On Main-memory Flushing in Microblogs Data Management Systems

Amr Magdy   Rami Alghamdi   Mohamed F. Mokbel

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455
{amr,alghamdi,mokbel}@cs.umn.edu

***Abstract*—Searching microblogs, e.g., tweets and comments, is practically supported through main-memory indexing for scalable data digestion and efficient query evaluation. With continuity and excessive numbers of microblogs, it is infeasible to keep data in main-memory for long periods. Thus, once allocated memory budget is filled, a portion of data is flushed from memory to disk to continuously accommodate newly incoming data. Existing techniques come with either low memory hit ratio due to flushing items regardless of their relevance to incoming queries or significant overhead of tracking individual data items, which limit scalability of microblogs systems in either cases. In this paper, we propose *kFlushing* policy that exploits popularity of top-k queries in microblogs to smartly select a subset of microblogs to flush. *kFlushing* is mainly designed to increase memory hit ratio. To this end, it identifies and flushes in-memory data that does not contribute to incoming queries. The freed memory space is utilized to accumulate more useful data that is used to answer more queries from memory contents. When all memory is utilized for useful data, *kFlushing* flushes data that is less likely to degrade memory hit ratio. In addition, *kFlushing* comes with a little overhead that keeps high system scalability in terms of high digestion rates of incoming fast data. Extensive experimental evaluation shows the effectiveness and scalability of *kFlushing* to improve main-memory hit by 26-330% while coping up with fast microblog streams of up to 100K microblog/second.**

I. INTRODUCTION

Microblogs, e.g., tweets, reviews, news comments, Facebook comments, and Foursquare check ins, have become incredibly popular among web users, where several billions microblogs are posted everyday [10, 27]. Microblogs come with rich contents and time-sensitive information that include textual contents, locations, and user information. The rich contents of microblogs have motivated several practical applications like news dissemination [3], rescue services [7], and tracking health-related issues [25]. Such important usage of microblogs has motivated researchers to spend major efforts to efficiently support search queries for large numbers of microblogs. Search queries on microblogs include keyword search queries "*Find microblogs that contain certain keyword(s)*" [5, 6, 16], location search queries "*Find microblogs that are posted within a certain location*" [19, 24], and user timeline search queries "*Find microblogs that are posted by a certain user*" [28]. Due to the large number of returned results for any of these search queries, all proposed techniques agree to put a limit k on the number of returned results. Hence, all search queries on microblogs turned out to be *top-k* queries, where the k results are selected based on a certain ranking criterion.

This work is partially supported by the National Science Foundation, USA, under Grants IIS-1525953, CNS-1512877, IIS-0952977 and IIS-1218168 and the University of Minnesota Doctoral Dissertation Fellowship.

Existing work for (*top-k*) search queries on microblogs [5, 16, 19, 28] mainly focus on building scalable indexing techniques *in main-memory* to digest incoming microblogs with their high arrival rates. Existing index structures along with their query processing techniques either explicitly or implicitly assume the following two assumptions: (1) Memory is so large that almost all queries of interest will be answered from in-memory contents. In case that the answer is not found in-memory, the search will continue in another disk-based index structure. However, only *in-memory query response time* is reported for performance evaluation, ignoring the disk access. (2) Once the memory is filled up, a chunk of *oldest* in-memory microblogs is flushed to disk, leaving their valuable memory space for new incoming microblogs.

Unfortunately, the implications of these two implicit assumptions are way underestimated in all prior work. Such assumptions are only geared towards *in-memory query response time*, while ignoring another critical performance measure, which is *memory hit ratio*, i.e., the ratio of queries that are completely answered from in-memory contents. With such two implicit assumptions, existing techniques may have a bad performance due to a low *memory hit ratio* as many of the incoming queries may not be answered from in-memory contents. Such queries are answered from disk with a very high cost. For example, the query "*Find most recent k tweets that has the keyword Obama*" would most likely be answered from in-memory contents because *Obama* is a popular (i.e., high-frequency) keyword. However, if the same query asks for the keyword "*concurrency*", which is not common in tweets, it is unlikely to find the answer in memory, and hence a visit to in-disk index has to be paid resulting in poor query latency.

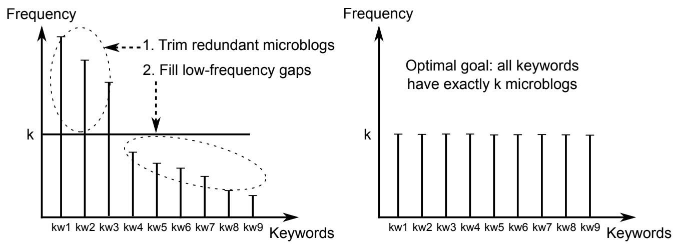
The rational of existing techniques is that most queries ask for popular keywords and will be answered from memory. Hence, it is reasonable to support such queries efficiently, and kind of ignore other queries that does not ask about popular words. However, such rational is not always favorable in practical scenarios. For example, web search engines optimize their performance to serve 95% of their search queries within a certain threshold, e.g., 50-100ms. So, it is important to optimize for worst case scenario, i.e., we need to ensure that 95% of our queries are answered below a certain threshold, which is favorable than optimizing for the average query response time. Considering the *memory hit ratio* as a major query performance in searching microblogs ensures that more queries are answered efficiently from memory, which matches the same optimization goal of major web search engines.

To illustrate the memory management problem in microblogs data management systems, Figure 1(a) depicts a typical snapshot of the memory contents. The figure shows nine keywords $kw1$ to $kw9$, on the horizontal axis, along with the number of microblogs containing each keyword on the vertical axis. The figure also has a horizontal line corresponds to the number k, where k is the default value used in any top-k query. Only three keywords, $kw1$, $kw2$, and $kw3$ appear more than k times, while the rest of keywords have appeared less than k times. Existing index structures and their query processors for search queries on microblogs work with such memory contents as is to retrieve their answers. Therefore, for any incoming query on any of the nine keywords in Figure 1(a), only the ones asking about the first three keywords can be answered form memory very efficiently as there are in-memory k keywords for each of them. However, any query asking about other keywords will have to encounter a disk access to retrieve k items, resulting in a very poor performance. Unfortunately, existing techniques have all their focus on how to query and index the first three keywords very efficiently while ignoring queries coming on the rest of keywords. The implicit assumption is that there is a background process that regularly evicts old memory contents to give room for new incoming ones. However, such process would still maintain the memory contents to be similar to Figure 1(a).

In this paper, we present *kFlushing*; a new flushing policy that is triggered once memory is full. The goal is to evict part of the in-memory contents to the disk storage, allowing new incoming microblogs to be digested in memory. *kFlushing* spots the problem in Figure 1(a), where a major part of the memory is consumed by useless microblogs that will not help in answering any top-k query. For example, consider the set of microblogs that include the first three keywords in Figure 1(a), but they are ranked above the k level according to the underlying ranking function. Such microblogs would never show up in a query answer for any top-k query with the same ranking function. Our observations on real Twitter data show that for k=20 and a temporal ranking function based on tweet arrival time, more than 75% of memory contents are consumed by tweets that will never show up in a query answer for a top-k keyword search query.

The goal of our proposed *kFlushing* policy is to ensure that all memory contents are useful. This is done by getting rid of the useless microblogs and use their space for the keywords that have less than k microblogs. Ultimately, with *kFlushing*, the memory contents should look like Figure 1(b), where each keyword has exactly k microblogs in memory. In that case, a query coming to any of the nine keywords $kw1$ to $kw9$ will be fully answered from in-memory contents, which significantly increases the system memory hit ratio. *kFlushing* enables existing algorithms for top-k microblog search queries (e.g., [5, 6, 16, 24, 19, 28]) to reach to their full potential and significantly increasing their memory hit ratio.

The concept of adjusting memory contents to increase memory hit ratio has been studied in different contexts under


Frequency
1. Trim redundant microblogs
2. Fill low-frequency gaps
k
kw1 kw2 kw3 kw4 kw5 kw6 kw7 kw8 kw9
Keywords
Frequency
Optimal goal: all keywords have exactly k microblogs
k
kw1 kw2 kw3 kw4 kw5 kw6 kw7 kw8 kw9
Keywords

(a) Before kFlushing (b) After kFlushing

Fig. 1. kFlushing main idea

different terminologies, e.g., *buffer management* in database management systems (DBMSs) [9], *anti-caching* in main-memory databases [8, 15, 30], and *load shedding* in data stream management systems (DSMSs) [1, 12, 13]. However, neither buffer management nor anti-caching techniques exploit top-k queries as they decide on flushing an item based on its latest access time regardless of other items. When optimizing for top-k queries, the decision to evict or keep an item in main-memory depends on the presence/absence of other items that satisfy the query. Meanwhile, the main focus of *load shedding* techniques is to drop a portion of incoming data to optimize memory contents for a set of registered continuous queries. This is different from the case of microblogs that removes from existing indexed data to optimize for any query that may come later on.

kFlushing employs a parameter B (default=10%) that represents the ratio of memory contents that need to be flushed. Then, the main idea of our *kFlushing* policy is to employ a three-phase strategy. In the first phase, we try to get the B% from those microblogs with keywords that have more than k microblogs. However, repeatedly doing so will result in a memory saturation, where we cannot get B% out. In that case, we employ the second phase that aims to get rid of keywords that have less than k microblogs, as they would require disk access in all cases. Again, another repetitive execution would result in another memory saturation case. In that case, we employ our third and final stage that checks on the query access pattern with the aim of having the memory contents as in Figure 1(b). We show that the *kFlushing* policy is extensible for: (a) various search attributes beyond the keyword search query, (b) various ranking functions, and (c) multiple keyword search queries. Extensive experimental evaluation using real Twitter data and various realistic query workloads shows that *kFlushing* improves the memory hit ratio for up to 330%, while keeping the in-memory query performance intact.

The rest of this paper is organized as follows. Section II formulates the problem. Sections III presents the *kFlushing* policy for keyword search queries and temporal ranking function. The extensibility of *kFlushing* to other query types and ranking functions is discussed in Section IV. Section V provides experimental evaluation. Section VI highlights related work. Finally, Section VII concludes the paper.

II. Preliminaries

This section gives important preliminaries for our proposed flushing policy that includes the underlying environment (Section II-A), the queries of interest (Section II-B), and problem formulation (Section II-C).

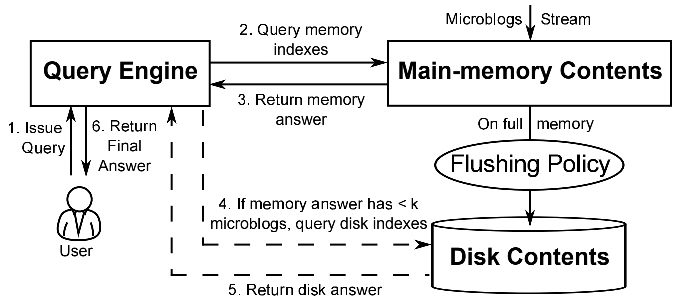


Fig. 2. Microblogs queries environment

A. Environment

Figure 2 gives the underlying environment of indexing and querying microblogs, in which our proposed flushing policy will be applied. The data input to this environment is a stream of microblogs, with high arrival rates, that is directly digested into an in-memory data structure. Once the memory becomes full, the *flushing policy* is triggered to select parts of the memory contents and flush it to the disk storage. All existing work in querying microblogs assume that such flushing works in a *temporal* way where the oldest in-memory contents are flushed to disk. This is in contrast to our proposed *kFlushing* policy, where we tune this flushing module to go beyond temporal flushing and smartly selects the flushing victims in a way that increases the *memory hit ratio* for incoming top-k queries. Meanwhile, incoming top-k search queries are posed to the query engine module, which first tries to get the answer from in-memory contents. If the answer could not be found in memory, e.g., less than k items are found, then this query is considered a *miss* and needs to check on disk contents to decide on its final answer. Going to the disk storage is an expensive process. Hence, the objective of our proposed flushing policy is to increase the *memory hit ratio,* which means reducing the ratio of queries that need to access the disk.

Figure 3 gives typical data structures for either in-memory or disk contents. The data structure includes a raw data store, which is basically a container for complete microblogs records as a raw data received from the input stream. The data structure also includes an attribute (e.g., keyword) index, which is basically a hash inverted table where each keyword entry has a list of microblog IDs of those microblogs that contain this keyword. Microblogs IDs are pointers to the raw data store where complete microblog records reside.

B. Queries

Our focus is supporting basic search queries on microblogs [5, 16, 19, 28]. Such queries retrieve individual microblogs that are associated with certain key value(s), e.g., keywords or user IDs. With excessive numbers of microblogs that could satisfy any query predicate, basic search queries are always considered as top-k queries that return only k microblogs, ranked based on certain ranking function F, where k is a reasonable number for human users to navigate, e.g., k=20. Formally, a basic search query is defined as follows:

Basic Search Query: *Given a search criteria A, integer k, and a ranking function F, a microblog basic search query finds k individual microblogs such that: (1) The k microblogs satisfy the search criteria A. (2) The k microblogs are the top ranked ones according to the ranking function F.*

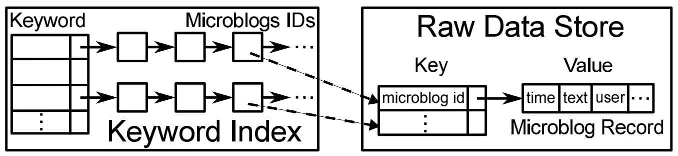


Fig. 3. Structure of main-memory and disk contents

The above query definition can be translated to a query on keyword *"Find k microblogs that contain certain keyword(s)"* [5], a query on locations "*Find k microblogs that are posted at a certain location*" [19], or a query on a user timeline "*Find k microblogs that are posted by a certain user*" [28]. Such basic search queries are most common for end users and supported in major commercial microblogging platforms. For example, Twitter supports retrieving microblogs of an individual user u timeline where the search criteria A is the ID of user u, k=20, and the ranking function F is temporal (i.e., most recent). Also, Twitter supports having the search criteria A as a set of search keyword(s), k=20, and F as one of two ranking functions, denoted as *All* (ranking by time) and *Top* (ranking over some popularity function). Finally, basic search queries on microblogs represent the basic building blocks for a wide spectrum of applications, e.g., event detection, user recommendation, or geo-targeted advertising.

C. Problem Formulation

Our problem in this paper can be defined as follows:

Problem Formulation: *Given a set S of in-memory microblogs and a flushing budget B, find a subset of microblogs $s \subseteq S$ to flush to disk storage such that: (1) s consumes at least B of main-memory, and (2) flushing s maximizes the memory hit ratio for incoming basic top-k search queries.*

The problem formulation imposes a minimum memory amount B to flush. If the amount of flushed data is not constrained, it may happen that few microblogs are flushed to disk. This means that the flushing procedure will be triggered more frequently as the memory will be filled faster. Performing a flushing operation every few seconds is not acceptable from a system efficiency and scalability point of view. This would involve expensive disk access that possibly causes system slowdown and limit its scalability. Thus, guaranteeing a minimum amount of flushed memory prevents filling the main-memory every few seconds and reduce the total number of flushing operations to sustain system scalability.

III. *kFlushing* POLICY

This section introduces our proposed *kFlushing* policy. *kFlushing* is triggered once the main-memory is full to decide on which microblogs to flush from memory to disk. *kFlushing* flushes a specified B percentage of memory contents to ensure a minimum amount of free memory, and hence continuity of digesting incoming data without re-invoking the flushing process frequently. *kFlushing* is composed of three consecutive phases, namely, *regular flushing* (Section III-A), *aggressive flushing* (Section III-B), and *forced flushing* (Section III-C). Each phase is invoked only if its preceding phase(s) cannot

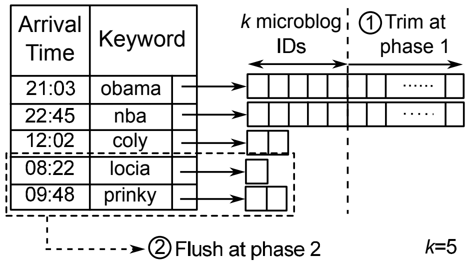


Fig. 4. Flushing Example

flush enough memory to meet the budget B. For ease of illustration and without loss of generality, we describe our proposed flushing policy considering basic search queries where the search criteria A is on keywords and the ranking function F is temporal (i.e., we return most recent k microblogs). In Section IV, we discuss the implications of changing the search criteria and/or the ranking function.

A. Phase 1: Regular Flushing

Motivation. *Regular flushing* is motivated by the large amounts of under utilized memory under temporal flushing scheme, that is currently used in microblogs systems [5]. As described in Figure 1(a), the frequency distribution of keywords in microblogs is very skewed. Thus, few keywords have very high frequency, much more than k, while the rest of keywords have low frequency, below k. For incoming top-k queries, microblogs that are beyond k in any keyword are useless microblogs as they would not contribute to any query answer. Such useless microblogs are observed to be 75% of the memory contents, for k=20, in real Twitter data. This means that only one quarter of the available memory is utilized.

Main idea. The main idea of the *regular flushing* phase is to trim extra useless microblogs that are above the k threshold line in Figure 1(a). For a trimmed microblog M, if M has only a single keyword, then M is removed from both index and raw data store and flushed to disk right away. In case M has more than one keyword, then Phase 1 removes it only from keyword index entries in which M is not among top-k microblogs. Yet, M data record might remain in the raw data store if it is still referenced by other index entry. This case would mean that M is still among top-k microblogs in other in-memory index entries. Whenever M is not referenced by any in-memory index entry, its record is removed from the raw data store and it is flushed to disk right away.

By removing useless microblogs, we clear significant memory space that can be utilized in a better way for low-frequency keywords to increase the memory hit ratio for incoming top-k search queries. Optimally, we aim to reach a memory snapshot that looks like Figure 1(b), where all keywords have appeared in exactly k microblogs, i.e., there are no extra useless microblogs and no shortage to retrieve from disk.

Example. Figure 4 gives an example of a simple hash index that contains five entries. Each entry has: (1) a keyword, (2) the latest arrival time for any microblog that includes the keyword (to be used in Phase 2), and (3) a list of microblog IDs that include the keyword, ordered by their arrival time. Two keywords (*obama* and *nba*) are considered popular as they has more than k=5 microblogs. In this case, the *regular flushing* phase removes from the index all microblogs that are beyond the most recent k in *obama* and *nba*. If a removed microblog is not referenced in other index entries, it is removed from the raw data store as well and flushed to disk right away. Otherwise, it remains in the raw data store until all its references are removed from the index.

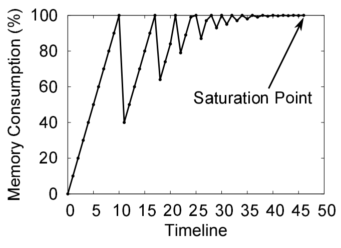
Algorithm. Incoming data is continuously digested in the main-memory data store and index described in Section II. On arrival of a new microblog M, it is stored in the data store with an auxiliary attribute $M.pcount$ initialized to the number of M's keywords. Then, M is inserted in the keyword index in each entry that corresponds to any of its keywords. If any of M's keywords kw has more than k microblogs, a pointer to kw index entry is added to a list L. The list L maintains pointers to keyword index entries that have more than k microblogs, i.e., have useless data. Practically speaking, due to the high skewness in keyword distribution in microblogs, L is a very short list as few keywords manipulate the memory contents. Maintaining L saves significant efforts of iterating over all keywords when Phase 1 is invoked.

On full memory, Phase 1 is invoked. For each keyword index entry W in the list L, W contains more than k microblogs. Then, Phase 1 shrinks W to contain only k microblog ids and trims the rest of its microblogs from the index. A trimmed microblog M would be removed from the index entry W all together and its $M.pcount$ would be decreased by one. In case $M.pcount > 0$, this means that M is still referenced by other index entries. Hence, in that case M's id is removed from list of microblog ids in W, while it is still kept in the main-memory data store as other index entries may need to retrieve it. This means that M data record is still physically in-memory, however, M id is not associated with W anymore. Whenever $M.pcount$ reaches zero, this means that M is no longer referenced by any index entry, and hence M entry in the data store is flushed as well. All flushed data are collected in a temporary main-memory buffer before writing them to disk. This is mainly to reduce the number of I/O operations. The list L is wiped out after the completion of Phase 1.

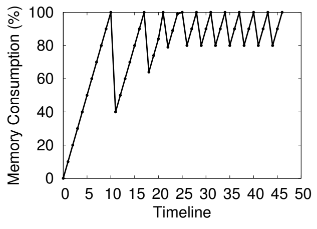
It is important to note that shrinking an index entry does not disturb continuous digestion of incoming microblogs within the same index entry. This is mainly because incoming microblog IDs are added to the list head while the trimmed IDs are removed from the list tail. The separation between insertion and deletion positions allows Phase 1 to be invoked in a separate thread without causing contention on index entries. This ensures continuous digestion of incoming microblogs in real time with high rates as shown in our experiments.

B. Phase 2: Aggressive Flushing

Motivation. Figure 5(a) shows the effect of employing only Phase 1 (*regular flushing*) on memory consumption over time. The horizontal axis is a time line while the vertical axis is the percentage of memory consumption. In the beginning, it takes about 10 time units to fill 100%of the memory. The first execution of Phase 1 flushes 60% of memory contents, leaving only 40% of memory consumed. It then takes only six time units to fill the memory again. Then, on a second call to Phase 1, there are less microblogs beyond top-k, and



(a) With only Phase 1



(b) With Phase 1 and Phase 2

Fig. 5. Memory consumption behavior

hence only 45% of memory contents are flushed. As time goes by, memory is filled much faster, and the amount of flushed memory becomes much less. This is because Phase 1 tunes the memory contents to make more keywords have exactly k items, and hence, there is not much microblogs to flush. Continuing like this will reach to a saturation point, where there are only few microblogs each time, which would be very costly to invoke a flushing process very frequently.

Overall, Figure 5(a) shows that we cannot rely solely on Phase 1 for flushing in-memory contents. Ultimately, we would like to have a memory behavior similar to Figure 5(b), where in a steady state, a fixed percentage, e.g., 20%, of the memory is flushed. This ensures that the flushing will be invoked in regular intervals, and will end up in flushing a reasonable part of the memory every time, which eliminates the overhead of running the flushing process very frequently. To achieve this goal, if Phase 1 fails to flush B percent of memory contents, we employ Phase 2 (*aggressive flushing*).

Main idea. Triggering the execution of Phase 2 means that all in-memory microblogs are useful as we have already trimmed all microblogs that do not participate in any top-k list in Phase 1. In this case, keyword entries in the in-memory keyword index fall in one of these two categories: (1) keywords that have exactly k microblogs, and (2) keywords that have less than k microblogs. Phase 2 only focuses on the keywords of the second category. The rationale is that queries that come on a keyword in the second category would not find their answers in memory anyway and would encounter expensive disk access. Thus, flushing these microblogs would not cause additional disk access, and will not degrade the memory hit ratio. Phase 2 flushes microblogs with keywords of the second category till we reach our target memory budget B. Since, there may be many keywords in this category, we select a subset of them that barely achieves the target B. Keywords are flushed in the order of their *least recently arrived*. So, keywords that did not receive any microblogs for the longest time are flushed to disk first. These keywords are less likely to accumulate k microblogs soon, and hence would have the least effect on memory hit ratio. This flushing order comes with a little overhead of assigning a single timestamp with each keyword rather than a timestamp per each data item as in traditional DBMS policies, e.g., LRU. This reduces both memory and CPU overheads of tracking flushing candidates as the experimental evaluation shows.

For each flushed keyword in Phase 2, we remove its entry all together from the in-memory index. This includes trimming all its microblog ids from the index. For each trimmed microblog M, its reference count $M.pcount$ is decreased by one. Whenever $M.pcount$ reaches zero, M data record is removed from the raw data store and flushed to the disk right away. This repeats until we flush total of the requested budget B.

Example. Following on the example in Figure 4, after Phase 1, keywords *obama* and *nba* have k=5 microblogs, *coly* and *prinky* have two microblogs each, and *locia* has one microblog. Since the first two keywords have exactly k microblogs, they are not considered by Phase 2. Assuming that Phase 2 needs to flush three more microblogs to reach B, we select the three microblogs that are associated with *locia* and *prinky* as they have the least arrival timestamps.

Algorithm. A straight forward implementation of Phase 2 is to sort the list of in-memory keywords based on their last arrival time. Then, we flush keywords from the top of the list till we reach our target B. That takes $O(nlogn)$, where n is the number of in-memory keywords. That is still expensive given the large number of keyword entries in memory, which is in terms of millions. Hence, we employ a smarter algorithm that is only $O(n)$. The main idea is to traverse all keywords that have less than k microblogs while maintaining on-the-go buffer of keywords so that: (1) Total memory consumption of buffered keywords at least equals the target B. (2) The buffer contains keywords with least arrival time. We maintain a max heap H of keywords and their memory consumption, sorted on keyword's arrival time. First, we add to H the first traversed keywords whose memory consumption adds up to at least the requested memory budget. Then, for each remaining keyword kw, if kw is less recent than H's most recent keyword, then kw replaces the most recent keyword in H. This is repeated until all keywords are exhausted. With each keyword replacement, H keywords total memory consumption must equal or exceed the requested budget, otherwise, the new keyword is inserted without removing H's most recent keyword. At the end, H contains the final set of keywords to be flushed, along with their microblogs.

For each keyword W in H, W's entry is removed all together from in-memory index. This includes removing W and trimming *all* its associated microblog ids from the index. For each trimmed microblog M, $M.pcount$ is decreased by one. If $M.pcount = 0$, then M record is removed from the raw data store and flushed to disk. If $M.pcount > 0$, M record remains in the raw data store until $M.pcount$ falls to zero. This repeats for all trimmed microblogs and keywords.

Phase 2 is executed in a separate thread so that it does not noticeably interrupt the continuous digestion of incoming data. In particular, on selecting its victims, Phase 2 is a reader-only for data structures that digest new data and does all its changes to temporary data structures, e.g., heap H. In addition, during flushing its victims, Phase 2 does the minimal possible interruption to the index. To illustrate, a new insertion may come on a keyword that is being removed at the same time. To avoid data inconsistency, each keyword's entry is moved from the index to a temporary buffer in a single atomic step, i.e., the entry is locked so that no microblogs can be inserted at the same time. The entries are locked once at a time so that atomicity overhead is negligible, especially with least recent entries that are less likely to receive new data at the time of flushing. Thus, data integrity is preserved with minimal overhead on data digestion.

C. Phase 3: Forced Flushing

Main idea. Triggering the execution of Phase 3 means that: (a) Both Phases 1 and 2 failed to flush at least B percentage of memory contents, and hence it is the goal of Phase 3 to flush more microblogs to reach to the goal of flushing memory budget B. (b) All keywords in memory have exactly k microblogs, where a snapshot of the in-memory keyword frequency looks like Figure 1(b). As a result, Phase 3 has no option other than removing keywords with exactly k microblogs. Consequently, any flushed data could reduce the memory hit ratio. To limit such reduction, we flush those microblogs that are less likely to be queried. This is accomplished by flushing *least recently queried* microblogs, i.e., microblogs that are associated with least recently queried keywords. With this preference order, Phase 3 keeps recently popular keywords in main-memory. This preference order is based on a previous study [17] that shows that real-time distribution of microblogs queries exhibits a strong temporal locality. So, recent queries behavior predicts the near future effectively. Similar to its preceding phases, trimmed microblogs in this phase are removed from the index, and their reference counts $pcount$'s are decreased. A microblog M is removed from the data store and flushed to disk whenever its $M.pcount$ falls to zero. Like Phase 2, the flushing order in Phase 3 comes with a little overhead that assigns a single timestamp to all microblogs that are associated with each keyword. This reduces both memory and CPU overhead of tracking flushing candidates.

Algorithm. The algorithm of Phase 3 is similar to that of Phase 2 single pass algorithm except that: (a) flushed entries are selected based on last querying time instead of arrival time, and (2) all keywords are candidates for flushing instead of only the low-frequency keywords. It is worth mentioning that although the newly attached timestamp can be updated from multiple querying threads simultaneously, it does not need any concurrency control overhead. The reason is that if two queries try to update this timestamp simultaneously, both of them would be trying to assign it to the same value, which is NOW. Thus, any race happens would not cause problems.

IV. Extensibility of *kFlushing*

We have discussed the *kFlushing* policy assuming keyword search queries that retrieve most recent k microblogs. However, *kFlushing* is a generic flushing policy and is designed to work for top-k queries in general, regardless their search attributes, ranking function, and/or value of k. Also, *kFlushing* could support single-keyword and multiple keyword queries. In this section, we discuss the extensibility of *kFlushing* for other search attributes beyond the *keyword* attribute (Section IV-A) and other ranking functions beyond the *most recent* one (Section IV-B). In addition, we discuss the possibility of changing the value of k during run time (Section IV-C). Finally, we discuss supporting multiple-keyword queries through *kFlushing* (Section IV-D).

A. Supporting Different Attributes

kFlushing is a generic concept that can be applied for any search attribute other than keyword attribute. Similar to the case of keyword index, we assume the existence of an index structure for the search attribute in our top-k queries. This is a practical assumption as current platforms already include hash index structures on users' IDs to support user time line search queries on the form: "*Find k microblogs that are posted by a certain user*" [28]. Meanwhile, recent research suggest to add spatial index structures to microblogging platforms as a means of supporting spatial search queries on the form: "*Find k microblogs that are posted in certain location*" [19]. For the case of user IDs, *kFlushing* aims to flush those microblogs that are not among the most recent k posts from any user. Similarly, for the case of locations, *kFlushing* aims to flush those microblogs that are not among the most recent k posts from each indexed area.

The *kFlushing* algorithm can actually be applied regardless of the underlying index structure. In particular, Phase 1 mainly keeps track of pointers to index entries that contain data beyond top-k answers. Such tracking can actually be used as is within the insertion procedure of any index structure. So, when inserting new items in any index cell C, e.g., a spatial index cell or a user index cell, C is checked for having useless data. In Phases 2 and 3, the algorithm mainly iterates over all index entries to select their victims. This also has nothing specific to do with our hash index and can be used in any index structure.

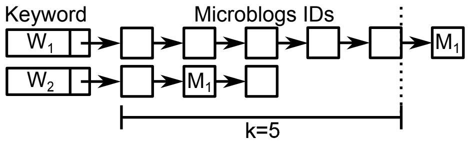
B. Supporting Different Ranking Functions

Throughout the paper, *kFlushing* was discussed in the context of a temporal ranking function, i.e., queries are looking for the *most recent k* microblogs that satisfy the query predicates. Though temporal ranking is the most widely used in microblogs [5], microblogs queries can still use other ranking functions. For example, a query may ask about tweets that are recent and posted by most popular users, where popularity is measured by number of followers on Twitter. Other ranking functions include ranking functions that combine timestamp with spatial attributes [19], combine timestamp with microblog popularity and textual relevance [28], or combine timestamp with user social graph and textual relevance [16].

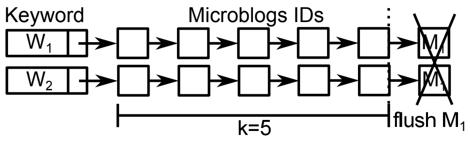
kFlushing can accommodate any ranking function either based on one single attribute or multiple attributes, given that *the ranking score can be all computed upon the microblog arrival*. In this case, we already know the top-k items in each index entry upon their arrival before any query comes. Thus, we can order data inside each index cell, e.g., the list of microblogs IDs in the index structure of Figure 3, so that top-k items are quickly accessible. Hence, Phase 1 would still keep only top-k microblogs and trims the rest that are beyond top-k. Phases 2 and 3 have nothing to do with the underlying ranking function, as they work only with the last arrival time and last query time, respectively.

C. Supporting Dynamic k Values

kFlushing policy can easily adapt itself dynamically with changing the value of k in the middle of the system operations. The only constraint is that the k should be fixed along all phases of each single execution of *kFlushing*. This means that if k is changed during the flushing operation, the change will actually take place in the next time the flushing procedure will be triggered. In case k is decreased, *kFlushing* can instantly adapt to the new k as existing in-memory data can still fulfill new queries answers as they ask for less data. Existing



(a) Snapshot at time t_1



(b) Snapshot at time $t_2 > t_1$

Fig. 6. Example of Multiple-Keyword Extension of *kFlushing*

microblogs that are beyond new k and below old k are marked to be flushed in the next flushing cycle. In case k is increased, *kFlushing* adaptation to the new k will be lagged a bit. In particular, as the new k is greater than the old one, existing in-memory data would not instantly fulfill incoming queries with the new value of k. However, as microblogs are arriving with high rates, missed data will be caught up quickly.

D. Supporting Multiple-keyword Queries

Our discussion up to this point considers queries that search for a single keyword. In this section, we introduce an extension for the proposed *kFlushing* to effectively support queries that search for any number of keywords. This shows the applicability of *kFlushing* for all practical scenarios with minor tweaks. In our discussion, we consider both types of multiple-keywords queries that are supported in the major web services: (i) OR queries that return a microblog if it has any of the keywords (*"Find most recent k microblogs that contain* ***any*** *of the keywords* W_1 *OR* W_2 *OR...*W_n*"*), and (ii) AND queries that return a microblog only if it has all the keywords (*"Find most recent k microblogs that contain* ***all*** *the keywords* W_1 *AND* W_2 *AND...*W_n*"*). Both queries use the ranking function F as *most recent*. For the rest of this section, we refer to them as *OR queries* and *AND queries*.

OR queries. *kFlushing* work perfectly fine with OR queries without any modifications. The reason is that in-memory contents under *kFlushing* would be enough to find all answers that could exist in memory for OR queries. To illustrate, in order to answer an OR query with two keywords W_1 and W_2, we retrieve the two index entries of W_1 and W_2, get the union of their microblogs, in a chronological ordered list L_m. If both keywords have k microblogs, so L_m is guaranteed to contain the k answer microblogs and causes memory hit. If any of the keywords has less than k microblogs, there is a possibility that L_m may not contain the final answer. This is mainly caused by the low-frequency keywords that have less than k microblogs. As *kFlushing* goal is maintain k microblogs in each keyword, then *kFlushing* achieves maximum hit ratio for OR queries.

AND queries. The described *kFlushing* policy has a limitation to achieve the maximum possible memory hit ratio for AND queries. The policy keeps with each index entry a maximum of k microblog ids. This leads many AND queries answers to have less than k microblogs from in-memory contents, and hence obligates to visit disk contents and causes memory misses. To illustrate, when an AND query comes on two keywords W_1 and W_2, an answer microblog must exist in both W_1 and W_2. Thus, we retrieve in-memory index entries of W_1 and W_2, scan their microblog ids list, and any microblog that is associated with both W_1 and W_2 is added to a chronological ordered list L_m. In most cases, L_m will be shorter than either lists, as L_m is represents the intersection of the two lists. As *kFlushing* keeps each list with maximum length of k, then, in most cases, L_m contains less than k microblogs and causes memory miss.

To overcome this limitation and increase memory hit in AND queries, we slightly extend *kFlushing* so that it allows (in certain cases) to have more than k microblog ids in each index entry. The main idea here is that microblogs that are allowed to be indexed while they are beyond the top-k microblogs must be *potential candidates* to increase memory hit ratio of AND queries. The candidate microblog would be the one that is still ranked among the top-k in other index entries. So, the extended *kFlushing* keeps a microblog in all index entries as long as it is among top-k microblogs in any of its keywords.

An illustration example is given in Figure 6. The example shows a microblog M_1 with two keywords W_1 and W_2 in Figure 6(a). M_1 is outside top-k microblogs in W_1 and among top-k microblogs in W_2, and $M_1.pcount = 2$. If the original Phase 1 (Section III-A) is executed, then M_1 id would be trimmed from index entry of W_1 and kept in W_2, and $M_1.pcount$ becomes 1. Now, assume an AND query comes on W_1 and W_2. The intersection of W_1 and W_2 microblogs would not find M_1 in memory, because its id is not associated with W_1 anymore. So, we have to visit in-disk entry of W_1 to get M_1 in the answer, while actually M_1 is still physically in the main-memory data store as it is still referenced by at least one in-memory keyword, i.e., $M_1.pcount > 0$. On the contrary, if we keep M_1 id associated with W_1 entry, M_1 would satisfy the AND condition and appear in the answer list without a need to access disk contents. This causes W_1 entry to have more than k microblogs. Yet, M_1 would lead to increase memory hit ratio, without significantly degrading memory utilization because it is a memory resident as long as $M_1.pcount > 0$. This extension affects the three phases of *kFlushing* as follows.

In Phase 1, the flushing rule is extended so that a microblog id M is trimmed from an index entry W if it satisfies two conditions: (1) M is beyond top-k microblogs in W, and (2) M is not among top-k microblogs in any other index entry. The second condition is added to prevent M to be trimmed from any index entry as long as it would remain in the in-memory data store. This means that $M.pcount$ would not decrease until M is outside top-k microblogs in all its keywords. Once this happen, $M.pcount$ would fall to zero in the following execution of Phase 1 and would be trimmed from all index entries and from the in-memory data store. Continuing to the example in Figure 6(a), when the extended Phase 1 is executed, it keeps M_1 in W_1 as is, and then $M_1.pcount = 2$ remain intact. When M_1 becomes outside top-k for both keywords as in Figure 6(b), it is trimmed from all keywords, its $M_1.pcount$ falls to zero, and it is flushed from the memory contents.

In Phase 2, the flushing rule is extended so that a microblog M is trimmed from an index entry W if it satisfies three conditions: (1) W has less than k microblogs, (2) W is selected based on *least recently arrived* order, and (3) M does not exist in any index entry that has $\geq k$ microblogs. The reason to add the third condition is that trimming M in that case may cause a memory miss and causes an additional disk access, violating the assumption of the original Phase 2 (Section III-B) that flushing all microblogs of low-frequent keywords would not cause additional disk access. Elaborating on M_1 in Figure 6(a), assume that the extended Phase 2 is invoked and W_2 is selected for flushing. Then, all W_2 microblogs are trimmed except M_1 as it exists in the frequent keyword W_1. So, when AND query comes on W_1 and W_2, M_1 would appear in the in-memory answer list. This Phase 2 extension prevents low-frequency keywords to hurt memory hit ratio of frequent keywords if they are involved in the same AND query.

Phase 3 is kept intact as described in Section III-C. The reason is that the original assumption of Phase 3 is still valid. In specific, Phase 3 is executed while reaching a saturation point in which all in-memory microblogs could cause memory hit. Thus, Phase 3 already flushes microblogs that may hurt the hit ratio, however, with minimal probability. This assumption is still valid with the extended Phase 1 and Phase 2. The difference here is that when Phase 3 is executed, not all in-memory keywords would have exactly k microblogs. Instead, it might find keywords that has either more than k microblogs (left by extended Phase 1) or less than k microblogs (left by extended Phase 2). However, this does not affect Phase 3 as all these microblogs still could cause memory hit. Thus, Phase 3 would remain intact and consider all in-memory keywords for flushing in *least recently queried* order.

Although the proposed modifications do not guarantee that all multiple-keyword queries would be answered entirely from memory contents, they improve the memory hit ratio and utilization as shown in our experimental evaluation with various realistic query workloads. Also, applying this extension slightly degrade the efficiency of *kFlushing* phases as they are invoked in separate threads that keep minimal interaction with real-time digestion thread, as described in Section III.

V. Experimental Evaluation

This section provides experimental evaluation of *kFlushing* policy and its multi-key queries extension, denoted as *kFlushing-MK*, that is described in Section IV-D to show their effect in increasing the memory hit ratio without sacrificing the performance of the underlying index. We compare our proposed policy with two policies: (1) The default temporal flushing policy (denoted as *FIFO*) used implicitly or explicitly in all existing techniques for microblogs [5, 16, 28]. *FIFO* always flushes the oldest data and is implemented based on a temporally-segmented hash index that consists of multiple temporally disjoint segments. On full memory, the oldest index segments are completely flushed out from memory. (2) The popular *least recently used* policy (denoted as *LRU*), implemented as H-Store anti-cache [8], where a global doubly-linked list is maintained to order microblogs in *least recently used* order. To reduce memory overhead, pointers of LRU list are embedded in the index entry of each microblog. H-Store is selected as it is designed for fast data environments, similar to microblogs environments.

Experimental setup. We compare *kFlushing*, *kFlushing-MK*, *FIFO*, and *LRU* for different values of k, different main-memory budgets, and different flushing budgets. Unless mentioned otherwise, we use a default k value of 20, main-memory budget of 30 GB, and flushing budget of 10% of the memory budget. We have collected 2+ Billion tweets from public Twitter Streaming APIs over the course of more than a year. We run these real tweets with an arrival rate of 6,000 tweets/second, which matches the current Twitter rate. By default, the presented experiments are performed using *keyword* attribute and *most recent* ranking function, where we use hashtags, if available, as keywords. All results are collected only in the steady state, i.e., after filling the main-memory budget and have multiple data flushes. Our performance measures include memory hit ratio for incoming queries and flushing overhead in terms of memory overhead and effect on digestion rate of incoming data. All experiments are based on Java 7 implementations for evaluated flushing policies and using an Intel Core i7 machine with CPU 2.40GHz and 64GB RAM that run Ubuntu 12.04. Synchronization between threads is handled through Java synchronization features.

Query workloads. In lack of standard query workload for microblogs keyword queries, we generate the following two workloads out of our real Twitter dataset:

1. *Correlated Query Load*: a query workload where keyword queries are selected at random from all keywords associated with our tweets without removing duplicates. Hence, the probability of a certain keyword to be queried equals its occurrence probability in the dataset. This query workload favors frequent keywords, which is a realistic assumption as active topics are likely to be the ones being queried.

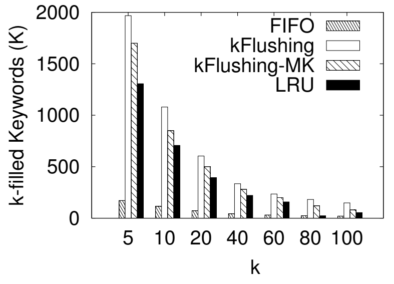
2. *Uniform Query Load*: a query workload where keyword queries are selected from the whole pool of possible keywords with equal probability regardless of their frequency in the incoming data. Although such query workload does not simulate the actual behavior of real users, yet it is practically used for testing the quality of performance for major systems, e.g., Twitter, and major search engines, e.g., Google and Bing. The rationale here is that such systems measure their performance for extreme cases to guarantee a minimum level of quality of service. In other words, the objective of such systems is not only to make the query search faster on the average, but also to guarantee that 99% of their queries are answered within reasonable latency [5].

Each of the two workloads consists of ten million queries. Each workload has one third of single-keyword queries, 2-keyword AND queries, and 2-keyword OR queries. Queries are posted as a stream of high rate of 25,000 query/second, similar to Twitter high query rates [26]. For extensibility experiments, similar query workloads are generated for spatial and user attributes replacing *keyword* with *latitude/longitude coordinates* and *userid*, respectively. Yet, all queries on *user* attribute are single-key queries as they are in practice.

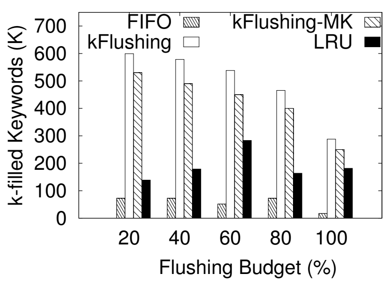
In the rest of this section, Section V-A analyzes a snapshot of memory contents. Sections V-B and V-C evaluates the memory hit ratio and the flushing overhead, respectively. Section V-D evaluates *kFlushing* extensibility.

A. Snapshot of In-Memory Contents

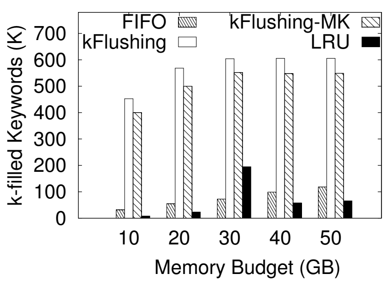
As was indicated earlier in Figure 1, the optimal scenario is to remove useless microblogs in a way that allows other



(a) Varying k

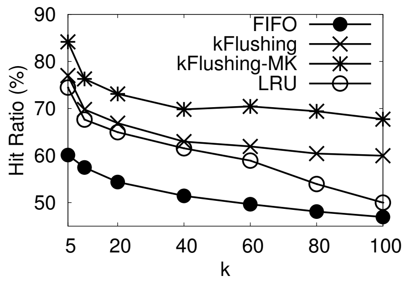


(b) Varying Flushing Budget

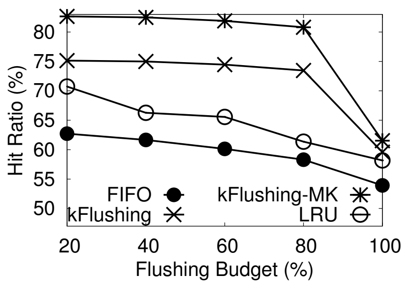


(c) Varying Memory Budget

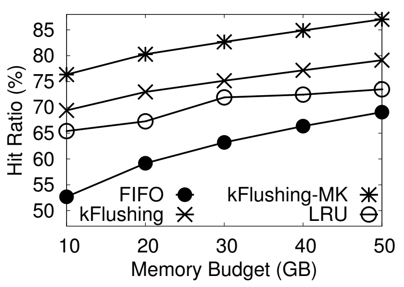
Fig. 7. Number of Memory-hit Keywords



(a) Varying k



(b) Varying Flushing Budget



(c) Varying Memory Budget

Fig. 8. Hit Ratio on Correlated Query Load

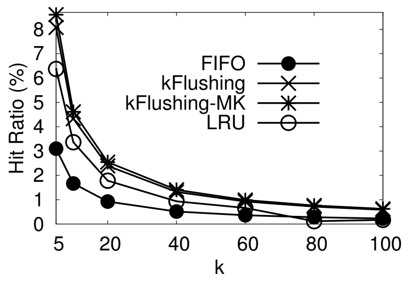
keywords to accumulate k microblogs, and hence they would not need a disk access if queried. Figure 7 gives the effect of running *FIFO*, *kFlushing*, *kFlushing-MK*, and *LRU* policies on the number of keywords that have accumulated at least k microblogs, in a steady state point. Queries on these keywords cause memory hit, and hence, the more of these keywords the much better the flushing policy. Figure 7(a) gives the number of k-filled keywords when varying k from 5 to 100. With increasing k, the number of k-filled keywords is noticeably decreasing for all policies as less keywords can accumulate k microblogs with larger k. However, for all k values, both *kFlushing* and *kFlushing-MK* outperforms both *FIFO* and *LRU*. In specific, *kFlushing* accumulates *at least* 7 times k-filled keywords more than *FIFO* and up to 3 times more than *LRU*. *kFlushing-MK* always accumulates slightly lower than *kFlushing* due to intentionally overlooking potential posts and keeping them in memory, which reduces the amount of memory available for low-frequent keywords to accumulate k. Yet, *kFlushing-MK* still outperforms the other two competitors. This experiment can be translated that up to 700K queries that could cause a memory hit with *kFlushing* variations, would miss their answers with *LRU*, and similarly 1800K queries with *FIFO*, which is a significant improvement over both policies.

Figure 7(b) gives the number of k-filled keywords when varying the flushing budget from 20 to 100% of allocated memory. With increasing flushing budget, number of k-filled keywords is decreasing as memory looses more content. Only *LRU* shows a kind of unexpected behavior with varying flushing budget, as it depends on incoming queries in real time and does really follows a certain pattern. However, different flushing budgets give 8 to 10 times more k-filled keywords in *kFlushing* variations compared to *FIFO* and 2 to 9 times compared to *LRU*, so at least it doubles the number of k-filled keywords in its worst cases, which shows superiority over both policies. Finally, Figure 7(c) gives the number of k-filled keywords when varying memory size from 10GB to 50GB. For 10GB memory, both *kFlushing* variations accumulate $\sim$13 times more k-filled keywords than *FIFO* and $\sim$50 times more than *LRU*. This ratio decreases with increasing memory budget, as *FIFO* and *LRU* accumulate more k-filled keywords with having more memory space, while *kFlushing* gives consistent superior performance for different memory budgets. This shows the robustness of *kFlushing* to give high performance in tight memory environments. *LRU* still shows a kind of unpredictable pattern as a result for depending on query distribution in real time, which is arbitrary.

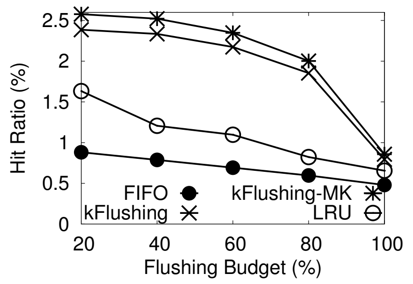
B. Memory Hit Ratio

In this section, we evaluate the effectiveness of *kFlushing* in improving *memory hit ratio*, i.e., the ratio of queries that find their k microblogs in memory contents. As *memory hit ratio* is heavily dependent on the incoming query workload, we perform our experiments twice, once for the correlated query workload (Figure 8) and another for the uniform query workload (Figure 9).

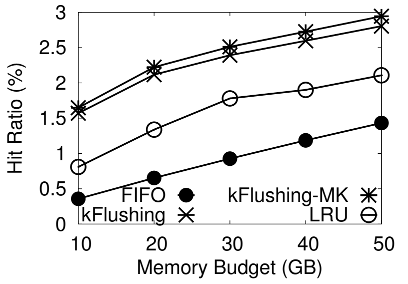
Figure 8 gives memory hit ratio for correlated query workload. For all parameters, *kFlushing* variations consistently achieves 12 to 20% higher hit ratio over *FIFO* which represents 20 to 44% improvement, and 2 to 18% higher hit ratio over *LRU* which represents 3 to 35% improvement. Thus, with 10 millions queries in our query load, *kFlushing* variations hit 1.2 to 2 million queries in main-memory that are not hit using *FIFO*, and 200 thousands to 1.8 million queries that are not hit using *LRU*, which is a significant improvement. In addition, *kFlushing-MK* is always superior to *kFlushing* with 7 to 9% increase in hit ratio which represents 9 to 15% improvement.



(a) Varying k



(b) Varying Flushing Budget



(c) Varying Memory Budget

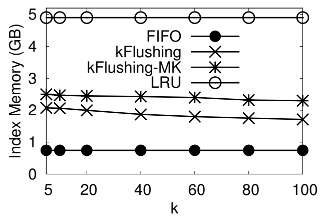
Fig. 9. Hit Ratio on Uniform Query Load

This clearly shows the effectiveness of the proposed multiple-key extension in answering more hundreds of thousands of queries (average of 800 thousands) from in-memory contents. Figure 8(a) gives memory hit ratio when varying k from 5 to 100. With increasing k, hit ratio of all policies decrease as queries ask about more data. Yet, *kFlushing-MK* answers 68 to 84% of all queries from in-memory contents, which is higher than all other alternatives significantly, where *kFlushing* achieves 61 to 77%, *LRU* achieves 50 to 74%, and *FIFO* achieves 46 to 60%. The superiority of *kFlushing* variations with large values of k shows the positive effect of accumulating much more k-filled keywords that is shown in the previous experiment. Figure 8(b) shows that with increasing flushing budget, hit ratio of all policies are also decreasing as memory loses more data. Still *kFlushing-MK* has up to 20% increase in hit ratio over *FIFO* and *LRU*. Finally, Figure 8(c) confirms the superior performance of *kFlushing-MK* and *kFlushing* over all alternatives especially with tight memory budgets. *kFlushing-MK* always achieves ∼10% improvement over *kFlushing*. For 10 GB memory, *kFlushing* gives 18% increase in hit ratio over *FIFO*, where we go up from only 52% to 70%.

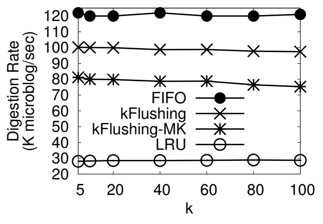
Figure 9 evaluates memory hit ratio on uniform query workload. It is noticeable that the hit ratio of uniform workload is consistently low, below 9%, due to the low percentage of frequent keywords in Twitter data. Both *kFlushing-MK* and *kFlushing* give almost similar performance for different parameters. However, for all parameters values, *kFlushing* variations are superior and provide significant *relative improvement* in memory hit ratio, which ranges from 100% to 330% compared to *FIFO* and 26 to 240% compared to *LRU*. In specific, Figure 9(a), at k=40, shows 0.42% hit ratio for *FIFO* and 1.41% for *kFlushing*, which means 3.3 times more queries answered from memory. Even with such low hit ratio, this 1% improvement, for 10 millions queries workload, gives 100,000 more queries answered from memory, which is a significant improvement. Similar to the results on correlated workload, Figure 9(a) and Figure 9(b) show decreasing hit ratio with increasing k and flushing budget, respectively, while Figure 9(c) shows increasing hit ratio with increasing memory budget. This experiment also confirms efficient *kFlushing* performance in tight memory environments (10GB).

C. Flushing Overhead

In this section, we evaluate the overhead encountered by the flushing policy along with its effect on the system scalability to digest incoming tweets with high rates. To do so, we do not limit the arrival rate, instead we stress our system and let the tweets arrive to the system as fast as it tolerates.



(a) Memory Overhead

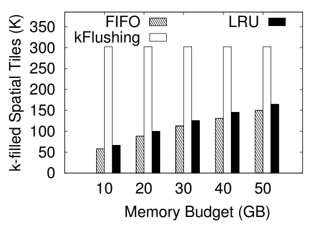


(b) Digestion Rate

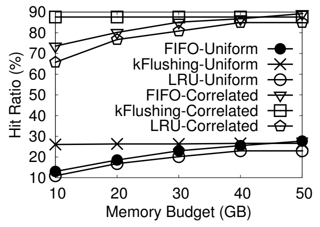
Fig. 10. Flushing Overhead vs. k

Figure 10 gives the flushing process overhead in terms of indexing memory overhead (Figure 10(a)) and the effect on the underlying index digestion rate (Figure 10(b)), when varying k. Figure 10(a) shows that different policies give stable memory overhead for k ranges from 5 to 100. *kFlushing* overhead decreases very slowly with increasing k due to decrease in total number of keyword entries in the index. Yet, *LRU* gives the highest overhead, 4.9GB, which is around 2 times *kFlushing-MK* and 2.5 times *kFlushing* overhead, while *FIFO* gives the lowest overhead, ∼0.75GB for all k values. This is interpreted by the big overhead of LRU list that tracks individual microblogs, while *kFlushing* variations do not track individual items. Instead, it uses the natural index grouping, based on keyword, to track usage of microblogs in groups that significantly reduce the tracking overhead. Yet, during the flushing process, a large amount of temporary buffering memory, ∼2GB, is needed to collect the scattered victim items to flush to disk. In *FIFO*, this temporary buffer is not even needed as the index is segmented based on arrival timestamp and hence the oldest index segment is used as the buffer.

Figure 10(b) shows the effect of the flushing policies on the digestion rate of incoming microblogs to underlying index. For all values of k, *FIFO* allows its underlying index to digest ∼120K tweets/second. Due to its insertion and book keeping overhead, the two variations of our *kFlushing* policy perform worse than *FIFO*. This is mainly because of accessing the index from two threads simultaneously, which includes a minimum level of concurrency control. However, *kFlushing* can still digest ∼100K tweets/second and *kFlushing-MK* digest ∼80K tweets/second, for all k values. This is 13 to 17 times higher arrival rates than Twitter firehose; a stream that contains all Twitter data. This shows that *kFlsuhing* policy could efficiently isolate its CPU overhead from the

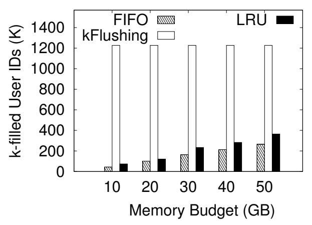


(a) Number of Memory-hit Keys

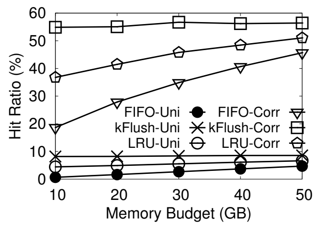


(b) Hit Ratio

Fig. 11. *kFlushing* Performance on Spatial Attribute



(a) Number of Memory-hit Keys



(b) Hit Ratio

Fig. 12. *kFlushing* Performance on User Attribute

underlying index and keeps its high performance. *kFlushing-MK* consumes more efforts in the flushing threads and so it causes more contention and less digestion rate. On the contrary, *LRU* encounters significant contention on the global LRU list, which limits the digestion rate to only 29K tweets/second. This list is accessed by querying threads, so that recently used tweets are moved to the head of the list, and insertion thread where new tweets are also inserted to the list head. In real-time operation, both insertions and querying are running most of the time, and hence a significant contention is introduced and limits index scalability. This shows the superiority of *kFlushing* that significantly improves memory hit over *LRU*, and also sustains high digestion scalability.

D. kFlushing *Extensibility*

This section shows the extensibility and effectiveness of *kFlushing* policy when employed with different attributes. To evaluate *kFlushing* extensibility, we use the commonly used microblogs attributes: spatial and user attributes. These two attributes are used to support queries "*Find most recent k microblogs that are posted in a certain location*" [19] and "*Find most recent k microblogs that are posted by a certain user*" [28], respectively. For spatial attribute, we use a spatial grid index that is composed of equal-area spatial tiles, each of 4 mile^2. For user attribute, we use a hash index that has a similar structure to our keyword index (Figure 3), however, it indexes user ids instead of keywords. In this section, *kFlushing-MK* is omitted because all user queries are single-key queries and spatial queries have no AND queries, as they are semantically invalid in the spatial context. This is because AND queries look for a microblog that exits in two spatial regions at the same time and each of our tweets associated with only one point location. Therefore, *kFlushing-MK* performs exactly as *kFlushing* in all queries of this section.

Figure 11 evaluates memory hit on spatial attribute with different memory budgets. Figure 11(a) shows the number of keys, i.e., spatial tiles, that cause memory hit. *kFlushing* outperforms both *FIFO* and *LRU* with 2 to 5 times higher number of memory-hit keys. In addition, *kFlushing* gives high performance even in tight memory budgets (10GB), which confirms the superiority in tight memory environments that are shown in previous experiments. Figure 11(b) shows memory hit ratio for both uniform and correlated query workload. Both query workloads show a stable superior performance for *kFlushing* over the other policies even with low memory budgets. The hit ratio in other policies starts relatively low with tight memory budgets ($\leq$ 30GB) and then improves noticeably with memory $>$ 30GB. At memory $\leq$ 30GB, *kFlushing* improves 15 to 100% over *FIFO* and 30 to 138% over *LRU* for uniform queries, while improves 10 to 20% over *FIFO* and 8 to 33% over *LRU* for correlated queries. This is still significant improvement that cause over two millions queries to hit memory with *kFlushing* and miss with its competitors. We omit the flushing overhead and digestion rate scalability results for space limitations, however, the results are exactly the same as shown with keyword attribute.

Figure 12 evaluates memory hit on user attribute with different memory budgets. The figure shows pretty similar performance improvements like the drawn conclusions from both keyword and spatial attributes. However, it is noticeable that *kFlushing* gives much better improvement for correlated query workload in user attribute compared to keyword and spatial attribute. This reflects the more skewness of data according to the user attribute. In other words, highly active users, who tweet frequently, cause higher percentage of useless microblogs than popular keywords and popular spatial regions. Otherwise, the improvement patterns and conclusions are pretty much the same. *kFlushing* still gives scalable digestion rate of 100K microblog/second.

In nutshell, extensibility experiments show the superiority of *kFlushing* on the three attributes, keyword, spatial, and user, for different parameters and performance measures. This shows the generality and effectiveness of *kFlushing*.

VI. Related Work

In this section, we highlight three areas related to our work: *DBMS buffer management and anti-caching*, *real-time microblogs data management* and *load shedding in data streams*.

DBMS buffer management and anti-caching. Evicting data from main-memory has been studied in both buffer management in database systems [9] and anti-caching in main-memory databases [8, 15, 30]. Our problem can be considered a variation of the anti-caching problem, applied in microblogs platforms rather than relational main-memory databases. However, existing techniques [8, 15, 30] have limitation to solve our problem. First, none of them addresses top-k queries, which is a major component of microblogs systems [5, 19, 28]. In addition, they suffer from significant overhead that limits the scalability of microblogs systems. Specifically, Hekaton [15] depends on offline processing which cannot scale for high velocity data like microblogs. On the contrary, H-Store [8] anti-cache is optimized for fast data environments. Though, it still uses a traditional policy (LRU) that requires tracking usage of individual data items. This pose a significant overhead to maintain LRU-ordered list for all data items in the system [30]. Unlike this work, *kFlushing* uses top-k queries as a guide to smartly select flushing victims with minimal overhead that does not limit system scalability.

Real-time microblogs data management. Due to its popularity and high application needs, managing real-time microblogs has attracted several research efforts in industry and academia. However, the main focus was on either indexing (e.g., [28, 29]), querying (e.g., keyword search [5, 6, 16, 28] or location-based search [4, 19, 24]), analysis (e.g., event and trend detection [11, 22], news and topic extraction [14, 23], or semantic and sentiment analysis [2]), or query languages [18, 21]. In all this work, it is assumed that queries are all answered from in-memory contents. Thus, the main performance measure is the query response time from in-memory contents. Only our prior work [19, 20] have studied the effect of having a flushing policy, in terms of the memory consumption. However, this work was tailored to a specific spatio-temporal queries and has nothing to do with any other attributes, ranking functions, or index structures. Therefor, this cannot fit in our vision to build a generic system [18]. Our work in this paper is the first to propose a generic flushing policy for microblogs. In addition, it is the first to address increasing the memory hit ratio of incoming queries, and hence significantly increase the overall system quality.

Load shedding in data streams. Selecting flushing victims is similar in spirit to the idea of load shedding that was extensively studied in data stream management systems (e.g., see [1, 12, 13]), where upon high system load, a portion of data is dropped from memory so that queries quality is minimally affected. However, these techniques cannot be applied to flushing microblogs for two main reasons: (1) Selected victims are chosen to optimize the performance for a set of continuous queries that are already registered in the system. This is not the case for microblogs where the focus is not continuous queries. So, we are adjusting the memory contents so that any query may arrive later. (2) Streaming load shedding is optimized for query accuracy as the removed data is just thrown away and not moved to disk. On the contrary, in microblogs, flushed data is moved to disk, and hence the answers are always accurate. Instead, we optimize for increasing the memory hit ratio, which is not considered at all in load shedding techniques.

VII. Conclusion

This paper has studied the problem of main-memory flushing in microblogs data management systems. Our study is motivated by existence of many useless data that is stored in main-memory under existing flushing schemes. This data does not contribute to incoming queries, which mostly ask for only top-k microblogs, where k is typically a small number. Thus, we exploit these characteristics to design effective flushing rules for microblogging environments. In particular, we have proposed *kFlushing*: an effective and scalable flushing policy that works for top-k search queries on microblogs. *kFlushing* policy frees the unutilized memory that are used to store useless data. The freed memory is used to accumulate more useful data so that much more queries can find their answers in memory. When all memory is utilized, *kFlushing* flushes microblogs that are less likely to degrade memory hit ratio. Using the same memory budget, *kFlushing* is able to significantly boost memory hit ratio by 26-330% compared to existing flushing schemes. In addition, it can work efficiently in tight memory environments and saves up to 75% of memory resources. *kFlushing* is shown to be efficient and scalable in digesting up to 100K microblog/second, which is an order of magnitude higher rate than current Twitter firehose rate.

References

[1] B. Babcock, M. Datar, and R. Motwani, “Load Shedding for Aggregation Queries Over Data Streams,” in *ICDE*, 2004, pp. 350–361.
[2] A. Bermingham and A. F. Smeaton, “Classifying Sentiment in Microblogs: Is Brevity an Advantage?” in *CIKM*, 2010.
[3] “After Boston Explosions, People Rush to Twitter for Breaking News. 2013,” http://www.latimes.com/business/technology/la-fi-tn-after-boston-explosions-people-rush-to-twitter-for-breaking-news-20130415,0,3729783.story.
[4] C. Budak, T. Georgiou, D. Agrawal, and A. E. Abbadi, “GeoScope: Online Detection of Geo-Correlated Information Trends in Social Networks,” in *VLDB*, 2014.
[5] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, “Earlybird: Real-Time Search at Twitter,” in *ICDE*, 2012.
[6] C. Chen, F. Li, B. C. Ooi, and S. Wu, “TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets,” in *SIGMOD*, 2011.
[7] “Sina Weibo, Chinas Twitter, comes to rescue amid flooding in Beijing. 2012,” http://thenextweb.com/asia/2012/07/23/sina-weibo-chinas-twitter-comes-to-rescue-amid-flooding-in-beijing/.
[8] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, “Anti-Caching: A New Approach to Database Management System Architecture,” in *VLDB*, 2013.
[9] W. Effelsberg and T. Härder, “Principles of Database Buffer Management,” *TODS*, vol. 9, no. 4, pp. 560–595, 1984.
[10] “Facebook Statistics,” http://newsroom.fb.com/company-info/, 2015.
[11] W. Feng, J. Han, J. Wang, C. Aggarwal, and J. Huang, “STREAMCUBE: Hierarchical Spatio-temporal Hashtag Clustering for Event Exploration Over the Twitter Stream,” in *ICDE*, 2015.
[12] B. Gedik, K. Wu, P. S. Yu, and L. Liu, “A Load Shedding Framework and Optimizations for M-way Windowed Stream Joins,” in *ICDE*, 2007, pp. 536–545.
[13] Y. He, S. Barman, and J. F. Naughton, “On Load Shedding in Complex Event Processing,” in *ICDT*, 2014, pp. 213–224.
[14] L. Hong, A. Ahmed, S. Gurumurthy, A. J. Smola, and K. Tsioutsiouliklis, “Discovering Geographical Topics In The Twitter Stream,” in *WWW*, 2012.
[15] J. J. Levandoski, P. Larson, and R. Stoica, “Identifying Hot and Cold Data in Main-memory Databases,” in *ICDE*, 2013.
[16] Y. Li, Z. Bao, G. Li, and K.-L. Tan, “Real Time Personalized Search on Social Networks,” in *ICDE*, 2015.
[17] J. Lin and G. Mishne, “A Study of ”Churn” in Tweets and Real-Time Search Queries,” in *ICWSM*, 2012.
[18] A. Magdy and M. Mokbel, “Towards a Microblogs Data Management System,” in *MDM*, 2015.
[19] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He, “Mercury: A Memory-Constrained Spatio-temporal Real-time Search on Microblogs,” in *ICDE*, 2014, pp. 172–183.
[20] ——, “Venus: Scalable Real-time Spatial Queries on Microblogs with Adaptive Load Shedding,” *TKDE*, vol. 28, no. 2, pp. 1–15, 2016.
[21] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller, “Tweets as Data: Demonstration of TweeQL and TwitInfo,” in *SIGMOD*, 2011.
[22] T. Sakaki, M. Okazaki, and Y. Matsuo, “Earthquake Shakes Twitter Users: Real-Time Event Detection by Social Sensors,” in *WWW*, 2010.
[23] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling, “TwitterStand: News in Tweets,” in *SIGSPATIAL*, 2009.
[24] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen, “Scalable Top-k Spatio-temporal Term Querying,” in *ICDE*, 2014, pp. 148–159.
[25] “Tracking Disease Trends,” http://nowtrending.hhs.gov/, 2015.
[26] “The Engineering Behind Twitter New Search Experience,” blog.twitter.com/2011/engineering-behind-twitter%E2%80%99s-new-search-experience, 2011.
[27] “Twitter Statistics,” https://about.twitter.com/company, 2015.
[28] L. Wu, W. Lin, X. Xiao, and Y. Xu, “LSII: An Indexing Structure for Exact Real-Time Search on Microblogs,” in *ICDE*, 2013.
[29] J. Yao, B. Cui, Z. Xue, and Q. Liu, “Provenance-based Indexing Support in Micro-blog Platforms,” in *ICDE*, 2012.
[30] H. Zhang, G. Chen, B. C. Ooi, W. Wong, S. Wu, and Y. Xia, “”Anti-Caching”-based Elastic Memory Management for Big Data,” in *ICDE*, 2015.