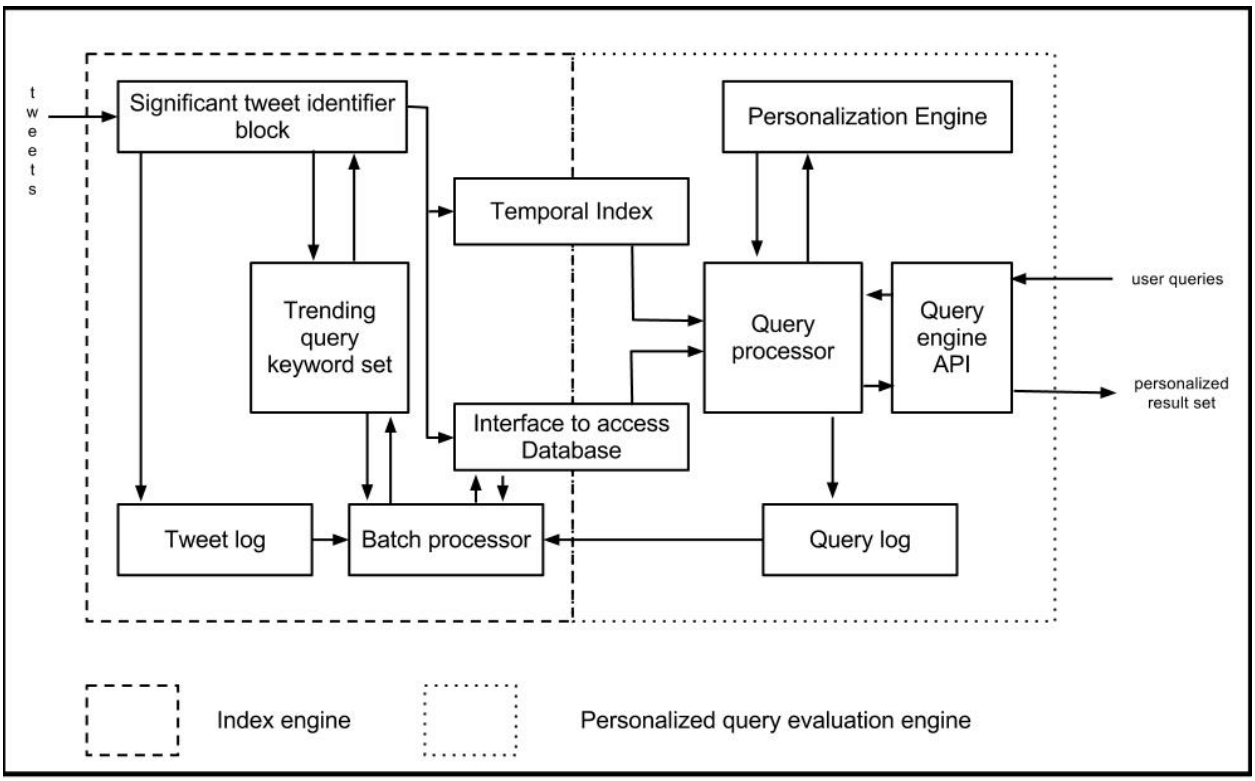

Figure 4.1: REPLETE System Architecture

Interface to access the database, which are shared with the Index engine. The Query Engine API is responsible for accepting user queries and returning personalized result sets of tweets back to the user. The query processor is responsible for finding important tweets that match search criteria given by the user. These tweets are then forwarded to the Personalization engine, which ranks the tweets based on factors such as user preferences, the temporal significance of the tweet, and the importance of the author posting the tweet. We also maintain a query log to keep track of queries submitted to the system. This query log is used to update the Trending query keyword set during each batch cycle, so that the system is in sync with current trends on Twitter and hence can better identify tweets that need to be indexed immediately. The Temporal index is an in-memory inverted index that is used

for retrieving tweets that match keywords in the query. Sometimes the keyword may not be present in the index, in which case we need to access the database to fetch the relevant tweet; this is achieved using the Interface to access the database. We explain the PQE in detail in *Chapter 5, Personalized Query Evaluation.*

The Index engine is responsible for selectively indexing the incoming tweets. It has six components: 1) Significant tweet identifier block, 2) Trending query keyword set, 3) Tweet log, and 4) Batch processor which are its dedicated components, while the remaining two components 5) Temporal index and 6) Interface to access the database, which are shared with the query evaluation engine. Our system performs partial indexing in that, it identifies significant tweets using the Significant tweet identifier block and indexes them immediately, while the unimportant tweets are forwarded to the Tweet log where they are stored temporarily and indexed in a batch cycle. The Trending query keyword set is used by the significant tweet identifier block to make a decision of whether to index the tweet immediately or forward it to the tweet log. In order for the system to be in sync with the current trends on Twitter it is necessary that we periodically update the Trending query keyword set. The Batch processing block refers to the Query log to perform this task. Apart from this, the Batch processing block is also responsible for periodically indexing the tweets present in the Tweet log. We explain the Index engine in detail in *Chapter 6, Scalable Indexing.*

Before discussing the novel features of the REPLETE framework, we present some terms used in the rest of the thesis.

Author Influence: The influence of an individual author in Twitter is defined as the ratio of the particular author's followers to the total number of currently active Twitter users[1]. The greater the number of followers, the higher the influence.

Super User: A user whose author influence score exceeds a system-defined threshold (Λ) is a super user.

Trending Query Keyword Set Ω: This set represents a set of keywords that are encountered relatively often in recent queries submitted to the query processing engine. The size of this set is system defined and the set is updated during each batch processing cycle (using the well-known Least Recently Used (LRU) replacement policy). The query log is used for updating the set Ω to replace less frequent keywords with more recent queries keywords from the query log.

[1]Users who publish tweets regularly

Chapter 5

Personalized Query Evaluation

In this section, we explain our personalized query evaluation engine(PQE) which operates in two phases [8]. First, as with most existing search engines [21], we use a simple keyword-based filter to retrieve tweets that are relevant to a given query. This phase is intentionally kept simple so as to filter-out large fractions of obviously-irrelevant tweets. The tweets that are deemed relevant are processed by the second phase, which generates personalized search results by using our novel personalized tweet ranking algorithm.

5.1 Personalized Tweet Ranking

One of the main challenges in ranking tweets is that several factors influence the relative importance of tweets. These factors have to be taken into account when ranking the tweets. In our work, we have identified three main factors that influence the relative significance of tweets to a given query. The first is the strength of the Social Networking (SN) relationship between the tweet's author and the user issuing the query. We call this the *personalization factor*. The second is the *temporal significance factor*, which capture the temporal relevance of a tweet to the query. The third is the *author influence* factor, which measures social

influence of a tweet's author. Each of these factors have very distinct significance. We combine these factors into a personalized tweet ranking function as follows.

$$\begin{aligned} CRS(Tw_k) =& w_1 * PS(Tw_k) + w_2 * TS(Tw_k) + w_3 * AIS(Tw_k) \\ & s.t. \quad w_1 + w_2 + w_3 = 1.0 \end{aligned} \tag{5.1}$$

In this function, $CRS(Tw_k)$ indicates the cumulative ranking score for tweet Tw_k, $PS(Tw_k)$ denotes the tweet's personalization score, $TS(Tw_k)$ denotes its temporal score and $AIS(Tw_k)$ indicates its author influence score. w_1, w_2, and w_3 are system-defined weight parameters that determine the relative importance of the different factors, which must sum to 1.0. We set these values to 0.33 in our system evaluation (i.e., all three factors receive equal importance). Although the weights used while computing the $CRS(Tw_k)$ are manually set, it is possible to optimize these using machine learning algorithms [16]. We now discuss our mechanisms for quantifying each of these factors.

Personalization Score (PS)**:** This score measures the *social affinity* of the user issuing the query to the tweet's author. To compute the social affinity, we model the follower relationship as an unweighted directed graph in which vertices correspond to users and edges correspond to the follower relationship between users. If user U_1 is currently following U_2, there will be a directed edge from U_1 to U_2.

There are several meaningful ways to quantify social affinity between users, including:

- inverse of the length of the shortest path between them
- number of non-overlapping paths between them

For REPLETE, we use the inverse of the length of the shortest path to quantify the personalization score. Our choice is influenced by (a) computational efficiency and (b) stability

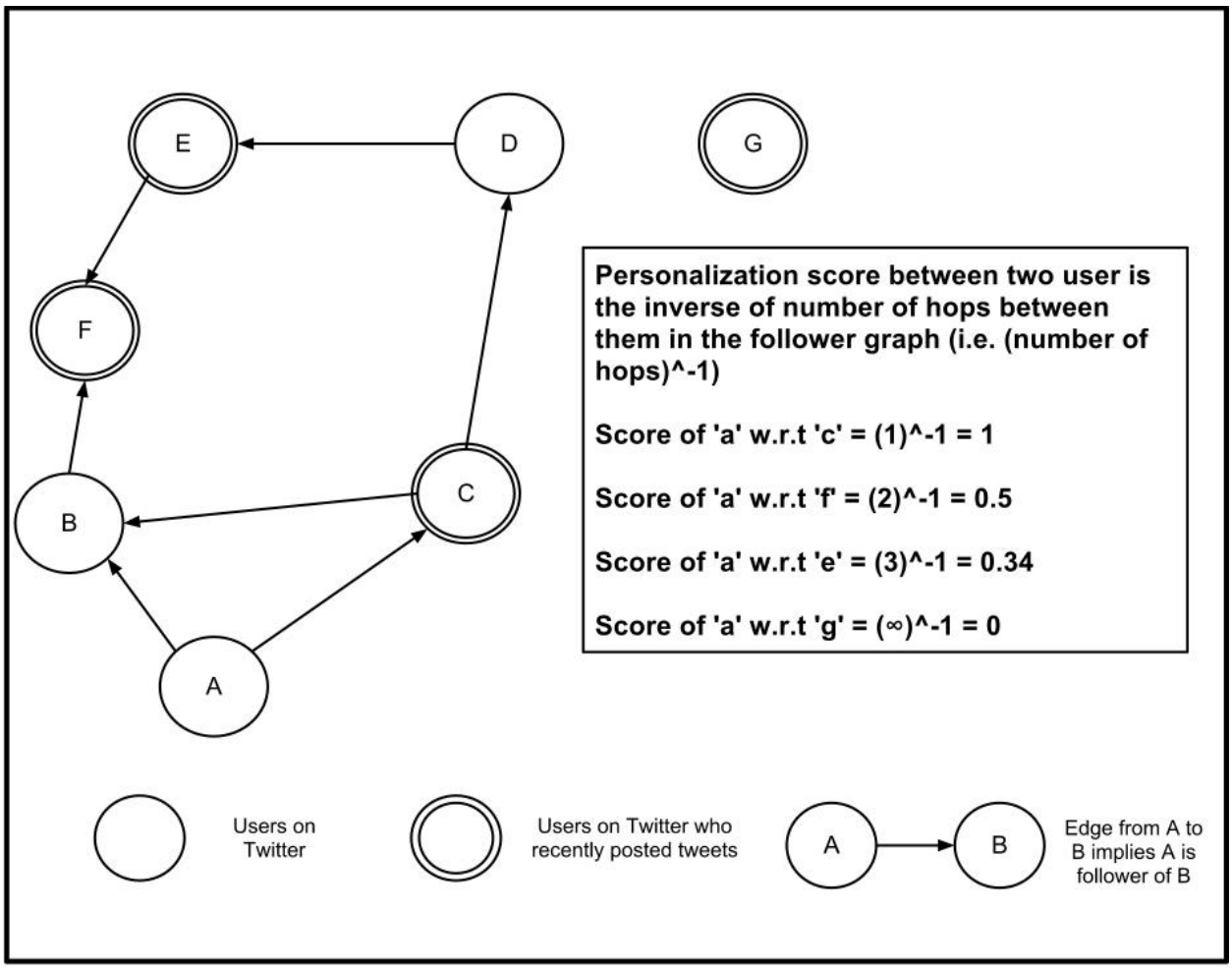

Figure 5.1: Personalization using relationship property.

of the parameter over time. Since the graph is unweighted, the personalization score is the inverse of the shortest number of hops between users in the directed follower graph. Figure 5.1 illustrates the personalization score computation on a hypothetical follower graph. Consider a scenario where A issues a particular query Q_A and users C, E, F, and G recently published tweets Tw_C, Tw_E, Tw_F, and Tw_G, respectively, containing keywords present in query Q_A. In such a scenario the result set obtained by PQE would contain these tweets and the personalization scores for each of these tweets would be as follows. The personalization

score of A with respect to C would be 1 because A is a direct follower of C. The personalization score of A with respect to F would be 0.5 because A is two hops away from F. The personalization score of A with respect E would be 0.33 since A is three hops away from E, whereas the personalization score of A with respect to G would be zero because A is neither a direct nor an indirect follower of G.

Temporal Score (TS)**:** The temporal score measures the temporal closeness of a tweet to a query. The rationale is to provide higher preference to more recent tweets. Suppose a tweet Tw_i was posted at time T_x. This tweet's temporal score with respect to query Q_j issued at time T_y is quantified as $\frac{1}{T_y - T_x + 1}$.

Author Influence Score (AIS)**:** The author influence score measures the influence of a tweet's author on the Twitter user community. The rationale is to give higher ratings to tweets from more popular Twitter users. The author influence score of a tweet is defined as the ratio of the number of followers of the tweet's author to the total number of currently active Twitter users. The greater the number of followers the higher the influence.

Ranking Algorithm: At a conceptual level, the ranking algorithm is quite simple – the CRS of each matching tweet is calculated and the tweets are ranked in decreasing order of their CRS values. However, this naive implementation does not scale well because the personalization score computation will quickly become a bottleneck. Thus, we need smarter ways to generate the ranking. The REPLETE system incorporates an approximation strategy to accelerate the ranking process. Our algorithm proceeds as follows. First, we select only tweets whose temporal score exceeds a certain threshold μ. Computing this list is computationally scalable because essentially we are selecting tweets that are more recent than a certain time-point. Our temporal index (described in the next section) further speeds up the selection process. We compute the PS and the AIS values only for the tweets whose

TS value exceeds μ. Once these values are available, we generate the ranked list. Since PS values (which are the most expensive) are computed only for a limited number of tweets.

5.2 Personalized Query Evaluation engine algorithms

The algorithm shown below is used by the Personalized Query Evaluation (PQE) engine to fetch the personalized result sets corresponding to queries submitted by users. The algorithm performs a two step process in which it fetches the top-K tweets from the temporal index and/or database, and these result are re-ranked by the *Personalization engine*. In the algorithm mentioned below U_{userId} corresponds to the id of the user submitting the query, Qj and Qj_{TS} corresponds to the query and its timestamp, respectively.

Data: U_{userId}, Qj, Qj_{TS}

Data: Temporal index

Result: top-K personalized tweets

list = fetch_tweets(Qj, Qj_{TS});

list = personalize_the_result(list, U_{userId}, Qj_{TS});

Algorithm 1: Query Evaluation

The algorithm described below is used to fetch the candidate top-K tweets corresponding to the query. The system fetches these tweets from the temporal index unless the temporal index is not currently storing tweets or has fewer than K tweets corresponding to the query. In those cases, the system fetches the remaining tweets from the database. Fetching tweets from the database is a computationally intensive process, and it is likely that the database would return an extremely large number of tweets, making it difficult for the system to perform further computation. To make the system more efficient, *REPLETE* incorporates an approximation strategy to accelerate the ranking process. The system only selects those tweets from the database whose temporal score (TS) exceeds a threshold μ. Using the value of this threshold μ we can determine the oldest timestamp a tweet can have.

$$\mu = \frac{1}{Qj_{TS} - Tw_{oldestTS}} \tag{5.2}$$

Again Qj_{TS} is the timestamp of the query Qj, μ is the system parameter that was set to $\frac{1}{6}$ and $Tw_{oldestTS}$ is the oldest timestamp a tweet can have.

The first step in query evaluation is to fetch all the keywords from the query. Next, the system finds tweets corresponding to these keywords from the temporal index, and creates a new list for each keyword. Finally, the system finds the common tweets in all the lists obtained in the previous step. If the final list contains fewer than K tweets or happens to be empty, the system then turn to the database to fetch tweets that contain these keywords and have a temporal score of at least μ.

Data: U_{userId}, Qj, Qj_{TS}

Data: Temporal Index, K

S = fetchAllKeywords(Qj);

count = S.length;

Let "list" be an empty *linked list*;

word = S[count];

$count = count - 1$;

if *index contains word* **then**
  list = getTweetsfromIndex(word)

end

while *count > 0* **do**
  word = S[count];
  new_list = getTweetsfromIndex(word);
  list = list ∩ new_list;
  if *list is empty* **then**
    break;
  end
  count = count − 1;

end

```
if list is not empty then
    if list.size < K then
        new_list = getRemainingTweetsfromDB(S, μ);
        list = list ∪ new_list;
    end
    return list;
else
    list = getAllTweetsFromDB(S, μ);
    return list;
end
```

Algorithm 2: Fetch Tweets

The algorithm described below is used to generate the personalized result set for query Qj. The top-K tweets obtained from the Fetch Tweets algorithm are ranked and reordered before the final result is returned to the user. For each tweet the system computes a cumulative ranking score (CRS) as explained in Chapter 5. This ranking score is a weighted sum of the personalization score (PS), temporal score (TS), and author influence score (AIS). Once the CRS for each tweet has been computed the system reorders them in descending order of their CRS and returns the personalized result back to the user.

Data: list, U_{userId}, Qj, Qj_{TS}

Let "personalized_results" be an empty map of tweet & its rank;

length = list.length;

while *length > 0* **do**

Tw_{length} = list[length];

$CRS(Tw_{length}) = w_1 * PS(Tw_{length}) + w_2 * TS(Tw_{length}) + w_3 * AIS(Tw_{length})$;

personalized_results.add(Tw_{length}, $CRS(Tw_{length})$);

length = length − 1;

end

sorted_result = sortByCRS(personalized_results);

return sorted_result;

Algorithm 3: Personalize the result

Chapter 6

Scalable Indexing

An effective indexing scheme is extremely important for fast generation of personalized search results. However, at the same time it is also necessary to ensure that the indexing mechanism is lightweight to be able to ingest the thousands of tweets generated per second. In order to reconcile these somewhat contradictory objectives, we have designed an indexing scheme that temporally indexes selective tweets in real-time while postponing indexing of less important ones. As mentioned in the *Chapter 4, System Architecture*, our indexing system has dedicated blocks for identifying significant tweets, real-time indexer for processing significant tweets and a batch processor for indexing non-significant tweets. Our indexing scheme itself is in memory and it is temporal in nature.

6.1 In-Memory Temporal Index

Earlybird, Twitter's current search engine, uses Lucene [17] for indexing. However, using Lucene has a significant limitation for the application at hand. First, recall that one of the early steps in our ranking strategy is to select tweets whose temporal score exceeds the threshold μ. In order to efficiently support this, we need an index that preserves temporal

order. The only way to implement this using Lucene is to create a wrapper around it. This, we believe, will add to the complexity of the indexing framework, thereby making it less efficient. Therefore, we chose to design a highly efficient, low-overhead indexing scheme using elementary data structures such as hash maps and linked lists.

Our temporal tweet index is a HashMap of configurable size (using the system parameter Γ). Frequently occurring tweet keywords act as the keys to this hash map. Each keyword is associated with a linked list that includes tweets that contain the respective keyword along with important meta-data attributes such as tweet Id, timestamp, authorId[1], and author's influence score. Maintaining this meta-data helps to speed up the query evaluation process [4][19]. These tweet objects are arranged in descending order of their timestamp (i.e., latest tweets are at the beginning of the list). The maximum size of each of these linked lists is dictated by the parameter γ. Figure 6.1 illustrates the structure of our temporal tweet index.

The keywords that are selected as the keys of the hash map are determined by the current popularity trends. We use a sliding time window to determine the most popular keywords. The least recently occurring keywords are replaced with keywords that exhibit growing popularity.

6.2 Realtime Indexing of Significant Tweets

The Significant tweet identifier block processes each item in the incoming tweet stream. This block determines if the tweet is a significant tweet that must be indexed immediately. For each new tweet that arrives, the system performs several checks to determine if it is a significant tweet. First, we determine if the tweet is from a "super user" (recall that a super user is a user whose influence exceeds the threshold Λ). This can be done efficiently by periodically

[1]Author Id is an identifier provided by Twitter to uniquely identify an Author.

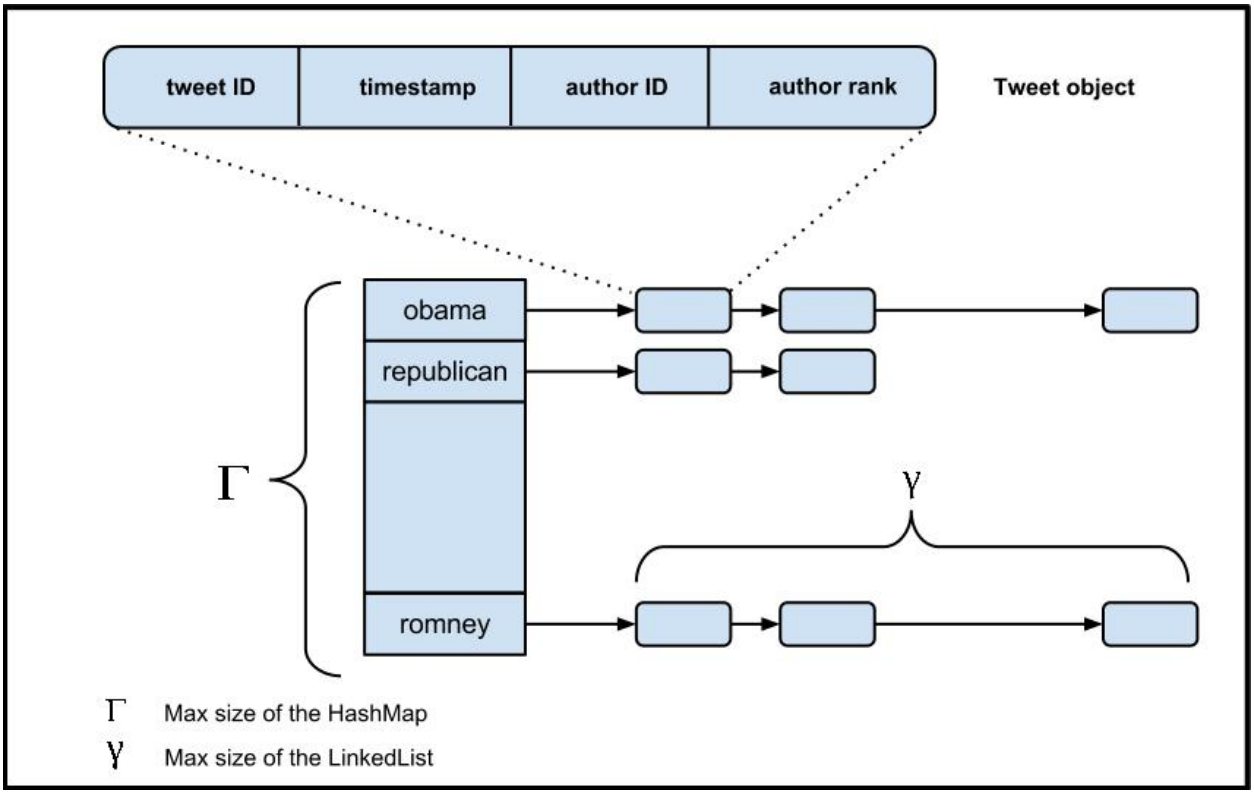

Figure 6.1: Temporal tweet index

computing the author influence scores for various users in the background. The number of followers of various users are quite stable, which allows for offline influence computation. If the tweet is indeed from a super user, it is immediately labelled as a significant tweet and forwarded to the real-time indexer. Second, we parse the tweets and extract keywords from the tweet. If any of the keywords of a tweet belong to the Trending Query Keyword Set Ω, it is also labelled as a significant tweet and forwarded to the real-time indexer. All other tweets are put onto the tweet log which will be indexed by the batch processor.

While indexing significant tweets, the first step is to eliminate any stop words present in the tweet. Then for each remaining word in the tweet, we query the temporal index to check

if the word is present in the key set. If it is not present, we create a new linked list with just one tweet object and insert this entry into the temporal index. However, if we find that the word is already present in the index we add the tweet to the front of the corresponding list. If the linked list is full (i.e., it already has γ tweets), we compute the score of the new tweet as well as for the tweet corresponding to the last element of the list using the formulas shows below. If the score of the new tweet is greater than the score of the last tweet in the list, the new tweet gets inserted at the head of the list while the last element gets deleted from the list.

$$CRS(Tw_{lasttweet}) = AIS(Tw_{lasttweet}) + TS(Tw_{lasttweet}) \tag{6.1}$$

$$CRS(Tw_{newtweet}) = AIS(Tw_{newtweet}) + 1 \tag{6.2}$$

6.3 Batch Processor

The Batch processor is responsible for periodically performing batch cycles. During each batch cycles two tasks are performed. The first task is to index the unimportant tweets stored in the Tweet log. Each tweet from the tweet log is added to the database. The other task performed during the batch cycle is updating the Trending query keyword set Ω so that the system is in sync with the current trends on Twitter. In order to perform this task the batch processor uses the Query log module. The batch processor computes the top 40% of the queries from the query log and uses them to update Ω using a least recently used (LRU) replacement scheme.

6.4 Scalable Indexing algorithms

The algorithm described below is used for identifying significant tweets. All the incoming tweets must pass through a series of tests to determine if they are significant and thus should be indexed immediately. Whenever a new tweet Tw_k arrives the system would determine the author influence score $AIS(Tw_k)$ for that particular tweet. If that score is greater than a system defined threshold Λ, the tweet is labeled significant, and is indexed immediately. However, if the $AIS(Tw_k)$ is less that Λ the system performs further testing. It finds all the keywords present in Tw_k and checks if any of them are a part of the Trending Query Keyword Set 'Ω'. If the system finds that the tweet Tw_k contains keywords present in Ω, Tw_k is then labeled as a significant tweet and indexed immediately. If all these test fail, the tweet is then added to the *Tweet Log* to be indexed by the batch processor at a later stage.

Data: Tw_k, $Tw_{k_{authorId}}$, $AIS(Tw_k)$, $Tw_{k_{TS}}$

Data: User Importance Threshold Λ, Trending Query Keyword Set Ω

Result: True: Tweet is Significant or False: Tweet is not Significant

if $AIS(Tw_k) > \Lambda$ **then**

    Index the Tweet;

    return True;

else

    S = fetchAllKeywords(Tw_k);

    count = S.length;

    while *count > 0* **do**

        word = S[count];

        if *word is in* Ω **then**

            Index the Tweet;

            return True;

        end

        count = count -1;

    end

    Send the tweet to Tweet log to be indexed in batch cycle;

    return False;

end

Algorithm 4: Significant tweet identifier

When a tweet is labeled as significant, it should be indexed immediately. The algorithm below describes the indexing process. The first step in the algorithm is to fetch keywords from the tweet Tw_k. For each keyword the system checks if the temporal index contains an entry for that keyword so system fetches the corresponding linked list. If the size of the linked list is less than γ, a new node containing the tweet Tw_k along with its meta-data is added to the head of the list. If the size of the list is γ, new scores are computed for the tweet Tw_k and the last tweet in the list using equations 6.1 and 6.2, respectively. If the rank of Tw_k is greater than that of the last tweet, Tw_k gets added to the head of the list while the current last node in the list is removed. On the other hand, if the temporal index does not contain the keyword, then a new linked list containing just Tw_k, along with its meta-data is created and added to temporal index.

Data: Tw_k, $Tw_{k_{authorId}}$, $AIS(Tw_k)$, $Tw_{k_{TS}}$

Data: temporal index, K

W = fetchAllKeywords(Tw_k);

size = W.length;

```
while size > 0 do
    word = W[size];
    if word present in Index Structure then
        Obtain the existing linked list "list" from Index Structure;
        if list.size = K then
            CRS(Tw_k) = AIS(Tw_k) + 1;
            Tw_lasttweet = list[K];
            CRS(Tw_lasttweet) = AIS(Tw_lasttweet) + TS(Tw_lasttweet);
            if CRS(Tw_k) > CRS(Tw_lasttweet) then
                Remove lastTweet from list;
                Add Tw_k to the head of the list;
                Add the list back to the index;
            end
        else
            Append the Tw_k to the head of the list along with its meta-data;
            Add the list back to the Index Structure;
        end
    else
        Create a new linked list "list" corresponding to word and add Tw_k along with
        its meta-data into the list;
        Add "list" to the index structure;
    end
    size = size -1;
end
```

Algorithm 5: Index the Tweet

Chapter 7

Experimental Evaluation

In this section, we discuss the experiments we performed on our system. We used the tweet data set containing about 25 million tweets collected from October 2006 to November 2009 [1]. In this dataset, about 465K users were used as seeds and tweets were crawled every 24 hours for each user. In our evaluation, we chose a set of about 2 million tweets in the increasing order of their timestamps. We created a client-server environment where our system acts as a server while the client sends tweets in the increasing order of their timestamps. We ran our system in two modes warming up mode and testing mode. The first million tweets sent by the client where used for warming up the system, they were used to build the temporal index as well as for building the social graph and computing the author influence score. The next batch of million tweets were used for testing the system, we collected different parameters for indexing as well as query processing during this mode. The results we obtained are mentioned below. It is a known fact that in a real-time search queries tend to have a skewed distribution [13]. For generating queries we used the later million tweets, by eliminating stop words in them and randomly combining the top 60% of the remaining keywords to generate queries. We restricted the maximum number of keywords in a query to five. Out of the total queries generated 80% were single word queries 16% were two word

queries and the remaining where multi-word queries having keywords between three to five.

During our evaluation we use two version of temporal index, one which stores linked list of tweet ids and the other which stores the linked list of tweet object. In case of the foremost version most of the information needed for query evaluation is fetched from the database, while in case of the later version most of the information needed is already present in the tweet object and additional information like author influence score is cached. We refer to the first system as non-cached system while the second as cached system. Figure 7.1 depicts the temporal index used by the non-cached system. Further, in both systems we set the maximum size of trending query keyword set Ω to 200 (i.e., $\omega = 200$). We vary the maximum size of the temporal index Γ from 2000, 3000, to 4000. We also vary maximum tweets per keyword inside temporal index (i.e., γ) from 20, 30, to 40. For each query, we only fetch top-20 tweets. For every query submitted we measure the query latency. Furthermore, in order to gain better understanding of the performance, we split the query latency into time spent personalizing results and time spent fetching tweets from the database. We also keep track of number of queries answered completely by the temporal index, those answered partially by it and the ones answered directly from the database. Finally we also measure the total time spent indexing tweets and collect similar parameter for the same.

Figure 7.2 and 7.3 show the reduction in query latency for the cached system in case of single keyword queries and all queries respectively. In both the graphs the x-axis represents the size of linked list (γ) used in the temporal index and the y-axis represent the percentage decrease in query latency. Each graph contains three line graphs, the blue solid line graph represents the percentage decrease in query latency when the size of the temporal index is set to 2000 (i.e., $\Gamma = 2000$), the red dashed line represents the percentage decrease in query latency when the size of the temporal index is set to 3000 (i.e., $\Gamma = 3000$), and finally the

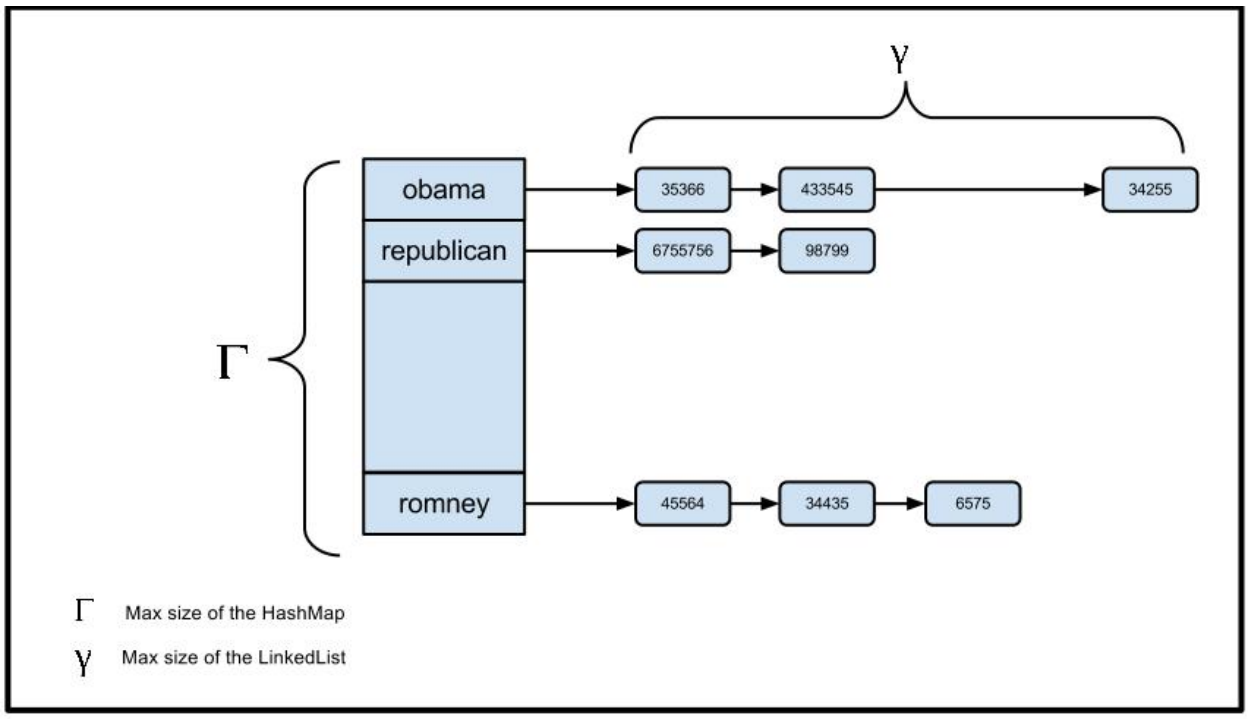

Figure 7.1: Temporal tweet index for the non-cached system

black dotted line represents the percentage decrease in query latency when the size of the temporal index is set to 4000 (i.e., $\Gamma = 4000$). We found that the cached system performed better while answering queries, since it had additional information with in the temporal index. The maximum performance gain of 52% was achieved when temporal index was set to $\Gamma = 4000$ and $\gamma = 30$ for all queries as seen in Figure 7.3, this gain was amplified to 67% for single keyword queries as seen in Figure 7.2. This additional gain for single keyword query was expected since amount of computation needed in terms of database access is significantly less when compared to multi keyword queries. We also noticed a particular pattern in the query latency, we found that when we changed γ from 20 to 30 there was an additional gain but it reduced when we further increased γ to 40. We believe that the system reached its saturation point at γ=30 when searching for top-20 results. An interesting result that

we noted for the cached system was 81% of time was spent accessing the database fetching candidate tweets and the remaining 19% time was spent personalizing the result set. On an average the queries answered completely by the temporal index took about 6 milliseconds while the one partial answered by the index took about 2 seconds, while the slowest ones were those answered completely by the database taking on an average 8 seconds. This behavior was predicted as the number of database access is directly proportional to query latency. Figure 7.4 represents the reduction in the query response time by varying the size of the temporal index (Γ). The x-axis in this graph represents the size of the temporal index, while, the y-axis represents the Percentage reduction in query latency. The blue solid line represent this percentage reduction in query latency for all queries, while, the red dashed line represent the percentage reduction for single keyword queries. Overall we found that the cached system performed better than the non-cached system and the performance seemed to get better as we increased temporal index size Γ from 2000 to 3000 and seemed to go on increasing as we further increased it to Γ=4000. This behavior is also expected since as the size of temporal index increases its able to keep track of more keywords and hence the need to access the database is reduced.

Figure 7.5 and 7.6 represent the reduction in database access for cached system when answering single keyword queries and all queries respectively. In both the graphs the x-axis represents the size of the linked list (γ) used in the temporal index and the y-axis represent the percentage decrease in database access needed to answer queries. Each graph contains three line graphs, the blue solid line graph represents the percentage decrease in database access when the size of the temporal index is set to 2000 (i.e., $\Gamma = 2000$), the red dashed line represents the decrease in database access when the size of the temporal index is set to 3000 (i.e., $\Gamma = 3000$), and finally the black dotted line represents the decrease in database access when the size of the temporal index is set to 4000 (i.e., $\Gamma = 4000$). There was a

significant decrease in database access for the cached system while answering queries. A maximum decrease of 91% for all queries and 94% for single keyword query was achieved for Γ = 3000,4000 at γ=30 as seen in Figure 7.6 and 7.5 respectively. This reduction in database access was expected, since, the cached system maintains meta-data along with the tweet ids, allowing the query engine to process queries at a faster rate. We noticed a similar pattern when measuring database access as we noticed with the query latency the gain increased as we increased γ from 20 to 30, however on further increasing γ = 40 we found that the gain dropped. We also noticed that the graph for Γ 3000, 4000 seemed to overlap. We believe this happened since the saturation point for the system was reached.

Figure 7.7, 7.8, and 7.9 represent the HIT rate of the temporal index when the size of the linked list with in the index is set to 20, 30, and 40 respectively (i.e. varying γ). In these bar graphs the x-axis represents the size of the temporal index, while the y-axis represents number of queries answered. Each bar in the bar graph is divided into three section, the bottom most section represents the number of queries answered completely from the temporal index, the middle portion represent the number of queries answered partially by the temporal index, while the top most portion of the graph represents the number of queries answered directly from database. We noticed that the amount of queries answered completely by the index, and those answered partially by index seemed to go on increasing as we increased the index size from Γ=2000 to 3000 and then from 3000 to 4000. This behavior was predictable; as we increase the size of the index it is able to store more keywords and their corresponding tweets. When measuring performance gain for indexing, we found both the cached and non-cached system performed equally well.

Figure 7.10 represents the reduction in indexing cost due to caching. The x-axis represents the size of the linked list (γ) used in the temporal index and the y-axis represent the

percentage decrease in indexing cost. The blue solid line represents the percentage decrease in indexing cost when the temporal index size is set to 2000 (i.e., $\Gamma = 2000$). The red dashed line represents the percentage decrease when $\Gamma = 3000$ and finally, the black dotted line represents the percentage decrease when $\Gamma = 4000$. We found that there is not a significant decrease in the indexing cost in both the systems. We believe this behavior is acceptable since we are not targeting to reduce the indexing cost, but, we are reducing the number of tweets being indexed by identifying the one that are significant and indexing them immediately. Figure 7.11 represents the reduction in database access required while indexing. The x-axis represents the size of the linked list (γ) used in the temporal index and the y-axis represent the percentage decrease in database access needed while indexing. The blue solid line represents the percentage decrease in database access when the temporal index size is set to 2000 (i.e., $\Gamma = 2000$). The red dashed line represents the percentage decrease when $\Gamma = 3000$ and finally, the black dotted line represents the percentage decrease when $\Gamma = 4000$. We found that the amount of database access decreased on an average by 67%. The maximum decrease was of 70% when $\Gamma = 2000$ and $\gamma = 20$. The reason of this decrease in cached system is due to caching of author influence score (AIS), in main memory. On the other hand the non-cached system has to fetch this score from the database. Figure 7.12 highlights the cost of personalization in terms of query response time. The y-axis represents the average query response time, while the x-axis represents the number of keywords present in the query. For each type of query (single-keyword, multi-keyword) we determine the average query response time with personalization, as well as without personalization. For each type of query in the Figure 7.12 the left bar-graph represents the average query response time with personalization, while the right one represents the average query response time without personalization. For single keyword queries the query response time increases on an average by just 1 second when personalization is incorporated, while for multi-keyword queries it increases by 3 seconds.

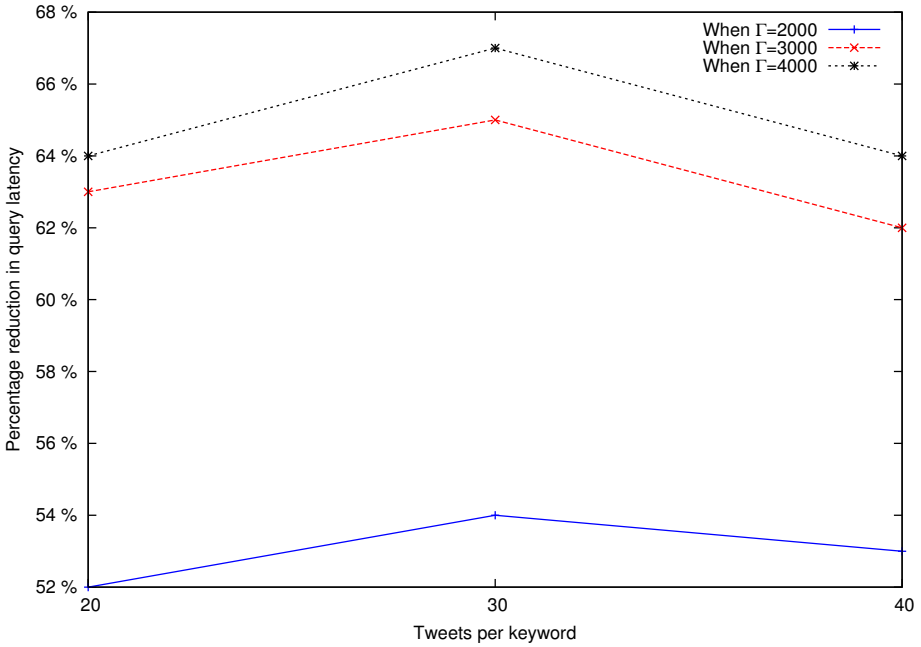

Figure 7.2: Reduction in query response time due to caching for one word queries

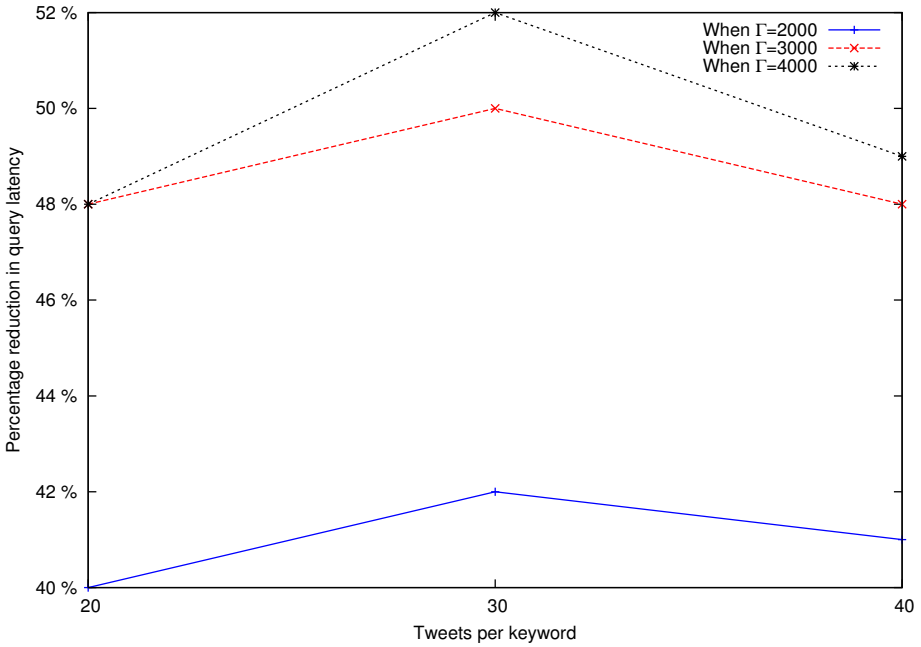

Figure 7.3: Reduction in query response time due to caching for all queries

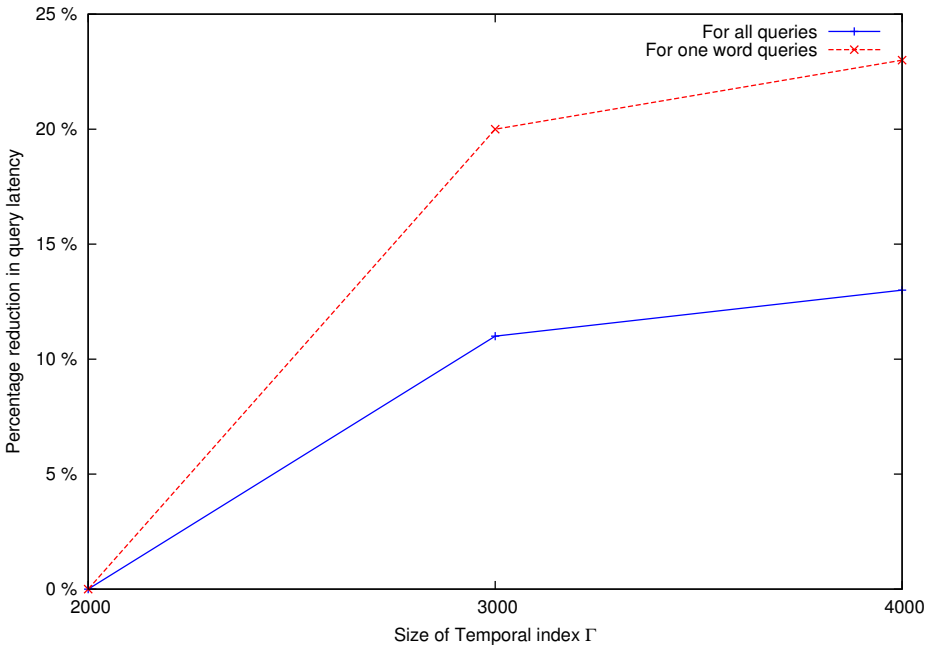

Figure 7.4: Reduction in query response time by varying size of the temporal index Γ

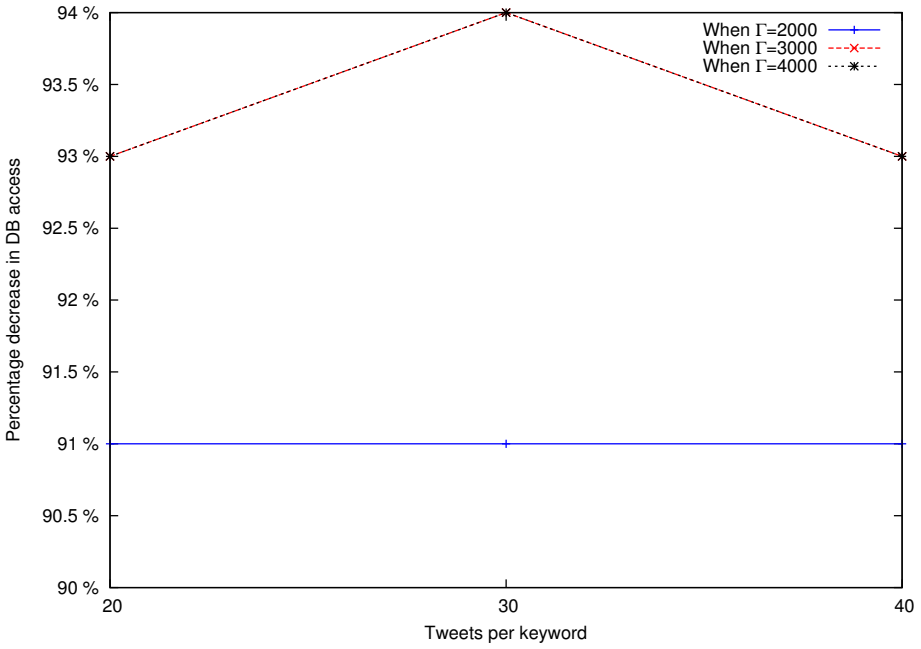

Figure 7.5: Reduction in DB access due to caching for one word queries

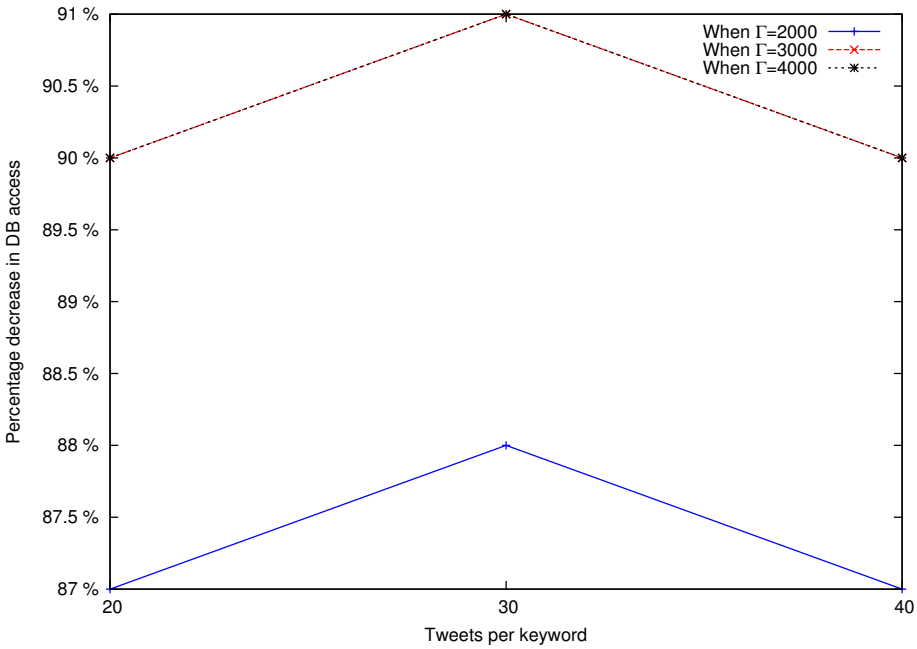

Figure 7.6: Reduction in DB access due to caching for all queries

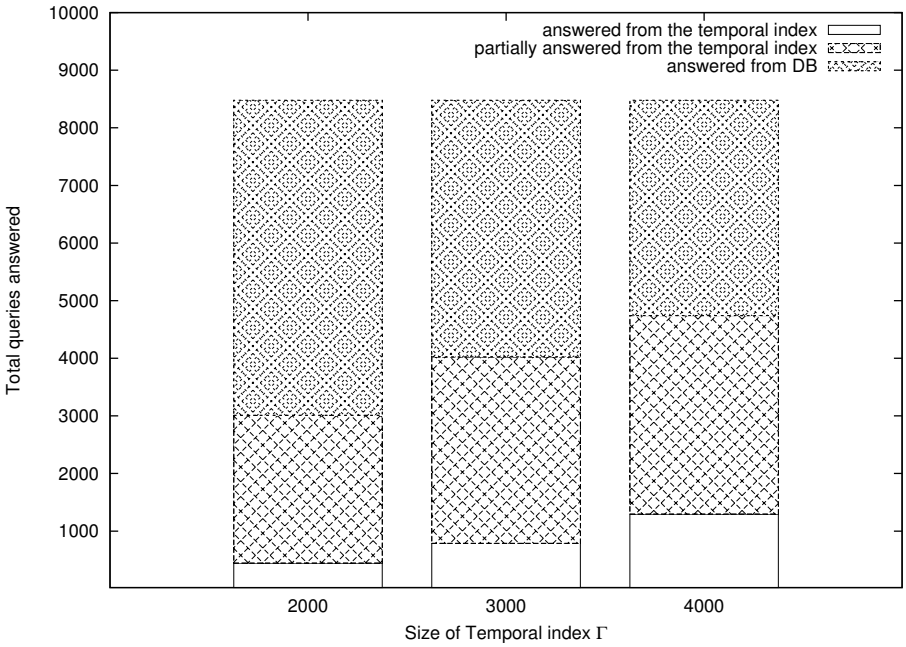

Figure 7.7: Temporal index hit statistics for γ=20

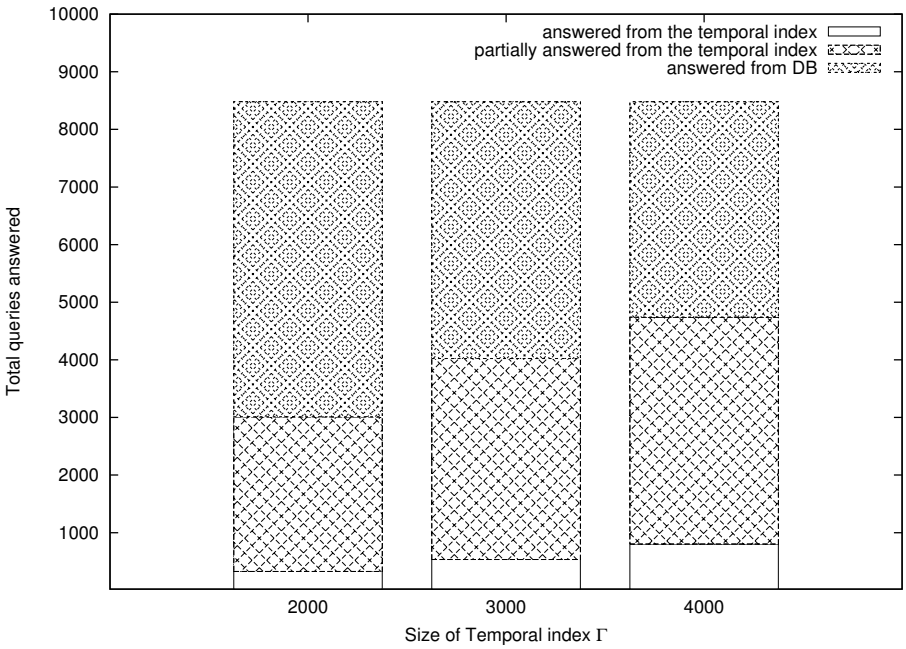

Figure 7.8: Temporal index hit statistics for γ=30

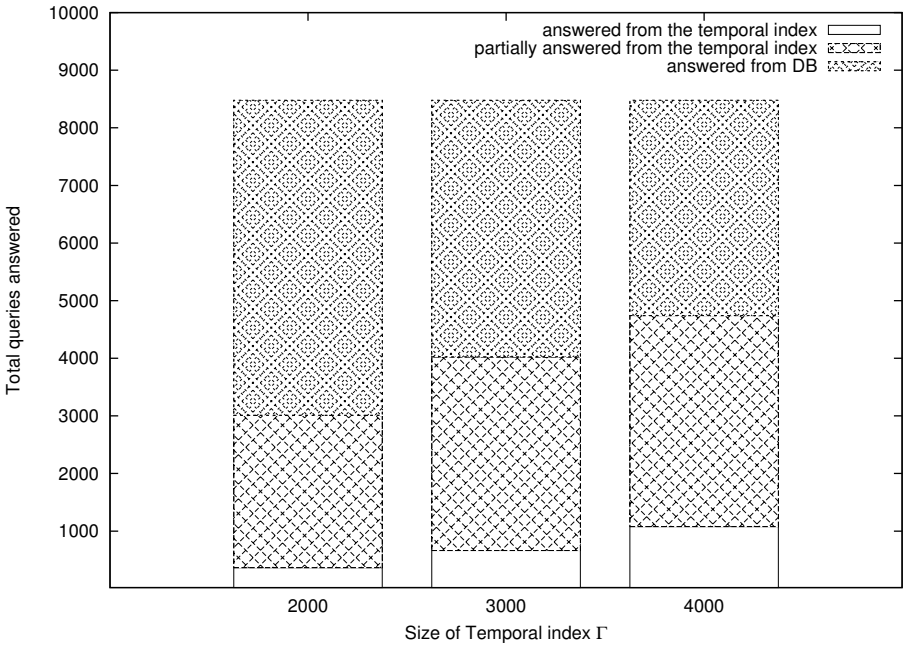

Figure 7.9: Temporal index hit statistics for γ=40

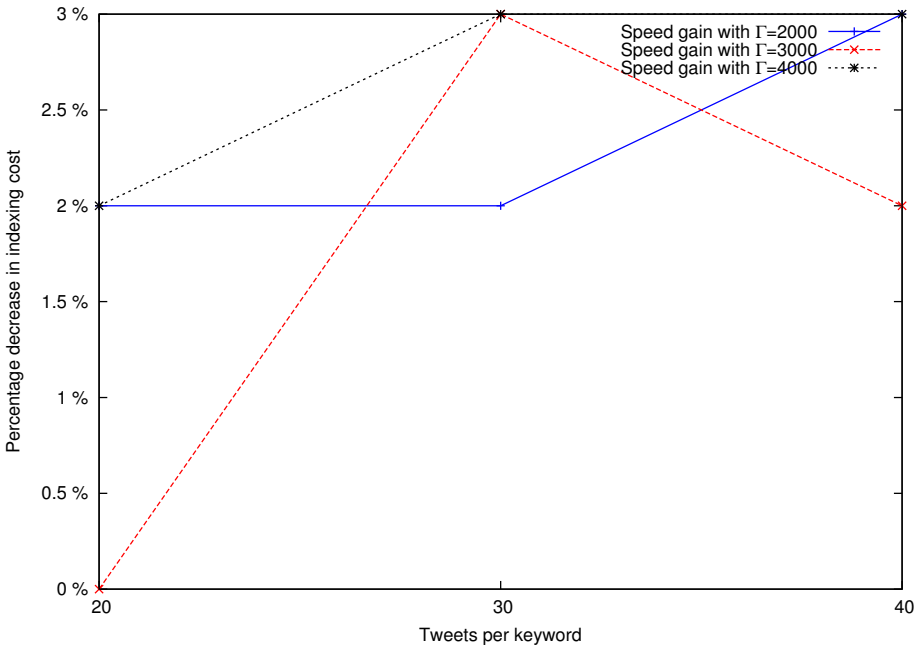

Figure 7.10: Reduction in indexing cost

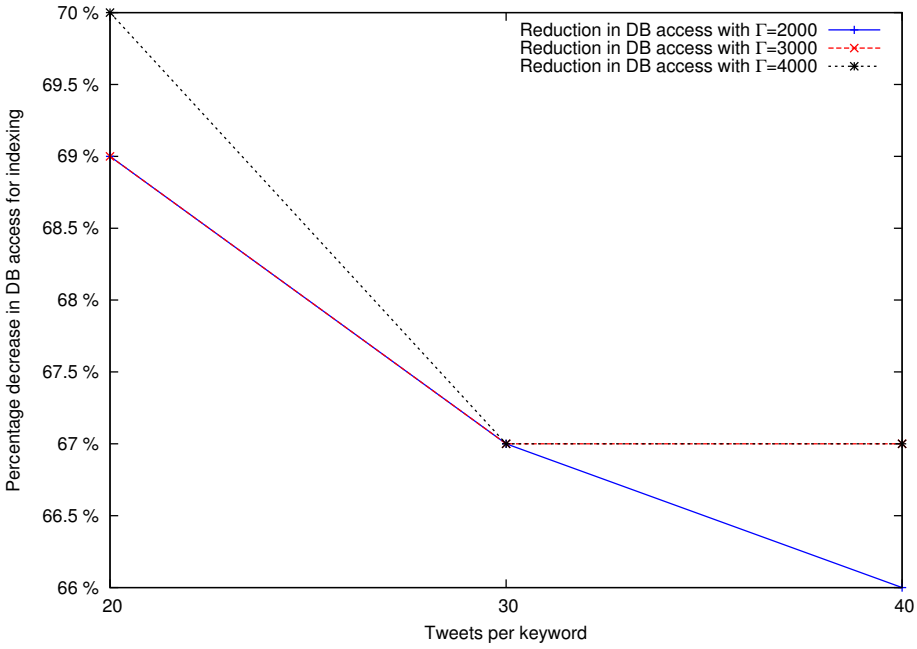

Figure 7.11: Reduction in DB access due to caching for indexing

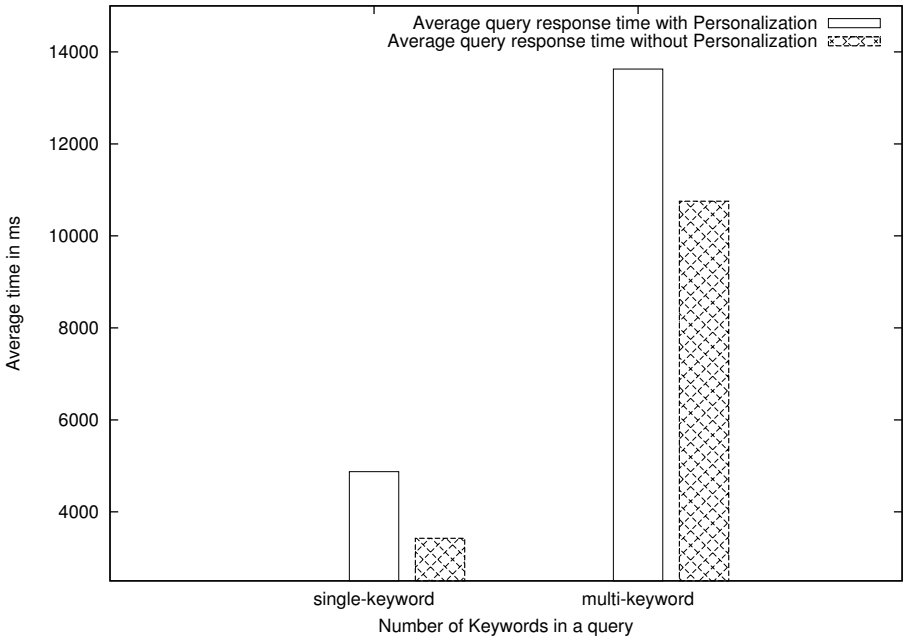

Figure 7.12: Personalization cost

Chapter 8

Related Work

Information Retrieval (IR) over the web is a domain which has matured in recent years. It mainly focuses on the use of web-crawler [22][9] for finding new contents, and indexing them in batch cycle. Modern day web-crawlers are able to achieve high throughput, by using batch systems such as MapReduce [6] for running the indexing jobs. The index maintained by IR engine are mainly static read only and get updated during these batch cycles. They are able to index content in batch since a delay of minutes, hours or even days may be acceptable while indexing depending on the type of content [2].

Another interesting topic which has gain attention recently is the Ranking functions used for tweets. Ranking tweets is not a trivial task [5] 'Ranking approaches for microblog search' [15] discuss a sophisticated ranking function which not only considers the temporal significance of tweets (which happens to be of prime importance for real-time search) but takes into account SN properties of authors of tweets in addition to the properties of tweets. They use a two stage query evaluation process [8] where they fetch the top-K results [23] using the existing query engine provided by Twitter and re-rank this result using their own ranking function. Twitter's real-time search engine takes into account the syntactic match between

query and the tweets and the temporal recency of matching tweet. To determine SN properties of authors they define ranking factors such as AuthorRank which is determined based on number of tweets the author uploads and couple it with FollowerRank which is determined by the number of followers the author has and number of users the author follows. To determine the properties of tweet they propose two more ranking factors LengthRank which is determined using the length of the tweet coupled with URLRank which is determined based on; if the tweet contains URL.

Many new systems have be proposed for real-time search over tweets. Earlybird [2] is the search engine used by Twitter, it uses a single writer multiple reader architecture for implementing real-time search. Earlybird considers two major factors in producing search results, namely, syntactic match between query and the tweets and the temporal recency of matching tweet. Another such real-time searching system is TI [1] which performs partial indexing of incoming tweets and supports an adaptive index. Since TI uses partial indexing it is able to save some portion of computational power used for indexing which it redirects towards ranking. TI's ranking system is far more sophisticated than Earlybird, it considers many different factors such as importance of author posting tweet, current trend on Twitter as well as the temporal significance of the tweet.

Chapter 9

Conclusion

We presented REPLETE – a scalable, real-time search framework for micro-blogs that incorporates personalization by analyzing the follower relationships in Twitter. We discussed the three novel features that our REPLETE framework has to offer. Firstly, we came up with a unique tweet ranking metric for generating personalized search results. This ranking metric took into consideration three major tweet relevance factors, the syntactical similarities between the tweet and the query, the distance of follower relationships, and the temporal significance of the tweet. The distances of the follower relationships helped us incorporate personalization with in our framework, and to best of our knowledge none of the existing works incorporate personalization into their search strategies. The second unique feature of our framework was our in-memory temporal index which helped us speed up query evaluation process by preserving temporal significance of the tweets. This index helped us address the scalability challenges posed by personalization described previously. Finally the third feature offered by our framework ensured the overall quality and timeliness of search results, we performed partial indexing by identifying important tweets, and prioritize their indexing. The experimental evaluations that we performed on our framework demonstrated the scalability and efficiency of our framework.

Currently the system performs personalization by measuring the *social affinity* of user issuing the query to the tweet's author. However there are other factors that can be used while computing personalization score such as geo-preference. We could measure the geographical distance between the user issuing the query and the author of the tweet, and integrate it in a ranking function. Higher preference could be given to the tweets that were posted by authors geographically closer to the user issuing the query. Personalization is an unexplored area and we believe we have just started exploring it and hope that this work can act as a motivation to other researcher to pursue this area of research.

There are couple of other enhancement that can be implemented in our system. We would like to implement this system with a multithreaded architecture. We could have two groups of worker threads, each containing significant number of threads. One group would be responsible for handling tweets as and when they enter the system in a concurrent fashion. The other would be responsible for handling user queries in a concurrent way. We think these enhancement would definitely help us speed up the entire process of indexing and query evaluation. We would also like to implement the complete system using Lucene for indexing purpose. This will allow us to compare the performance of our temporal index implemented using HashMaps and LinkedList with a modified Lucene library which considers temporal significant while indexing. In our evaluation of the system we use MySQL[20] as our primary database, we would like to experiment with different databases to see how the performance varies in term of time spent reading from or writing into a databases. Finally we would also like to implement a persistent index which can complement our in-memory index. We believe having such a persistent index would help reduce the latency when the query evaluation engine is not able to find a match using the temporal index.

Bibliography

[1] Chen, Chun, Feng Li, Beng Chin Ooi, and Sai Wu. "TI: an efficient indexing mechanism for real-time search on tweets." In Proceedings of the 2011 international conference on Management of data, pp. 649-660. ACM, 2011.

[2] Busch, Michael, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. "Earlybird: Real-time search at Twitter." In Data Engineering (ICDE), 2012 IEEE 28th International Conference on, pp. 1360-1369. IEEE, 2012.

[3] De Choudhury, Munmun, Yu-Ru Lin, Hari Sundaram, K. Selcuk Candan, Lexing Xie, and Aisling Kelliher. "How does the data sampling strategy impact the discovery of information diffusion in social media." In Proceedings of the 4th International AAAI Conference on Weblogs and Social Media, pp. 34-41. 2010.

[4] Baeza-Yates, Ricardo, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. "The impact of caching on search engines." In Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 183-190. ACM, 2007.

[5] Dong, Anlei, Ruiqiang Zhang, Pranam Kolari, Jing Bai, Fernando Diaz, Yi Chang, Zhaohui Zheng, and Hongyuan Zha. "Time is of the essence: improving recency ranking using twitter data." In Proceedings of the 19th international conference on World wide web, pp. 331-340. ACM, 2010.

[6] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[7] Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank citation ranking: bringing order to the web." (1999).

[8] Broder, Andrei Z., David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. "Efficient query evaluation using a two-level retrieval process." In Proceedings of the twelfth international conference on Information and knowledge management, pp. 426-434. ACM, 2003.

[9] Olston, Christopher, and Marc Najork. "Web crawling." Foundations and Trends in Information Retrieval 4, no. 3 (2010): 175-246.

[10] eMarketer (2009). U.S. Twitter usage surpasses earlier estimates. http://www.emarketer.com/(S(nuv2kg55sxyth245rhij2l55))/Article.aspx?R=1007271

[11] Teevan, Jaime, Daniel Ramage, and Merredith Ringel Morris. "# TwitterSearch: a comparison of microblog search and web search." In Proceedings of the fourth ACM international conference on Web search and data mining, pp. 35-44. ACM, 2011.

[12] Google search engine http://www.google.com

[13] Jansen, Bernard J., Gerry Campbell, and Matthew Gregg. "Real time search user behavior." In Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems, pp. 3961-3966. ACM, 2010.

[14] Twitter http://www.twitter.com

[15] Nagmoti, Rinkesh, Ankur Teredesai, and Martine De Cock. "Ranking approaches for microblog search." (2010): 153-157.

[16] Middlemiss, Melanie J., and Grant Dick. "Weighted feature extraction using a genetic algorithm for intrusion detection." Evolutionary Computation, 2003. CEC'03. The 2003 Congress on. Vol. 3. IEEE, 2003.

[17] Hatcher, Erik, Otis Gospodnetic, and Michael McCandless. "Lucene in action." (2004).

[18] Peng, Daniel, and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications." In Proceedings of the 9th USENIX conference on Operating systems design and implementation, pp. 1-15. USENIX Association, 2010.

[19] Skobeltsyn, Gleb, Flavio Junqueira, Vassilis Plachouras, and Ricardo Baeza-Yates. "ResIn: a combination of results caching and index pruning for high-performance web search engines." In Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval, pp. 131-138. ACM, 2008.

[20] MySQL Database http://www.mysql.com

[21] Wen, Ji-Rong, Jian-Yun Nie, and Hong-Jiang Zhang. "Query clustering using user logs." ACM Transactions on Information Systems 20, no. 1 (2002): 59-81.

[22] Pinkerton, Brian. "Finding what people want: Experiences with the WebCrawler." In Proceedings of the Second International World Wide Web Conference, vol. 94, pp. 17-20. 1994.

[23] iProspect Search Engine User Behavior Study. `http://district4.extension.ifas.ufl.edu/Tech/TechPubs/WhitePaper_2006_SearchEngineUserBehavior.pdf`

[24] The Next Web. http://thenextweb.com/socialmedia/2012/03/21/twitter-has-over-140-million-active-users-sending-over-340-million-tweets-a-day/

[25] Maximum number of users that one can follow on Twitter. https://support.twitter.com/articles/66885-follow-limits-i-can-t-follow-people#

[26] Yue, Sui, and Yang Xuecheng. "The potential marketing power of microblog." Communication Systems, Networks and Applications (ICCSNA), 2010 Second International Conference on. Vol. 1. IEEE, 2010.

[27] Java, Akshay, et al. "Why we twitter: understanding microblogging usage and communities." Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis. ACM, 2007.

[28] Kwak, Haewoon, et al. "What is Twitter, a social network or a news media?." Proceedings of the 19th international conference on World wide web. ACM, 2010.

[29] Sankaranarayanan, Jagan, et al. "Twitterstand: news in tweets." Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. ACM, 2009.

[30] Salton, Gerard, and Christopher Buckley. "Term-weighting approaches in automatic text retrieval." Information processing and management 24.5 (1988): 513-523.

[31] Ramos, Juan. "Using tf-idf to determine word relevance in document queries." Proceedings of the First Instructional Conference on Machine Learning. 2003.

[32] Ari, Ismail, et al. "Managing flash crowds on the Internet." Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on. IEEE, 2003.

[33] Elson, Jeremy, and Jon Howell. "Handling flash crowds from your garage." USENIX 2008 Annual Technical Conference on Annual Technical Conference. USENIX Association, 2008.

[34] Thelwall, Mike. "WEB CRAWLERS AND SEARCH ENGINES." Library and Information Science 4 (2010): 9-22.

[35] Brin, Sergey, and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine." Computer networks and ISDN systems 30.1 (1998): 107-117.

[36] Cho, Junghoo, Hector Garcia-Molina, and Lawrence Page. "Efficient crawling through URL ordering." Computer Networks and ISDN Systems 30.1 (1998): 161-172.

[37] Baeza-Yates, Ricardo, Paolo Boldi, and Carlos Castillo. "Generalizing pagerank: Damping functions for link-based ranking algorithms." Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2006.