# Mercury: A Memory-Constrained Spatio-temporal Real-time Search on Microblogs

Amr Magdy[1§], Mohamed F. Mokbel[2§], Sameh Elnikety[3], Suman Nath[4], Yuxiong He[5]

[1,2]*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455*

[3,4,5]*Microsoft Research, Redmond, WA 98052-6399*

{amr[1],mokbel[2]}@cs.umn.edu,{samehe[3],suman.nath[4],yuxhe[5]}@microsoft.com

***Abstract*—This paper presents *Mercury*; a system for real-time support of top-$k$ spatio-temporal queries on microblogs, where users are able to browse recent microblogs near their locations. With high arrival rates of microblogs, *Mercury* ensures real-time query response within a tight memory-constrained environment. *Mercury* bounds its search space to include only those microblogs that have arrived within certain spatial and temporal boundaries, in which only the top-$k$ microblogs, according to a spatio-temporal ranking function, are returned in the search results. *Mercury* employs: (a) a scalable dynamic in-memory index structure that is capable of digesting all incoming microblogs, (b) an efficient query processor that exploits the in-memory index through spatio-temporal pruning techniques that reduce the number of visited microblogs to return the final answer, (c) an *index size tuning* module that dynamically finds and adjusts the minimum index size to ensure that incoming queries will be answered accurately, and (d) a *load shedding* technique that trades slight decrease in query accuracy for significant storage savings. Extensive experimental results based on a real-time Twitter Firehose feed and actual locations of Bing search queries show that *Mercury* supports high arrival rates of up to 64K microblogs/second and average query latency of 4 msec.**

## I. Introduction

Microblogs, e.g., tweets, Facebook comments, and Foursquare check-in's, are among the most popular web services nowadays. For example, Twitter has 140+ Million active users who generate 400+ Million daily tweets [38], while Facebook has 1+ Billion users who post 3.2+ Billion daily comments [12]. Combined with the advances in wireless communication and GPS-equipped handheld devices, microblogs have entered a new era where locations can be attached to each posted microblog to indicate the whereabouts of the microblog issuer. Consequently, Facebook added the options of location check-ins and *near* where users can state a nearby location of their status messages, Twitter automatically captures the GPS coordinates from mobile devices, while Foursquare is a microblog service that is all around the location information and the whereabouts of its users.

In this paper, we aim to take advantage of the combination of location information with microblogs to support spatio-temporal search queries on microblogs, where users are interested in *getting a set of recent microblogs (within the last $T$ time units) and within a certain spatial region*. Due to the large numbers of microblogs that can satisfy the given constraints, we limit the query answer to $k$ microblogs, deemed most relevant to the querying user based on a ranking function $F$ that combines the time recency and the spatial proximity of each microblog to the querying user.

§The work of these authors is partially supported by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168.

Users of our proposed spatio-temporal queries include news agencies (e.g., CNN and Reuters) to have a first-hand knowledge on events in a certain area, advertising services to serve geo-targeted ads to their customers based on nearby events, or individuals who want to know what is currently going on in a certain area. For example, in April 2013, Los Angeles Times reported [4] how people rush to Twitter for real-time breaking news about Boston Marathon explosions. Such users may not know the appropriate keyword or hash tag to search for. Instead, they want to know what are the recently posted microblogs in a certain particular area. Thus, our goal here is not to replace the traditional keyword search in microblogs, but rather to provide another important search option for localized microblogs. The answer of our spatio-temporal queries can be fed to other modules for further processing, which may include event detection, keyword search, entity resolution, sentiment analysis, or visualization.

We present *Mercury*: a system for real-time support of spatio-temporal queries on microblogs. *Mercury* faces two main challenges: high arrival rates of microblogs and the need for real-time query response. Both challenges call for relying on *only* in-memory data structures to index and query incoming microblogs, where memory is a scarce resource. Hence, *Mercury* employs an in-memory partial pyramid index [2], equipped with efficient bulk insertion, bulk deletion, speculative cell splitting, and lazy cell merging operations that make the index able to digest the high arrival rates of incoming microblogs. Incoming queries efficiently exploit the in-memory index through spatio-temporal pruning techniques that minimize the number of visited microblogs to return the final answer. *Mercury* bounds its search space by a *spatial* boundary $R$ as a search area around the user location of interest and a *temporal* boundary $T$ as the search past time horizon. Within $R$ and $T$, a ranking function $F$ is employed to score each microblog, per its spatial proximity and time recency, to produce the top-$k$ microblogs as the query answer. *Mercury* is optimized for a preset default values of $T$, $R$, and $k$. Queries with less values than the default can still be satisfied with the same performance. Yet, queries with higher values may encounter higher cost as they may need to visit a secondary storage. This goes along with the design choices of major web services, e.g., Bing and Google return, by default, the top-$k$ ($k$=10) most related search results, while Twitter gives the most recent $k$ tweets to a user upon logging on. If a user would like to get more than $k$ results, an extra query response time will be paid.

A direct way to ensure that all incoming queries will be satisfied from in-memory contents is to store and index all incoming microblogs from the last default $T$ time units. However, that may require a very large main memory, which can be prohibitively expensive. Hence, we propose two effective memory optimization techniques: (1) We develop an *index size tuning* technique that achieves significant memory savings (up to 50%) without sacrificing the query answer quality (more than 99% accuracy). The main idea is to exploit the diversity of arrival rates per regions, e.g., city centers have higher arrival rates than suburban areas. Hence, the top-$k$ microblogs would have arrived more recently in city centers than suburban areas. We maintain only the items that may appear in user queries, delete items that are dominated by others. (2) For scarce memory configurations, we develop a parameterized *load shedding* technique that trades significant reduction in the memory footprint (up to 75% less storage) for a small loss in query accuracy (up to 5% accuracy loss). The idea is to expel from memory a set of victim microblogs that are less likely to contribute to a query answer.

We evaluate the system experimentally based on a real system deployment of *Mercury* and using a real-time feed of US tweets (via access to Twitter Firehose) and actual locations of Bing web search queries. Our measurements show that *Mercury* supports arrival rates of up to 64K microblogs/second, average query latency of 4 msec, minimal memory footprints, and a very high query accuracy of 99%.

In addition to introducing *Mercury* as well as providing a crisp definition for spatio-temporal search queries over microblogs (Section III), the contributions of this paper are summarized as follows:

1) We propose efficient spatio-temporal indexing/expelling techniques that are capable of inserting/deleting microblogs with high rates (Section IV).
2) We introduce an efficient spatio-temporal query processor that minimizes the number of visited microblogs to return the final answer (Section V).
3) We introduce an *index size tuning* module that dynamically adjusts the index contents to achieve significant memory savings without sacrificing the query answer quality (Section VI).
4) We introduce a *load shedding* technique that trades significant reduction in memory footprint for a slight decrease in query accuracy (Section VII).

Section VIII gives experimental evidence, based on real system prototype, microblogs, and queries, showing that *Mercury* is scalable and accurate with minimal memory consumption. Finally, Section IX concludes the paper.

## II. Related Work

Due to its widespread use, recent research efforts have explored various research directions related to microblogs. This goes along the way of the system stack starting from logging [18] and machine learning techniques [21] to indexing [5], [7], [42], [43] and designing a SQL-like query language interface [24]. In addition, several efforts have focused on analyzing microblog data, which include semantic and sentiment analysis [3], [28], [30], decision making [6], news extraction [35], event and trend detection [1], [19], [27], [34], [37], understanding the characteristics of microblog posts and search queries [22], [33], microblogs ranking [11], [39], and recommending users to follow or news to read [14], [32]. Meanwhile, recent work [35], [40] exploited microblogs contents to extract location information that is used to visualize microblog posts on a map [25], [26] and model the relationship between user interests, locations, and topics [15].

With such rich work in microblogs, up to our knowledge, there is no existing work that address real-time indexing and querying microblogs locations; which is the main focus of this paper. However, the two most related topics to our work are *microblog search queries* and *spatio-temporal streams*.

**Microblog Search Queries.** Real-time search on microblogs often refers to keyword search [5], [7], [42], [43]. The difference of one technique over the other is mainly in the query type, accuracy, ranking function, and memory management. None of these work have addressed the case of location-aware search. On the other hand, spatial keyword search is well studied on web documents and web spatial objects [9], [10], [20], [41], [44]. However, they use offline disk-based data partitioning indexing, which cannot scale to support the dynamic nature and arrival rates of microblogs [5], [8].

**Spatio-temporal Streams.** Microblogs can be considered as a spatio-temporal stream with very high arrival rates, where there exist a lot of work for spatio-temporal queries over data streams [16], [23], [29], [31], [45]. However, the main focus of such work is on continuous queries over moving objects. In such case, a query is registered first, then its answer is composed over time from the incoming data stream. Such techniques are not applicable to spatio-temporal search queries on microblogs, where we retrieve the answer from existing stored objects that have arrived prior to issuing the query.

*Mercury* shares with microblogs keyword search its environment (i.e., queries look for existing data, in-memory indexing, and the need for efficient utilization of the scarce memory resource), yet, it is different from keyword search in terms of the functionality it supports, i.e., spatio-temporal queries. In the mean time, *Mercury* shares similar functionality with spatio-temporal queries over data streams, yet it is different in terms of the environment it supports, i.e., query answer is retrieved from existing data rather than from new incoming date to arrive later. Finally, *Mercury* shares with both keyword search and spatio-temporal queries the need to support incoming data with high arrival rates and the need to support real-time search query results.

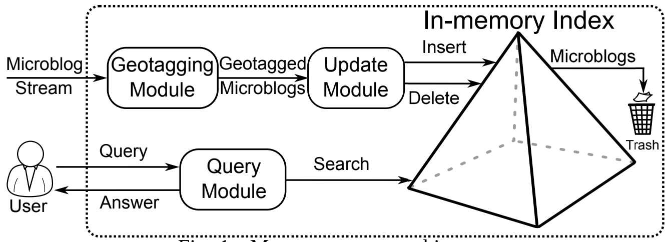


Fig. 1. Mercury system architecture.

## III. System Overview

This section gives an overview of *Mercury* system architecture, supported queries, and ranking function.

### A. System Architecture

Figure 1 gives *Mercury* system architecture with three main modules around an in-memory index, namely, *geotagging*, *update*, and *query* modules, described briefly below:

**Geotagging module.** This module receives the incoming stream of microblogs, extracts the location of each microblog, and forwards each microblog along with its extracted location to the *update module* with the form: *(ID, location, timestamp, content)* that presents the microblog identifier, location, issuing time, and textual contents. Location is either a precise *latitude* and *longitude* coordinates (if known) or a Minimum Bounding Rectangle (MBR). We extract the microblog locations through one or more of the following: (1) *Exact locations*, if already associated with the microblog, e.g., posted from a GPS-enabled device. (2) *User locations*, extracted from the issuing user profile. (3) *Content locations*, by parsing the microblog contents to extract location information. If the microblog ends up to be associated with more than one location, we output multiple versions of it as one per each location. If no location information can be extracted, we set the microblog MBR to the whole space. As we use existing software packages and public datasets for geocoding and location extraction, this module will not be discussed further in this paper.

**Update module.** The *update* module ensures that all incoming queries can be answered accurately from indexed in-memory contents with the minimum possible memory consumption. This is done through three main tasks: (1) Inserting newly coming microblogs into the in-memory index structure, (2) Smartly deciding on the set of microblogs to expire from memory without sacrificing the query answer quality, and (3) In cases of very tight memory, a load shedding module is triggered to smartly trade slight decrease in query accuracy with significant savings in memory consumptions. Details of index operations, index size tuning, and load shedding are discussed in Sections IV, VI, and VII, respectively.

**Query module.** Given a location search query, the *query* module employs spatio-temporal pruning techniques that reduce the number of visited microblogs to return the final answer. As the *query* module just retrieves what is there in the index, it has nothing to do in controlling its result accuracy, which is mainly determined by the decisions taken at the *update* module on what microblogs to expire from the in-memory index. Details of the *query* module are described in Section V.

### B. Supported Queries

*Mercury* users (or applications) issue queries on the form: "*Retrieve a set of recent microblogs near this location*". Internally, four parameters are added to this query: (1) $k$; the number of microblogs to be returned, (2) a range $R$ around the user location, where any microblog located outside $R$ is considered too far to be relevant, (3) a time span $T$, where any microblog that is issued more than $T$ time units ago is considered too old to be relevant, and (4) a spatio-temporal ranking function $F_\alpha$ that employs a parameter $\alpha$ to combine the temporal recency and spatial proximity of each microblog to the querying user. Then, the query answer consists of $k$ microblogs posted within $R$ and $T$, and top ranked according to $F_\alpha$. Formally, our query is defined as follows:

**Definition:** *Given $k$, $R$, $T$, and $F_\alpha$, a microblog spatio-temporal search query from user $u$, located at $u.loc$, finds $k$ microblogs such that: (1) The $k$ microblogs are posted in the last $T$ time units, (2) The (center) locations of the $k$ microblogs are within range $R$ around $u.loc$, and (3) The $k$ microblogs are the top ranked ones according to the ranking function $F_\alpha$.*

Our query definition is a natural extension to traditional spatial range and $k$-nearest-neighbor queries, used extensively in spatial and spatio-temporal databases [17], [36]. A range query finds all items within certain spatial and temporal boundaries. With the large number of microblogs that can make it to the result, it becomes natural to limit the result size to $k$, and hence a ranking function $F_\alpha$ is provided. Similarly, a $k$-nearest-neighbor query finds the *closest* $k$ items to the user location. As the relevance of a microblog is determined by both its time and location, we change the term *closest* to be *most relevant*, hence we define a ranking function $F_\alpha$ to score each microblog within our spatial and temporal boundaries.

Upon initialization, a system administrator sets default values for parameters $k$, $R$, $T$, and $\alpha$. Users may still change the values of the default parameters, yet a query may have less performance if the new parameters present larger search space than the default ones. This goes along with the design choices of major web services. For example, web search engines, e.g., Bing and Google, return the top-$k$ most related search results where $k$ is 10 by default. Similarly, Twitter gives the most recent $k$ tweets to a user upon logging on or in a keyword search result. If a user would like to get more than $k$ results, an extra query response time will be paid.

### C. Ranking Function

Given a user $u$, located at $u.loc$, a microblog $M$, issued at time $M.time$ and associated with location $M.loc$, and a parameter $0 \le \alpha \le 1$, *Mercury* employs the following ranking function $F_\alpha(u, M)$ that gives the relevance score of $M$ to $u$, where lower scores are favored:

$$\begin{aligned} F_\alpha(u,M) = & \quad \alpha \times SpatialDist(M.loc, u.loc) \\ + & \quad (1-\alpha) \times TemporalDist(M.time, NOW) \end{aligned}$$

$\alpha$=1 indicates that the user cares only about the spatial proximity of microblogs, i.e., query result includes the $k$ closest microblogs issued in the last $T$ time units. $\alpha$=0 gives

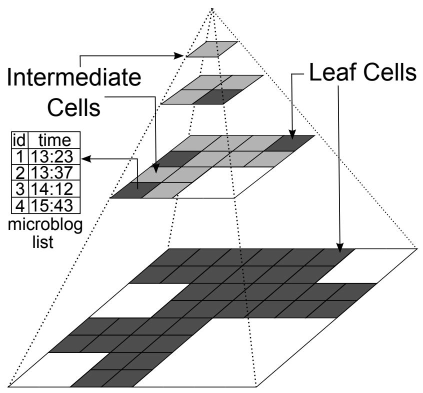


Fig. 2. Main memory pyramid index structure in Mercury.

the $k$ most recent microblogs within range $R$. A compromise between the two extreme values gives a weight of importance for the spatial proximity over the temporal recency.

*TemporalDistance(M.time,NOW)* ( *SpatialDistance(M.loc, u.loc)*) is any normalized <u>*monotonic*</u> function in the range $[0,1]$, applied only for microblogs in the last $T$ time units and within area $R$, where smaller values indicate more recent (closer) microblogs. The largest possible value 1 takes place when $M$ is posted exactly $T$ time units ago (on the boundary of region $R$). For simplicity, we employ the following functions, yet, other functions can be accommodated as long as they are monotonic:

$$TemporalDistance(M.time, NOW) = \begin{cases} \frac{NOW - M.time}{T} & NOW - M.time \le T \\ N/A & NOW - M.time > T \end{cases}$$

$$SpatialDistance(M.loc, u.loc) = \begin{cases} \frac{Distance(M.loc - u.loc)}{Radius(R)} & M.loc \textit{ inside } R \\ N/A & M.loc \textit{ outside } R \end{cases}$$

*Distance(M.loc-u.loc)* is the Euclidian distance between two points where *M.loc* and *u.loc* are either precise location coordinates or center points of their minimum bounding rectangles.

## IV. Spatio-temporal Indexing

We have two main objectives to satisfy in our *Mercury* indexing. First, the employed index has to be able to digest high arrival rates of incoming microblogs. Second, the employed index should be able to expel (delete) microblogs from its contents with the same rate as the arrival rate. This will ensure that the index size is fixed in a steady state, and hence all available memory is fully utilized. The need to support high arrival rates immediately favors space-partitioning index structures (e.g., quad-tree [13] and pyramid [2]) over data-partitioning index structures (e.g., R-tree). This is because the shape of data-partitioning index structures is highly affected by the rate and order of incoming data, which may trigger a large number of cell splitting and merging with a sub performance compared to space-partitioning index structures that are more resilient to the rate and order of insertions and deletions.

To this end, *Mercury* employs a partial pyramid structure [2] (Figure 2) that decomposes the space into $H$ levels. For a given level $h$, the space is partitioned into $4^h$ equal area grid cells. At the root, one grid cell represents the entire geographic area, level 1 partitions the space into four equi-area cells, and so forth. Dark cells in Figure 2 present leaf cells, which could lie in any pyramid level, light gray cells indicate non-leaf cells that are already decomposed into four children, while white cells are not actually maintained, and just presented for illustration. We favor the pyramid structure over quad-trees as it involves storing data in non-leaf nodes, which significantly helps in query processing. Each maintained pyramid cell $C$ has a list of microblog $M\_List$ that have arrived within the cell boundary in the last $T$ time units, ordered by their timestamps. A microblog with location coordinates is stored in the leaf cell containing its location, while a microblog with MBR is stored in the lowest level enclosing cell, which could be non-leaf. The pyramid index is spatio-temporal, where the whole space is *spatially* indexed (partitioned) into cells, and within each cell, microblogs are *temporally* indexed (sorted) based on timestamp.

Though it is most suitable to *Mercury*, existing pyramid index structures [2] are not equipped to accommodate the needs for high-arrival insertion/deletion rates of microblogs. To support high-rate insertions, we furnish the pyramid structure by a *bulk insertion* module that efficiently digests incoming microblogs with their high arrival rates (Section IV-A) and a *speculative cell splitting* module that avoids skewed cell splitting (Section IV-B). To support high-rate deletions, we provide a *bulk deletion* module that efficiently expels from the pyramid structure a set of microblogs that will not contribute to any query answer (Section IV-C) and a *lazy cell merging* module that decides on when to merge a set of cells together to minimize the system overhead (Section IV-D).

### A. Bulk Insertion

Inserting a microblog $M$ (with a point location) in the pyramid structure can be done traditionally [2] by traversing the pyramid from the root to find the leaf cell that includes $M$ location. If $M$ has an MBR location instead of a point location, we do the same except that we may end up inserting $M$ in a non-leaf node. Unfortunately, such insertion procedure is not applicable to microblogs due to its high arrival rates. While inserting a single item, new arriving items may get lost as the rate of arrival would be higher than the time to insert a single microblog. This makes it almost infeasible to insert incoming microblogs, as they arrive, one by one. To overcome this issue, we employ a *bulk insertion* module as described below.

The main idea is to buffer incoming microblogs in a memory buffer $B$, while maintaining a minimum bounding rectangle $B_{MBR}$ that encloses the locations of all microblogs in $B$. Then, the bulk insertion module is triggered every $t$ time units to flush all microblogs in $B$ to the pyramid index. This is done by traversing the pyramid structure from the root to the lowest cell $C$ that encloses $B_{MBR}$. If $C$ is a leaf node, we append the contents of $B$ to the top of the list of microblogs in $C$ ($C.M\_List$). This still ensures that $M\_List$ is sorted by timestamp as the oldest microblog in $B$ is more recent than the most recent entry in $M\_List$. On the other hand, if $C$ is a non-leaf node, we: (a) extract from $B$ those microblogs that are presented by MBRs and cannot be enclosed by any of $C$'s children, (b) append the extracted MBRs to the list of microblogs in $C$ ($C.M\_List$), (c) distribute the rest of microblogs in $B$, based on their locations, to four quadrant buffers that correspond to $C$'s children, and (d) execute bulk insertion recursively for each child cell of $C$ using its corresponding buffer.

The parameter $t$ is a tuning parameter that trades-off insertion overhead with the time that an incoming microblog becomes searchable. A microblog is searchable (i.e., can appear in a search result), only if it is inserted in the pyramid structure. So, the larger the value of $t$ the more efficient is the insertion, yet, an incoming microblog may be held in the buffer for a while before being searchable. A typical value of $t$ is a couple of seconds, which is enough to have few thousands microblogs inside $B$. Since a typical arrival rate in Twitter is 4K+ microblogs/second, setting $t$ = 2 means that each two seconds, we will insert 8,000 microblogs in the pyramid structure, instead of inserting them one by one as they arrive. Yet, a microblog may stay for up to two seconds after its arrival to be searchable, which is a reasonable time.

Bulk insertion significantly reduces insertion time as instead of traversing the pyramid for each single microblog, we group thousands of microblogs into MBRs and use them as our traversing unit. Also, instead of inserting each single microblog in its destination cell, we insert a batch of microblogs by attaching a buffer list to the head of the microblog list.

## B. Speculative Cell Splitting

Each pyramid index cell has a maximum capacity; set as an index parameter. If a leaf cell $C$ has exceeded its capacity, a traditional cell splitting module would split $C$ into four equi-area quadrants and distribute $C$ contents to the new quadrants according to their locations. Unfortunately, such traditional splitting procedure may not be suitable to microblogs. The main reason is that microblog locations are highly skewed, where several microblogs may have the same exact location, e.g., microblogs tagged with a hot-spot location like a stadium. Hence, when a cell splits, all its contents may end up going to the same quadrant and another split is triggered. The split may continue forever unless with a limit on the maximum pyramid height, allowing cells with higher capacity at the lowest level. This gives a very poor insertion and retrieval performance due to highly skewed pyramid branches with fat cells at the lowest level.

To avoid long skewed tree branches, we employ a *speculative cell splitting* module, where a pyramid cell $C$ is split into four quadrants only if two conditions are satisfied: (1) $C$ exceeds its maximum capacity, and (2) If split, microblogs in $C$ will span at least two quadrants. While it is easy to check the first condition, checking the second condition is more expensive. To this end, we maintain in each pyramid cell a set of split bits (*SplitBits*) as a four-bits variable; one per cell quarter (initialized to zero). We use the *SplitBits* as a proxy for non-expensive checking on the second condition.

After each bulk insertion operation in a cell $C$, we first check if $C$ is over capacity. If this is the case, we check for the second condition, where there could be only two cases for *SplitBits*: (1) Case 1: *The four SplitBits are zeros*. In this case, we know that $C$ has just exceeded its capacity during this insertion operation. So, for each microblog in $C$, we check which quadrant it belongs to, and set its corresponding bit in *SplitBits* to one. Once we set two different bits, we stop scanning the microblogs and split the cell as we now know that the cell contents will span more than one quadrant. If we end up scanning all microblogs in $C$ with only one set bit, we decide not to split $C$ as we are sure that a split will end up having all entries in one quadrant. (2) Case 2: *One of the SplitBits is one*. In this case, we know that $C$ was already over capacity before this bulk insertion operation, yet, $C$ was not split as all its microblogs belong to the same quadrant (the one with a *SplitBits* one). So, we only need to scan the new microblogs that will be inserted in $C$ and set their corresponding *SplitBits*. Then, as in Case 1, we split $C$ only if two different bits are set. In both cases, when splitting $C$, we reset its *SplitBits*, create four new cells with zero *SplitBits*, and distribute microblogs in $C$ to their corresponding quadrants. This shows that we would never face a case where two (or more) of the *SplitBits* are zeros, as once two bits are set, we immediately split the cell, and reset all bits to zeros.

Using *SplitBits* significantly reduces insertion and query processing time as: (a) we do not have dangling skewed tree branches, and (b) we avoid the expensive checking step for whether cell contents belong to the same quadrant or not, as the check is now done infrequently on a set of bits. In the mean time, maintaining the integrity of *SplitBits* in each cell $C$ comes with very little overhead. First, as long as $C$ is under capacity, we do not read or set the value of *SplitBits*. Second, deleting entries from $C$ has no effect on its *SplitBits*, unless $C$ becomes empty, where we reset all bits to zero.

## C. Bulk Deletion

As we have finite memory, *Mercury* needs to delete older microblogs to give room for newly incoming ones. Deleting an item $M$ from the pyramid structure can be done in a traditional way [2] by traversing the pyramid from its root till cell $C$ that encloses $M$, and then removing $M$ from $C$'s list. Unfortunately, such traditional deletion procedure cannot scale up for *Mercury* needs. Since we need to keep index contents to only objects from the last $T$ time units, we may need to keep pointers to all microblogs, and chase them one by one as they become out of the temporal window $T$, which is a prohibitively expensive operation. To overcome this issue, we employ a *bulk deletion* module where all deletions are done in bulk. We exploit two strategies for bulk deletion, namely, *piggybacking* and *periodic* bulk deletions, described below.

**Piggybacking Bulk Deletion.** The idea is to piggyback the deletion operation on insertion. Once a microblog is inserted in a cell $C$, we check if $C$ has any items older than $T$ time units in its microblog list ($M\_List$). As $M\_List$ is ordered by timestamp, we use binary search to find its most recent item $M$ that is older than $T$. If $M$ exists, we trim $M\_List$ by removing everything from it starting from $M$. Piggybacking deletion on insertion saves significant time as we share the pyramid traversal and cell access with the insertion operation.

**Periodic Bulk Deletion.** With piggybacking bulk deletion, a cell $C$ may still have some microblogs that should have been deleted, yet, they are still there as there is no recent insertions in $C$. To avoid such cases, we trigger a light-weight periodic bulk deletion process every $T'$ time units (we use $T' = 0.5T$). In this process, we go through each cell $C$, and only check for the first (i.e., most recent) item $M \in C.M\_List$. If $M$ has arrived more than $T$ time units ago, we wipe the cell by deleting its $M\_List$ and setting its number of items to zero. If $M$ has arrived within the last $T$ time units, we do nothing and skip $C$. It may be the case that $C$ still has some expired items, yet we intentionally overlook them in order to make the deletion light-weight. Such items will be deleted soon either in the next insertion or in the next periodic bulk deletion process.

Deleted microblogs are moved from our main pyramid structure to another similar index structure of larger size, stored in a lower storage tier. Deleted microblogs will be retrieved only if an issued query has a time boundary larger than $T$, which is an uncommon case, as most of our incoming queries use the default $T$ value.

### D. Lazy Cell Merging

After deletion, if the total size of $C$ and its siblings is less than the maximum cell capacity, a traditional cell merging algorithm would merge $C$ with its siblings into one cell. However, with the high arrival rates of microblogs, we may end up in spending most of the insertion and deletion overhead in splitting and merging pyramid cells, as the children of a newly split cell may soon merge again after deleting few items. To avoid such overhead, we employ a *lazy merging* strategy, where we merge four sibling cells into their parent only if three out of the four quadrant siblings are empty.

The idea is that once a cell $C$ becomes empty, we check its siblings. If two of them are also empty, we move the contents of the third sibling to its parent, mark the parent as a leaf node, and remove $C$ and its siblings from the pyramid index. This is lazy merging, where in many cases it may happen that four siblings include few items that can all fit into their parent. However, we avoid merging in this case to provide more stability for our highly dynamic index. Hence, once a cell $C$ is created, it is guaranteed to survive for at least $T$ time units before it can be merged again. This is because $C$ will not be empty, i.e., eligible for merging, unless there are no insertions in $C$ within $T$ time units. Although the lazy merging causes underutilized cells, this has a slight effect on storage and query processing, compared to saving 90% of redundant split/merge operations (which is measured practically) that leads to a significant reduction in index update overhead.

## V. Query Processing

This section discusses the query processing module, which receives a query from user $u$ with spatial and temporal boundaries, $R$ and $T$, and returns the top-$k$ microblogs according to a spatio-temporal ranking function $F_\alpha$ that weights the importance of spatial proximity and time recency of each microblog to $u$. A simple approach is to exploit the pyramid index structure to compute the ranking score for all microblogs within $R$ and $T$ and return only the top-$k$ ones. Unfortunately, such approach is prohibitively expensive due to the large number of microblogs within $R$ and $T$. Instead, *Mercury* uses the ranking function to prune the search space and minimize the number of visited microblogs through a two-phase query processor. The *initialization* phase (Section V-B) finds an initial set of $k$ microblogs that form a basis of the final answer. The *pruning* phase (Section V-C) keeps on tightening the initial boundaries $R$ and $T$ to enhance the initial result and reach to the final one.

### A. Query Data Structure

The query processor employs two main data structures; a heap priority queue of cells and a sorted list of microblogs:

**Heap priority queue of cells $H$:** A heap priority queue of all cells that overlap with query spatial boundary $R$. A cell in $H$ has the form (*C*, *index*, *BestScore*); where *C* is a pointer to the cell, *index* is the position of the first non-visited microblog in $C$ (initialized to one), and *BestScore* is the best (i.e., lowest) possible score, with respect to user $u$, that any non-visited microblog in $C$ may have. Cells are inserted in $H$ ordered by *BestScore*, computed as:

$$\begin{aligned}BestScore(u,C) &= \alpha \times MinSpatialDistance(C,u.loc)\\ &+ (1-\alpha) \times TemporalDistance(C.M\_List[index].time, NOW)\end{aligned}$$

where *MinSpatialDistance(C,u.loc)* is the minimum distance between $u$ and $C$ and *C.M_List[index]* is the most recent non-visited microblog in $C$.

**Sorted list of microblogs *AnswerSet*:** A sorted list of $k$ microblogs of the form (*MID*, *Score*), as the microblog id and score, sorted on score. Upon completion of query processing, *AnswerSet* contains the final answer.

### B. The Initialization Phase

The *initialization* phase gets an initial set of $k$ microblogs that form the basis of pruning in the next phase. One approach is to get the most recent $k$ microblogs from the pyramid cell $C$ that includes the user location. Yet, this is inefficient as: (1) $C$ may contain less than $k$ microblogs within $T$, and (2) Other microblogs outside $C$ may provide tighter bounds for the initial $k$ items, which leads to faster pruning later.

**Main Idea.** The main idea is to consider all cells within the spatial boundary $R$ in constructing the initial set of $k$ microblogs. We initialize the heap $H$ by one entry for each cell $C$ within $R$. Entries are ordered based on best scores computed as discussed in Section V-A. Then, we take the top cell entry $C$ in $H$ as our strongest candidate to contribute to the initial top-$k$ list. We remove $C$ from $H$ and check on its microblogs

**Algorithm 1** Query Processor

```
1: Function Query Processor ($u$, $k$, $T$, $R$, $\alpha$)
2: $H \leftarrow \phi$; $AnswerSet \leftarrow \phi$; $MIN \leftarrow \infty$; $R' \leftarrow R$; $T' \leftarrow T$
3: for each leaf cell $C$ overlaps with $R$ do
4:     BestScore $\leftarrow \alpha \frac{Dist(u.loc,C)}{R}$ + (1-$\alpha$) $\frac{NOW-C.M\_List[1].time}{T}$
5:     Insert ($C$, 1, BestScore) into $H$
6: end for
7: $TopH$ ← Get (and remove) first entry in $H$
8: while $TopH$ is not NULL and $TopH.score < MIN$ do
9:     Score ← TopH.score; $M$ ← TopH.C.M_List[TopH.index]
10:    NextScore ← score of current top entry in $H$
11:    while Score < NextScore and $M$ is not NULL do
12:        if $M.loc$ inside $R'$ then
13:            Score $\leftarrow \alpha \frac{Dist(u.loc,M.loc)}{R} + (1-\alpha) \frac{NOW-M.time}{T}$
14:            if Score $< MIN$ then
15:                Insert ($M$,Score) in $AnswerSet$
16:                if $|AnswerSet| \geq k$ then
17:                    Trim $AnswerSet$ size to $k$
18:                    $MIN \leftarrow AnswerSet[k].score$;
19:                    if $MIN < \alpha$ then $R' \leftarrow \frac{MIN}{\alpha} \times R'$
20:                    if $MIN < (1-\alpha)$ then $T' \leftarrow \frac{MIN}{1-\alpha} \times T'$
21:                end if
22:            end if
23:        end if
24:        $M$ ← Next microblog in $TopH.C.M\_List$
25:        if $M.time$ outside $T'$ then $M$ ← NULL
26:    end while
27:    if $M \neq$ NULL then Insert ($C$, index($M$), BestScore) in $H$
28:    $TopH$ ← Get (and remove) first entry in $H$
29: end while
30: Return $AnswerSet$
```

one by one in their temporal order. For each microblog $M$, we compare its score against the best score of the current top cell $C'$ in $H$. If $M$ has a smaller (better) score, we insert $M$ in our initial top-$k$ list, and check on the next microblog in $C$. Otherwise, (a) we conclude that the next cell entry $C'$ in $H$ has a stronger chance to contribute to top-$k$, so we repeat the same procedure for $C'$, and (b) If $M$ is still within the temporal boundary $T$, we insert a new entry of $C$ into $H$ with a new best score. We continue doing so till we collect $k$ items in the top-$k$ list.

**Algorithm.** Algorithm 1 starts by populating the heap $H$ with an entry for each cell $C$ that overlaps with the query boundary $R$. Each cell entry has its pointer, the index of the first non-visited microblog as one, and the best score that any entry in $C$ can have (Lines 2 to 6). Then, we remove the top entry $TopH$ from $H$, and keep on retrieving microblogs from the cell $TopH.C$ and insert them into our initial answer set till any of these three stopping conditions take place: (1) We collect $k$ items, where we conclude the *initialization* phase at Line 16, (2) The next microblog in $C$ is either outside $T$ or does not exist, where we set $M$ to NULL (Line 25) and retrieve a new top entry $TopH$ from $H$ (Line 28), or (3) The next microblog $M$ in $C$ is within $T$, yet it has a higher score than the current top entry in $H$. So, we insert a new entry of $C$ with a new score and current index of $M$ in $H$, and retrieve a new top entry $TopH$ from $H$ (Lines 27 to 28). The conditions at Lines 8 and 14 are always True in this phase as *MIN* is set to $\infty$.

## C. The Pruning Phase

The *pruning* phase takes the *AnswerSet* from the *initialization* phase and enhances it contents to reach the final $k$.

**Main Idea.** The *pruning* phase keeps on tightening the original search boundaries $R$ and $T$ to new boundaries, $R' \leq R$ and $T' \leq T$, till all microblogs within the tightened boundaries are exhausted. Microblogs outside the tightened boundaries are early pruned without looking at their scores. The idea is to maintain a threshold *MIN* as the score of the $k$th element in *AnswerSet*. For a microblog $M$ to be included in *AnswerSet*, $M$ has to have a lower score than *MIN*, i.e.,:

$$\alpha \frac{Dist(u.loc, M.loc)}{R} + (1-\alpha)\frac{NOW - M.time}{T} < MIN$$

This formula is used for spatial and temporal boundary tightening as follows: (1) *Spatial boundary tightening.* Assume that $M$ has the best possible temporal score, i.e., *M.time* = *NOW*. In order for $M$ to make it to *AnswerSet*, we should have: $\alpha\frac{Dist(u.loc,M.loc)}{R} < MIN$, i.e., $M$ has to be within distance $\frac{MIN}{\alpha}R$ from the user. Hence, we tighten our spatial boundary to $R' = Min(R, \frac{MIN}{\alpha}R)$. (2) *Temporal boundary tightening.* Assume that $M$ has the best possible spatial score, i.e., *Dist(u.loc,M.loc)* = 0. In order for $M$ to make it to *AnswerSet*, we should have: $(1-\alpha)\ \frac{NOW-M.time}{T} < MIN$, i.e., $M$ has to be issued within the last $\frac{MIN}{1-\alpha}T$ time units. Hence, we tighten our temporal boundary to $T' = Min(T, \frac{MIN}{1-\alpha}T)$.

**Pruning steps.** Based on our bound tightening and the values of $MIN$ and $\alpha$, the *pruning* phase goes through three pruning steps in order: (1) *No Pruning*: When $MIN$ > *Max*($\alpha$,1 − $\alpha$), then we search within our original boundaries $R$ and $T$. (2) *One-dimensional Pruning*: When $MIN$ lies between $\alpha$ and $(1-\alpha)$, we start to employ either *spatial* or *temporal* pruning, based on the value of $\alpha$. If $\alpha > (1-\alpha)$, we only apply *spatial* pruning, otherwise, we go for *temporal* pruning. (3) *Spatio-temporal Pruning*: When $MIN <$ *Min*($\alpha$,1−$\alpha$), we tighten both *spatial* and *temporal* boundaries till we exhaust all microblogs in the tightened boundaries.

**Algorithm.** Line 16 in Algorithm 1 is the entry point for the *pruning* phase, where we already have $k$ microblogs in *AnswerSet*. We first set *MIN* to the minimum score in *AnswerSet*. Then, we check if we can apply spatial and/or temporal pruning based on the values of *MIN* and $\alpha$ as described above. Pruning and bound tightening are continuously applied with every time we find a new microblog $M$ with a lower score than *MIN*, where we insert $M$ into *AnswerSet* and update *MIN* (Lines 14 to 22). The algorithm then continues exactly as in the *initialization* phase by checking if there are more entries in the current cell or we need to get another cell from the heap. The algorithm concludes and returns the final answer list if any of two conditions takes place (Line 8): (a) Heap $H$ is empty, which means that we have exhausted all microblogs in the boundaries, or (b) The best score of top entry of $H$ is larger than *MIN*, which means all microblogs in $H$ cannot make it to the final answer.

## VI. Index Size Tuning

Our discussion so far assumed that all microblogs posted in the last $T$ time units are stored in the in-memory pyramid structure. Hence, a query with any temporal boundary $\leq T$ guarantees to find all its answer in memory. In this section, we introduce the *index size tuning* module that takes advantage of the natural skewness of data arrival rates over different pyramid cells to achieve its storage savings (~50% less storage) without sacrificing the query accuracy (accuracy ~99%). Our *index size tuning* is motivated by two main observations: (1) The top-$k$ microblogs in areas with high microblog arrival rates can be obtained from a much shorter time than areas of low arrival rates, e.g., top-$k$ microblogs in downtown Chicago may be obtained from the last 30 minutes, while it may need couple of hours to get them in a suburb area. (2) $\alpha$ plays a major role on how far we need to go back in time to look for microblogs. If $\alpha = 1$, top-$k$ microblogs are the closest ones to the user locations, regardless of their time arrival within $T$. If $\alpha = 0$, top-$k$ microblogs are the most recent ones posted within $R$, so, if we look back only for the time needed to issue $k$ microblogs. Then, for each cell $C$, we find the minimum search time horizon $T_c \leq T$ such that an incoming query to $C$ finds its answer in memory. Section VI-A derives the values of $T_c$, for each cell $C$, while Section VI-B discusses the impact of the *index size tuning* module on various *Mercury* components.

### A. Reducing the Cell Size

This section aims to find the value $T_c$ for each cell $C$ such that only those microblogs that have arrived in $C$ in the last $T_c$ time units are kept in memory. Per the following Lemma, $T_c$ is computed based on the default values of $k$, $R$, $T$, and $\alpha$, and uses the microblog arrival rate $\lambda_c$ for each cell $C$. We assume that the locations of incoming microblogs are uniform within each cell boundary, yet they are diverse across various cells, hence each cell $C$ has its own microblog arrival rate $\lambda_c$

**Lemma 1:** *Given query parameters $k$, $R$, $T$, and $\alpha$, and the average arrival rate of microblogs in cell $C$, $\lambda_c$, the spatio-temporal query answer from cell C can be retrieved from those microblogs that have arrived in the last $T_c$ time units, where:*

$$T_c = Min\left(T, \frac{\alpha}{1-\alpha}T + \frac{k}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_c}\right)$$

**Proof:** The proof is composed of three steps: First, we compute the value of $\lambda_R$ as the expected arrival rate of microblogs to query area $R$, among the microblogs in cell $C$ with arrival rate $\lambda_c$. This depends on the ratio of the two areas *Area(R)* and *Area(C)*. If *Area(R) < Area(C)*, then $\lambda_R = \frac{Area(R)}{Area(C)}\lambda_c$, otherwise, all microblogs from $C$ will contribute to $R$, hence $\lambda_R = \lambda_c$. This can be put formally as:

$$\lambda_R = Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_c$$

Second, we compute the shortest time $T_k$ to form a set of $k$ microblogs as an initial answer. This corresponds to the time to get the first $k$ microblogs that arrive within cell $C$ and area $R$. Since $\lambda_R$ is the rate of microblog arrival in $R$, i.e., we receive one microblog each $\frac{1}{\lambda_R}$ time units, then we need $T_k = \frac{k}{\lambda_R}$ time units to receive the first $k$ microblogs.

Finally, we compute the maximum time interval $T_c$ that a microblog $M$ within cell $C$ and area $R$ can make it to the list of top-$k$ microblogs according to our ranking function $F$. In order for $M$ to make it to the top-$k$ list, $M$ has to have a better (i.e., lower) score than the microblog $M_k$ that has the $k$th (i.e., worst) score of the initial top-$k$, i.e., $F(M) < F(M_k)$. To be conservative in our analysis, we assume that: (a) $M$ has the best possible spatial score: zero, i.e., $M$ has the same location as the user location. In this case, $F(M)$ will rely only on its temporal score, i.e., $F(M) = (1-\alpha)\frac{T_c}{T}$, where $T_c = NOW - M.time$ indicates the search time horizon $T_c$ that we are looking for, and (b) $M_k$ has the worst possible spatial and temporal scores among the initial $k$ ones. While the worst spatial score would be one, i.e., $M_k$ lies on the boundary of $R$, the worst temporal score would take place if $M_k$ arrives $T_k$ time units ago. So, the score of $M_k$ can be set as: $F(M_k) = \alpha + (1-\alpha)\frac{k}{\lambda_R T}$. Accordingly, to satisfy the condition that $F(M) < F(M_k)$, the following should hold:

$$(1-\alpha)\frac{T_c}{T} < \alpha + (1-\alpha)\frac{k}{\lambda_R T} \qquad (1)$$

This means that in order for $M$ to make it to the answer list, $T_c$ should satisfy:

$$T_c < \frac{\alpha}{1-\alpha}T + \frac{k}{\lambda_R}$$

By substituting the value of $\lambda_R$, and bounding $T_c$ by the value of $T$, as we cannot go further back in time than $T$, the maximum value of $T_c$ would be:

$$T_c = Min\left(T, \frac{\alpha}{1-\alpha}T + \frac{k}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_c}\right)$$

■

Lemma 1 means that in order for a microblog $M$ in cell $C$ to make it to the top-$k$ answer, $M$ has to arrive within the last $T_c$ time units, where $T_c \leq T$. Therefore, we save memory space by storing fewer microblogs.

### B. Impact on Mercury Components

This section discusses the impact of employing the $T_c$ values on *Mercury* components, namely, index structure, index operations, index maintenance, and query processing.

**Index Structure.** Each pyramid cell $C$ will keep track of two additional variables: (1) $\lambda_c$; the arrival rate of microblogs in $C$, which is continuously updated with new microblog arrivals, and (2) $T_c$; the temporal boundary in cell $C$ computed from Lemma 1, and updated with every update of $\lambda_c$.

**Index Operations.** Insertion in the pyramid index will have the following two changes: (1) For all visited cells in the insertion process, we update the values of $\lambda_c$ and $T_c$, (2) If $T_c$

is updated with a new value, we will have one of two cases: (a) The value of $T_c$ is decreased. In this case, microblogs that were posted in the time interval between the old and new values of $T_c$ are immediately deleted. (b) The value of $T_c$ is increased. In this case, we have a temporal gap between the new and old values of $T_c$, where there are no microblogs there. However, with the rate of updates of $T_c$, such gap will be filled up soon, and hence would have very little impact on query answer. On the other side, deletion module deletes microblogs from each cell $C$ based on the value of $T_c$ rather than based on one value $T$ for all cells.

**Index Maintenance.** When a cell $C$ splits into four quadrant cells, the value of $\lambda_c$ in each new child cell $C_i$ is set based on the ratio of microblogs from cell $C$ that goes to cell $C_i$. As *Mercury* employs a lazy merging policy, i.e., four cells are merged into a parent cell $C$ only if three of them are empty, the value of $\lambda_c$ at the parent cell $C$ is set to the arrival rate of its only non-empty child.

**Query Processor.** The query processor module is left intact as it retrieves its answer from the in-memory data regardless of the temporal domain of the contents.

## VII. Load Shedding

Even with the *index size tuning* module, there could be cases where there is no enough memory to hold all microblogs from the last $T_c$ time units in each cell, e.g., very scarce memory or time intervals with very high arrival rates. Also, some applications are willing to trade slight decrease in query accuracy with a large saving in memory consumption. In such cases, *Mercury* triggers a *load shedding* module that smartly selects and expires a set of microblogs from memory such that the effect on query accuracy is minimal. The main idea of the *load shedding* module is to use less conservative analysis than that of the *index size tuning* module, discussed in Section VI. In particular, Equation 1 was very conservative in assuming that there is a microblog $M$ that lies exactly on the same location of the querying user, and hence $M$ has a spatial score of zero. The *load shedding* module relaxes this assumption and assumes that $M$ has a spatial score of $0 \le \beta \le 1$ (instead of zero), and hence Equation 1 will be re-formulated as:

$$\alpha\beta + (1-\alpha)\frac{T_{c,\beta}}{T} < \alpha + (1-\alpha)\frac{k}{\lambda_R T} \qquad (2)$$

We use the term $T_{c,\beta}$ instead of $T_c$ to indicate the search time horizon for each cell $C$ when the *load shedding* module is employed. $\beta$ acts as a tuning parameter that trades-off significant savings of storage with slight loss of accuracy. $\beta = 0$ means that there is no load shedding, hence no storage savings over the *index size tuning* module ($T_{c,0} = T_c$). On the other hand, $\beta = 1$ means that the memory is barley enough to hold only the most recent $k$ microblogs per cell. As will be shown below and in our experiments, a storage saving of $\beta$ results in accuracy loss of $\beta^3$. For example, if $\beta = 0.3$, a 30% saving of storage is traded with only 2.7% of accuracy loss.

### A. Storage Savings

Starting from Equation 2, in order for a microblog $M$ to make it to the answer list, $T_{c,\beta}$ should satisfy:

$$T_{c,\beta} < \frac{\alpha(1-\beta)}{1-\alpha}T + \frac{k}{\lambda_R}$$

By substituting the value of $\lambda_R$, and bounding $T_{c,\beta}$ by the value of $T$, the maximum value of $T_{c,\beta}$ would be:

$$T_{c,\beta} = Min\left(T, \frac{\alpha(1-\beta)}{1-\alpha}T + \frac{k}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_R}\right) \quad (3)$$

Per Equation 3, $T_{c,\beta}$ gives a tighter temporal coverage for each cell as $T_{c,\beta} \le T_c$. Depending on the values of query parameters, $k$, $R$, $T$, and $\alpha$, storage savings ranges from 0 to $\beta$, i.e., if $\beta = 0.3$, we achieve up to 30% storage savings. This means that storage saving goes linear with $\beta$. Based on our extensive experiments in Section VIII, we have experimentally found that we always achieve storage saving of $\frac{\beta}{2}$ with much better accuracy than the theoretical bound discussed below.

### B. Accuracy Loss

Given the less conservative assumption in Equation 2, there is a chance to miss microblogs that could have made it to the final result. In particular, there is an area $A_x$ in the spatio-temporal space that is not covered by our analysis. A microblog $M$ in area $A_x$ satisfies two conditions: (1) The spatial score of $M$ is less than $\beta$, and (2) The temporal distance of $M$ is between $T_{c,\beta}$ and $T_c$. We measure the accuracy loss in terms of the ratio of the area covered by $A_x$ to the whole spatio-temporal area covered by $R$ and $T$, i.e., $R \times T$. This is measured by multiplying the ratios of the $A_x$'s temporal and spatial dimensions, $T_{ratio}$ and $R_{ratio}$, to the whole space. The temporal ratio $T_{ratio}$ can be measured as:

$$T_{ratio} = \frac{T_c - T_{c,\beta}}{T_c} = \frac{\left(\frac{\alpha}{1-\alpha}T + \frac{k}{\lambda_R}\right) - \left(\frac{\alpha(1-\beta)}{1-\alpha}T + \frac{k}{\lambda_R}\right)}{\left(\frac{\alpha}{1-\alpha}T + \frac{k}{\lambda_R}\right)}$$

$$This\ leads\ to: \qquad T_{ratio} = \beta \times \frac{\frac{\alpha}{1-\alpha}T}{\frac{\alpha}{1-\alpha}T + \frac{k}{\lambda_R}} \quad \le \beta$$

This means that the temporal ratio is bounded by $\beta$.

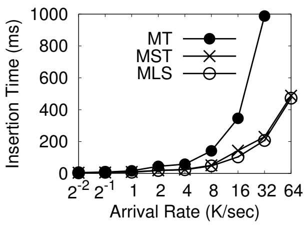
For the spatial ratio, consider that $A_x$ and $R$ are represented by circular areas around the querying user location with radius *Radius($A_x$)* and *Radius(R)*. Since a microblog $M$ at distance *Radius($A_x$)* has spatial score of $\beta$ while a microblog at distance *Radius(R)* has spatial score of 1, then *Radius($A_x$)* = $\beta$ *Radius(R)*. Hence, the ratio of the spatial dimension is:

$$R_{ratio} = \frac{Area(A_x)}{Area(R)} = \frac{\pi Radius(A_x)^2}{\pi Radius(R)^2} = \frac{\beta^2 Radius(R)^2}{Radius(R)^2} = \beta^2$$

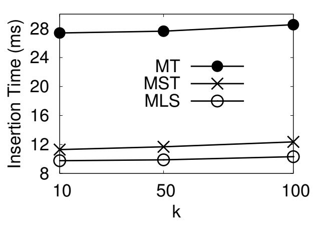
Hence, the accuracy loss can be formulated as:

$$AccuracyLoss_{\beta} = T_{ratio} \times R_{ratio} \le \beta^3 \qquad (4)$$

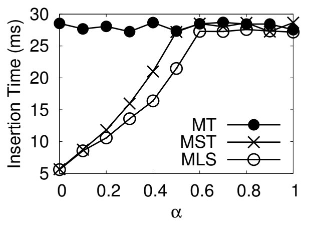
This shows a cubic accuracy loss in terms of $\beta$, e.g., if $\beta = 0.3$, we have 2.7% loss in accuracy for 30% storage saving.



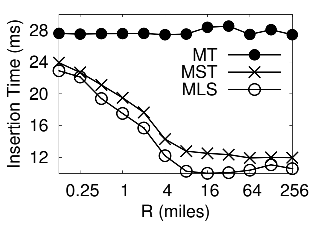
(a) Varying arrival rate



(b) Varying $k$



(c) Varying $\alpha$



(d) Varying $R$

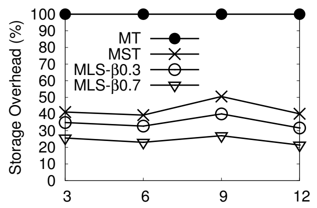
Fig. 3. Index insertion time.

## VIII. Experimental Results

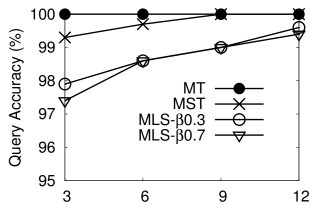
This section provides experimental evaluation of *Mercury* based on an actual system implementation. With lack of direct competitors, see Section II, we evaluate various versions of *Mercury* to show the effectiveness of its components. All experiments are based on a real deployment of *Mercury* and using a real-time feed of US tweets (via access to Twitter Firehose) and actual locations of web search queries from Bing. We have stored 340+ million tweets and one million Bing search queries in files. Then, we have read and timestamped them to simulate an incoming stream of real microblogs and queries. Unless mentioned otherwise, the default value of $k$ is 100, microblog arrival rate $\lambda$ is 1000 microblogs/second, range $R$ is a 30 miles, $T$ is 6 hours, $\alpha$ is 0.2, $\beta$ is 0.3, and cell capacity is 150 microblogs. The default values of cell capacity, $\alpha$, and $\beta$ are selected experimentally and show to work best for query performance and result significance, respectively, while default $\lambda$ is the effective rate of US geotagged tweets. As microblogs are so timely that Twitter gives only the most recent tweets (i.e., $\alpha$=0), we set $\alpha$ to 0.2 as the temporal dimension is more important than spatial dimension. All results are collected in the steady state, i.e., after running the system for at least $T$ time units. We use an Intel Core i7 machine with CPU 3.40GHZ and 64GB RAM. Our measures of performance include insertion time, storage savings, query accuracy, and response time. The rest of this section evaluates index maintenance (Section VIII-A), load shedding (Section VIII-B), and query processing (Section VIII-C).

### A. Index Maintenance

Figure 3 gives the performance of *Mercury* insertion time, which entails piggypacked/bulk deletion, cell splitting, and cell merging, if needed. We compare three alternatives for *Mercury* index: (a) storing all microblogs of last $T$ time units (denoted as *MT*), (b) using the *index size tuning* module (Section VI), denoted as *MST*, and (c) using the *load shedding* module (Section VII), denoted as *MLS*. Figure 3(a) gives the performance when varying the arrival rate from 250 to 64,000 micoblogs/second. The figure presents the time of *Mercury* bulk insertion every 1 second, i.e., all microblogs that have arrived in the last second are inserted in bulk. Both *MLS* and *MST* perform much better than *MT*. While *MT* is able to digest only 32K micorblogs/second, *MLS* and *MST* are able to bulk insert 64K microblogs in less than 0.5 second. For the current Twitter rate (4,600 microblogs/second), *MLS* and *MST* are able to insert all the incoming 4,600 items in less than 34 msec.



(a) Storage Overhead



(b) Query Accuracy

Fig. 4. Effect of $T$ on storage vs. accuracy

This is mainly because both *MLS* and *MST* reduce the index size, and hence less cells are visited in bulk insertion.

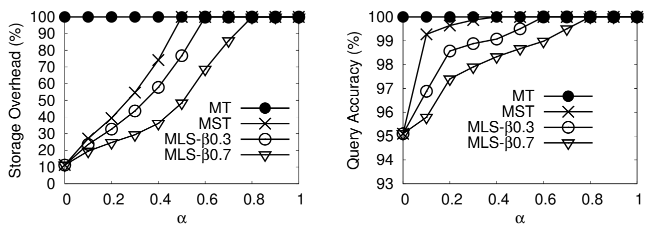
Figure 3(b) gives the same experiment with varying $k$ from 10 to 100. The performance is stable for all alternatives with 28, 12, and 10 msec for *MT*, *MST*, and *MLS*, respectively. This shows the practical dominance of the first term ($\frac{\alpha}{1-\alpha}T$) in the equations of $T_c$ and $T_{c,\beta}$ over the term that contains $k$.

Figure 3(c) gives the bulk insertion time with varying $\alpha$ from 0 to 1. *MT* has a stable performance as $\alpha$ does not affect its storage. On the other side, lower values of $\alpha$ strongly favors *MST* and *MLS* as it plays a major role in deciding the values of $T_c$ and $T_{c,\beta}$, and hence the storage consumed. With the increasing $\alpha$, *MST* and *MLS* degenerate to be equivalent to *MT* at $\alpha$ = 0.5 and 0.6, respectively. Figure 3(d) shows that increasing $R$ significantly enhances the performance of both *MST* and *MLS*. The reason is that increasing $R$ gives more search space to look for the result, hence, no need to look much back in time which needs less storage.

### B. Load Shedding

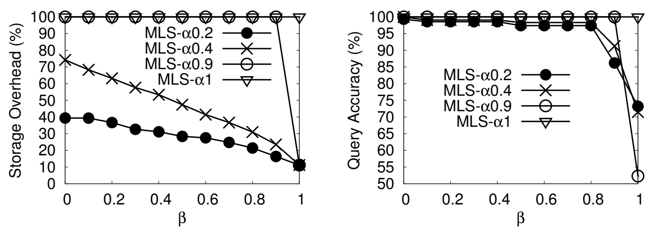
In this section, we evaluate the impact of load shedding on storage savings and query accuracy. Storing all the 340+ million tweets consumes more than 8GB from memory just to store tweet id and latitude/longitude coordinates while encounters much higher storage overhead, ~56GB, with text and .NET framework overhead, yet it guarantees 100% query accuracy. We show the effect of varying $T$, $\alpha$, and $\beta$ on the storage overhead and accuracy of both *index size tuning* (*MST*) and *load shedding* (*MLS*).

Figure 4 gives the storage overhead ratio and query accuracy of *MST* and *MLS* while varying $T$ from 3 to 12 hours. We depict two curves for *MLS* that correspond to two values of $\beta$ as 0.3 and 0.7, while *MT* is depicted in the Figure as 100% storage overhead and query accuracy. For all values of $T$, *MLS* with $\beta$ = 0.3 consumes only 35% of the storage required by *MT* (Figure 4(a)). This takes place with a very high accuracy of 98% to 99.5% (Figure 4(b)). Similarly, *MLS* with $\beta$ = 0.7



(a) Storage Overhead (b) Query Accuracy

Fig. 5. Effect of $\alpha$ on storage vs. accuracy



(a) Storage Overhead (b) Query Accuracy

Fig. 6. Effect of $\beta$ on storage vs. accuracy

consumes only 25% of the storage with accuracy 97.5% to 99.3%. *MST* consumes 40% of the storage with almost 100% accuracy. These practical results confirm our earlier analysis in Section VII, where we anticipated that a linear reduction of storage ($\beta$) will result in a cubic loss of accuracy ($\beta^3$). As shown in the figure, *MST* gives less than 100% accuracy for $T$ = 3, where we have only 99.2%. Theoretically, *MST* should be able to provide 100% accuracy regardless of the parameters values. However, the theoretical model assumed spatial uniformity of microblogs within individual pyramid cells, which is not 100% true. This leads to missing few microblogs and hence a slight drop in *MST* accuracy.

Figure 5 gives the effect of varying $\alpha$ from 0 to 1 on the storage overhead and query accuracy of *MST* and *MLS* with $\beta$ = 0.3 and 0.7. Figure 5(a) shows that increasing $\alpha$ leads to increasing the storage overhead of both *MST* and *MLS*. While *MST* storage degenerates to be as *MT* when $\alpha \geq 0.5$, *MLS* still keeps its storage gain till $\alpha$ approaches 0.6 and 0.8, respectively for the two values of $\beta$. This shows that *load shedding* still can find reasonable storage savings even for large values of $\alpha$. Meanwhile, Figure 5(b) shows that all alternatives have query accuracy of more than 95%. The worst case takes place when $\alpha$ = 0, in which the value of $\beta$ does not play any role. For all alternatives, once we have the same storage overhead as *MT*, we obtain 100% accuracy.

Figure 6 focuses only on the load shedding (*MLS*), where it studies the effect of varying $\beta$ from 0 to 1 on the storage overhead and query accuracy. Per Figure 6(a), the storage overhead saving is linear in $\beta$ with line slope that depends on value of $\alpha$. The lower $\alpha$, the lower $T_{c,\beta}$, and hence more storage savings can be achieved. The extreme case $\alpha$ = 1 makes *MLS* runs exactly as *MT*, while with $\alpha$ = 0.2, we can achieve from 60% to 90% storage saving.

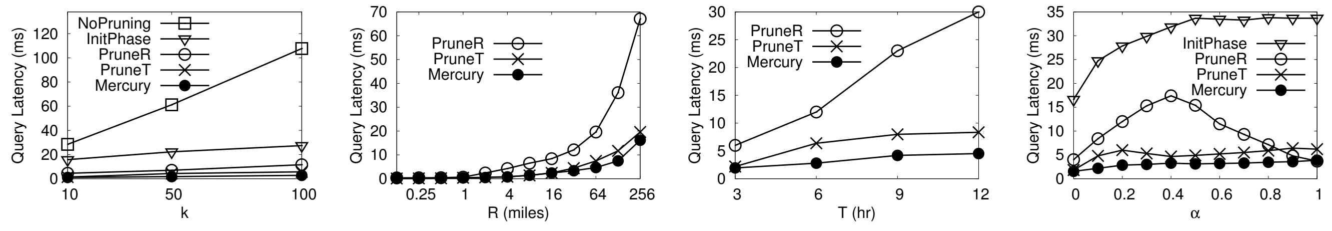
Figure 6(b) shows that the query accuracy is directly proportional with the storage overhead for different values of $\beta$. With $\alpha$ = 1, $T_{c,\beta}$ = $T$ and hence the accuracy is 100% regardless of $\beta$. The accuracy shown is much higher than the theoretical expectations and has a practical lower bound of 52% accuracy at $\beta$ = 1 with $\alpha$ = 0.9 which has a storage saving of 90% (Figure 6(a)). This happens because our theoretical model uses very conservative assumptions on spatial and temporal maximum distances. Thus, even with significant load shedding, in-memory microblogs can provide good quality query answer, which shows the applicability of *Mercury* in memory-constrained environments.

## C. Query Evaluation

In this section, we show the effect of the query processing techniques, where we contrast *Mercury* against: (a) *NoPruning*, where all microblogs within $R$ and $T$ are processed, (b) *InitPhase*, where only the *initialization* phase of *Mecury* is employed, (c) *PruneR*, where only spatial pruning is employed, and (d) *PruneT*, where only temporal pruning is employed. Figure 7(a) gives the effect of varying $k$ from 10 to 100 on the query latency. It is clear that variants of *Mercury* give order of magnitude performance over *NoPruning*, which shows the effectiveness of the employed strategies. With this, we are not showing any further result to *NoPruning* as it is clearly non-competitive. Also, *InitPhase* gives much worse performance than *Mercury*, which shows the strong effect of the *pruning* phase. Finally, it is important to note that with $k = 100$, *Mercury* gives a query latency of only 3 msec.

Figures 7(b) and 7(c) give the effect of varying $R$ and $T$, respectively, on the query latency for *Mercury*, *PruneR*, and *PruneT*. Both figures show that *Mercury* takes advantage of both spatial and temporal pruning to get to its query latency of up to 4 msec for 12 hours and 64 miles ranges. Increasing $R$ and $T$ increases the query latency of all alternatives, however, *Mercury* still performs much better when using its two pruning techniques. It is also clear that *PruneT* achieves better performance than *PruneR*, i.e., temporal pruning is more effective than spatial pruning, which is a direct result of the default value of $\alpha$=0.2 that favors the temporal dimension.

Figure 7(d) gives the effect of varying $\alpha$ from 0 to 1 on the query latency, where *Mercury* consistently has a query latency under 4 msec, while *InitPhase* has an unacceptable performance that varies from 15 to 35 msec. This shows the strong effect of the *pruning* phase in *Mercury*. Meanwhile, with increasing $\alpha$, the temporal boundary of *PruneR* increases and hence it visits more microblogs inside each cell. For low values of $\alpha$ (< 0.5), the number of additional microblogs visited due to increasing the temporal boundary is more than the number of microblogs that are pruned based on spatial pruning. This increases the overall latency of *PruneR*. When $\alpha \geq 0.5$, the number of microblogs that *PruneR* prunes based on the spatial pruning becomes larger than the additional visited microblogs due to enlarging the temporal horizon. Hence, *PruneR* latency becomes quickly better and beats *PruneT* at $\alpha > 0.8$. This means that for all values of $\alpha < 0.8$, temporal pruning is still more effective than spatial pruning. *PruneT* has a stable performance with respect to varying $\alpha$. In



(a) Varying $k$ (b) Varying $R$ (c) Varying $T$ (d) Varying $\alpha$

Fig. 7. Average query latency.

all cases, *Mercury* takes advantage of both spatial and temporal pruning to achieve its overall performance of around 4 msec.

## IX. Conclusion

We have presented *Mercury*; a system for real-time support of spatio-temporal queries on microblogs, where users request a set of recent $k$ microblog near their locations. *Mercury* works under a challenging memory-constrained environment, where microblogs arrive with very high arrival rates. *Mercury* employs efficient in-memory indexing to support up to 64K microblogs/second and spatio-temporal pruning techniques to provide real-time query response of 4 msec.

.

## References

[1] H. Abdelhaq, C. Sengstock, and M. Gertz. EvenTweet: Online Localized Event Detection from Twitter. In *VLDB*, 2013.
[2] W. G. Aref and H. Samet. Efficient Processing of Window Queries in the Pyramid Data Structure. In *PODS*, 1990.
[3] A. Bermingham and A. F. Smeaton. Classifying Sentiment in Microblogs: Is Brevity an Advantage? In *CIKM*, 2010.
[4] After Boston Explosions, People Rush to Twitter for Breaking News. http://www.latimes.com/business/technology/la-fi-tn-after-boston-explosions-people-rush-to-twitter-for-breaking-news-20130415,0,3729783.story, 2013.
[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. C. Cao, J. She, Y. Tong, and L. Chen. Whom to Ask? Jury Selection for Decision Making Tasks on Micro-blog Services. *PVLDB*, 2012.
[7] C. Chen, F. Li, B. C. Ooi, and S. Wu. TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets. In *SIGMOD*, 2011.
[8] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial Keyword Query Processing: An Experimental Evaluation. In *VLDB*, 2013.
[9] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient Query Processing in Geographic Web Search Engines. In *SIGMOD*, 2006.
[10] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *PVLDB*, 2(1), 2009.
[11] A. Dong, R. Zhang, P. Kolari, J. Bai, F. Diaz, Y. Chang, Z. Zheng, and H. Zha. Time is of the essence: Improving recency ranking using twitter data. In *WWW*, 2010.
[12] Facebook Statistics. http://newsroom.fb.com/Key-Facts, 2012.
[13] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *ACTA*, 4(1), 1974.
[14] J. Hannon, M. Bennett, and B. Smyth. Recommending twitter users to follow using content and collaborative filtering approaches. In *RecSys*, 2010.
[15] L. Hong, A. Ahmed, S. Gurumurthy, A. J. Smola, and K. Tsioutsiouliklis. Discovering Geographical Topics In The Twitter Stream. In *WWW*, 2012.
[16] S. J. Kazemitabar, U. Demiryurek, M. H. Ali, A. Akdogan, and C. Shahabi. Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight. *PVLDB*, 3(2), 2010.
[17] M. Koubarakis, T. Sellis, A. U. Frank, S. Grumbach, R. H. Gting, C. S. Jensen, and N. Lorentzos. *Spatio-Temporal Databases: The CHOROCHRONOS Approach*. Springer, 2003.
[18] G. Lee, J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy. The Unified Logging Infrastructure for Data Analytics at Twitter. *PVLDB*, 5(12), 2012.
[19] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang. TEDAS: A Twitter-based Event Detection and Analysis System. In *ICDE*, 2012.
[20] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-Tree: An Efficient Index for Geographic Document Search. *TKDE*, 23(4), 2011.
[21] J. Lin and A. Kolcz. Large-scale machine learning at twitter. In *SIGMOD*, 2012.
[22] J. Lin and G. Mishne. A Study of ”Churn” in Tweets and Real-Time Search Queries. In *ICWSM*, 2012.
[23] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing. Discovering Spatio-temporal Causal Interactions in Traffic Data Streams. In *KDD*, 2011.
[24] 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.
[25] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Twitinfo: Aggregating and Visualizing Microblogs for Event Exploration. In *CHI*, 2011.
[26] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Processing and Visualizing the Data in Tweets. *SIGMOD Record*, 40(4), 2012.
[27] M. Mathioudakis and N. Koudas. TwitterMonitor: Trend Detection over the Twitter Stream. In *SIGMOD*, 2010.
[28] E. Meij, W. Weerkamp, and M. de Rijke. Adding semantics to microblog posts. In *WSDM*, 2012.
[29] E. Meskovic, Z. Galic, and M. Baranovic. Managing Moving Objects in Spatio-temporal Data Streams. In *MDM*, 2011.
[30] G. Mishne and J. Lin. Twanchor Text: A Preliminary Study of the Value of Tweets as Anchor Text. In *SIGIR*, 2012.
[31] M. F. Mokbel and W. G. Aref. SOLE: Scalable On-Line Execution of Continuous Queries on Spatio-temporal Data Streams. *VLDB Journal*, 17(5), 2008.
[32] O. Phelan, K. McCarthy, and B. Smyth. Using twitter to recommend real-time topical news. In *RecSys*, 2009.
[33] D. Ramage, S. T. Dumais, and D. J. Liebling. Characterizing Microblogs with Topic Models. In *ICWSM*, 2010.
[34] T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake shakes twitter users: Real-time event detection by social sensors. In *WWW*, 2010.
[35] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. TwitterStand: News in Tweets. In *GIS*, 2009.
[36] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.
[37] V. K. Singh, M. Gao, and R. Jain. Situation Detection and Control using Spatio-temporal Analysis of Microblogs. In *WWW*, 2010.
[38] Twitter Statistics. http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/, 2013.
[39] I. Uysal and W. B. Croft. User Oriented Tweet Ranking: A Filtering Approach to Microblogs. In *CIKM*, 2011.
[40] K. Watanabe, M. Ochi, M. Okabe, and R. Onai. Jasmine: A Real-time Local-event Detection System based on Geolocation Information Propagated to Microblogs. In *CIKM*, 2011.
[41] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint Top-K Spatial Keyword Query Processing. *TKDE*, 24(10), 2012.
[42] L. Wu, W. Lin, X. Xiao, and Y. Xu. LSII: An Indexing Structure for Exact Real-Time Search on Microblogs. In *ICDE*, 2013.
[43] J. Yao, B. Cui, Z. Xue, and Q. Liu. Provenance-based Indexing Support in Micro-blog Platforms. In *ICDE*, 2012.
[44] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword Search in Spatial Databases: Towards Searching by Document. In *ICDE*, 2009.
[45] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal and Spatio-temporal Aggregations over Data Streams using Multiple Time Granularities. *Information Systems*, 28(1-2), 2003.