LSII: An Indexing Structure for Exact Real-Time Search on Microblogs

Lingkun Wu[1,2], Wenqing Lin[1], Xiaokui Xiao[1], Yabo Xu[3]
[1]*Nanyang Technological University, Singapore*
{wulingkun, wlin1, xkxiao}@ntu.edu.sg
[2]*A*STAR, Singapore*
[3]*Sun Yet-Sen University, China*
xuyabo@mail.sysu.edu.cn

***Abstract*—Indexing microblogs for real-time search is challenging given the efficiency issue caused by the tremendous speed at which new microblogs are created by users. Existing approaches address this efficiency issue at the cost of *query accuracy*, as they either (i) exclude a significant portion of microblogs from the index to reduce update cost or (ii) rank microblogs mostly by their timestamps (without sufficient consideration of their relevance to the queries) to enable append-only index insertion. As a consequence, the search results returned by the existing approaches do not satisfy the users who demand timely and high-quality search results.**

To remedy this deficiency, we propose the *Log-Structured Inverted Indices (LSII)*, a structure for *exact* real-time search on microblogs. The core of LSII is a sequence of inverted indices with exponentially increasing sizes, such that new microblogs are (i) first inserted into the smallest index and (ii) later moved into the larger indices in a batch manner. The batch insertion mechanism leads to a small amortize update cost for each new microblog, without significantly degrading query performance. We present a comprehensive study on LSII, exploring various design options to strike a good balance between query and update performance. In addition, we propose extensions of LSII to support personalized search and to exploit multi-threading for performance improvement. Extensive experiments demonstrate the efficiency of LSII with experiments on real data.

I. Introduction

Microblogging is an increasingly popular form of online communication where users broadcast short textual messages to the public or selected peers. Major microblogging sites, such as Twitter, have tremendous numbers of users that create hundreds of millions of microblogs per day [1]. Such a high rate of microblog generation poses significant challenge to real time search on microblogs, as conventional indices for keyword queries (e.g., inverted indices) are not designed to accommodate high-frequency updates. To address this challenge, two more advanced indexing structures, TI [2] and EarlyBird [1], have been proposed to reduce update overhead at the cost of *query accuracy*.

In particular, TI adopts a partial indexing approach that distinguishes *frequent* keyword queries from the *infrequent* ones. For any new microblog, if it is relevant to at least one frequent query, it is inserted into an inverted index immediately; otherwise, it is inserted to an append-only log, the content of which is periodically flushed into the inverted index. In addition, any keyword query is processed using the inverted index without looking into the log, i.e., the results for an infrequent query may be *incomplete* if a microblog relevant to the query is recorded in the log but not the inverted index.

Contact authors: Yabo Xu and Xiaokui Xiao

The above partial indexing approach of TI improves update efficiency as it reduces the number of microblogs that need to be processed in a real-time manner. However, it also causes two severe deficiencies as follows. First, as illustrated in existing study [3] on search logs, infrequent queries could account for up to 70% of the queries issued by the users. Hence, treating infrequent queries as second-class citizens could lead to negative search experience for a large number of users. Second, recent work [4] shows that the distribution of real-time queries on microblogs could change substantially within minutes, and hence, an infrequent query could become frequent even before a microblog relevant to the query is moved from the append-only pool of TI to the inverted index. In other words, TI may return incomplete results even for a query that is frequent.

In contrast to TI (which does not guarantee completeness of query results), EarlyBird [1] ensures that all new microblog can be searchable. Specifically, EarlyBird employs an append-only inverted index where each posting list stores microblogs in ascending order of their timestamps. Given a keyword query, EarlyBird scans each relevant posting list in descending order of the timestamps of the microblogs, after which it returns the top-k microblogs that match the query, with newer microblogs being ranked higher than older ones. The append-only nature of EarlyBird allows new microblogs can be efficiently inserted. Nevertheless, as EarlyBird ranks query results based only by time, it overlooks several important factors that could significantly affect the ranking of results, such as the popularity of a microblog's author and the popularity of the microblog itself (i.e., the number of times that the microblog is replied or forwarded by other users). As a consequence, EarlyBird could miss important results for a query if the results are not ranked high enough in terms of time. This issue may be partially alleviated if we first retrieve a relatively large set of microblogs based on time, and then re-rank the microblogs in the set using a more comprehensive approach. However, this method still does not guarantee the completeness of query results.

Contributions and Organization. To address the deficiency of the existing methods, we propose the *Log-Structured In-*

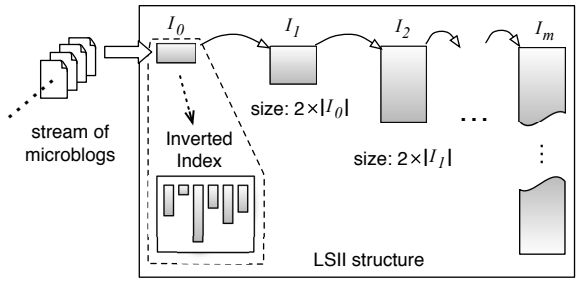

Fig. 1. The LSII Structure

verted Indices (LSII), an indexing method that returns *exact* results for real-time search on microblogs. LSII is similar in spirit to the *Log-Structured Merge-Tree* [5]: It contains a sequence of inverted indices $I_0, I_1, \ldots, I_m$ with exponentially increasing sizes, as illustrated in Figure 1. When a new microblog arrives, it is first inserted into the smallest inverted index I_0 – this incurs little overhead given the tiny size of I_0. When the number of microblogs indexed by I_0 exceeds a certain threshold, we merge the entries of I_0 into I_1, and we empty I_0; in turn, the entries in I_1 are flushed into I_2 when I_1 gets to a certain size, and so on. This considerably reduces the amortized update cost per microblog, since (i) each microblog can be only involved in a small number of index mergers, and (ii) the index mergers can be performed in a much more efficient manner than inserting all microblogs into an inverted index one by one. In addition, LSII also provides accurate and timely answers to keyword queries. Each inverted index maintained by LSII records not only the timestamp of each microblog but also other relevant information that may affect the ranking of query results. During query processing, all inverted indices are scanned simultaneously using the Threshold Algorithm [6] to efficiently compute an answer set that takes into account both the microblog timestamps and other relevant factors.

We present a comprehensive study on the formulation and extensions of LSII. In particular, in Section IV, we explore various design options of LSII to strike a good balance between query and update performance. In Section V, we propose an extension of LSII for *personalized search*, i.e., real-time search on the microblogs posted by a selected set of users. Such queries are of practical importance as users of microblogging services are usually interested only in the microblogs posted by their friends or celebrities. In Section VII, we devise a multi-threading approach that enables LSII to parallelize query and update processing, so as to reduce system response time. Finally, in Section IX, we present extensive experimental results that demonstrate the superiority of LSII over several alternative solutions.

II. Problem Formulation

Assume that we have a microblogging server that receives a finite stream of tuples in increasing order of timestamps, such that each tuple represents either a microblog or a query. Each microblog is defined as a multi-set of words (referred to as *terms*) from an alphabet Ω. A microblog d can be the *descendant* of another microblog d' with a smaller timestamp, i.e., d replies or forwards d'.

A query q is defined as a set of terms along with a positive number k_q. It asks for k_q microblogs that are not only relevant to the query terms but also *fresh*, i.e., the timestamps of the retrieved microblogs should be smaller than and close to the timestamp of q. We emphasize on freshness (in addition to relevancy), as users of microblogging service are generally more interested in the latest available microblogs, e.g., the microblogs talking about breaking news [1], [4]. Accordingly, we propose to rank each microblog d with a score function in the following form:

$$f(d,q) = w_1 \cdot sig(d) + w_2 \cdot sim(d,q) + w_3 \cdot fresh\,(ts_d, ts_q) \quad (1)$$

such that

1) $w_1 + w_2 + w_3 = 1$ and $w_1, w_2, w_3 > 0$,
2) $sig(d)$ quantifies the *significance* of d, e.g., d is considered significant if it is posted by a famous user or is replied or forwarded a large number of times,
3) $sim(d,q)$ measures the relevance of d to q,
4) $fresh(ts_d, ts_q)$ gauges the freshness of d with respect to q based on d's timestamp ts_d and q's timestamp ts_q.

In other words, $f(d,q)$ is a linear function of d's scores in terms of its significance, freshness, and relevance to q. This formulation of $f(d,q)$ generalizes several score functions used in previous work [7], [8].

Our solution is not specific to the definitions of $sig(d)$ and $fresh(ts_d, ts_q)$, as long as $fresh(ts_d, ts_q)$ is a monotonically decreasing function of $ts_q - ts_d$. In our experiments, we compute $sig(d)$ using the algorithm adopted in TI [2] (see Section VIII for details), and we follow previous work [9] to set $fresh(d,q) = e^{c\cdot(ts_d - ts_q)}$, where c is a constant deciding how fast the freshness of d degrades with time.

On the other hand, we consider that $sim(d,q)$ equals the *cosine similarity* between d and q, as suggested in previous work [2]. Specifically, let t_i ($i \in [1, |\Omega|]$) denote the i-th term in the alphabet Ω, and let v_d and v_q be two vectors with length $|\Omega|$, such that $\|v_d\|_1 = \|v_q\|_1 = 1$ and the i-th element in v_d (resp. v_q) quantifies the importance of t_i in d (resp. q). Then,

$$sim(d,q) = v_d \cdot v_q. \quad (2)$$

We refer to v_d (resp. v_q) as the *term vector* of d (resp. q), and we use $v_d[t]$ (resp. $v_q[t]$) to denote the element in v_d (resp. v_q) that corresponds to a term t. In our experiments, we compute the term vectors based on the tf-idf model [10].

In addition, we require that only microblogs d with $sim(d,q) > 0$ can be returned as query results, so as to avoid giving completely irrelevant answers to users' queries. This implies that d and q should have at least one common term.

In summary, each query q asks for the k_q microblogs d that maximizes $f(d,q)$ among those microblogs with timestamps smaller than ts_q and with $sim(d,q) > 0$. Our objective is to minimize the total time required to process the stream of tuples

TABLE I
TABLE OF NOTATIONS

Notation	Description
$t \in \Omega$	a term t in an alphabet Ω
d	a microblog consisting of terms in Ω
q	a query consisting of terms in Ω and a positive number k_q
k_q	the number of microblogs that q asks for
ts_d, ts_q	the timestamps of d and q, respectively
v_d, v_q	the term vectors of d and q, respectively
$v_d[t]$	the element in v_d that corresponds to a term t
$v_q[t]$	the element in v_q that corresponds to a term t
$f(d, q)$	the ranking function that computes the score of d with respect to q (see Eqn. 1)
$sig(d)$	the significance score of d
$sim(d, q)$	the cosine similarity between d and q (see Eqn. 2)
$fresh(ts_d, ts_q)$	the freshness score of d with respect to q
I_i $(i \in [0, m])$	the inverted indices in LSII
τ_i $(i \in [0, m])$	the maximum number of microblogs indexed by LSII

received by the server. We assume that the server has sufficient memory to accommodate all indices built on the microblogs, as is the case for modern search engines [1]. For ease of exposition, we also assume that the timestamp, term vector, significance score of each microblog are fixed, and we will discuss how this assumption can be removed in Section VI. Table I lists the notations that we will frequently use.

Remark. The above problem formulation assumes that tuple stream is finite, while in practice the number of microblogs received by the server increases monotonically with time. However, as users are generally not interested in *stale* microblogs, the server only needs to retain the microblogs whose timestamps are larger than a certain threshold. For example, Twitter only indexes 6-9 days' microblogs [11]. Our solution for finite tuple streams can be easily adopted to handle this case.

III. FIRST-CUT SOLUTIONS

Before presenting our solution, we first discuss the possibility of using conventional inverted indices for real-time search on microblogs. Given a set D of documents, an inverted index on D consists of a hash table and a set of *posting lists*. Each posting list corresponds to a term t in the alphabet Ω, and it contains an entry for each document in which t appears. The hash table, on the other hand, maps each term to its corresponding posting list. Given the posting lists and the hash table, if we are to retrieve the k_q documents with highest cosine similarly to a query q, we can first use the hash table to locate the set S of posting lists corresponding to terms in q, and then we scan the entries in those posting lists to identify the relevant documents. The efficiency of the scanning process depends on the order of entries in each posting list, as will be explained shortly. For convenience, we say that posting lists in S are *relevant* to q.

A. Append-Only Approach

To adopt an inverted index for microblog search, the simplest approach (as adopted in EarlyBird [1]) is to treat each microblog d as a document, and sort the entries in each posting list in ascending order of the corresponding microblogs' timestamps. This approach is highly efficient in terms of microblog insertions, as new microblogs can be easily appended to the ends of posting lists without affecting the ordering of entries. In terms of query efficiency, however, the aforementioned approach is rather unfavorable. To explain this, recall that our ranking function $f(d, q)$ evaluates a microblog d based on three factors: its significance $sig(d)$, relevance $sim(d, q)$, and freshness $fresh(d, q)$. If the entries in a posting list are sorted in ascending order of timestamps, the corresponding microblogs would be in ascending order of freshness, regardless of their significance or relevance. In other words, the entry order does not provide any hint on the overall score of each microblog. As a consequence, when answering a query q, we have to examine all entries in all posting lists relevant to q, since the omission of any entry may render the query results incomplete. This leads to significant query overhead.

B. Triple-Posting-List Approach

To improve query efficiency, a natural idea is to extend the inverted index to incorporate information about the significance and relevance of microblogs. In particular, we can maintain three posting lists for each term t, such that (i) the first and second lists sort microblogs in descending order of their timestamps and significance scores, respectively, and (ii) the third list arranges the microblogs d in descending order of $v_d[t]$, i.e, the weight of t in the term vector of d. Such entry ordering makes it possible to compute query answers without traversing all entries in the relevant posting lists. For example, if we have visited the first x entries in a posting list that records significance scores, and the x-th entry has a significance score y, then we know for sure that the significance scores of the unvisited entries in the posting list are upper-bounded by y. By combining the score upperbounds from all relevant posting lists, we can derive (based on Equation 1) an upperbound of $f(d, q)$ for any unseen microblog d. If this upperbound is smaller than the scores of the top-(k_q) microblogs that we have examined, then we can stop traversing the posting lists without missing query results.

More specifically, we process each query q with the Threshold Algorithm (TA) [12], a standard technique for top-k queries. Given q, TA first identifies the three posting lists corresponding to each term in q. This results in $3 \cdot |q|$ posting lists, where $|q|$ denotes the number of terms in q. After that, TA traverses the posting lists in an iterative manner. In the i-th iteration, TA examines the i-th microblog d in each of the $3 \cdot |q|$ posting lists, and computes d's overall score $f(d, q)$. A microblog would be retained in a candidate pool if its overall score ranks among the top-(k_q) microblogs found in the first $i - 1$ iterations.

In addition, the i-th iteration of TA also derives an upperbound $sig_i^\top$, $sim_i^\top$, and $fresh_i^\top$ for the significance, relevance, and freshness scores of any unseen microblog. In particular,

$$sig_i^\top = \max_{d \in D_i} \{sig(d)\}, \quad (3)$$

where D_i contains the i-th microblog in each of the $|q|$ posting lists sorted by significance scores. Similarly,

$$fresh_i^\top = \max_{d \in D'_i} \{fresh(ts_q - ts_d)\}, \quad (4)$$

where D'_i contains the i-th microblog in each of the $|q|$ posting lists sorted by timestamps. Furthermore,

$$sim_i^\top = \sum_{t \in q} v_{di}[t] \cdot v_q[t], \quad (5)$$

where v_{di} denotes the term vector of the i-th microblog in the posting list storing term weights corresponding to t. Given $sig_i^\top$, $sim_i^\top$, and $fresh_i^\top$, TA computes the upperbound of the overall score of any unseen microblog based on Equation 1. If the upperbound is no better than the scores of the microblogs in the candidate pool, then TA terminates and returns the content of the candidate pool as the final results.

In summary, by maintaining sorted posting lists for microblogs' significance, relevance, and timestamps, we can achieve efficiency in processing real-time search. For convenience, we refer to this indexing approach as the *Triple-Posting-List (TPL)* approach. Although TPL is efficient in terms of query processing, it causes much update overhead, as we need to retain the sorted order of posting lists under frequent microblog insertions. One may attempt to address this issue by organizing each posting list into a main-memory B-tree (so as to reduce update cost), but it would significantly degrade query performance. To explain this, recall that the TA algorithm computes query results using linear scans on the posting list entries. When the entries are stored in a B-tree, the linear scan would be done at the leaf level of the tree. As the leaf-nodes of the B-trees are often not located in a contiguous memory space, the linear scan is likely to incur a large number cache line misses, leading to increased query processing time.

The deficiencies of TPL and the append-only approach demonstrate that conventional inverted indices fail to strike a balance between query and update efficiency for microblog search. This motivates our LSII structure which combines the advantages of TPL and the append-only method without suffering from their drawbacks, as will be elaborated in Section IV.

IV. Log-Structured Inverted Indices

This section presents the Log-Structured Inverted Indices (LSII) for real-time search on microblogs. Section IV-A first explains the basic idea of LSII, and then, Sections IV-B and IV-C discuss the implementation details and design considerations.

A. Basic Idea

LSII consists of a sequence of inverted indices $I_0, I_1, \ldots, I_m$, each of which indexes a disjoint subset of microblogs, as illustrated in Figure 1. The structure of I_0 is identical to the append-only approach in Section III-A, i.e., I_0 contains one posting list for each term t, and the entries in the list are sorted in ascending order of timestamps. Meanwhile, I_i ($i \in [1, m]$) has the same structure with TPL, i.e., I_i maintains three posting lists for each term t, such that the first, second, and third lists sort microblogs d in descending order of $sig(d)$, $v_d[t]$, and ts_d, respectively. We impose a threshold τ_i ($i \in [0, m]$) on the number of microblogs that I_i is allowed to index, and we set $\tau_i = 2\tau_{i-1}$ ($i \in [1, m]$), i.e., the maximum size of I_i is 2 times the maximum size of I_{i-1}.

Whenever a new microblog arrives, it is first inserted into I_0. This entails little overhead given the append-only nature of I_0. If I_0 already reaches its maximum size before the insertion, we first merge the posting lists in I_0 with the posting lists in I_1. After that, I_0 becomes empty, and we will proceed to insert the new microblog. In general, if the number of microblogs indexed by I_i ($i \in [0, m-1]$) reaches τ_i, we combine the posting lists in I_i with those in I_{i+1}. When I_m becomes full, we create a new inverted index I_{m+1} with size threshold $\tau_{m+1} = 2\tau_m$, and we move all posting lists in I_m to I_{m+1}. Initially, when no microblog has arrived, we set $m = 0$, in which case LSII contains only one index I_0.

The aforementioned update strategy of LSII has several important implications. First, the timestamp of any microblog in I_i ($i \in [1, m]$) is larger than the timestamp of any microblog in I_{i-1}. As a consequence, the microblogs in I_i have smaller freshness scores than those in I_{i-1}, which makes the microblogs in I_i less likely to be query results.

Second, as all microblog insertions are handled by I_0, the structure of the other indices $I_1, \ldots, I_m$ are usually unchanged (except when index mergers are performed). Therefore, we can implement the posting lists in $I_1, \ldots, I_m$ as sorted arrays, which ensures that linear scans of the posting lists would be efficient – this is crucial for the performance of our query algorithm (i.e., the TA approach). In addition, the sorted arrays also enable efficient index merger: merging any two indices in $I_1, \ldots, I_m$ would only incur linear cost, since we can combine two sorted arrays in linear time.

Third, the number of inverted indices in LSII (i.e., $m + 1$) is $O(\log n)$, where n is the number of microblogs that has arrived. To understand this, recall that $m = 0$ initially, and LSII would increase m (i.e., create a new index) only when I_m reaches its maximum size τ_m. Therefore, if LSII contains an index I_m, then I_{m-1} must have become full at least once. This indicates that $n \geq \tau_{m-1} = 2^{m-1}\tau_0$, which leads to $m + 1 = O(\log n)$.

Fourth, each microblog d can be involved in less than $O(m)$ index mergers. To explain this, consider that d is currently indexed by I_i. Observe that I_i can be merged with I_{i-1} at most twice before it becomes full (since $\tau_i = 2\tau_{i-1}$), after which it would be merged with I_{i+1}. Hence, d can be involved in a

Algorithm *LSII-Query* (q, I_0, ..., I_m)

1. let C be a candidate pool of size k_q; set $C = \emptyset$
2. scan I_0 to identify the k_q microblogs in I_0 that with the largest overall score, and put them into C
3. let $d^{\perp}$ be the microblog in C with the smallest overall score
4. let $j = 1$
5. initialize m upperbounds $ub_1 = ub_2 = \ldots = ub_m = +\infty$
6. while $\max_i\{ub_i\} > f(d^{\perp}, q)$
7. for any $i \in [1, m]$ such that $ub_i > f(d^{\perp}, q)$
8. for the j-th microblog d in any posting list in I_i that is relevant to a term in q and contains at least j entries
9. if $f(d, q) > f(d^{\perp}, q)$
10. remove $d^{\perp}$ from C and insert d into C
11. let $d^{\perp}$ be the microblog in C with the smallest overall score
12. set ub_i to the upperbound of the overall score of any unseen microblog in I_i (based on Eqn. 1, 3-5)
13. $j = j + 1$
14. return C

Fig. 2. The *LSII-Query* Algorithm

constant number of index merger before it is moved to I_{i+1}. Therefore, the total number of index mergers that involves d is $O(m)$. Given that the cost of each merger is linear to the number of microblogs in the merged indices, it can be verified that the amortized update cost for each microblog is $O(m) = O(\log n)$.

The above analysis shows that LSII can efficiently handle updates. Meanwhile, the query cost of LSII is also small. Given a real-time query q, LSII first scans through I_0 to identify the k_q microblogs in I_0 with the highest overall score, and puts those microblogs into the candidate pool. After that, LSII locates the posting lists in $I_1, I_2, \ldots, I_m$ that are relevant to the terms in q, and then LSII invokes the TA algorithm on the posting lists. In particular, the i-th iteration of TA examines the i-th microblog in each of the posting lists, computes their overall scores, and updates the candidate pool accordingly. For each I_i ($i \in [1, m]$), TA also computes an upperbound of the overall score of any unseen microblog in I_i, based on Equations 1, 3, 4, and 5. If the upperbound for I_i is no more than the scores of the top-(k_q) microblogs that TA has seen, then TA stops traversing the posting lists in I_i (the traversal on the other inverted indices may continue). When the traversal on all indices are terminated, TA returns the microblogs in the candidate pools as the final results. Figure 2 shows the pseudo-code of the query algorithm of LSII.

Compared with the TPL approach (which answers query with a single inverted index), LSII needs to traverse a larger number of inverted indices for query processing. Nevertheless, the query cost of LSII can still be smaller than that of TPL, due to two reasons. First, as we have mentioned, the microblogs in the larger indices in LSII (e.g., I_m) have small freshness scores (which in turn lead to small overall scores), and hence, the TA algorithm tends to terminate the traversal on those indices quite early. In other words, those indices do not entail much query overhead. Second, as the posting lists in $I_1, \ldots, I_m$ are implemented as arrays, TA would perform much better with $I_1, \ldots, I_m$ than with TPL, whose posting lists cannot support efficient linear scan without incurring significant update overhead.

To sum up, LSII achieves a good balance between query and update efficiency, and hence, remedies the drawbacks of the append-only approach and TPL. In what follows, we will discuss the implementation details of LSII, and explores the possibility of alternative designs.

B. Implementations

For each inverted index I_i ($i \in [0, m]$), LSII maintains a hash table that maps the ID of a microblog d indexed by I_i to a triplet $\langle sig(d), v_d, ts_d \rangle$. This facilitates the computation of a microblog's overall score given its ID. Accordingly, each entry e in a posting list in I_i records only the ID of the microblog corresponding to e, so as to reduce space consumption.

To ensure that new microblogs can be efficiently inserted into I_0, we implement each posting list in I_0 as a dynamic array A where each entry e records the ID of the microblog corresponding to e. Initially, A is allocated a small memory space of constant size. Whenever a new microblog is inserted to the posting list, its ID will be put into the first available entry in A. If A becomes full after the insertion, we allocate a new memory space that doubles the current size of A, and we move all elements in A to the new memory space. Meanwhile, the posting lists in $I_1, \ldots, I_m$ are implemented as static arrays, as explained in Section IV-A.

Whenever we need to merge two posting lists (during an index merger), we first use the hash table to obtain the microblogs corresponding to the entries in the posting lists. If the two posting lists are sorted (e.g., neither of the posting lists is from I_0), the merger is performed with two linear scans on the microblogs. On the other hand, if one of the posting lists is not sorted in the desired order (i.e., it is from I_0), then we first sort the entries in the posting lists, and then merge them with the entries from the other list.

C. Design Considerations

As explained in Section IV-A, the posting lists in I_0 are sorted by time only, which ensures efficient microblog insertion but requires us to scan all entries in all relevant posting lists when answering queries. We argue that the complete scans of relevant posting lists in I_0 would not have a detrimental effect on query performance, as the number of microblogs indexed by I_0 is small. But this leads to an interesting question: Would it be more beneficial if we adopt the TPL approach in I_0 to maintain three sorted posting lists about the significance, term weights, and freshness of microblogs? At the first glance, this would lead to more efficient query processing (as we can apply the TA algorithm on I_0 to answer queries), and the cost of maintaining sorted posting list (in face of frequent update) would not be substantial due to the small size of I_0.

As we observe in our experiments, however, adopting the TPL approach in I_0 only increases update cost without improving query performance. To explain this, observe that

the microblogs indexed by I_0 are the newest among the microblogs indexed by LSII. As a consequence, their freshness scores are high, due to which they tend to have large overall scores and are likely to be query answers. Therefore, even if we apply the TPL approach in I_0 and employ the TA algorithm for query processing, we may still need to visit a large portion of the microblogs in I_0 before the traversal on I_0 can be terminated. This renders the query cost comparable to the case when the posting lists in I_0 are sorted by time. Hence, we do not adopt the TPL approach in I_0.

The above analysis also sheds light on how we should set the size threshold τ_0 for I_0. As the microblogs with large timestamps are likely to be query results, we may as well put them all into I_0 to reduce update cost without degrading query performance. This indicates that τ_0 should not be too small, so as to accommodate all reasonably new microblogs. On the other hand, if τ_0 is excessively large, then I_0 may contain a large number of microblogs that are relatively new but not significant or highly relevant to users' queries, in which case I_0 would causes considerable query overhead. In Section VII, we will discuss how we can dynamically tune τ_0 based on run-time statistics.

Finally, one may concern about the space consumption of $I_1, \ldots, I_m$ as each of them maintains three posting lists for each term. We argue that such space overhead is reasonable given that accurate query results can only be derived by taking into account the significance, relevance, and freshness scores of microblogs. In addition, the space cost of $I_1, \ldots, I_m$ may be reduced by compressing each posting list [13], [14] (as $I_1, \ldots, I_m$ are relatively static). A thorough treatment of this issue is beyond the scope of this paper.

V. Extension for Personalized Search

In microblogging services, users are often interested only in the microblogs posted by a subset of other users, e.g., their friends and celebrities. For example, on average, a Twitter user only *follows* less than 40 other users (i.e., he/she is interested in less than 40 other users' microblogs). As such, it is useful to provide users *personalized* real-time search that processes queries based on the microblogs posted by a selected set of other users. More formally, given a query q and a set U of users, we aim to retrieve the k_q microblogs posted by the users in U that maximizes $f(q, d)$. To process such a query on LSII, a naive approach is to invoke the algorithm in Figure 2 with one modification: a microblog d can be put into the candidate result pool, only if d is posted by a user in U. This approach, albeit simple, is highly inefficient due to the large number of irrelevant microblogs visited during query processing. Motivated by this, we propose a *User-Specific Links (USL)* approach, which augments the posting lists in LSII with auxiliary information, such that we can traverse posting lists in a manner that avoids visiting any entries from irrelevant users.

In particular, USL augments LSII by (i) associating each posting list L with a hash table and (ii) adding a pointer to each entry in L. The hash table maps each user u to the first entry in L that corresponds to a microblog posted by u. Meanwhile, the pointer added in each entry e points to the next entry e' in L, such that e and e' correspond to the same user. For example, if the entries in L is sorted in descending order of timestamps, then the hash table maps u to the newest entry from u that is indexed by L. In addition, if u has three entries e_1, e_2, e_3 in L, such that e_1 (resp. e_3) has the largest (resp. smallest) timestamp, then e_1 has a pointer to e_2, and e_2 contains a pointer to e_3. Figure 3 shows an example. With such pointers, each posting list is conceptually divided into a number of sublists, such that each sublist links the microblogs from the same user in a sorted order. As such, we can perform personalized search efficiently by traversing only the sublists that correspond to the selected users.

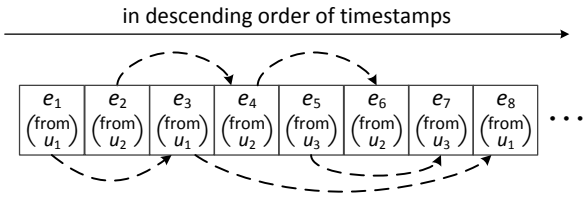

Fig. 3. A posting list with user-specific links

More specifically, given a query q on the microblogs of a set U of users, we first retrieve the posting lists in $I_0, \ldots, I_m$ that correspond to the terms in q, and then we traverse the posting lists using the TA algorithm in Figure 3, with one modification: when the algorithm visits a posting list L in the i-th iteration, it would examine the i-th entry in L *that corresponds to some user in* U, without looking at irrelevant entries. For example, given the posting list in Figure 3, if we are to search on the microblogs posted by u_2 and u_3, then algorithm would only visit entries e_2, e_4, e_5, e_6, e_7 (as the other entries correspond to u_1 instead of u_2 or u_3).

The above traversal approach is implemented as follows. Assume without loss of generality that the posting list L is sorted in descending order of timestamps. We first examine the hash table H associated with L, and use H to map each user $u \in U$ to the first entry e_u in L corresponding to u. Then, we insert all those entries into a priority queue Q that sorts the entries in descending order of timestamps. After that, in each iteration, TA would (i) remove the top entry e in Q, (ii) visit the microblog corresponding to e, and then (iii) insert into Q the entry in L that is pointed to by e. For instance, assume that L is as illustrated in Figure 3, and $U = \{u_2, u_3\}$. We would first put into Q two entries e_2 and e_5, as e_2 (resp. e_5) is the first entry in L from u_2 (resp. u_3). After that, the first iteration of TA would extract e_2 from Q (as e_2's timestamp is larger than e_5's), visit the microblog corresponding to e_2, and then insert e_4 into Q (as e_2 points to e_4). Next, the second iteration of TA would remove e_4 from Q and insert e_6 instead, so on and so forth.

Our traversal method takes $O(\log |U|)$ time to retrieve an entry from a posting list L in each iteration of TA. This mild cost is justified when L contains a large number of entries irrelevant to the users in U (as those entries would be omitted by the algorithm). However, when the number of

entries in L is small or when U contains most of the users who have microblogs in L, then our method may be inferior to a sequential traversal of L. As a rule of thumb, we apply our method on L only when the number of entries in L is at least ten times the number of users in U.

It remains to clarify how we can construct the hash table H associated with each sorted posting list L and the user-specific link in each list entry. Assume without loss of generality that L is sorted. We start with an empty H, and scan L from the end to the beginning. For each entry e that we encounter, if e is from a user u that is not in H, then we insert u into H and map u to e. On the other hand, if u is already in H, then we retrieve the entry e' that H maps u to, and we let e point to e'. After that, we let H map u to e instead. A similar algorithm can also be used to construct user-specific links when merging two sorted posting lists.

VI. Handling Score Updates

Our solution in the previous sections assumes that each microblog's timestamp, term vector, and significance score are fixed upon arrival. This assumption is partially true: In practice, the timestamps and term vectors of microblogs are generally static, but the significance score $sig(d)$ of a microblog d may change over time. For example, when $sig(d)$ is computed based on previous work [2] (as is the case in our experiments), $sig(d)$ is determined by some static factors as well as the *popularity score* of d, which monotonically increases when there are new microblogs referring to d. In other words, the arrival of a new microblog may cause changes in some older microblogs' significance scores. Such score updates may pose a challenge to LSII, depending on the microblog d whose significance score is to be modified.

In particular, if d is a relatively new microblog indexed by I_0, then we only need to update the hash table associated with I_0, to change the significance score of the hash table entry that corresponds to d. The posting lists in I_0 need not be changed, as they are sorted by timestamps and are independent of the microblogs' significance scores. On the other hand, if d is indexed by $I_1, \ldots, I_m$ (where one third of the postings are sorted by significance scores), then updating the associated hash tables alone would not be sufficient, as we need to maintain the sorted order of the posting lists. A naive solution is to directly re-arrange the posting list entries after the score update, but it would cause substantial overhead since the posting lists in $I_1, \ldots, I_m$ are implemented as sorted arrays.

To address this issue, we extend LSII by adding a *buffer* B to each posting list L in $I_1, \ldots, I_m$ that is sorted by significance scores of microblogs. Each entry in B is a pair $\langle id, s \rangle$, where id is the identifier of a microblog whose significance score needs to be changed, and s is the new significance score of the microblog. For efficiency, we index the buffer entries with a main-memory B-tree on their significance scores. For any microblog d in the posting list L, if we need to update its significance score from s_{old} to s_{new}, then we insert an entry $\langle d, s_{new} \rangle$ into B, and we remove $\langle d, s_{old} \rangle$ from B (if it exists). In other words, we use the buffer to record any update in significance scores, without changing the structure of posting list L itself.

Query processing with buffers. With the buffers added, each query is processed with a slightly modified TA algorithm that takes into account the entries in the buffers. For example, let us consider a personalized query q on the microblogs posted by a set U of users. The algorithm first retrieves each posting list L that is relevant to q, and then checks whether L's buffer is empty. If L has an empty buffer, then it would be traversed in the same way as in the algorithm in Section V, i.e., we will follow the user-specific links in L to examine only the entries in L relevant to the users in U.

On the other hand, if L's buffer B is not empty, then before the first iteration of TA, we create a priority queue Q that sorts entries in descending order of significance scores, and we insert into Q the entry in B with the largest significance score. In addition, for each user $u \in U$, we retrieve the first entry in L that corresponds to u, and we put the entry into Q, too. After that, in each iteration of TA, we remove the top entry e in Q, and we examine the microblog corresponding to e. If d has been visited in the previous iterations of TA or if d is not posted by any user in U, then we ignore d. Otherwise, we compute the overall score of d (based on the information in the hash table that maps d to $\langle sig(d), v_d, ts_e \rangle$), and we update the candidate pool accordingly. Next, if e is an entry in L, then we would retrieve the entry e' that e's user-specific link points to, and we insert e' into Q. If e is from B, however, we will insert into Q the next unvisited entry e^* in B, i.e., e^* has smaller (resp. larger) significance score than e (resp. any other unvisited entry in B). In other words, we visit the entries in B and the relevant entries in L in descending order of their significance scores.

To explain why the above traversal algorithm is correct, let us consider a microblog d whose significance score is s_{old} when L is constructed. Assume that d corresponds to an entry e_L in L, and that $sig(d)$ is later increased to s_{new}, which is recorded in an entry e_B in the buffer B. Depending on whether $s_{new} > s_{old}$ or $s_{new} < s_{old}$, our algorithm may visit e_B or e_L before the other one. But regardless of whether e_B or e_L is visited first, our algorithm calculates the overall score of d according to the hash table that records $\langle sig(d), v_d, ts_e \rangle$, which is always updated whenever $sig(d)$ changes. Therefore, the overall score of d would be accurate, which ensures the correctness of the query results. Note that, although we have focused our discussion on personalized search, our algorithm can be easily modified for non-personalized queries.

Index merger with buffers. Besides the query algorithm, the index merging method in LSII also needs to be slightly modified to incorporate updates in the buffers. Without loss of generality, assume that we are to merge two posting lists L_1 and L_2, whose associated buffers are B_1 and B_2, respectively. We treat B_1 and B_2 as two sorted lists, given that each of them is indexed by a B-tree. Accordingly, the merger of the posting lists can be treated as a merger of four sorted list L_1,

L_2, B_1, and B_2, which can be done in linear time with some bookkeeping efforts to avoid inserting outdated entries into the merged results.

One natural question is: How would the buffers of the posting lists affect query and update performance? Apparently, if each buffer contains a large number of entries, then the query and update efficiency of LSII would be severely degraded. Fortunately, this issue is alleviated by the facts that (i) most microblogs are not referred to by other microgblogs, and (ii) even if a microblog d is referred to by others, the references are usually made shortly after d is posted. For example, recent research [15] shows that only 29% of the microblogs on Twitter were referred to by other microblogs; in addition, 92.4% of the references were made within one hour after the original microblogs are posted. That is, only a small fraction of microblogs will receive updates in their significance scores, and most of the updates will concern the microblogs in I_0 (as they have the largest timestamps), for which the updates can be processed efficiently without being put into a buffer. Therefore, the number of entries in the buffers of $I_1, \ldots, I_m$ are unlikely to be large, which ensures query efficiency.

VII. Concurrency Management

In the previous sections, we have shown that LSII ensures a small amortized update cost for each microblog. Nevertheless, the *worst-case* update cost for a microblog can still be high, since the insertion of a microblog may cause mergers of the inverted indices in LSII. Therefore, if we process both microblog insertions and queries with a single thread, then the queries that immediately follow a microblog insertion may need to wait for an extended period of time, which results in negative user experience. This section addresses the issue by presenting a multi-threaded approach that concurrently processes microblog insertions and queries to reduce system response time. In what follows, we assume that our system has three types of threads: a *reader thread* that processes queries, a *writer thread* that handles microblog insertions, and multiple *merger threads* that deal with index mergers. Our solution can be easily extended for the case of multiple reader threads.

When I_0 is NOT full. Let us first consider the simple case when I_0 is not full. In this case, a microblog insertion would not trigger any index merger, and hence, we only need to focus on the interaction between the reader and writer threads. To coordinate these two threads, we maintain a variable ts_{max}, which records the timestamp of the last microblog that has been inserted into LSII. (This approach is also adopted in EarlyBird [1].) With this variable, the writer thread inserts a microblog d into LSII in a straightforward manner: it first appends d to the relevant posting lists in I_0 (recall that the posting lists in I_0 are sorted in ascending order of time); after that, it sets ts_{max} to the timestamp of d.

Meanwhile, the reader thread processes any (personalized or non-personalized) query q as follows. It first reads ts_{max}, and then performs the queries on LSII using the algorithms presented in the previous sections, with one additional rule: When it traverses a posting list in I_0, if it encounters a microblog whose timestamp equals ts_{max}, then the traversal will be terminated. As such, we ensure that the reader thread would not read the microblog d that is being written by the writer thread[1].

When I_0 is full. Now consider that I_0 is full, in which case the insertion of a new microblog d triggers a merger of I_0 and I_1. The basic idea of our solution for this case is to merge the indices in a manner that does not block the reader and writer threads. For this purpose, we create an index I'_0 that has the same structure with I_0 (i.e., each posting list in I'_0 is also sorted in descending order of timestamps), and we also set size threshold of I'_0 to τ_0. We refer to I'_0 as the *shadow* of I_0. Given a microblog d to be inserted, the writer thread would put d into I'_0 (instead of I_0), and then update ts_{max} accordingly. In other words, we use I'_0 as a buffer to accommodate new microblogs that arrive during the merger of I_0 and I_1.

To facilitate queries, we only impose shared locks on I_0 and I_1 when the merger is being performed (as will be clarified shortly). As such, the reader thread can still answer any query q, just that the query processing is under two additional conditions: First, all entries in both I_0 and I'_0 need to be examined, so that the new microblogs inserted during the merger are considered; Second, any traversal on I'_0 would terminate upon visiting a microblog whose timestamp equals ts_{max}, so as to avoid conflicts with the writer thread.

The merging of I_0 and I_1 is handled by a merger thread. It first creates a new inverted index I'_1 that has the same structure with I_1, i.e., I'_1 is a shadow of I_1. After that, it imposes shared locks on I_0 and I_1, and merges the contents of I_0 and I_1 into I'_1. Once the merging process completes, the merger thread requests exclusive locks on I_0, I'_0, and I_1. Upon acquiring the locks, the merger thread sets $I_1 = I'_1$ and $I_0 = I'_0$, and frees the space previously allocated to I_0 and I_1. In other words, the merger uses the shadow index I'_1 to buffer the results of merging I_0 and I_1, so as to allow queries to be performed on I_0 and I_1 during the merging process.

The merger of any other I_i and I_{i+1} ($i \in [1, m-1]$) are also performed by an individual merger thread, in an almost identical manner. Specifically, the merger thread starts by creating a shadow index I'_{i+1}. After that, it imposes shared locks on I_i and I_{i+1}, and it merges I_i and I_{i+1} into I'_{i+1}. During the merging process, the reader thread is allowed to perform queries using I_i and I_{i+1}. Finally, when the merger completes, the merger thread requests exclusive locks on I_i and I_{i+1}, and then it sets $I_{i+1} = I'_{i+1}$ and empties I_i.

When two mergers overlap. It remains to clarify how two merger threads may interact with each other. Assume that, when we are about to merge I_i and I_{i+1} ($i \in [0, m-2]$), the

[1]In effect, this will exclude d from the query results for q, even if d's timestamp ts_d is smaller than q's timestamp ts_q. This deviates from our problem definition, which requires that all documents with timestamps smaller than ts_q must be considered for q. Nevertheless, such a discrepancy is acceptable in practice, as the difference between ts_d and ts_q is extremely small (given that inserting d into I_0 takes negligible time), and hence, the two threads can run in parallel without interfering with each other.

index I_{i+1} happens to be full. In that case, we would need to merge I_{i+1} and I_{i+2} first, i.e., we need to combine all posting lists in I_{i+1} into I_{i+2}. After that, we can empty I_{i+1} and move all of I_i's posting lists into I_{i+1}. Finally, we can empty I_i.

To parallelize the above two mergers, we create two merger threads θ_1 and θ_2. θ_2 handles the merger of I_{i+1} and I_{i+2}, and it proceeds as a normal merger thread. In other words, θ_2 would create a shadow index I'_{i+2}, and would merge the information in I_{i+1} and I_{i+2} into I'_{i+2}.

At the same time, θ_1 creates a shadow index I'_{i+1}, and it requests a shared lock on I_i. After that, it copies all I_i's posting lists into I'_{i+1}. During this process, the reader thread can still answer any query that involves I_i. Once the moving of posting lists completes, θ_1 checks whether θ_2 has finished the merger of I_{i+1} and I_{i+2}. If θ_2 has terminated, then I_{i+1} should have been emptied. In that case, θ_1 would impose exclusive locks on I_i and I_{i+1}, after which it would set $I_{i+1} = I'_{i+1}$ and empty I_i. This completes the merger of I_i and I_{i+1}.

On the other hand, if θ_2 is still merging I_{i+1} and I_{i+2} when θ_1 finishes moving posting lists into I'_{i+1}, then θ_1 would request an exclusive lock on I_i only. Upon acquiring the lock, θ_1 empties I_i and makes I'_{i+1} visible to the reader thread. That is, whenever the reader thread processes a query, it would consider the entries in both I_{i+1} and I'_{i+1}. After that, when θ_2 completes the merger of I_{i+1} and I_{i+2}, it would set $I_{i+1} = I'_{i+1}$, as the posting list previously in I_{i+1} has been merged into I_{i+2}.

Summary. In summary, our multi-threaded approach uses a variable ts_{max} to ensure that the reader and writer threads do not interfere with each other, and we employ shadow indices to buffer any updates that occur during index mergers. This enables LSII to concurrently perform microblog insertions and queries, which improves system response time. There are only two circumstances when the reader or writer threads need to be blocked. First, as explained before, when a merger thread finishes merging two indices I_i and I_{i+1} into I'_{i+1}, it would impose exclusive locks on I_i and I_{i+1} to set $I_{i+1} = I'_{i+1}$ and to empty I_i, during which the reader thread cannot access I_i and I_{i+1}. However, this would only block the reader thread for a tiny period of time, as it takes negligible cost to set $I_{i+1} = I'_{i+1}$ and empty I_i. Note that the writer thread would not be affected in this case, since it never needs to visit an index that is involved in a merger.

Second, if the shadow index I'_0 becomes full when I_0 is being merged with I_1, then the writer thread needs to be suspended until I_0 becomes empty (since I'_0 cannot accommodate new microblogs any more). The reader thread would also need to be blocked, so as to ensure that it would not miss the microblogs that should have been inserted into LSII when the query arrives. This issue can be alleviated by setting τ_0 (i.e., the size threshold of I'_0 and I_0) to a reasonably large value, so that I'_0 would not become full during the merger of I_0 and I_1. In addition, we can also dynamically tune the value of τ_0 at runtime. For example, we may start LSII with a certain τ_0, and whenever we witness that the reader and writer threads are blocked as I'_0 becomes full, then we can double the value of τ_0 and adjust the structure of LSII accordingly, so as to accommodate more updates in I'_0 during the next merger of I_0 and I_1. On the other hand, if we observe that I'_0 is never more than half-full when I_0 is being merged with I_1, then we can halve the value of τ_0. As such, we can set τ_0 to a value adaptive to the frequency of microblog insertions.

VIII. Related Work

Microblog search is a relatively new research topic that only starts to attract research interests in recent years. Existing work on microblog search mainly focuses on three issues: (i) ranking microblogs with respect to queries [7], [8], [16], [17], (ii) indexing microblogs for efficient search [1], [2], [18], and (iii) analyzing the characteristics of microblog data and queries [4], [15], [19], [20]. Our work addresses the second issue, and is most related to previous work on the same topic, i.e., TI [2] and EarlyBird [1] (which have been discussed extensively in Section I), as well as the method by Yao et al. [18]. In particular, Yao et al.'s method is designed for retrieving *provenance information* from microblogs, such as the origin of a thread of microblog discussion, the evolvement of a microblog topic, etc. Although Yao et al. also consider efficiency issues, their approach cannot be applied for our problem as we focus on real-time microblog search instead of provenance retrieval.

As mentioned in Section II, the ranking function used in our paper is similar to the one adopted by TI [2]. Specifically, TI's ranking function is defined as follows:

$$f'(d,q) = (w_1 \cdot sig(d) + w_2 \cdot sim(d,q)) / (ts_q - ts_d),$$

where $sim(d,q)$ is as defined in Equation 2, and $sig(d)$ is computed as a weighted sum of two factors: (i) the popularity of the user who posted d, as inferred from the social network associated with the microblogging service, and (ii) the popularity of the topic thread that d belongs to, as inferred from the microblogs that refer to d and the microblogs that d refers to. Compared to TI's ranking function, our function provides more flexibility in modeling how fast the freshness of a microblog degrades with time.

Finally, our LSII structure borrows ideas from the *Log-Structured Merge-tree (LSM-tree)* [5]. In particular, The LSM-tree also contains a sequence of indices with exponentially increasing sizes, and it is designed to support efficient one-dimensional search under frequent tuple updates. Although LSM-tree has been adopted for various other applications (e.g., [21], [22]), to the best of our knowledge, we are the first to apply it for microblog search.

IX. Experiments

A. Experimental Settings

We experimentally compare LSII with two methods: (i) the append-only approach in Section III-A, which minimizes update cost but incurs significant query overhead; (ii) the TPL approach in Section III-B (with a B-tree implementation), which improves over the append-only approach in terms of

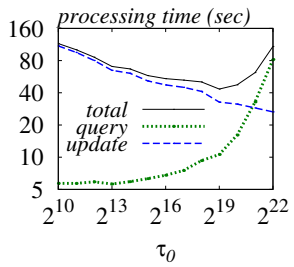

Fig. 4. LSII's performance vs. τ_0

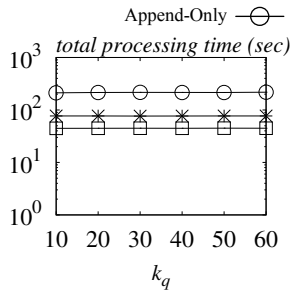

Fig. 5. Processing time vs. k_q

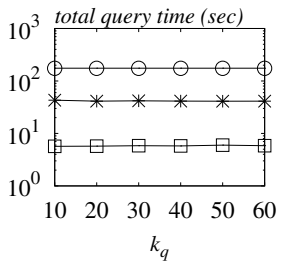

Fig. 6. Query time vs. k_q

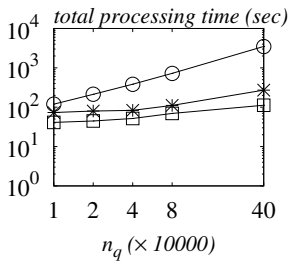

Fig. 7. Processing time vs. n_q

TABLE II
DEFAULT VALUES OF EXPERIMENTAL PARAMETERS

Parameter	Description
$w_1 = 2/7$, $w_2 = w_3 = 5/14$	the weights used in the ranking function $f(d, q)$ (See Eqn. 1)
$n_q = 20,000$	the number of queries
$k_q = 10$	the number of microblogs asked for by each query
$\|U\| = 40$	the size of the user set associated with each personalized search

query performance. We use a dataset that contains 10.4 million microblogs posted by 260 thousand users on Twitter from March 26, 2012 to April 8, 2012. Totally, there are 2.6 million distinct terms in the dataset (excluding stop words), and the average number of terms in each microblog is 9.

We sort the microblogs by the time that are posted, and we divide the sorted sequence into two subsequences S_1 and S_2, such that the S_1 contains the first 10 million microblogs in the sorted sequence, and S_2 contains the rest. In each experiment, we first insert the microblogs in S_1 into each indexing structure, after which we measure the time required by the structure to process a mixed query and update stream generated from S_2. In other words, we evaluate the performance of an index after a sufficient number of microblogs have been inserted, i.e., after the index has become relatively stable.

The mixed query and update stream is generated by inserting queries into random positions in S_2. For experiments on non-personalized queries, each query inserted into S_2 is created in a manner similar to previous work [2]. In particular, each query contains a number k of terms that are randomly selected from the $50,000$ terms in the dataset with the highest tf-idf scores [10]. The value of k is a random variable that equals $1, 2, 3, 4, 5$ with probabilities 50%, 25%, 15%, 7.5%, and 2.5%, respectively.

For experiments on personalized queries, each query added into S_2 is generated based on a non-personalized queries explained above. Specifically, given a non-personalized query q, we transform it into a personalized query, by associating q with a set U of users. 50% of the users in U are randomly selected, while the other 50% are randomly chosen from the set of users who have posted some microblogs that contains at least one term in q. (This ensures that the terms in q are not completely irrelevant to the users in U.) The cardinality of U is varied in our experiments.

The number of queries inserted into S_2 is controlled by a parameter n_q. We only consider the case when n_q is smaller than 0.4 million (i.e., the number of microblogs in S_2), as users of microblogging services usually post microblogs more frequently than submitting queries. All of our experiments are performed on a Windows machine with 48GB of RAM and a Intel Xeon 6-core CPU running at 2.4GHz. Table II shows the parameters of our experiments and their default values.

B. Experimental Results for Non-Personalized Queries

In the first set of experiments, we study the performance of all methods in processing a mixed stream of non-personalized queries and updates in a single-threaded environment. In particular, we first investigate the effect of τ_0 on the performance of LSII. (Recall that τ_0 is the threshold on the number of microblogs that can be indexed by the index I_0 in LSII.) Figure 4 shows the time required by LSII to process a mixed stream, with τ_0 varying from 2^{10} to 2^{22}. When τ_0 is small, the processing overhead of LSII is mainly due to updates, and it decreases with the increase of τ_0 until $\tau_0 = 2^{19}$. After that, LSII's computation cost is mostly due to queries, and it increases monotonically with τ_0. This is consistent with our analysis in Section IV-C that (i) an excessively small τ_0 incurs substantial update overheads without reducing query cost, and (ii) too large a τ_0 would result in significant query costs, leading to degraded overall performance. Based on this result, we set $\tau_0 = 2^{19}$ in the rest of our experiments.

Figure 5 (resp. Figure 6) illustrates the total processing time (resp. total query time) of all three indexing methods, with k_q (i.e., the number of microblogs that each query asks for) varying from 10 to 60. The append-only approach entails enormous overhead, since (i) it only maintains posting lists where entries are sorted by their timestamps, and hence (ii) it needs to scan the entries in all relevant posting lists during query processing. TPL considerably improves the append-only approach, as the three posting lists that it facilitates efficient query processing using the TA algorithm. Nevertheless, as discussed in Section III-B, the posting list entries in TPL are usually located in non-contiguous memory space, which is not ideal for the efficiency of sequential scan performed by TA. This explains why TPL is consistently outperformed by LSII by a large margin. In particular, LSII's running time is only one-fourth of TPL's.

Figure 7 (resp. Figure 8) shows the total computation time (resp. total query time) of all methods as a function of n_q, the number of queries in the mixed stream of queries and updates.

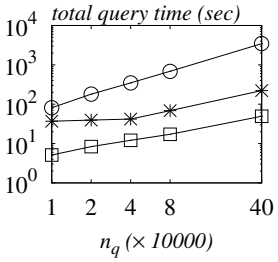

Fig. 8. Query time vs. n_q

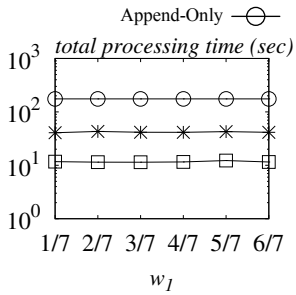

Fig. 9. Processing time vs. w_1

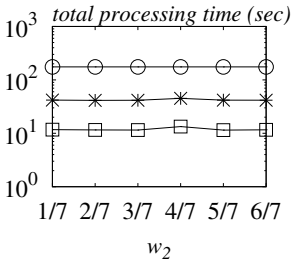

Fig. 10. Processing time vs. w_2

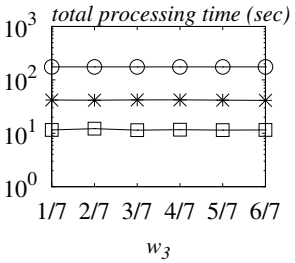

Fig. 11. Processing time vs. w_3

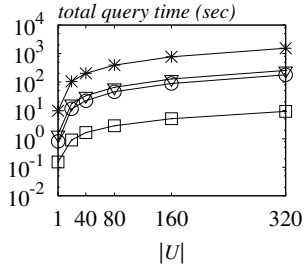

Fig. 12. Query time vs. $|U|$

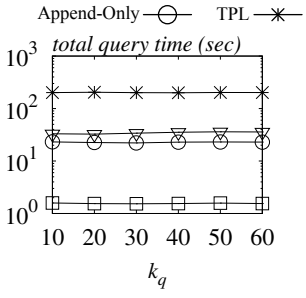

Fig. 13. Query time vs. k_q

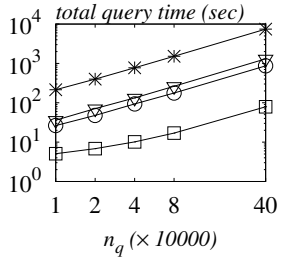

Fig. 14. Query time vs. n_q

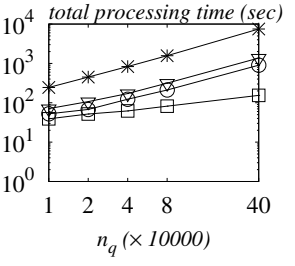

Fig. 15. Processing time vs. n_q

(Recall that the number of updates are fixed to 0.4 million.) Again, LSII incurs much smaller overhead than the other two techniques in all cases, which demonstrates that LSII strikes a better balance between query and update costs. The running time of all methods increases with n_q, since a larger number of queries entail a higher computation cost.

In the next experiment, we vary w_1 from $1/7$ to $6/7$, setting $w_2 = w_3 = (1 - w_1)/2$. Figure 9 illustrates the processing cost of each method as a function of w_1. Observe that the relative performance of each method remains the same as in the previous experiments. This shows that the efficiency of LSII is not sensitive to the value of w_1. Figure 10 and 11 illustrate the results of two experiments where we vary w_2 and w_3, respectively. Evidently, neither w_2 nor w_3 has a significant impact on the performance of LSII.

C. Experimental Results for Personalized Queries

In the second set of experiments, we focus on the processing of personalized queries in a single-threaded environment. Figure 12 illustrates the time required by each algorithm to process the queries in the mixed query and update stream, varying $|U|$, the number of selected users associated with each query. (The update costs are omitted from the figure, so as to facilitate a clear comparison of query overheads.) In addition to TPL, LSII, and the append-only approach, we also include LSII-NoUSL, a revised version of LSII that does not incorporate user-specific links. Observe that LSII significantly outperforms all other methods, which shows the effectiveness of user-specific links in accelerating user-specific queries. On the other hand, LSII-NoUSL performs no better than the append-only approach – this indicates that, without user-specific links, LSII needs to examine a large number of non-relevant entries in its posting lists, which entails substantial overheads. Interestingly, the running time of TPL is even higher than that of the append-only approach. The reason is that, for each term t in the query, TPL needs to traverse three posting lists, while the append-only approach only needs to scan one posting list for each term. For user-specific queries, TPL tends to examine a large portion of the posting lists before it can terminate, and hence, it can be inferior to the append-only approach in terms of query efficiency.

Figures 13 plots the query time of each method as a function of k_q. Again, LSII consistently incurs a smaller query cost than any other method does. The running time of LSII, LSII-NoUSL, and TPL increases with k_q, since a larger k_q requires each of those methods to examine more posting list entries. In contrast, the query time of the append-only approach does not change with k_q, since it scans all entries in the relevant posting lists, regardless of the value of k_q. Figure 14 repeats a similar experiment by varying the number of queries n_q, while Figure 15 illustrates the total processing time of each method as a function of n_q. The relative performance of each method remains the same.

D. Experimental Results for the Multi-threaded Approach

In the final set of experiments, we compare the single-threaded and multi-threaded implementations of LSII in terms of efficiency. For the multi-threaded implementation, we adopt one reader thread, one writer thread, and multiple merger threads. For convenience, we use LSII$^+$ (LSII) to refer to the multi-threaded (single-threaded) implementation. Figure 16 illustrates total time required by LSII and LSII$^+$ in processing a mixed stream of non-personalized queries and updates, with τ_0 varying. Observe that the processing time of LSII$^+$ is considerably smaller than that of LSII, when τ_0 is not excessively large. On the other hand, when τ_0 is large, both LSII$^+$ and LSII could entail significant query overhead, and hence, the performance gap between LSII$^+$ and LSII becomes

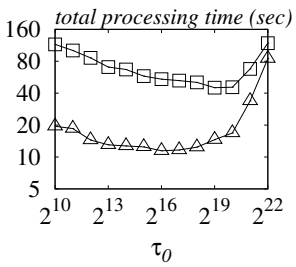

Fig. 16. Processing time vs. τ_0

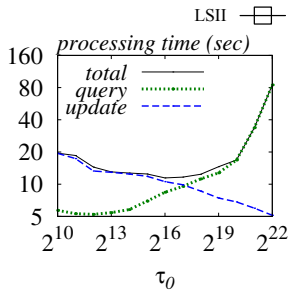

Fig. 17. LSII$^+$'s performance vs. τ_0

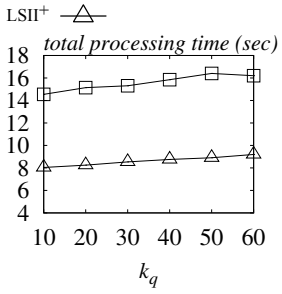

Fig. 18. Processing time vs. k_q

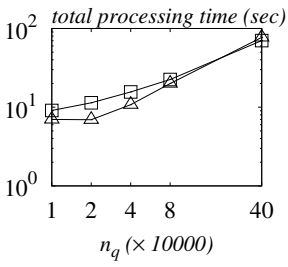

Fig. 19. Processing time vs. n_q

smaller. Figure 17 illustrates the query, update, and total costs of LSII$^+$ when τ_0 varies. Observe that the total overhead of LSII$^+$ roughly equals the larger one of its query and update overheads, due to the effect of multi-threading.

Figure 18 plots the processing time of LSII$^+$ and LSII as a function of k_q. Observe that LSII$^+$ consistently outperforms LSII. Finally, Figure 19 illustrates the running time of LSII$^+$ and LSII, with a varying n_q (i.e., the number of queries in the mixed query and update stream). Notice that, as n_q increases, the performance of LSII$^+$ and LSII becomes similar. This is because, when n_q is large, the update cost in LSII$^+$ and LSII becomes relatively insignificant, and the query cost becomes the dominating factor. Since LSII$^+$ has only one reader thread, its query performance is not better than LSII, and hence, the running time of the two methods is comparable.

X. Conclusion

In this paper, we study the problem of real-time search on microblogs under frequent updates. While previous work on this problem achieves efficiency at the cost of query accuracy, we show that it is possible to retrieve exact query results without incurring significant query and update overheads. The core of our solution is the Log-Structured Inverted Indices (LSII), which maintains a sequence of inverted indices with exponentially increasing sizes, such that new microblogs are (i) first inserted into the smallest index and (ii) later moved into the larger indices in a batch manner. The batch insertion mechanism leads to a small amortize update cost for each new microblog, without significantly degrading query performance. We present a comprehensive study on the design of LSII, and we demonstrate its efficiency with extensive experiments.

There are several interesting directions for future work. First, our paper has assumed all microblog insertions and queries are handled by a single sever. We plan to investigate how our solution can be extended to the case of multiple servers. Second, it is interesting to explore how the posting list of LSII can be compressed without severely degrading query and update performance. Third, we would also like to extend LSII for more advanced types of queries, e.g., retrieval of provenance information from microblogs.

XI. Acknowledgement

This work was supported by the Nanyang Technological University under SUG Grant M58020016, by the Agency for Science, Technology, and Research (Singapore) under SERC Grant 102-158-0074 and Grant 112-172-0010, and by the Sun Yet-Sen University under Natural Science Youths Funding Grant 62000-4103033.

References

[1] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *ICDE*, 2012, pp. 1360–1369.
[2] C. Chen, F. Li, B. C. Ooi, and S. Wu, "TI: An efficient indexing mechanism for real-time search on tweets," in *SIGMOD*, 2011, pp. 649–660.
[3] E. Adar, "User 4xxxxx9: Anonymizing query logs," in *Workshop on Query Log Analysis at the 16th World Wide Web Conference*, 2007.
[4] J. Lin and G. Mishne, "A study of "churn" in tweets and real-time search queries," in *ICWSM*, 2012.
[5] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.
[6] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.
[7] R. Nagmoti, A. Teredesai, and M. D. Cock, "Ranking approaches for microblog search," in *Web Intelligence*, 2010, pp. 153–157.
[8] A. Dong, Y. Chang, Z. Zheng, G. Mishne, J. Bai, R. Zhang, K. Buchner, C. Liao, and F. Diaz, "Towards recency ranking in web search," in *WSDM*, 2010, pp. 11–20.
[9] G. M. D. Corso, A. Gulli, and F. Romani, "Ranking a stream of news," in *WWW*, 2005, pp. 97–106.
[10] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.
[11] (2012) Twitter developer documentation. [Online]. Available: https://dev.twitter.com/docs/using-search
[12] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *Journal of Computer and System Sciences*, vol. 66, pp. 614–656, 2003.
[13] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *WWW*, 2009, pp. 401–410.
[14] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," in *WWW*, 2008, pp. 387–396.
[15] (2010) Replies and retweets on twitter. [Online]. Available: http://www.sysomos.com/insidetwitter/engagement/
[16] Y. Duan, L. Jiang, T. Qin, M. Zhou, and H.-Y. Shum, "An empirical study on learning to rank of tweets," in *COLING*, 2010, pp. 295–303.
[17] J. Weng, E.-P. Lim, J. Jiang, and Q. He, "Twitterrank: finding topic-sensitive influential twitterers," in *WSDM*, 2010, pp. 261–270.
[18] J. Yao, B. Cui, Z. Xue, and Q. Liu, "Provenance-based indexing support in micro-blog platforms," in *ICDE*, 2012, pp. 2183–2187.
[19] J. Teevan, D. Ramage, and M. R. Morris, "#twittersearch: a comparison of microblog search and web search," in *WSDM*, 2011, pp. 35–44.
[20] A. Java, X. Song, T. Finin, and B. L. Tseng, "Why we twitter: An analysis of a microblogging community," in *WebKDD/SNA-KDD*, 2007, pp. 118–138.
[21] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *VLDB*, 1997, pp. 16–25.
[22] Y. Li, B. He, J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *PVLDB*, vol. 3, no. 1, pp. 1195–1206, 2010.