TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets *

Chun Chen #1, Feng Li §2, Beng Chin Ooi §3, Sai Wu §4
College of Computer Science, Zhejiang University, China, 321100
1 chenc@zju.edu.cn
§ School of Computing, National University of Singapore, Singapore, 117590
2,3,4 {li-feng, ooibc, wusai}@comp.nus.edu.sg

ABSTRACT

Real-time search dictates that new contents be made available for search immediately following their creation. From the database perspective, this requirement may be quite easily met by creating an up-to-date index for the contents and measuring search quality by the time gap between insertion time and availability of the index. This approach, however, poses new challenges for micro-blogging systems where thousands of concurrent users may upload their micro-blogs or tweets simultaneously. Due to the high update and query loads, conventional approaches would either fail to index the huge amount of newly created contents in real time or fall short of providing a scalable indexing service.

In this paper, we propose a tweet index called the *TI* (Tweet Index), an adaptive indexing scheme for microblogging systems such as Twitter. The intuition of the *TI* is to index the tweets that may appear as a search result with high probability and delay indexing some other tweets. This strategy significantly reduces the indexing cost without compromising the quality of the search results. In the *TI*, we also devise a new ranking scheme by combining the relationship between the users and tweets. We group tweets into topics and update the ranking of a topic dynamically. The experiments on a real Twitter dataset confirm the efficiency of the *TI*.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design

*In Twitter, tweet refers to the microblog published by users. In this paper, we use it as a common phrase for microblogs.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

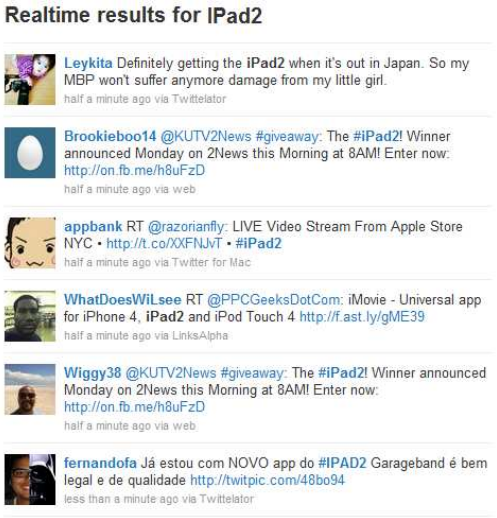


Figure 1: Example of Twitter Search on 3/11/2011

Keywords

Real-time Search, Index, Ranking

1. INTRODUCTION

The increasing popularity of social networking systems changes the form of information sharing. Instead of issuing a query to a search engine, the users log into their social networking accounts and retrieve news, URLs and comments shared by their friends. This is in part caused by the failure of conventional search engines in providing real-time search service for social networking systems. For example, it is difficult to search a new blog or tweet uploaded a few minutes ago using a conventional search engine. The problem is further amplified in the microblogging systems such as Twitter due to unprecedented amount of tweets or microblogs being posted each day. For example, Tumblr (http://www.tumblr.com) estimated that there were more than 2 million posts and fifteen thousands new users every day[1]; and based on a latest report from Twitter[2], it handled more than 50 million tweets per day.

Providing real-time search service is indeed very challenging in large-scale microblogging systems. In such a system,

[1] http://staff.tumblr.com/post/434982975/a-billion-hits
[2] http://thenextweb.com/socialmedia/2010/02/22/twitter-statistics-full-picture/

thousands of new updates need to be processed per second. To make every update searchable, we need to index its effect in real time and provide effective and efficient keyword-based retrieval at the same time. The objectives are therefore contradictory since maintenance of up-to-date index will cause severe contention for locks on the index pages.

Another problem of real-time search is the lack of effective ranking functions. Figure 1 illustrates an example on the search results of Twitter for the keyword "IPad2". The query was submitted a few minutes later after IPad2's sale starts. The user is perhaps looking for the reviews and comments about the IPad2, or he is trying to find out the length of queue at the apple stores around his neighborhood. However, most search results are advertisements and most of the returned tweets do not even provide any useful information. This is because the current Twitter search engine sorts the results based on time, and therefore, the latest tweets have the higher rankings. Recall that one key factor of Google's early success is its PageRank [14] algorithm. Without proper ranking functions, the search results are meaningless. However, defining a ranking function for real-time search is not trivial, and the function must have the following two desiderata:

1. The ranking function must consider both the timestamp of the data and the similarity between the data and the query. As an example, for a given query submitted to Twitter, we do not want to get tweets posted many weeks ago, even though they may contain the keywords of the query. On the other hand, newer tweets with less information are not preferred either. Hence, the ranking function is composed of two independent factors, time and similarity.
2. The ranking function should be cost-efficient. As we want to support real-time search using a ranking function partially based on time, we have to compute the rankings during query time. Thus, the computation of the ranking function should not incur high overhead.

In this paper, we propose the Tweet Index (*TI*), a novel indexing and ranking mechanism for enabling real-time search in microblogging systems such as Twitter. The *TI* is designed based on the observation that most tweets will not appear in the search results. Therefore, we can significantly reduce the indexing cost by delaying indexing less useful tweets. In essence, the *TI* classifies the tweets into two types, *distinguished* tweets and *noisy* tweets. The *TI* consists of two indexing schemes: a real-time indexing scheme for distinguished tweets and a background batch indexing scheme for noisy tweets. Given a new tweet, *TI* analyzes its contents and determines its type. If it is a distinguished tweet, we will index it immediately. Otherwise, it is grouped with other noisy tweets and periodically, the batch indexing scheme is invoked to index all the noisy tweets in one go. The design principle of the *TI* is similar in spirit to the partial indexing scheme [20, 18], and is also related to the view selection problem [1]. To the best of our knowledge, this is the first proposal that addresses the index issues for the real-time search.

In the *TI*, the ranking function plays the major role in deciding whether the tweets are distinguished tweets or noisy tweets and in retrieving meaningful answers. We therefore propose a new ranking function by combining the user graph and tweet graph. In social networks, each user can be considered as a node and different nodes are connected together via the friend links. The user graph denotes the relationship among the users. Naturally, a popular user will have more friends and his/her blogs/tweets also attract wider readership. Therefore, we run a PageRank algorithm for the user graph to compute the ranking for each user. Besides the user graph, the tweets also form a graph, as some tweets are exchanged between people and some tweets reply to the other tweets. We group tweets into topics based on their relationship, and we measure the popularity of the topics based on their statistics. Finally, our proposed ranking function is composed of the user's PageRank, the popularity of topics, the TF (Term Frequency) and the timestamp. The IDF (Inverse Document Frequency) is not used in the *TI*, since the length of a microblog is fairly small and often capped at certain length (e.g. in Twitter, it is capped at 140 characters).

We evaluate the *TI* by using a real Twitter dataset collected for a user group within the last three years. The experiments examine the performance of our indexing scheme and the effect on the quality of query results. We also compare our ranking function with the other relevant ranking functions.

The rest of the paper is organized as follows. In Section 2, we review the previous work in social network search and the corresponding database techniques. In Section 3, we introduce the overview architecture of *TI*. And the details of the *TI*'s indexing scheme and ranking function are discussed in Section 4 and Section 5, respectively. We evaluate the performance of the proposed schemes in Section 6. And the paper is concluded in Section 7.

2. RELATED WORK

2.1 Partial Indexing and View Materialization

In database systems, indexes are created to facilitate efficient query processing. However, existing indexes designed for similarity and KNN search such as iDistance [28] could not be directly applied to tweet indexing, since they have not been designed for very high insertion load. Instead of indexing the whole dataset, a partial index was proposed for indexing the records that may be queried with high probability. The idea of partial indexing was first proposed in [20], where the advantages of a partial index are analyzed. In [18], a statistical model is built to monitor the query distribution and the partial index is created adaptively. Partial indexing technique is also adopted in the distributed environment. In PIER [13], only rare items are indexed in the DHT (Distributed Hash Table), while the popular items are searched via flooding. In PISCES [25], a just-in-time indexing scheme that can be dynamically tuned to follow query patterns was proposed to facilitate query processing in a peer-to-peer based data management system on BATON [9].

View materialization shares some similar principles with the partial indexing technique. [3] and [27] discuss how to adaptively materialize the views in multi-dimensional databases and data warehouse systems. Cost models were proposed in [1] and [5] to automatically select views for materialization. In [19], the adaptive view materialization strategy is applied to reduce the overhead of stream feeding systems. The proposed *TI* adopts a similar design philosophy with the above work. In the *TI*, only data that are deemed essential for the queries are indexed in real-time, while the remaining data are processed in bulk and batch mode.

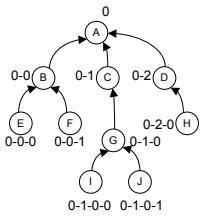


Figure 2: Tree Structure of Tweets

2.2 Microblog Search

Google and Twitter have released their real-time search engines recently. Google designs its web crawler to adaptively crawl the microblogs, while Twitter relies on an existing technique, such as Lucene[3], to provide the search service. Both of them treat a query as a continuous query and update the results in real time. However, the ranking function only considers the time dimension, and as a result, the results are sorted by time. By studying the users' behavior in the microblogging systems [11], more sophisticated ranking schemes, such as [23] and [15], were proposed. However, most ranking schemes are too complex and therefore too expensive and time consuming. They are precomputed in an offline manner. To address this problem, in [16], noisy tweets are pruned and similar tweets are clustered together. Ranking is computed for the tweets of the same cluster so that the computation cost can be significantly reduced.

In the *TI*, we also group tweets into some topics by examining their relationships captured in a tree structure. In particular, tweets replying to the same tweet or belonging to the same thread are organized as a tree. Similar schemes were adopted for forum search [17, 26]. To reduce the ranking cost, *TI* maintains the popular topics in memory and modifies the structure of an inverted index. Compared to the previous work, *TI*'s ranking function is more efficient and incurs less overhead.

3. SYSTEM OVERVIEW

3.1 Social Graphs

As the *TI* is proposed to support efficient search in microblogging systems, we first review the features of social networks that influence the design of the index.

In social networks, users are connected together by friend links (in Twitter, it's following/follower link). Typically, a popular and famous user will have more friends than an ordinary or low-profile user. Here, we define a user graph $G_u = (U, E)$, where U is set of users in the system and E is the friend links between them.

Apart from the user graph, we have another graph that is induced by the relationship of microblogs or tweets. Figure 2 shows a tree structure of tweets, where each node denotes a tweet and the directed edge indicates that one tweet replies to or retweets another tweet. For example, tweet B replies to tweet A and thus A is the parent node of B in the tree. The tweet that does not reply to others becomes the root of the tree. In this paper, we use a tweet tree to represent a discussion topic. When searching, tweets in the same topic can be grouped together and returned. We do not explicitly maintain the tweet tree, as it may incur too much overhead.

[3]http://lucene.apache.org

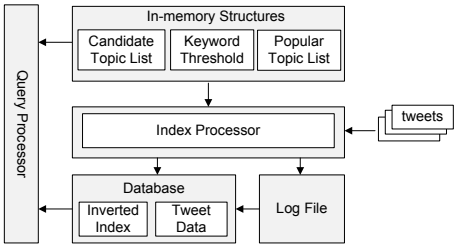


Figure 3: Architecture of *TI*

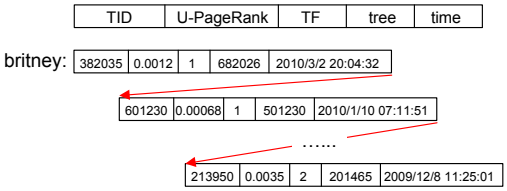


Figure 4: Structure of Inverted Index

Instead, we assign each tweet a tree encoding ID, which is similar to the Dewey Order ID [22] in XML search. Given tweet t_i, we sort its child nodes by their timestamps (the time that the tweet is inserted into the system). Suppose the encoding of t_i is "x" and tweet t_j is t_i's kth child, t_j's encoding is "x"+"-"+"k", where + indicates the string concatenation. With the help of tree encoding, we can easily reconstruct the tree structure.

3.2 Design of the TI

The *TI* provides its search via an inverted index. When a new tweet is inserted into the microblogging system, the indexing process determines whether it should be indexed or not. To facilitate the fast index maintenance and search process, some statistics are maintained in memory. Figure 3 shows the architecture of the *TI*.

In the *TI*'s database, we keep an inverted index for the tweet data. Given a keyword, the inverted index returns a tweet list, T. T consists of a set of tweet IDs and tweets in T are sorted by their timestamps (the time when a tweet is inserted into the system). Figure 4 shows the index structure of the inverted index. For each record in the index, we keep its tweet ID, TID (inherited from the status ID provided by Twitter), to identify different tweets. Then, for the ranking purpose, we keep the U-PageRank of a tweet (to be defined in Section 5), the TF (Term Frequency) value, the tree ID and the timestamp of the tweet. Tree ID is the TID of the root node in a tweet tree. Records of the same keyword are maintained as a list and the latest record is inserted into the head of the list. As a result, the records are sorted by their timestamps in the list.

To facilitate our ranking scheme, we also keep the metadata of a tweet. We define a tweet table as follows.

Table 1 Example of Tweet Table

TID	RID	tree	time	count	coding	UID	pointer
26476	76732	25742	...	0	0-0-0	...	null
57380	76732	25742	...	0	0-0-1	...	null
26980	null	26980	...	1	0	...	1022

Based on a tweet's content, we know whether the tweet replies/re-tweets another tweet. We maintain the ID of the

replied tweet as RID, and it can be used to retrieve the parent tweet. If a tweet belongs to an existing tree, we keep the root ID of the tree, which can be obtained from its parent tweet. Otherwise, we create a single node tree by using the tweet itself as the root. We also keep the timestamp of each tweet and the *count* attribute denotes the number of tweets that reply to this tweet. To enable efficient reconstruction of the tree, the encoding is stored with each tweet. The author ID UID of a tweet is defined as the foreign key in the tweet table. Finally, if a tweet is not indexed and written back to the log file, we keep a pointer to its offset in the log file. To support efficient retrieval via tweet ID and user ID, we build a B^+-tree index for TID and UID in the database.

Besides the tweet table, the TI keeps a log file for recording the unindexed tweets. The TI selectively indexes the inserted tweets, the distinguished tweets. The noisy tweets are appended to the log file and periodically, a background batch indexing process will scan the log file to index the noisy tweets.

To support the TI's indexing and ranking algorithm, we keep some useful information in the memory, such as keyword threshold, candidate topic list and popular topic list. Keyword threshold records the statistics of recent popular queries. The candidate topic list maintains the information about recent topics, while popular topic list represents the hotly discussed topics. Based on above information, we can quickly classify a tweet as a distinguished or noisy tweet and adopt different indexing scheme accordingly. Moreover, based on our in-memory structures, we rank the tweets in the querying time by combining the time, popularity and similarity.

4. CONTENT-BASED INDEXING SCHEME

The basic idea of the TI's indexing scheme is to index the tweets based on their contents and their rankings with respect to existing queries. Intuitively, it streams a new tweet into an existing set of popular queries, and based on its ranking, determines if it should be indexed in real-time or in batch periodically. To improve the quality of search results, our ranking function considers the user's pagerank, the popularity of a topic and the similarity between queries and tweets. Figure 5 shows the data flow in TI's index processor. In this section, we present how we classify the tweets and apply the adaptive tweet indexing strategy. The details of ranking function $\mathcal{F}$ will be discussed in next section.

4.1 Tweet Classification

The first challenge in the design of TI's indexing strategy on the measurement of the importance of a tweet. Limited by its size, a tweet itself does not provide too much information. Therefore, we apply a query-based classification approach. We assume that users are only interested in the top-K results. This assumption can easily be verified by the statistics of search engines [8] where 62% of the users click a result in the first page and more than 90% of the users stop their browsing after three pages of results.

In particular, the problem can be formalized as follows.

Definition 1. Tweet Classification
Given a tweet t and a user's query set $\mathcal{Q}$, t is said to be a distinguished tweet, if $\exists q_i \in \mathcal{Q}$ and t is a top-K result for q_i based on the ranking function $\mathcal{F}$. Otherwise, t is a noisy tweet.

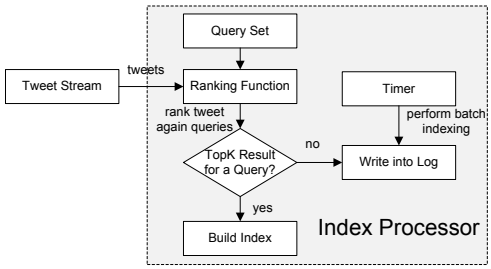


Figure 5: Data Flow of Index Processor

To answer top-K queries in query set $\mathcal{Q}$, we just need to index the distinguished tweets, while the noisy tweets can be indexed periodically. In this way, we avoid high real-time update costs.

Obviously, for a different query set $\mathcal{Q}$, the classification result will be different. Ideally, when all possible queries are considered, the classification will provide an accurate result for every query. However, the maintenance cost may neutralize the benefit of partial indexing. Fortunately, it has been confirmed that, like any social phenomenon, the search engine queries [2] and social networking queries [21] do in fact follow the well known Zipf's distribution. In other words, the top 20% queries represent 80% of the user requests. Therefore, only popular queries are maintained in $\mathcal{Q}$ to reduce maintenance cost. In particular, suppose the nth query appears with a probability of

$$p(n) = \frac{\beta}{n^\alpha} \tag{1}$$

where α and β are parameters that describe the Zipf's distribution. Let s be the number of submitted queries per second. The expected time interval of the nth query is

$$t(n) = \frac{1}{p(n)s} \tag{2}$$

That is, after $t(n)$ seconds, the nth query will be submitted to the system with high probability. Suppose we perform our batch indexing every t' seconds. We will keep the nth query in $\mathcal{Q}$, only if $t(n) < t'$. The intuition of this strategy is that for infrequent queries, we do not need to update the index frequently.

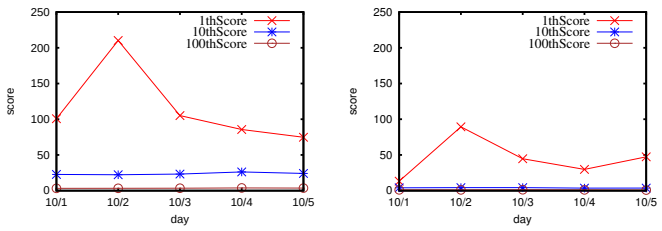
To estimate the query distribution, we keep a query log in disks. When a new unseen query arrives at the system, we assume it is an infrequent query and do not insert it into $\mathcal{Q}$. $\mathcal{Q}$ is updated during at the next batch indexing process. We search the query log to build a query histogram and simulate the distribution using Zipf's law. Based on Equation 2, popular queries are inserted into $\mathcal{Q}$.

After having defined the classification problem, a naive method can be designed directly from the definition. Suppose the tweet set is $\mathcal{T}$. Given a query $q_i \in \mathcal{Q}$, we use $\mathcal{F}(q_i, t_j)$ to denote the rank of a tweet $t_j \in \mathcal{T}$. To simplify the discussion, we define dominant set as:

Definition 2. Dominant Set
Given a tweet t, a query q and a tweet set $\mathcal{T}$, t's dominant set in relation to q is defined as the tweets that have higher ranks than t, namely

$$ds(q,t) = \{t_i | t_i \in \mathcal{T} \wedge \mathcal{F}(q,t_i) > \mathcal{F}(q,t)\}$$

A straightforward approach would compute t's dominant set for all queries in $\mathcal{Q}$. Algorithm 1 illustrates the idea. If



(a) Query "Coupon" (b) Query "Database"

Figure 6: Statistics of Keyword Ranking

there exists a query q_i satisfying $|ds(q_i, t)| < K$, we classify t as a distinguished tweet (line 3-4). Otherwise, it is a noisy tweet. Algorithm 1 suffers from two performance problems. First, to compute the dominant set, we need a full scan of the tweet set. Second, given a tweet t, we test it against every query in $\mathcal{Q}$. To address the above problems, two optimization approaches adopted.

Algorithm 1 NaiveClassifier(Tweet t, QuerySet $\mathcal{Q}$)

```
1: for ∀q_i ∈ Q do
2:     ds(q_i, t)=getDominantSet(Q, t)
3:     if ds(q_i, t).size< K then
4:         return distinguished tweet
5: return noisy tweet
```

4.1.1 Optimization 1: Top-K Threshold

The first optimization is to employ the query statistics to speed up the dominant set computation. Figure 6 shows the statistics of top-K query results in our Twitter dataset. The x-axis denotes the date of the ranking and the y-axis is the ranking score computed by our ranking function $\mathcal{F}$. The naive approach is invoked to compute the scores of pair (t_i, q_j), where t_i denotes an existing tweet by that specific day and q_j is a query in $\mathcal{Q}$. In Figure 6(a) and 6(b), we present the results for the query "coupon" and "database" respectively. Other queries share the same property. In particular, in the figures, we compare the scores of top 1 tweet, the top 10th tweet and the 100th tweet (our threshold). We find that although the score of top 1 tweet varies a lot with time, the scores of the top 10th and 100th tweet are quite stable. This is because in natural language, the words follow Zipf's distribution [12], where each word tends to appear in the text with certain frequency. Given a query, the expected number of hot tweets remains stable over time. We have the following theorem.

THEOREM 1. *Suppose each keyword appears in the tweets with a fixed probability and the tweets are inserted into the system with a stable rate. If query q_i has m results ($m >> K$), the variance of top-K score for q_i decreases for a larger K.*

PROOF. Suppose we have n tweets and there are m tweets ($m > K$) containing the search keyword. We try to estimate the Kth score of m resultant tweets, assuming they are randomly distributed in the tweet dataset. We sort the tweets by their ranks and have a list $\{t_1, t_2, ..., t_n\}$. The Kth tweet appears in the position x with probability of

$$p(x) = \frac{\binom{x-1}{K-1}\binom{n-x}{m-K}}{\binom{n}{k}}$$

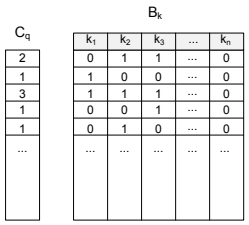


Figure 7: Matrix Index

And the expectation of top-K score is

$$E(K) = \sum_{i=k}^{n} p(i) score(i)$$

where $score(i)$ denotes the score of the ith tweet. The problem can be transformed into an order statistic problem. Based on the estimated bounds in [4], when m is sufficiently large, we get a more closer bound for $E(K)$ for a larger K. □

The above observation motivates our classification scheme. We keep a top-K threshold for each query $q \in \mathcal{Q}$, which is called threshold table T_θ. Given a query q, $T_\theta(q)$ returns the threshold for the top K tweets.

LEMMA 1. *For a tweet t, if $\mathcal{F}(q_i, t) < T_\theta(q_i)$, the size of t's dominant set is larger than K at the moment.*

PROOF. If $\mathcal{F}(q_i, t) < T_\theta(q_i)$, t's score is smaller than current Kth result. Therefore, more than K tweets have higher ranks than t. □

THEOREM 2. *For a tweet t, if $\mathcal{F}(q_i, t) < T_\theta(q_i)$ for all $q_i \in \mathcal{Q}$ and $\mathcal{F}(q_i, t)$ decreases with time, t is a noisy tweet.*

PROOF. If $\mathcal{F}(q_i, t)$ decreases with time, the tweet will never be a top-K result for a query. Thus, it is a noisy tweet. □

In Theorem 2, we require $\mathcal{F}(q_i, t)$ to be monotonically decreasing with time. In fact, in our ranking function, to catch the hotly discussed topics and discussion trend, $\mathcal{F}(q_i, t)$ may increase for a small number of hot tweets. We shall discuss how to handle such case in Section 5.2.

T_θ can be constructed and updated by Algorithm 2. Initially, T_θ's values are set to 0 for all queries. After a query is processed, we update its threshold based on the query result.

Algorithm 2 UpdateThreshold(T_θ, Query q)

```
1: Result R= getTopResult(K, q)
2: if R.size= K then
3:     Score s = R[K].score
4:     T_θ(q) = s
5: else
6:     T_θ(q) = 0
```

4.1.2 Optimization 2: Matrix Index for Queries

In Algorithm 1, computing the dominant set for every query in $\mathcal{Q}$ is time consuming. Therefore, our second optimization is to avoid unnecessary dominant set computation. We consider both queries and tweets as a bag of words. To simplify our discussion, we define the candidate query set as:

DEFINITION 3. Candidate Query
For a tweet $t = \{k_1, k_2, ..., k_n\}$ *and a query* $q = \{k'_1, k'_2, ..., k'_m\}$, *q is a candidate query for t, i.f.f.*

$$\forall k_i \in t \rightarrow \exists k'_j \in q \wedge k'_j = k_i$$

Instead of checking every query for an incoming tweet t, we just need to compute the dominant set for t's candidate queries. To facilitate the discovery of candidate queries, we propose a matrix index.

Figure 7 illustrates the index structure. B_k is a $m \times n$ matrix index (m is the size of $\mathcal{Q}$ and n is the number of unique keywords in $\mathcal{Q}$) and C_q is the counter vector for queries. Each row in B_k refers to a query and each column in B_k denotes a keyword. If the jth keyword appears in the ith query, we set $B_k[i][j]$ to 1. Otherwise, it is set to 0. C_q keeps the number of keywords in a query. The ith query has $C_q[i]$ keywords. Given a tweet t, we define its vector as $V_t = (v_1, v_2, ..., v_n)$, where $v_i = 1$ if t contains the ith keyword. Otherwise, $v_i = 0$. To find all candidate queries, we compute an evaluation vector as

$$V_e = V_t \times B_k^T \tag{3}$$

where B_k^T is the transpose of B_k. If $V_e[i] = C_q[i]$, then the ith query is a candidate query for tweet t. By applying the matrix index, we transform the discovery process of candidate queries into matrix computation. Because B_k is a sparse matrix, Equation 3 can be computed efficiently, which is shown in our optimized classification algorithm.

4.1.3 Optimized Classifier

Algorithm 3 Classifier(Tweet t, QuerySet $\mathcal{Q}$)

```
1: Array count=0
2: V_t=getTweetVector(t)
3: for j = 0 to n do
4:   if V_t[j] == 1 then
5:     for i=0 to m do
6:       if B_[i][j] == 1 then
7:         count(j)++
8:         if count[j] == C_q(j) then
9:           if t's ranking is larger than T_θ(j) then
10:            return distinguished tweet
11: return noisy tweet
```

Algorithm 3 shows our tweet classification algorithm. It is an evolution from Algorithm 1 by combining two optimization approaches. Given a tweet t, we first create a temporary counter for recording the queries that have been processed (line 1). Then we scan each column of matrix index (line 3-10). Once we detect the keyword is contained by a query (line 6), we will increase the count of the query in the temporary counter. If the counter indicates that all keywords of the queries have been seen (line 8), we will test the tweet's score against the query's threshold (line 8). If larger than the threshold, t is classified as the distinguished tweet.

In Algorithm 3, we use a temporary counter to simplify the matrix computation. As an example, in Figure 7, suppose a tweet t contains k_1, k_2 and k_3 as the keywords. We will start scanning the columns of the three keywords. By scanning the first column, we know that query q_1 and q_2 contain k_1. And after comparing with the value in counter C_q, we know q_1 is a candidate query, as it only has 1 keyword. Hence, we can compare its threshold with the score of the tweet.

We now discuss the complexity analysis of the above algorithm. Suppose we have m queries and n keywords. We need m bytes for the counter vector C_q and $\frac{nm}{8}$ bytes for the matrix index B_k. The top-K threshold is an array of floats. Therefore, its takes $4m$ bytes. Algorithm 3 incurs a storage overhead of

$$S = 5m + \frac{nm}{8} \tag{4}$$

As an example, when $m = 100000$ and $n = 5000$, we need approximately 60 MB memory. Suppose the average number of tweet's keywords is x, Algorithm 3 scans x columns of B_k. During scanning, instead of testing each bit one by one, we test the whole word. In a W-bit system, the time complexity is $\frac{xm}{W}$.

To further optimize the classification algorithm, we adopt compression technique. For each column in B_k, most bits are 0, as only a few queries contain the keyword. Therefore, we apply WAH (Word Aligned Hybrid) encoding [24] to compress the index.

4.2 Implementation of Indexes

For each incoming tweet, we will classify it as a distinguished or noisy tweet, and insert it into the index or log file for batch update. We shall present both indexing schemes in this subsection.

4.2.1 Real-Time Indexing

A new tweet that is identified as a distinguished tweet is indexed immediately. The indexing process entails the following steps,

1. If the tweet belongs to an existing tweet tree, we retrieve its parent tweet (2-3 I/Os via the index on TID) to get the root ID and generate the corresponding encoding. Then, we update the *count* number in the parent tweet. This incurs one I/O since the parent tweet has already been retrieved and cached in memory.
2. The tweet is subsequently inserted into the tweet data table, which incurs 1 I/O for the insertion and 2-3 I/Os for the index update.
3. Lastly, the tweet is inserted into the inverted index, which incurs a few I/Os depending on the number of keywords in the tweet. This is the dominant component of the indexing cost.

The first step is used to maintain the tree structure of tweets, which may incur one or two database operations. This cost can be saved, if the ranking function does not consider the effect of the tree structure. However, even in our case where the tree structure is used, this is not a major cost. Based on the statistics of [7], less than 23% of the tweets get replies, for which we need to maintain the tree structures. Furthermore, most of the tweets get replies in a relatively short period, and thus, caching the recent tweet records can significantly reduce the cost.

The main overhead of the indexing process is the cost of updating the inverted index. For a given tweet which has n keywords, we need to update the inverted list of each keyword.

4.2.2 Batch Indexing

When a noisy tweet is submitted to the microblogging system, instead of indexing it in the inverted index, we append

it to the log file. The operation is straightforward, and it incurs one I/O. The only update is to insert a new tweet tuple in the tweet data table with the cost of 2-3 I/Os. Hence, batch indexing is very efficient compared to the real-time indexing.

Periodically, the batch indexing process scans the log file and indexes the tweets in an offline manner. To reduce the cost of building the inverted index, we build an in-memory inverted index. We maintain a list for each encountered keyword in memory, and the list denotes the tweets that contain the keyword. If the memory is full, we combine the in-memory inverted index with the disk based index. In this manner, we can significantly reduce the I/Os, as the updates to an inverted list of a keyword can be performed in groups.

5. RANKING FUNCTION

In the *TI*, the indexing scheme is independent of the ranking function. The user can therefore define different ranking functions. In this section, we propose a computational efficient and effective ranking function tailored for the social networking systems by exploiting the features of user behaviors. Our proposed ranking function used is composed of the user's PageRank, popularity of the topic, the timestamp and the similarity between the query and the tweet.

5.1 User's PageRank

To capture the relationships between social networking users, we have a user graph $G_u = (U, E)$ where U denotes all the available users and E describe the links between them. In a system such as Twitter, there are two links defined for a user, the *followers* and *following*. Given a user u, its *followers* is a set of users, who follow u's tweets, while its *following* is another set of users that u currently follows. We use $f(u)$ and $f^{-1}(u)$ to denote the *followers* and *following* set of user u, respectively.

For ease of discussion, we consider G_u as a complete graph, where a user's *follower* or *following* must be another user in G_u. In a complete graph, the *following* link is analogical to the *follower* link. Therefore, in the remaining discussion, we only consider the *following* link. We build a matrix M_f to record the *following* links between users. As shown in Figure 8, if u_i follows u_j, we set $M_f[i][j]$ to 1. To compute PageRank, we also define a weight vector $V = (w_1, w_2, ..., w_n)$, where w_i is the weight of user u_i. Currently, w_i is set to 1 for all users, by assuming that every user is equally important initially. We then compute the user's PageRank as follows:

$$P_u = VM_f^x \tag{5}$$

x keeps increasing, until M_f^x converges. $P_u[i]$ denotes the PageRank value of user u_i. We normalize it as $P_u[i] = \frac{P_u[i]}{\sum_{1 \le i \le n} P_u[i]}$.

The PageRank values are stored in a user table, which is defined as *(UID, Name, PageRank)*, where UID is the ID of the user. We also have a follower and following table for capturing the friend links. In the ranking function, the tweet inherits the PageRank from its author. In particular, we define the tweet's U-PageRank as

DEFINITION 4. *U-PageRank*
Suppose the tweet t's author is u, t's U-PageRank is defined as u's PageRank value.

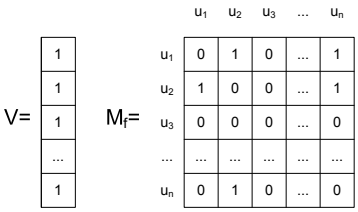


Figure 8: Following Matrix

A higher PageRank value indicates that the user has more friends and his tweets are probably more attractive than others. Therefore, we can use U-PageRank to decide whether a tweet is important for the users. In [23], an extended PageRank algorithm is also applied to rank Twitter data.

Computing the user's PageRank is costly. However, the active users in a system tend to be stable over time. Hence, the PageRank is computed in an offline manner. We can periodically, say every ten days, recompute the PageRank values. When a new user joins the system before the next computation, we set its PageRank value to 0.

5.2 Popularity of Topics

In Twitter, users retweet tweets of other people to broadcast the tweets to their friends. They also express their own ideas when replying to other's tweets. In the *TI*, tweets are grouped into a tree by the retweet/reply links. We define a tweet tree as a discussion topic or thread. To help users retrieve the popular topics, our ranking function is designed to favor the tweet trees with many discussions. This strategy is also adopted by the news group search [26] and community search [17]. In particular, given a tweet tree $\mathcal{T}$, we define its popularity as:

$$Pop(\mathcal{T}) = \sum_{\forall t_i \in T} t_i.UPageRank \tag{6}$$

As a result, the popularity of a tree is equal to the sum of U-PageRank values of all tweets in the tree. For a single node tree, the popularity of the tree is equal to the root's U-PageRank.

The tree's popularity can be computed fairly easily by joining the tweet table and user table. For example, the following query can be used for its computation.

```
SELECT SUM(U.PageRank) as Popularity, tree
FROM tweet T, user U
WHERE T.UID = U.UID
GROUP BY T.tree
```

However, processing such queries is costly, especially for a large-scale Twitter dataset. If we can reduce the number of records that need to be processed, we can effectively speed up the above query.

It is observed that more than 70% of tweets do not get any response (be replied or retweeted) [7]. For a majority of tweets, we do not need to compute the tree popularity, as the single node tree's popularity is equal to the root's U-PageRank, which can be directly obtained from the inverted index. Figure 9 verifies our assumption. It shows the changes of popularity values (without normalization). Most tweet trees exhibit the same behavior. When a tweet is published, it probably does not attract the interest of other users right away. As a result, in the first few hours, it has a low popularity. However, if the tweet discusses a popular topic,

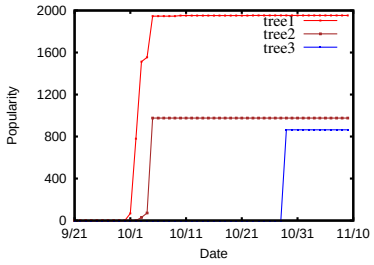


Figure 9: Popularity of Topics

it will gain continuous concern afterwards. The popularity of the corresponding tweet tree increases significantly, until the topic becomes stale some days later. Then, there will be no new tweets in this tree and the popularity remains stable after that.

We call a tweet topic that is being hotly discussed an *Active Tweet Tree*, which is defined as following:

DEFINITION 5. *Active Tweet Tree*
A tweet tree T is an active tweet tree, if the number of tree nodes keeps on increasing continuously.

For example, in Figure 9, tree 1 is an active tweet tree for tweets posted from October 1st to October 3rd. Instead of computing the popularities of all tweet trees, we just compute the popularities of active trees and maintain them in memory. By doing so, we can update the popularities of active trees efficiently when new tweets are submitted. To process the queries, we can look up the popularities kept in memory to rank the tweets.

Algorithm 4 isActiveTree(Tweet t)

```
1:  ID rid = getRootID(t)
2:  if rid is not null then
3:    if L_t.containsKey(rid) then
4:      L_t(rid).popularity += t.UPageRank
5:      L_t(rid).timestamp = t.timestamp
6:    if t.timestamp-L_c(rid).timestamp> θ then
7:      L_c(rid).count = 1
8:    else
9:      L_c(rid).count++
10:     if L_c(rid).count> γ then
11:       L_t.insert(rid, getPopularity(rid), t.timestamp)
12:       if some tweets in the tree are not indexed then
13:         create index for the tweets on the fly
14:   L_c(rid).timestamp = t.timestamp
```

In Algorithm 4, we outline the steps entailed in maintaining the active tree in memory. Initially, all the trees are assumed to be inactive trees. We keep two lists, a candidate tree list L_c and an active tree list L_t, and use hash tables to implement the lists. When a new tweet joins a tweet tree t, we use t's root ID to find its corresponding bucket in L_t and L_c. If t belongs to an active tree, we increase the tree's popularity and reset its timestamp (line 3-5). Otherwise, we retrieve t's record in L_c and compare the timestamp (line 6). If $t.timestamp - L_c(t.rid) > \theta$, we reset the counter to 1 (line 7). Otherwise, we update the timestamp and increase the value of counter by 1 (line 9). If the counter is larger than γ, we promote t as the active tree (line 10). In function $getPopularity(rid)$, we compute the popularity by issuing the query:

```
SELECT SUM(U.PageRank) as Popularity
FROM tweet T, user U
WHERE T.UID = U.UID AND T.tree= rid
```

To efficiently process the above query, we build B$^+$-tree indexes on attribute $T.UID$, $U.UID$ and $T.tree$. Recall that in Theorem 2, we require the ranking function to be decreasing with time. But for an active tree, its popularity may increase with time. Therefore, in line 12 and 13, we index all the tweets which are not yet indexed in the active tree, This can be done efficiently by following the pointers in the tweet table.

The active tree will be discarded, if it does not obtain any new tweet in more than δ time. In fact, in our ranking function, the popularity of a tree remains steady after a certain time. That is, after δ days, the rank of an inactive tree becomes too small and does not affect the top-K results. In that case, we remove it from L_t. The parameters θ, γ and δ are used to control the accuracy and memory overhead, which can be tuned based on statistics. In our experiment, θ, γ and δ are set to 8 hours, 3 tweets and 10 days respectively.

5.3 Time-based Ranking Function

The final part of our ranking function is the similarity between a query q and a tweet t. By using the bag-of-words model, we transform q and t into vectors. Their similarity is estimated as

$$sim(q,t) = \frac{q \times t}{|q||t|} \tag{7}$$

The general ranking function combines all the factors and are computed as

$$\mathcal{F}(q,t) = \frac{w_1 \times t.UPageRank + w_2 \times sim(q,t)}{q.timestamp - t.timestamp} + \frac{w_3 \times tree.popularity}{q.timestamp - tree.timestamp} \tag{8}$$

where q.timestamp denotes the time when the query is submitted, $tree$.timestamp is the timestamp of the tree that t belongs to (computed as the timestamp of the root node). In Equation 8, $UPageRank$, $sim(q,t)$ and $popularity$ are normalized into the same domain, [0, 1]. w_1, w_2 and w_3 are used to control the importances of different factors. Currently, w_1, w_2 and w_3 are set to 1, as we treat all factors equally important. If a tweet does not belong to a popular tree, we discard the second term in above formula, as in that case, the popularity should not contribute to its ranking. In our definition, a tweet's ranking is affected by its timestamp. An older tweet is less important than a newly inserted one. When searching, we prefer to the latest tweets with high similarity.

5.4 Adaptive Index Search

To process a query, the inverted index is employed to retrieve the result tweets based on the scores derived from the ranking function. In our ranking function, the PageRank value, the timestamp and the similarity can be computed based on the information in the inverted index, while the popularity can be obtained by querying the active tree list in memory. Hence, the ranking function is computationally efficient as it does not incur a significant overhead.

Nevertheless, the main problem that affects the search performance is the size of inverted index. Suppose the inverted index for keyword k_i is $\mathcal{I}_i$. The size of $\mathcal{I}_i$ will keep

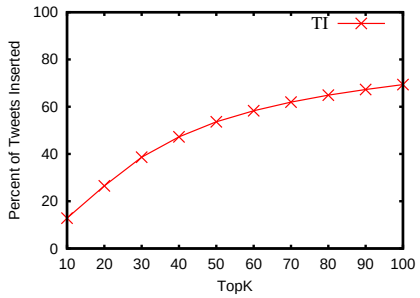


Figure 10: Number of Indexed Tweets in Real-Time

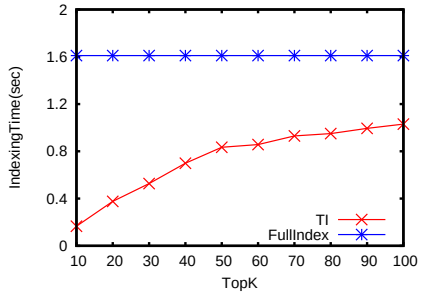


Figure 11: Indexing Cost (per 10,000 tweets)

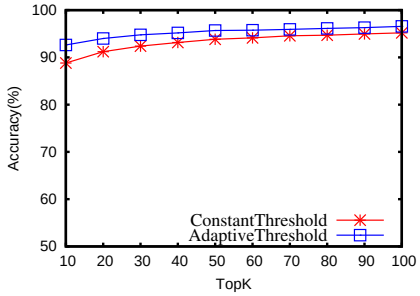


Figure 12: Accuracy of Adaptive Indexing

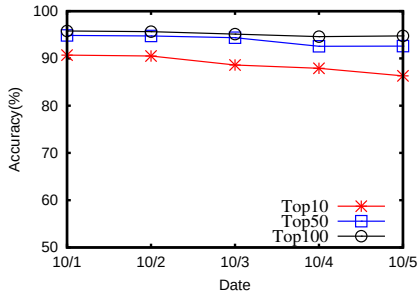


Figure 13: Accuracy by Time (constant threshold)

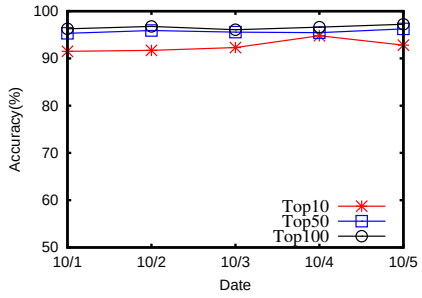


Figure 14: Accuracy by Time (adaptive threshold)

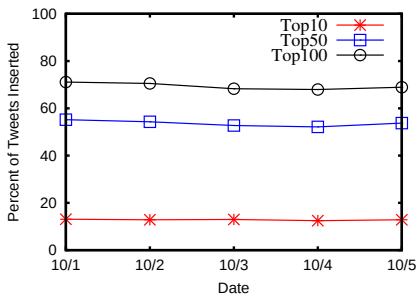


Figure 15: Effect of Adaptive Threshold

increasing, as more tweets are inserted.To address this problem, we propose an adaptive index searching scheme. The maximal possible score of a tweet at timestamp ts is estimated as:

$$score = \frac{w_1 \times UPageRank_{max} + w_2 + w_3 \times popularity_{max}}{q.timestamp - ts}$$

$UPageRank_{max}$ denotes the maximal user PageRank. We set similarity to 1. And $popularity_{max}$ is estimated by current active tree set. Let S_{tree} denote the active trees that have a timestamp before ts. If no such tree exists, $popularity_{max}$ is set to 0. Otherwise, $popularity_{max}$ equals to the maximal popularity in S_{tree}.

Let $\mathcal{T}_\theta(q)$ be the top-K threshold for query q. Instead of reading the whole inverted index blindly, we iteratively read a block of the index. If the last entry in the block has a timestamp ts and based on the above equation, the maximal score before ts is smaller than $\mathcal{T}_\theta(q)$, we will stop reading the index, since the remaining tweets will not contribute the the search results. This strategy effectively reduces the index search cost.

6. EXPERIMENTAL EVALUATION

In this section, we shall evaluate the performance of the TI indexing scheme and the effectiveness of the propose ranking functions. In the experiments, we use a Twitter dataset collected for three years [6] from October 2006 to November 2009. 500 random users are selected from Twitter as the seeds, including politicians, musicians, environmentalists and techies. Following the friend links, more users are discovered and added into the social graph. The total number of involved users is about 465K. For each user, the tweets are crawled every 24 hours. There are more than 25 million of tweets in the dataset.

In the experiments, we start from September 26 2009 and simulate users' behavior for ten days. The first five days are used to warm up the system (e.g. building the top-K threshold, learning the popularities of topics). The remaining five days are used to measure the performance. We collect keywords from the first five days' tweets. After removing the keywords in the stop-list and the infrequent words (frequency less than 10), we have less than 5K keywords left. Queries in real-time search engine follows a skewed distribution [10]. Therefore, in the experiments, queries are generated by randomly combining the keywords, and the number of keywords in queries follows Zipf's distribution. Approximately, 60% are 1-word queries; 30% are 2-word queries; and 10% are queries with more than two keywords. The queries are submitted to the system at random timestamps, while the tweets are inserted into the system based on their recorded timestamps. Each experiment is repeated for ten times and the average result is reported.

6.1 Effects of Adaptive Indexing

In the first set of experiments, we study how the adaptive indexing scheme affects the performance. In Figure 10, we show the percentage of tweets that are indexed in real-time. When only top-10 results are required, we can prune more than 80% of tweets (by using batch indexing scheme). As more results are returned to users, more tweets need to be indexed to be searchable. Because only a portion of tweets need to be indexed in real-time, the indexing cost is significantly reduced. Figure 11 compares the indexing time of TI and full indexing scheme. In TI, the cost of indexing is proportional to the number of indexed tweets. Therefore, when more tweets are required in the results, TI will incur higher indexing overhead.

To evaluate whether the adaptive indexing scheme reduces

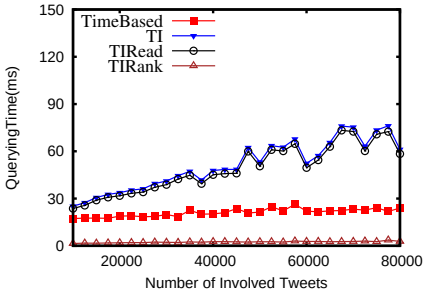


Figure 16: Performance of Query Processing

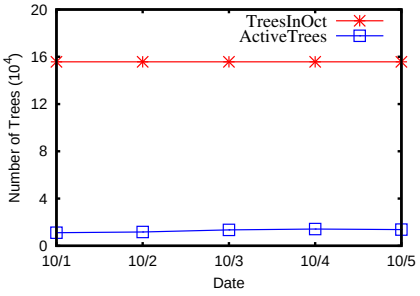


Figure 17: Popular Tree in Memory

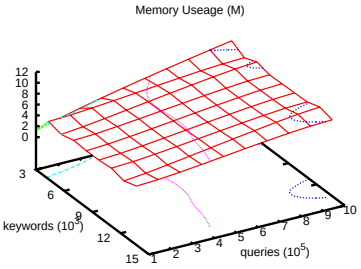


Figure 18: Size of In-memory Index

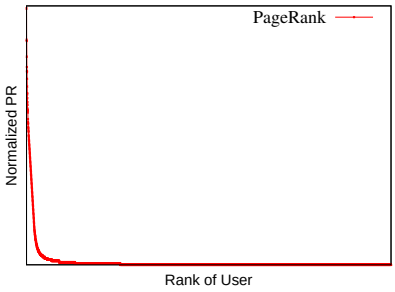


Figure 19: Distribution of PageRank

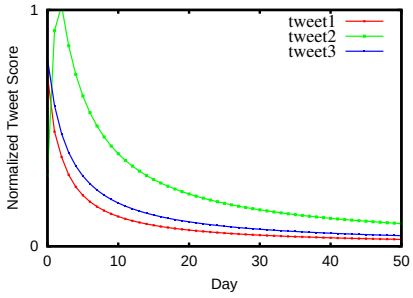


Figure 20: Popularity of Tweets by Time

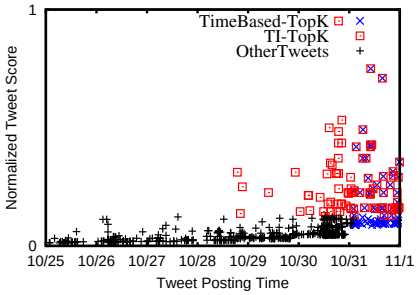


Figure 21: Distribution of Results

the quality of results, we compute the query accuracy as $accuracy = \frac{R \bigcap R'}{|R|}$, where R denotes the result set returned by full indexing scheme (all tweets are inserted in real-time), R' denotes the result set returned by TI, $R \bigcap R'$ represents the number of tweets in both result sets. Figure 12 shows the accuracy of TI's results. For comparison, we use two strategies. In *Constant Threshold*, we do not update the top-K threshold when processing queries. On the contrary, in *Adaptive Threshold*, we use Algorithm 2 to update the threshold adaptively. As shown in Figure 12, the accuracy of *Constant Threshold* is just slightly worse than *Adaptive Threshold*. The result verifies our observation made in Figure 6, where the top-K threshold remains stable in a period of time. The accuracy of both strategies decreases as K decreases. This can also be observed in Figure 6. When K is small, the top-K threshold changes more significantly. An extreme case is when $K = 1$. Thus, the TI may wrongly delay indexing some high ranking tweets. This problem can be fixed by setting a lower bound, e.g. 20, for K. Although user only requests for top 1 result, we always maintain the threshold for top 20 results.

In Figure 13 and Figure 14, we show the changes of accuracy by dates. The accuracy of *Constant Threshold* degrades, because it never updates its threshold values. However the quality of the results is still acceptable. For *Adaptive Threshold*, as the threshold is updated by the queries, we always get results with high accuracy. In Figure 15, we show the percentage of indexed tweets in *Adaptive Threshold* by dates. We can observe from the figure that the *Adaptive Threshold* scheme does lead to a stable performance, independent of K. As the *Adaptive Threshold* exploits the query results to update its threshold, which is almost free, we will always use *Adaptive Threshold* strategy in the TI indexing scheme.

6.2 Query Performance

To provide better search results, TI adopts a sophisticated ranking function. In this experiment, we study whether the ranking function leads to a better query performance. For comparison purposes, we implement a tweet search, which only ranks tweets via their timestamps. Such ranking strategy has been adopted by Twitter and Google's real-time search. As we sort the tweets in the inverted index by their timestamps, for a single keyword query, we just need to read the first K entries from the index, which is quite efficient. For a multi-keyword query, we iteratively read a block of the index for all keywords, and we stop when K results are obtained; otherwise, more blocks are searched.

Figure 16 shows the query performance of the TI and time-based ranking schemes. TI's costs are decomposed into two parts, the ranking cost $TIRank$ and the index search cost $TIRead$. We group queries by their total number of involved tweets. In Figure 16, the x-axis ranges from 0 to 80000, indicating that some popular queries get about 80000 hits in our dataset. Since the size of the inverted index for a keyword k_i is proportional to the number of tweets containing k_i, the index search cost increases as more tweets are involved. This is verified by the results. We have adopted some optimization approaches, such as the adaptive index search outlined in Section 5.4, in order to reduce the cost. As shown in Figure 16, $TIRead$ increases linearly with the number of involved tweets. We can further reduce the search cost by distributing the inverted index over a set of compute nodes and applying the parallel search. We will study the problem in our future work. On the contrary, the time-based ranking scheme only retrieves some top tweets, and hence, incurs less overhead. However, it achieves the efficiency by sacrificing the quality of results. Without a reasonable ranking scheme, the query results are less useful.

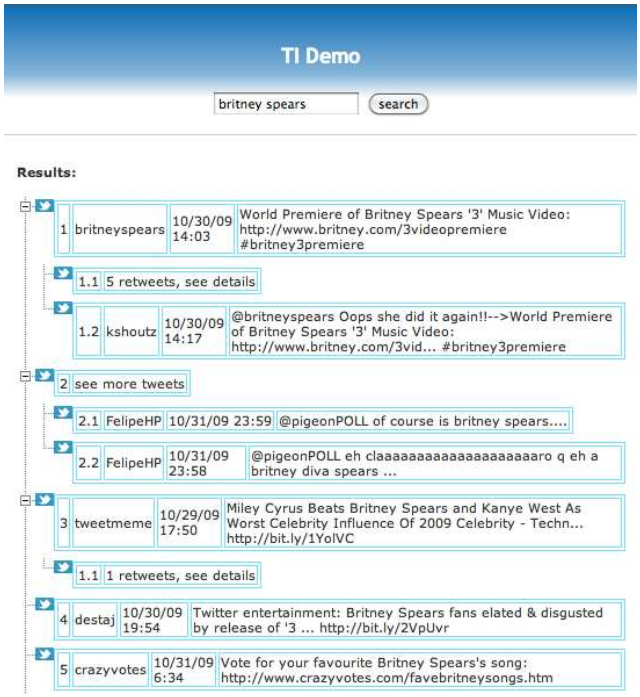


Figure 22: Search Result Ranked by TI

Figure 23: Search Result Ranked by Time

6.3 Memory Overhead

In this experiment, we evaluate the memory overhead in our system. We have maintained some memory structures to support adaptive indexing and efficient ranking. One structure is the active trees. Figure 17 shows the number of active trees. For comparison, we also show the number of total trees generated in October,2009, where less than ten percent of the trees, approximately 13000 trees, are identified as active trees. Moreover, we observe that the number of active trees does not increase with time. In conclusion, the memory requirement is well controlled and is not high.

Another memory structure is the matrix index. Given n keywords and m queries, we need $\frac{nm}{8}$ bytes to maintain the index. To reduce the overhead, we adopt WAH encoding to compress the matrix index. Figure 18 shows how the size of in-memory index changes for different n and m. We change the number of keywords from 3000 to 15000 and the number of queries from 100000 to 1 million. The maximal memory usage is only 12 MB, which indicates that the matrix index is very cost-efficient and we can maintain a much larger one for holding most keywords and queries. Another interesting observation is that the memory use does not necessarily increase even when more keywords and queries are used. This is because more keywords and queries lead to more 0s and 1s in the matrix index, which improves the compression performance of the WAH.

6.4 Ranking Comparison

In the ranking function, we have three components, the similarity between query and tweets, the PageRank of authors and the popularity of topics. Figure 19 shows the distribution of users' PageRanks in our dataset. It is not surprising that the PageRank value follows a highly skewed distribution, resembling that of Zipf's or power law distribution. Figure 20 shows the effects of time over the score of tweets. In the figure, X-axis represents the elapsed time, where 0 indicates the starting time of the tweets. Y-axis is a score computed by Equation 8. In our ranking function, the score is inversely proportional to time. Thus, the score of a specific tweet will decrease with time. However, a few popular tweets receive many replies within a short period of time after they are posted, contributing to a sudden rise in its score. Figure 21 illustrates the ranking scores over the post time of tweets. In the figure, the X-axis is the posting time of tweets, while the Y-axis is the score computed by removing the denominators in Equation 8. We use "Britney Spears" as our query. Based on observation of the results, time-based ranking scheme retrieves all recent queries as its top results, while our approach considers both time and other factors, which provides better results.

We show a demo result in Figure 22 and Figure 23. The search is processed by assuming the time is at *Nov 1, 2009 00:00:00*, when the last tweets in our dataset were crawled (some tweets after Nov 1 are considered as noisy and pruned). For each result, we show its ranking, author, timestamp and content. In Figure 22, we show the result of TI, where tweets are ordered by our ranking function. The first three tweets form a group, as they belong to the same tweet tree. The first tweet is posted by the official account of Britney Spears to publish a new video link. The second one represents 5 retweets. We aggregate them together, for all tweets have the same content. The third tweet is a reply to the first tweet, which shows the song name of the shared video. By grouping tweets via their tree structures, we provide a better visualization result.

In Figure 23, we show the result of time-based ranking, where tweets are strictly sorted by their timestamps. As a matter of fact, most results in Figure 23 also appear in Figure 22. And many results in Figure 23 are duplicates. This is because when a hot tweet is published, many users will

retweet it within a short time after that. Another problem of the time-based results is the lack of tree structures. Both the first and second tweets are replies to another tweet, but the time-based scoring function shows them individually, while the TI scheme groups them together, presenting the results more meaningfully.

7. CONCLUSION

The quest for real-time indexing has recently become more pressing due to the inability of search engines in indexing and retrieving the huge amount of social networking data as soon as they are produced. The problem is further exacerbated by the increasing popularity of microblogging systems where millions of tweets are produced each day. In this paper, we have proposed the Tweet Index (TI), a new indexing and ranking scheme for supporting real-time search in microblogging systems. The TI adopts an adaptive indexing scheme to reduce the update cost. To this end, a new tweet is indexed only if it may appear in the top-K results of some cached queries with high probability. Otherwise, it is grouped with other unimportant tweets, and a batch indexing scheme is used to reduce the indexing latency. We have also proposed a cost-efficient and effective ranking function, by taking the users' PageRank, the popularity of topics, the similarity between the data and the query, and the time into consideration. To evaluate the performance of the TI indexing scheme and ranking function, we have conducted an extensive experimental study using a real dataset from Twitter. The experimental results show that the TI is efficient in handling tweets as they are produced and is able to achieve high query effectiveness and efficiency at the same time.

8. ACKNOWLEDGEMENTS

The work of Chun Chen was in part supported by National Natural Science Foundation of China (Grant No. 61070155). The work of Feng Li, Beng Chin Ooi and Sai Wu was in part supported by Singapore MDA grant R-252-000-376-279.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[2] L. Backstrom, J. Kleinberg, R. Kumar, and J. Novak. Spatial variation in search engine queries. In *WWW*, pages 357–366, 2008.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.

[4] D. Bertsimas, K. Natarajan, and C.-P. Teo. Tight bounds on expected order statistics. *Probab. Eng. Inf. Sci.*, 20(4):667–686, 2006.

[5] R. Chirkova, C. Li, and J. Li. Answering queries using materialized views with minimum size. *The VLDB Journal*, 15(3):191–210, 2006.

[6] M. D. Choudhury, Y.-R. Lin, H. Sundaram, K. S. Candan, L. Xie, and A. Kelliher. How does the sampling strategy impact the discovery of information diffusion in social media? In *ICWSM*, 2010.

[7] S. Inc. Replies and retweets on twitter. 2010.

[8] iProspect. iprospect search engine user behavior study.

[9] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

[10] B. J. Jansen, G. Campbell, and M. Gregg. Real time search user behavior. In *CHI*, pages 3961–3966, 2010.

[11] A. Java, X. Song, T. Finin, and B. Tseng. Why we twitter: understanding microblogging usage and communities. In *WebKDD*, pages 56–65, 2007.

[12] W. Li. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, pages 1842–1845, 1992.

[13] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB*, pages 432–443, 2004.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Technical Report, Stanford University*, 1998.

[15] T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *WWW*, pages 851–860, 2010.

[16] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. Twitterstand: news in tweets. In *GIS*, pages 42–51, 2009.

[17] J. Seo, W. B. Croft, and D. A. Smith. Online community search using thread structure. In *CIKM*, pages 1907–1910, 2009.

[18] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE*, pages 420–427, 1995.

[19] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, pages 831–842, 2010.

[20] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.

[21] A. Sun, M. Hu, and E.-P. Lim. Searching blogs and news: a study on popular queries. In *SIGIR*, pages 729–730, 2008.

[22] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[23] J. Weng, E.-P. Lim, J. Jiang, and Q. He. Twitterrank: finding topic-sensitive influential twitterers. In *WSDM*, pages 261–270, 2010.

[24] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, 2002.

[25] S. Wu, J. Li, B. C. Ooi, and K.-L. Tan. Just-in-time query retrieval over partially indexed data on structured p2p overlays. In *SIGMOD*, pages 279–290, 2008.

[26] W. Xi, J. Lind, and E. Brill. Learning effective ranking functions for newsgroup search. In *SIGIR*, pages 394–401, 2004.

[27] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

[28] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001.