Venus: Scalable Real-time Spatial Queries on Microblogs with Adaptive Load Shedding

Amr Magdy*, Mohamed F. Mokbel*, Sameh Elnikety§, Suman Nath§ and Yuxiong He§
*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455
§Microsoft Research, Redmond, WA 98052-6399

Abstract—Microblogging services have become among the most popular services on the web in the last few years. This led to significant increase in data size, speed, and applications. This paper presents *Venus*; a system that supports real-time spatial queries on microblogs. *Venus* supports its queries on a spatial boundary R and a temporal boundary T, from which only the top-k microblogs are returned in the query answer based on a spatio-temporal ranking function. Supporting such queries requires *Venus* to digest hundreds of millions of real-time microblogs in main-memory with high rates, yet, it provides low query responses and efficient memory utilization. To this end, *Venus* employs: (1) an efficient in-memory spatio-temporal index that digests high rates of incoming microblogs in real time, (2) a scalable query processor that prune the search space, R and T, effectively to provide low query latency on millions of items in real time, and (3) a group of memory optimization techniques that provide system administrators with different options to save significant memory resources while keeping the query accuracy almost perfect. *Venus* memory optimization techniques make use of the local arrival rates of microblogs to smartly shed microblogs that are old enough not to contribute to any query answer. In addition, *Venus* can adaptively, in real time, adjust its load shedding based on both the spatial distribution and the parameters of incoming query loads. All *Venus* components can accommodate different spatial and temporal ranking functions that are able to capture the importance of each dimension differently depending on the applications requirements. Extensive experimental results based on real Twitter data and actual locations of Bing search queries show that *Venus* supports high arrival rates of up to 64K microblogs/second and average query latency of 4 msec.

Index Terms—Microblogs, Spatial, Location, Temporal, Performance, Efficiency, Scalability, Memory Optimization, Social.

1 Introduction

Social media websites have grabbed big attention in the last decade due to its growing popularity and unprecedentedly large user base. The new wave of user-interactive microblogging services, e.g., tweets, comments on Facebook or news websites, or Foursquare check-in's, has become the clear frontrunner in the social media race with the largest number of users ever and highest users activity in consistent rates. For example, Twitter has 288+ Million active users who generate 500+ Million daily tweets [1], [2], while Facebook has 1.35+ Billion users who post 3.2+ Billion daily comments [3]. Motivated by the advances in wireless communication and the popularity of GPS-equipped mobile devices, microblogs service providers have enabled users to attach location information with their posts. Thus, 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, per user permission, and Foursquare features are all around the location information and the whereabouts of its users. Consequently, a plethora of location information is currently available in microblogs.

We exploit of the availability of location information in microblogs to support spatio-temporal search queries where users are able to browse recent microblogs near their locations in real time. Users of our proposed 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 ongoing activities 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 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.

In this paper, we present *Venus*: a system for real-time support of spatio-temporal queries on microblogs. Due to large numbers of microblogs, *Venus* limits its query answer to only k *most relevant* microblogs so that it can be easily navigated by human users. Microblog relevance is assessed based on a ranking function F that combines the time recency and the spatial proximity to the querying user. In addition, *Venus* exploits the fact that the more recent microblogs data, the more important for real-time queries to bound its search space to include only those microblogs that have arrived during the last T time units within a spatial query range R. Thus, *Venus* users can post queries to *get a set of top-k relevant microblogs, ranked by a spatio-temporal function F, that are posted within a spatial range R in the last T time units*.

To support its queries, *Venus* faces two main challenges: high arrival rates of real-time microblogs and the need for low query response while searching millions of data items. Both challenges

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

call for relying on *only* main-memory indexing to digest and query real-time microblogs. Hence, *Venus* employs an in-memory partial pyramid index [5], 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.

Venus can employ different ranking functions to be able to serve requirements of different applications. Based on a certain ranking function, the different *Venus* components are optimized for preset default values of k, R, T, and α. 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 relevant search results, while Twitter gives the most recent k (k=20) 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.

As main-memory is a scarce resource, relying on main-memory indexing requires efficient management of the available memory resources. Although storing and indexing all incoming microblogs from the last default T time units ensures that all incoming queries will be satisfied from in-memory contents, which may require very large memory resources, which can be prohibitively expensive. Hence, we propose effective memory optimization techniques: (1) We analytically 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. For example, 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 then maintain only the items that may appear in user queries and delete items that are old enough to be dominated by others. (2) For tight memory configurations, we provide a parametrized *load shedding* technique that trades significant reduction in the memory footprint (up to 75% less storage) for a reasonable loss in query accuracy (up to 8% accuracy loss). The idea is to expel from memory a set of victim microblogs that are less likely to contribute to a query answer. (3) Building on our parametrized load shedding technique, we develop two parameter-free *adaptive load shedding* techniques that give the option to automatically tune the load shedding in different spatial regions adaptively with the incoming query loads. These techniques catch the spatial distribution of the incoming queries as well as the spatial access patterns of the stored microblogs so that they bring the storage overhead to its minimal levels (up to 80% less storage) while allow to answer queries with almost perfect accuracy (more than 99% in all cases).

Venus is the successor of *Mercury* [6], from which it is distinguished by: (1) Optimizing its index, query processor, and memory optimization techniques for different ranking functions, that rank its top-k answers, so that it is flexible to serve a wide variety of applications requirements. (2) Providing two parameter-free adaptive load shedding techniques that exploit the spatial distribution of incoming queries and data to automatically tune the load shedding adaptively so that they minimize the memory footprint significantly without (almost) compromising the query accuracy. (3) Providing experimental study that compares the performance of different system components, in terms of running time, storage overhead, and query accuracy, with the most two recognized ranking functions in the literature that satisfy most of the practical applications requirements.

We evaluate the system experimentally using a real-time feed of US tweets (via access to Twitter Firehose archive) and actual locations of Bing web search queries. Our measurements show that *Venus* 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%. The contributions of this paper are summarized as follows:

1) We provide a crisp definition for spatio-temporal search queries over microblogs (Section 3).
2) We propose efficient spatio-temporal indexing/expelling techniques that are capable of inserting/deleting microblogs with high rates (Section 4).
3) We introduce an efficient spatio-temporal query processor that minimizes the number of visited microblogs to return the final answer (Section 5).
4) 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 6).
5) We introduce a *load shedding* technique that trades significant reduction in memory footprint for a slight decrease in query accuracy (Section 7).
6) We introduce two *adaptive load shedding* techniques that exploit the spatial distribution of incoming queries and data to automatically tune the load shedding adaptively (Section 8).
7) We provide experimental evidence, based on real system prototype, microblogs, and queries, showing that *Venus* is scalable and accurate with minimal memory consumption (Section 9).

2 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 [7] and machine learning techniques [8] to indexing [9], [10], [11], [12] and designing a SQL-like query language interface [13]. In addition, several efforts have focused on analyzing microblog data, which include semantic and sentiment analysis [14], [15], [16], decision making [17], news extraction [18], event and trend detection [19], [20], [21], [22], [23], understanding the characteristics of microblog posts and search queries [24], [25], microblogs ranking [26], [27], and recommending users to follow or news to read [28], [29]. Meanwhile, recent work [18], [30] exploited microblogs contents to extract location information that is used to visualize microblog posts on a map [31], [32] and model the relationship between user interests, locations, and topics [33].

With such rich work in microblogs, the existing work on real-time indexing and querying of microblogs locations [34], [35] mostly address variations of aggregate queries, e.g., frequent keywords, that are posted on different regions. However, up to our knowledge, there is no existing academic work that support real-time indexing and querying to support non-aggregate spatial queries on individual microblogs locations; which is the main focus of this paper. Also, although Twitter search allows to embed spatial parameters in the query, they do not reveal the details

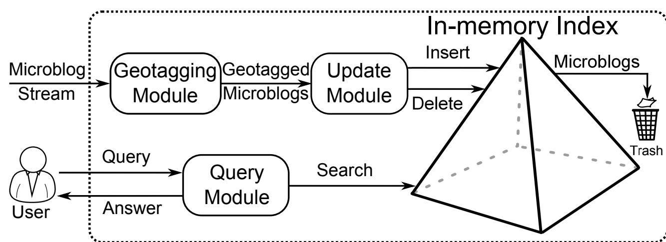


Fig. 1. Venus system architecture.

of how they are supporting their spatial search and hence we have no insights about their techniques. Generally, the two most related topics to our work are *microblog search queries* and *spatio-temporal streams*.

Microblog Search Queries. Real-time search on microblogs spans keyword search [9], [10], [11], [12] and location-aware search [6], [34], [35]. The difference of one technique over the other is mainly in the query type, accuracy, ranking function, and memory management. Other than *Mercury* [6], the predecessor of *Venus*, the existing location-aware search on microblogs mostly address aggregate queries. None of these work have addressed retrieving individual microblogs in real-time based on their location information. On the other hand, spatial keyword search is well studied on web documents and web spatial objects [36], [37], [38], [39], [40]. However, they use offline disk-based data partitioning indexing, which cannot scale to support the dynamic nature and arrival rates of microblogs [6], [9], [35], [41].

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 [42], [43], [44], [45], [46]. 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.

Venus 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 meantime, *Venus* 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 data that arrives later. Finally, *Venus* 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.

3 System Overview

This section gives an overview of *Venus* system architecture, supported queries, and ranking functions.

3.1 System Architecture

Figure 1 gives *Venus* 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 represents 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 two 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. Details of index operations and index size tuning are discussed in Sections 4, 6, 7, and 8.

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 5.

3.2 Supported Queries

Venus 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_α that employs a parameter α 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_α. Formally, our query is defined as follows:

Definition: *Given k, R, T, and F_α, 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_α.*

Our query definition is a natural extension to traditional spatial range and k-nearest-neighbor queries, used extensively in spatial and spatio-temporal databases [47], [48]. 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_α is provided. Similarly, a k-nearest-neighbor query finds the *closest* k items to the user location. As the relevance

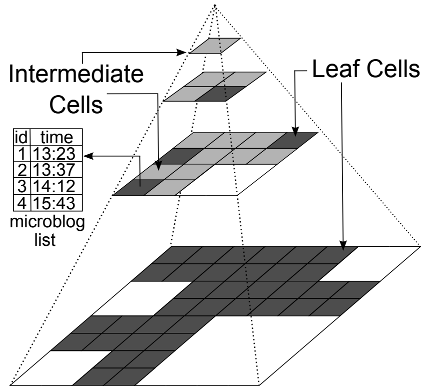


Fig. 2. Main memory pyramid index structure in Venus.

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_α to score each microblog within our spatial and temporal boundaries.

Upon initialization, a system administrator sets default values for parameters k, R, T, and α. Users may still change the values of the default parameters, yet a query may have worse performance if the new parameters present larger search space than the default ones. Setting default parameter values is adopted by major services, like Bing and Twitter, which return the top-k most related search results for a preset value of k. However, user can get more results upon request. Our system also can adapt for dynamically changing the preset values in the middle of operations as elaborated in Section 6.

3.3 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 $0 \leq \alpha \leq 1$, *Venus* employs the following ranking function $F_\alpha(u, M)$ that combines generic spatial and temporal scores in a weighted summation to give the relevance score of M to u, where lower scores are favored:

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

D_s and D_t are the spatial and temporal distances, respectively. In *Venus*, we use Euclidean distance and absolute timestamps difference, though, any other *monotonic* distance functions can be used without changing the presented techniques. The largest possible value takes place when M is posted exactly T time units ago and on the boundary of region R. α=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. α=0 gives 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.

TemporalScore and *SpatialScore* can be any functions that are: (1) monotonic, (2) have an inverse function with respect to the spatial and temporal distance, and (3) normalized in the same range of values where smaller values indicate more relevant microblogs. The inverse function is used in pruning search space and optimizing memory footprint as we discuss in the following sections. The normalization within the same range is not a correctness condition. However, as the scoring functions determine the decay pattern of the microblog relevance over time and space, normalization ensures that both spatial and temporal dimensions have the same effect on the final relevance score. In this paper, we employ the most two recognized scoring functions in the literature: the linear function (see [6]) and the exponential function (see [11]) to show the adaptivity of different *Venus* components with the different functions. However, other scoring functions, that satisfy the above conditions, can be adapted using exactly the same procedure that are explained in each component throughout the paper. The scores are defined by the following equations:

The linear scoring functions

$TemporalScore(D_t(M.time, NOW)) =$

$$\begin{cases} \frac{D_t(M.time, NOW)}{T} & D_t(M.time, NOW) \leq T \\ N/A & D_t(M.time, NOW) > T \end{cases}$$

$SpatialScore(D_s(M.loc, u.loc)) =$

$$\begin{cases} \frac{D_s(M.loc, u.loc)}{R} & D_s(M.loc, u.loc) \leq R \\ N/A & D_s(M.loc, u.loc) > R \end{cases}$$

Both functions are bounded in the range $[0, 1]$.

The exponential scoring functions

$TemporalScore(D_t(M.time, NOW)) =$

$$\begin{cases} e^{w \times \frac{D_t(M.time, NOW)}{T}} & D_t(M.time, NOW) \leq T, w > 0 \\ N/A & D_t(M.time, NOW) > T \end{cases}$$

$SpatialScore(D_s(M.loc, u.loc)) =$

$$\begin{cases} e^{w \times \frac{D_s(M.loc, u.loc)}{R}} & D_s(M.loc, u.loc) \leq R, w > 0 \\ N/A & D_s(M.loc, u.loc) > R \end{cases}$$

Both functions must have the same value of w to be bounded in the same range $[1, e^w]$.

4 SPATIO-TEMPORAL INDEXING

We have two main objectives to satisfy in *Venus* indexing. First, the employed index has to digest high arrival rates of incoming microblogs. Second, the employed index should expel (delete) microblogs from its contents with similar rates 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 [49] and pyramid [5]) 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, *Venus* employs a partial pyramid structure [5] (Figure 2) that consists of L levels. For a given level l, the whole space is partitioned into 4^l 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. The main reason to use a pyramid data structure is to handle the skewed spatial distribution of microblogs

efficiently, so that dense areas are split into deeper levels while sparse areas span only few levels. To elaborate, in a pyramid index, a large leaf cell represents large space, when the density is low. When the density is high, the depth increases so that a cell covers a much smaller area. If we use a simple spatial grid, for example, it is not clear what should be the grid size, and it will never be right, too small for some regions and too large for others. 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 *Venus*, existing pyramid index structures [5] 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 4.1) and a *speculative cell splitting* module that avoids skewed cell splitting (Section 4.2). 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 4.3) and a *lazy cell merging* module that decides on when to merge a set of cells together to minimize the system overhead (Section 4.4).

4.1 Bulk Insertion

Inserting a microblog M (with a point location) in the pyramid structure can be done traditionally [5] 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 insert all microblogs of B in 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 the average arrival rate in Twitter is 5.5K+ microblogs/second, setting $t = 2$ means that each two seconds, we will insert 11,000 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.

4.2 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 overloaded cells at the lowest level.

To avoid long skewed tree branches, we employ a *speculative cell splitting* module, where a 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 cell a set of split bits (*SplitBits*) as a four-bits variable; one bit 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 insertion operation, yet, C was not split as all its microblogs belong to the same quadrant (the one has its

bit set in *SplitBits*). 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.

Using *SplitBits* significantly reduces insertion and query processing time as: (a) we avoid dangling skewed tree branches, and (b) we avoid frequent expensive checking 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 meantime, maintaining the integrity of *SplitBits* comes with very little overhead. First, when cell is under capacity, we do not read or set the value of *SplitBits*. Second, deleting entries from the cell has no effect on its *SplitBits*, unless it becomes empty, where we reset all bits.

4.3 Bulk Deletion

As we have finite memory, *Venus* 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 [5] 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 *Venus* 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 useless microblogs that have not been deleted, yet, due to lack of 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 $C.M_List$. 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 either in the next insertion or in the next periodic cleanup.

Deleted microblogs are moved from our in-memory index structure to another index structure, 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.

4.4 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 redundant split/merge operations (which is measured practically to be 90% of the whole split/merge operations) that leads to a significant reduction in index update overhead.

5 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_α 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, *Venus* 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 5.2) finds an initial set of k microblogs that form a basis of the final answer. The *pruning* phase (Section 5.3) keeps on tightening the initial boundaries R and T to enhance the initial result and reach to the final answer.

5.1 Query Data Structure

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

Priority queue of cells H: A priority queue of all index cells that overlap with query spatial boundary R. An entry 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:

$$
\begin{aligned}
&BestScore(u, C) = \alpha \times SpatialScore(D_s(u.loc, C)) \\
&+(1-\alpha) \times TemporalScore(C.M_List[index].time, NOW)
\end{aligned}
$$

Where $D_s(u.loc, C)$ 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.

5.2 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 5.1. Then, we take the top entry's cell 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 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 entry's cell 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 entry has its cell 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 ∞.

5.3 The Pruning Phase

The *pruning* phase takes the *AnswerSet* from the *initialization* phase and enhances its 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 minimum acceptable score for a microblog to be included in *AnswerSet*, which corresponds to the current kth score in *AnswerSet*. Assume the linear scoring functions (as in Section 3.3), for a microblog M to be included in *AnswerSet*, M has to have a lower score than *MIN*, i.e.,:

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

Algorithm 1 Query Processor

1: **Function** Query Processor (u, k, T, R, α)
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* $\leftarrow \alpha$ $SpatialScore(D_s(u.loc, C))$ + (1-α) $TemporalScore(D_t(C.M_List[1].time, NOW))$
5:   Insert (C, 1, *BestScore*) into H
6: **end for**
7: $TopH \leftarrow$ Get (and remove) first entry in H
8: **while** $TopH$ is not NULL and $TopH.score < MIN$ **do**
9:   *Score* $\leftarrow$ *TopH.score*; $M \leftarrow$ *TopH.C.M_List*[*TopH.index*]
10:   *NextScore* $\leftarrow$ 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* $\leftarrow \alpha$ $SpatialScore(D_s(u.loc, M.loc))$ + (1-α) $TemporalScore(D_t(M.time, NOW))$
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:           $R' \leftarrow$ Min(R', $PruneRatio_s \times R'$)
20:           $T' \leftarrow$ Min(T', $PruneRatio_t \times T'$)
21:         **end if**
22:       **end if**
23:     **end if**
24:     $M \leftarrow$ Next microblog in $TopH.C.M_List$
25:     **if** $M.time$ outside T' **then** $M \leftarrow$ NULL
26:   **end while**
27:   **if** $M \neq$ NULL **then** Insert (C, index(M), *BestScore*) in H
28:   $TopH \leftarrow$ Get (and remove) first entry in H
29: **end while**
30: **Return** $AnswerSet$

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: $\alpha\frac{D_s(u.loc,M.loc)}{R} < MIN$, i.e., M has to be within distance $\frac{MIN}{\alpha}R$ from the user. We call the value $\frac{MIN}{\alpha}$ the spatial pruning ratio, for short $PruneRatio_s$. Hence, we tighten our spatial boundary to $R' = Min(R, PruneRatio_s \times R)$. (2) *Temporal boundary tightening.* Assume that M has the best possible spatial score, i.e., $D_s(u.loc, M.loc)$ = 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. We call the value $\frac{MIN}{1-\alpha}$ the temporal pruning ratio, for short $PruneRatio_t$. Hence, we tighten our temporal boundary to $T' = Min(T, PruneRatio_t \times T)$. Following the same steps, we can derive the values $PruneRatio_s$ and $PruneRatio_t$ for the exponential scoring functions to be: $PruneRatio_s = \frac{1}{w} \times ln(\frac{MIN-(1-\alpha)}{\alpha})$ and $PruneRatio_t = \frac{1}{w} \times ln(\frac{MIN-\alpha}{1-\alpha})$.

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 kth score in *AnswerSet*. Then, we check if we can apply spatial and/or temporal pruning based on the values of *MIN* and α 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 entry 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.

6 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 its answer entirely 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 answer quality (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) α 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, 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. Assume the microblog arrival rate for a cell C is λ_c and we use the linear scoring functions. Then, T_c is given by the following equation:

For linear scoring functions

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

The detailed derivation for T_c value can be revised in either [6] or Appendix A. Following the same steps, we can derive T_c using the exponential scoring function to be given by:

For exponential scoring functions

$$T_c = Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^w - 1) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_c T}}]\right) \quad (2)$$

We discuss next the impact of the *index size tuning* module on various *Venus* components. Equations 1 and 2 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 $T_c \leq T$, and so any older microblog can be safely shed without affecting the query accuracy. Therefore, we save memory space by storing fewer microblogs. We next discuss the impact of employing the T_c values on *Venus* components.

Index Structure. Each pyramid cell C will keep track of two additional variables: (1) λ_c; the arrival rate of microblogs in C, which is continuously updated on arrival of new microblogs, and (2) T_c; the temporal boundary in cell C computed from Equations 1 or 2, and updated with every update of λ_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 λ_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 global value T for all cells.

Index Maintenance. When a cell C splits into four quadrant cells, the value of λ_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 *Venus* 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 λ_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.

It also worth mentioning that optimizing the index for preset default parameters values does not limit *Venus* from adapting changes to these values in the middle of operations. In case a system administrator change the default values in the middle of operations, the index contents will be adapted for the new values in the following data insertion cycles (based on the new computed values of T_c). So, all what is needed is to plan changing the values ahead if the new queries require more data to fulfill their answers, i.e., if they lead into increasing T_c values.

7 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, *Venus* 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 that. In particular, Equations 1 and 2 consider the very conservative case that *every* stored microblog M may have a query that comes exactly at $M.loc$, i.e., $D_s(M.loc, u.loc) = 0$. The *load shedding* module relaxes this assumption and assumes that queries are posted βR miles away from M, i.e., $D_s(M.loc, u.loc) = \beta R$, where $0 \leq \beta \leq 1$. Using this relaxed assumption, we can revise the value of time horizon per cell to be:

For linear scoring functions

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

For exponential scoring functions

$$T_{c,\beta} = Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^w - e^{w\beta}) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \lambda_c T}}]\right) \quad (4)$$

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. The detailed derivation of $T_{c,\beta}$ can be revised either in [6] or Appendix B.1. Per Equations 3 and 4 , $T_{c,\beta}$ gives a

tighter temporal coverage for each cell as $T_{c,\beta} \leq T_c$. β acts as a tuning parameter that trades-off significant savings of storage with slight loss of accuracy. As Appendix B.2 shows, a storage saving of β results in accuracy loss maximum of β^3. For example, if $\beta = 0.3$, a 30% saving of storage is traded with maximum of 2.7% of accuracy loss. However, the experimental evaluation shows even much better performance.

8 Adaptive Load Shedding

As we show in Section 7, load shedding in *Venus* uses a global parameter β that represents the minimum spatial distance, as a ratio from R, between queries locations and microblogs locations. Choosing the right value for β is challenging as it should change across space and time: Microblogs queries change dynamically over time [24] and a single value limits the cost-benefit trade-off of load shedding. More importantly, the spatial distribution of both microblogs data and queries changes substantially across regions [6], and therefore using a global parameter may poorly treat sparse regions for which few queries are issued and aggressively treat dense regions where most queries are issued.

In this section, we introduce two methods, β-load shedding and γ-load shedding, which tune the load shedding process. Both methods extend the β-parameterized load shedding module in three aspects: (a) they tune the load shedding automatically so that it is not needed to preset a fixed value for β by the system administrator, (b) they keep one load shedding parameter value per each spatial index cell, instead of using one global value for all regions, to adapt with the localized distributions of incoming data and queries , and (c) they update the load shedding parameter values dynamically over time to reflect the changes in both data and queries. In the rest of this section, we develop the two methods and discuss their impact on the system components.

8.1 β-Load Shedding

In β-Load Shedding (β-LS for short), each index cell stores a parameter β to use in determining its temporal horizon $T_{c,\beta}$ (Equations 3 and 4). β has exactly the same meaning as described in Section 7, however, it is distinct per index cell instead of being a global parameter for the whole index. In addition, β-LS automatically tunes β values based on the incoming query loads. Then, each cell C uses its own auto-tuned β value to keep only microblogs from the last $T_{c,\beta}$ time units and shed older microblogs. In the rest of the section we describe the automatic tuning of β along with β-LS impact on *Venus* components.

Main idea. The main idea is to distinguish the spatial regions based on the percentage of queries they receive. Regions that receive a big percentage of the incoming queries are considered important spatial regions and so a small portion of data is shed, i.e., small β value is assigned, to reduce the likelihood to miss answer microblogs for a lot of queries. On the contrary, cells that receive small percentage of queries are considered less important, so that shedding more data will not significantly affect the query accuracy, and so a large β value is assigned. Thus, we use only the spatial distribution of incoming queries to estimate the β value, per cell, that is bounded in the range $[0, 1]$.

Implementation. For each index cell C, we keep the percentage of queries that process C microblogs out of all queries that are posted to the index. Specifically, for the whole index, we keep a single long integer Q_{total} that counts the total number of posted queries to the index so far. In addition, for each cell C, we maintain an integer $C.Q_c$ that counts the number of queries that process *one or more* microblog(s) from $C.M_List$. Then, whenever $C.T_{c,\beta}$ value is updated, on insertions in C, the value of β is estimated by $\beta = 1 - \frac{C.Q_c}{Q_{total}}$. Consequently, cells that did not receive any queries, i.e., $C.Q_c = 0$, are assigned β value of 1 and then a large amount of data is shed. On the other hand, cells that receive a big percentage of queries are assigned a small β value and hence shed much less data. Both Q_{total} and $C.Q_c$ are reset every T time units, measured from the system start timestamp, so that β values are estimated only based on the recent queries to adapt with the dynamic changes in query loads over time. By definition, $0 \leq C.Q_c \leq Q_{total}$, for all C, then β value is bounded in the range $[0, 1]$.

Impact on *Venus* components. β-LS implementation impacts index contents and query processing. For the index, each cell C maintains an additional integer $C.Q_c$, which ends up with a little impact on the overall index storage (less than 0.5MB extra which does not exceed 1^{-4}% of the overall storage) compared to the big storage saving that comes from shedding more microblogs. During the query processing, Q_{total} is maintained for the whole index and $C.Q_c$ is maintained for each cell. Although being concurrently accessed from multiple query threads, the concurrent operation on both of them is only a single atomic increment which causes a little overhead in query latency as our experiments show in Section 9.

8.2 γ-Load Shedding

In γ-Load Shedding method (γ-LS for short), we go one step beyond using only the query spatial distribution (as in β-LS) and use the access pattern of microblogs data inside the cell. β-LS increases the importance measure of a cell C as long as *one or more* of its microblogs is processed by the query regardless of the actual number of processed microblogs from $C.M_List$. On the contrary, γ-LS considers which microblogs are actually processed from the cell so that each cell stores only the useful data. To illustrate, we recall one of *Mercury* [6] findings that the analytical values of T_c and $T_{c,\beta}$ do not comply with the theoretical expectations. Specifically, T_c achieves $< 100\%$ query accuracy while it is expected to provide accurate results while $T_{c,\beta}$ achieves query accuracy much higher than the theoretical bound $((1 - \beta^3) \times 100)\%$. This means that each cell C stores either less or more data than it is needed. Motivated by this finding, γ-LS aims to adjust cell storage so that only microblogs that are sufficient to answer all incoming queries accurately are stored.

Main idea. The main idea is to estimate for each cell C the minimum search time horizon $T_{c,\gamma} \leq T$ such that C keeps only the useful data to answer incoming queries from main-memory contents. Unlike T_c, that is calculated analytically based only on the default query parameters as discussed in Section 6, $T_{c,\gamma}$ is calculated adaptively with the incoming query load. As T_c and $T_{c,\beta}$ are shown to be close to the optimal time horizon, to calculate $T_{c,\gamma}$, we make use of T_c and $T_{c,\beta}$ equations. Particularly, we replace the parameter β in $T_{c,\beta}$ (Equation 3) with another parameter γ. Unlike β, γ can take any value rather than being bounded in the range $[0, 1]$. Thus, γ is a tuning parameter where its values have three possible cases: (1) $\gamma \in [0, 1]$: in this case γ has the same effect as β (see Section 7) and controls the amount of shed data through controlling the value of $T_{c,\gamma} \leq T_c \leq T$. (2) $\gamma > 1$: in this case, the value $T_{c,\beta}$ at $\beta = 1$ is too large for the incoming queries to this cell, then the term $(1 - \gamma)$ gives a negative value and decreases the cell temporal coverage to shed the useless data that increases the storage overhead while does not contribute to the query answers. (3) $\gamma < 0$: in this case,

the value $T_{c,\beta}$ at $\beta = 0$, i.e. T_c, is too small to answer all the incoming queries to this cell, then the term $(1 - \gamma)$ gives a value larger than 1 and increases the cell temporal coverage to answer all the incoming queries accurately. Although Case 3 would lead to a slight increase in the storage overhead for some parameter setup, e.g., at $\alpha \approx 0$, it would consequently fill the gap between the theoretical assumptions of T_c value and the practical data distribution which lead to loss in query accuracy, as shown in *Mercury* [6] experiments.

Implementation. To implement γ-LS, *Venus* maintains a γ value in each index cell, that is changing adaptively with the incoming queries. To this end, γ value is calculated as follows. For an incoming query, we measure the time horizon $T_{c,\gamma}$ that spans all the processed microblogs, i.e., the useful data, in C. Obviously, $T_{c,\gamma}$ equals the difference between NOW and the oldest processed microblog. Based on the measured value of $T_{c,\gamma}$, we calculate a value γ using the following equations:

For linear scoring functions

$$\gamma = 1 - \left(T_{c,\gamma} - \frac{k}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_c}\right) \frac{1-\alpha}{\alpha T} \quad (5)$$

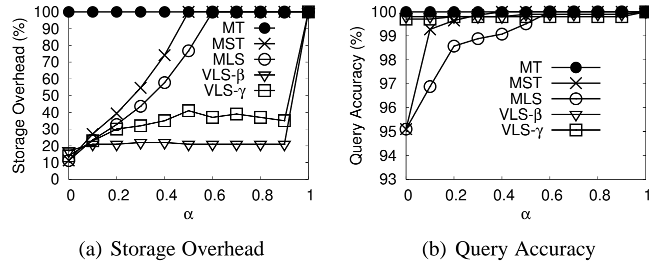
For exponential scoring functions

$$\gamma = \frac{1}{w} ln \left(e^{w} + \frac{1-\alpha}{\alpha}\left(e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\lambda_c T}} - e^{\frac{wT_{c,\gamma}}{T}}\right)\right) \quad (6)$$

Equations 5 and 6 are derived from Equations 3 and 4 by replacing β with γ and separating γ in the left hand side. If the ln parameter has a negative value, the negative sign is omitted and multiplied by the final result. Using a series of γ values, from subsequent queries, one estimated value of γ is calculated for each cell C. Then, the estimated γ value is used in Equations 3 and 4, replacing β, to calculate the actual cell temporal coverage $T_{c,\gamma}$.

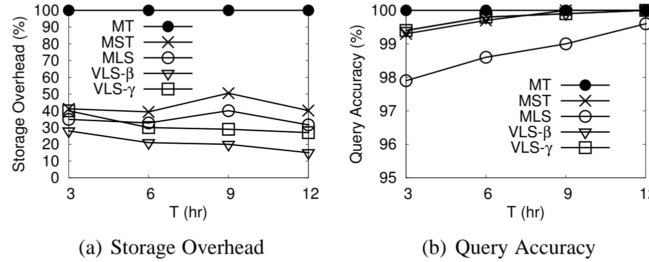
To estimate γ value per cell, we use a sample of the incoming queries to the cell. This sample is chosen randomly and independently per cell. For each query in the sample, a γ value is calculated, during the query processing, as described above. Then, the estimated γ is calculated by one of two methods: *min* or *average* where the minimum or the average value, respectively, so far is used. In both cases, γ value is reset every T time units, measured from the system start timestamp, so that it is estimated only based on the recent queries to adapt with the dynamic changes in query loads over time. The query sample is chosen randomly per cell for two reasons: (a) As γ is calculated during the query processing, then using all the incoming queries may be overwhelming to the query latency with a heavy query load in real time, so only a sample of queries are being used to reduce this overhead. (b) The query sample is chosen randomly and independently for each cell to eliminate any bias for a certain subset of the queries. As calculating γ value in each cell is independent from all other cells contents, choosing a different query sample for each cell is valid and leads to highly reliable load shedding as almost all incoming queries have a chance to contribute to tuning the load shedding in some cells.

Impact on *Venus* components. γ-LS implementation mainly impacts the query processing and slightly impacts index contents. For each index cell C, a single estimated value $C.\gamma$ is maintained incrementally. In addition, an additional integer is maintained per cell in case $C.\gamma$ is estimated using the incremental average. Both



(a) Storage Overhead (b) Query Accuracy

Fig. 3. Effect of α on storage vs. accuracy



(a) Storage Overhead (b) Query Accuracy

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

end up with maximum of 1MB extra storage on the average which is much less than storage saving of the shed microblogs and presents a negligible percentage of 1^{-4}% out of the overall storage consumption. $C.\gamma$ is incrementally maintained when incoming queries access some microblogs from $C.M_List$. For each query, a new γ value is calculated as described above and its estimated value $C.\gamma$ is updated accordingly. Although $C.\gamma$ is concurrently accessed from multiple query threads, only a single concurrent operation is needed to set the new value which is a little overhead compared to the expensive computation of γ values. Section 9 shows the query latency overhead of γ-LS compared to its storage saving and accuracy enhancement.

In both β-LS and γ-LS, the search time horizon is calculated based on Equations 3 and 4. To prevent the cancellation of the major term when $\alpha = 0$, which totally discards the automatic tuning of the adaptive load shedding module, we replace α in the numerator of these equations by $(\alpha + \epsilon)$ so that the values of β and γ work for adjusting the amount of load shedding. We set $\epsilon = 0.0001$. In Section 9, our experiments show that this heuristic increases the query accuracy, at $\alpha = 0$, from $\approx 95\%$, as in *Mercury* [6], to over 99% in *Venus*.

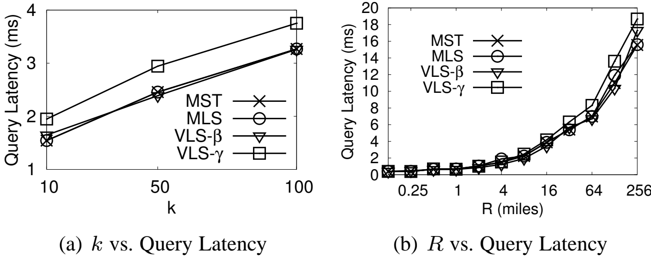
9 Experimental Evaluation

This section provides experimental evaluation of *Venus* based on an actual system implementation. As a successor of *Mercury* [6], and with lack of other direct competitors (see Section 2), *Venus* evaluation shows the effectiveness of its new components compared to *Mercury* components. This includes: (1) The adaptive load shedding module, with its two variations β-LS and γ-LS, as described in Section 8. (2) The flexible top-k ranking that employs both linear and exponential ranking functions. The experimental study in this section evaluates the effect of the different ranking functions on both effectiveness of spatio-temporal pruning in query processing and index storage overhead and its effect on query accuracy.

All experiments are based on a real prototype of *Venus* and using a real-time feed of US tweets (via access to Twitter Firehose archive) and actual locations of web search queries from Bing. We have stored real 340+ million tweets and one million Bing search queries in files. Then, we have read and timestamped

	Linear Scoring	Exponential Scoring
Spatial Score	$\frac{D_s(M.loc,u.loc)}{R}$	$e^{w\times\frac{D_s(M.loc,u.loc)}{R}}$
Temporal Score	$\frac{D_t(M.time,NOW)}{T}$	$e^{w\times\frac{D_t(M.time,NOW)}{T}}$
Spatial Pruning Boundary	$Min\left(R, \frac{MIN}{\alpha}R\right)$	$Min\left(R, \frac{1}{w}\times ln(\frac{MIN-(1-\alpha)}{\alpha})R\right)$
Temporal Pruning Boundary	$Min\left(T, \frac{MIN}{1-\alpha}T\right)$	$Min\left(T, \frac{1}{w}\times ln(\frac{MIN-\alpha}{1-\alpha})T\right)$
Index Size Tuning Time (T_c)	$Min\left(T, \frac{\alpha}{1-\alpha}T + \frac{k}{Min\left(\frac{Area(R)}{Area(C)},1\right)\times\lambda_c}\right)$	$Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^{w}-1) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\times\lambda_c T}}]\right)$
Load Shedding Time ($T_{c,\beta}$)	$Min\left(T, \frac{\alpha(1-\beta)}{1-\alpha}T + \frac{k}{Min\left(\frac{Area(R)}{Area(C)},1\right)\times\lambda_c}\right)$	$Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^{w}-e^{w\beta}) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\lambda_c T}}]\right)$
Adaptive γ-Load Shedding (γ)	$1-\left(T_{c,\gamma} - \frac{k}{Min\left(\frac{Area(R)}{Area(C)},1\right)\times\lambda_c}\right)\frac{1-\alpha}{\alpha T}$	$\frac{1}{w}ln\left(e^{w} + \frac{1-\alpha}{\alpha}\left(e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\lambda_c T}} - e^{\frac{wT_{c,\gamma}}{T}}\right)\right)$

TABLE 1
Summary of different system components equations using linear and exponential scoring.



(a) k vs. Query Latency (b) R vs. Query Latency

Fig. 5. Effect of adaptive load shedding on query latency

them to simulate an incoming stream of real microblogs and queries. Unless mentioned otherwise, the default value of k is 100, microblog arrival rate λ is 1000 microblogs/second, range R is 30 miles, T is 6 hours, α is 0.2, β is 0.3, w is 1, cell capacity is 150 microblogs, the spatial and temporal scoring functions are linear, and γ-LS uses *min* estimation. The default values of cell capacity, α, and β are selected experimentally and show to work best for query performance and result significance, respectively, while default λ is the effective rate of US geotagged tweets. As microblogs are so timely that Twitter gives only the most recent tweets (i.e., α=0), we set α 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 overhead, query accuracy, and query latency. Query accuracy is calculated as the percentage of correct microblogs in the obtained answer compared to the true answer. True answer is calculated when all microblogs of the last T time units are stored in the index. The rest of this section recaps *Mercury* results [6] (Section 9.1) and evaluates the adaptive load shedding (Section 9.2) and top-k ranking (Section 9.3).

9.1 *Mercury* Results Recap

In this section, we recap *Mercury* [6] results, the predecessor of *Venus*, as a context for evaluating the new components in *Venus*. *Mercury* has evaluated three alternatives of its index: (a) storing all microblogs of last T time units (denoted as *MT*), (b) using the *index size tuning* module (Section 6), denoted as *MST*, and (c) using *Mercury load shedding* module (Section 7), denoted as *MLS*.

Index Scalability: *Mercury* shows that *MT* digests 32K microblog/sec while *MST* and *MLS* digest 64K in ~ 0.5 sec. This shows an efficient digestion for arrival rates an order of magnitude higher than Twitter rate. Also, it shows more digestion scalability for indexes that store less data.

Memory Optimization: *Mercury* shows that *MST* can achieve storage savings of 90-25% for $\alpha < 0.5$. This corresponds to query accuracy of 95-99+%, where the lowest accuracy, i.e, 95%, is achieved at α = 0 due to the cancellation of the major term in T_c and so the index barely stores only k microblogs in each region R. Although *MST* is theoretically expected to achieve 100% accuracy consistently, the small accuracy loss comes from the gap between the theoretical assumption of uniformly distributed microblogs locations within the cell and the actual distribution which is not strictly uniform. On the other hand, *MLS* is shown to achieve 60-90% storage saving with 99-73% corresponding query accuracy at α = 0.2. The lowest achieved accuracy is 52% at β = 1 and α = 0.9 which is much higher than the theoretical bound.

Query Evaluation: *Mercury* shows that its spatial and temporal pruning are both very effective and significantly dominate the naive approaches. Also, the temporal pruning is more effective than spatial pruning even for large values of α (up to 0.8). The average query latency, when using both spatial and temporal pruning, for most parameters setup is 4 msec.

9.2 Adaptive Load Shedding

In this section, we evaluate the effectiveness of *Venus* adaptive load shedding module. We compare the two variations of *Venus* index with adaptive load shedding employed: (a) β-LS (Section 8.1), denoted as *VLS*-β, and (b) γ-LS (Section 8.2), denoted as *VLS*-γ, with three alternatives of *Mercury* [6] index (as in Section 9.1).

9.2.1 Effect on Querying and Storage

Figure 3 shows the effect of *Venus* adaptive load shedding on both storage overhead and query accuracy with varying α. For a wide range of varying α, Figure 3 shows the superiority of *VLS*-β and *VLS*-γ, for $\alpha > 0$, in saving a significant amount of storage (up to 80%) while keeping almost perfect accuracy (more than 99%). This is applicable even for large values of α (up to 0.9) which is a significant enhancement over *Mercury* alternatives (*MST* and *MLS*). With increasing α, *MST* and *MLS* keep more data as the spatial dimension is getting increasing weight in the

relevance score and hence older data are kept to account for being spatially close to incoming queries. However, as *VLS-*β and *VLS-*γ take the query spatial distribution into account and monitor the actual useful data localized per region instead of using a global parameter, they can smartly figure out almost all data that are not contributing to query answers and hence shed up to 80% without affecting the query accuracy. *MST* and *MLS* cannot sustain such large savings for $\alpha \geq 0.4$.

On the contrary to large α values, for $\alpha = 0$, *VLS-*β and *VLS-*γ come with a bit extra storage overhead, 14%, and 17%, respectively, compared to 11% in both *MST* and *MLS*, to increase the accuracy from $\sim 95\%$ to more than 99%. This is a result of the heuristic discussed in Section 8.2 which prevents cancellation of *VLS-*β and *VLS-*γ effect and hence they can automatically discover which data are useful for the incoming queries to keep them. This specific point, at $\alpha = 0$, shows that *VLS-*β and *VLS-*γ are adaptive so that they keep more data when needed as well as shedding useless data if exists.

Figure 4 shows the effect of varying T on storage and accuracy. For small values of T, *VLS-*γ encounters slightly more storage overhead than *MLS*. However, with increasing T, *VLS-*γ storage overhead becomes comparable to *MLS* while consistently maintains more than 99% accuracy. *VLS-*β dominates all other alternative for all values of T with almost perfect accuracy.

As *VLS-*β and *VLS-*γ come with an overhead during the query processing, Figure 5 shows the query latency with varying k and R. Both figures show higher query latency for both *VLS-*β and *VLS-*γ over *MST* and *MLS*. It is also noticeable that *VLS-*γ encounters higher latency than *VLS-*β due to the computational cost of calculating γ. However, the latency increase is acceptable and does not exceed 3 ms for large values of $R = 256$ miles, where many index cells are involved in β and γ computations. For average values of k and R, the increase in the order of 1 ms on the average. In nutshell, *VLS-*β and *VLS-*γ incur 12-14% increase in query latency to save up to 80% of storage, for wide ranges of parameters values, without compromising the accuracy. The 90, 95, and 99 percentiles of query latencies for all alternatives are under 15, 30, and 50 ms, respectively. More detailed analysis for query latency percentiles can be revised in Appendix C.

For *VLS-*γ, the presented results show *min* estimation method, which is more conservative than *average* method and leads to higher storage overhead. Generally, for all parameter values, *average* method behave pretty similar to *min* method and thus the same analysis of results would be applicable. For space limitations, results for *average* estimation method are moved to Appendix C.1.

9.2.2 Effect on Index Maintenance

With a significant amount of data shed from the index, *VLS-*β and *VLS-*γ significantly improve the index maintenance overhead. Figure 6 shows index insertion time with varying tweet arrival rate, k, α, and R. For all the parameter values, *VLS-*β and *VLS-*γ show lower insertion time due to the lighter index contents. As Figure 6(a) shows, *VLS-*γ is able to digest 64K microblog in ~ 400 ms while *VLS-*β does in less than a quarter of a second. For different values of tweet arrival rate, k, and R, *VLS-*γ insertion time is slightly better than *MLS* while *VLS-*β is significantly better than both of them. However, for a wide range of α values, both *VLS-*β and *VLS-*γ work significantly better than *MLS* as Figure 6(c) shows. This shows the superiority of *VLS-*β and *VLS-*γ and that the decrease in insertion time is proportional with the storage savings, so the lighter the index contents the more efficient it digests more data. It worth mentioning that the efficient bulk insertion techniques used in *Venus* significantly increase digestion rates four times for all alternatives. Detailed numbers and evaluation are presented in Appendix C.4.

9.3 Top-k Ranking

In this section, we study the effect of employing different ranking functions on *Venus* components. Specifically, the ranking function is affecting: (1) Index size tuning, as the values T_c and $T_{c,\beta}$ depend on the employed ranking function, and (2) the spatio-temporal pruning during the query processing. In this section, we study the effect of employing the linear and exponential functions (defined in Section 3.3), namely *F-Lin* and *F-Exp*, respectively, and their curves are denoted throughout the section by the suffixes *-Lin* and *-Exp*, respectively.

9.3.1 Ranking Effect on Index Size and Query Accuracy

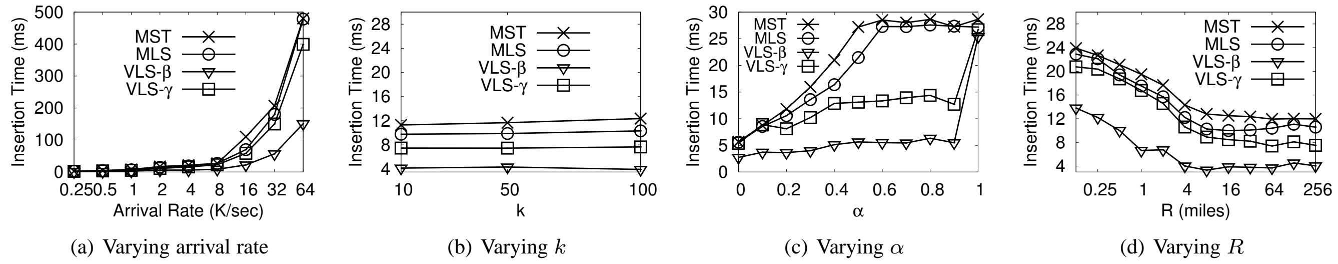
Table 1 summarizes the equations of different system components using both linear and exponential scoring. Mathematically, the values T_c and $T_{c,\beta}$ of *F-Lin* (Equations 1 and 3) give tighter temporal coverage than *F-Exp* (Equations 2 and 4). Consequently, Figures 7-9 show that *F-Exp* encounters more storage overhead than *F-Lin* for varying T, α and β. In Figure 7(a), *F-Exp* encounters larger storage overhead for smaller values of T while approach *F-Lin* storage with increasing T. However, for all T values, *F-Exp* achieves perfect accuracy that is almost 100% for both *MST* and *MLS*. The same observations hold for varying α and β in Figures 8 and 9, respectively. For T, α and β, the increase in *F-Exp* storage overhead between 7-15% more than *F-Lin*.

Two interesting points to discuss are at $\alpha = 0$ and at $\beta = 1$ in Figures 8 and 9, respectively. At these points, both T_c and $T_{c,\beta}$ almost vanish and the index barely stores only the most recent k microblogs in each region, which makes the query accuracy of *F-Lin* drops significantly, as shown in Figures 8(b) and 9(b), especially at $\beta = 1$ for large values of α where the spatial score is more important than the temporal score and so old microblogs matter. However, in all these cases, *F-Exp* accuracy remains almost perfect. This shows that *F-Exp* accuracy improvement is not a result for only storing more data in the index. Instead, the exponential scoring quickly demotes further microblogs, in either space, time or both, and hence less microblogs are needed to get the accurate answer. This is shown clearly while analyzing spatio-temporal pruning in Section 9.3.2.

Finally, it worth mentioning that employing *VLS-*β and *VLS-*γ with *F-Exp* gives pretty similar numbers to those in Section 9.2 in both storage overhead and query accuracy. For space limitations, we moved these results to Appendix C.2.

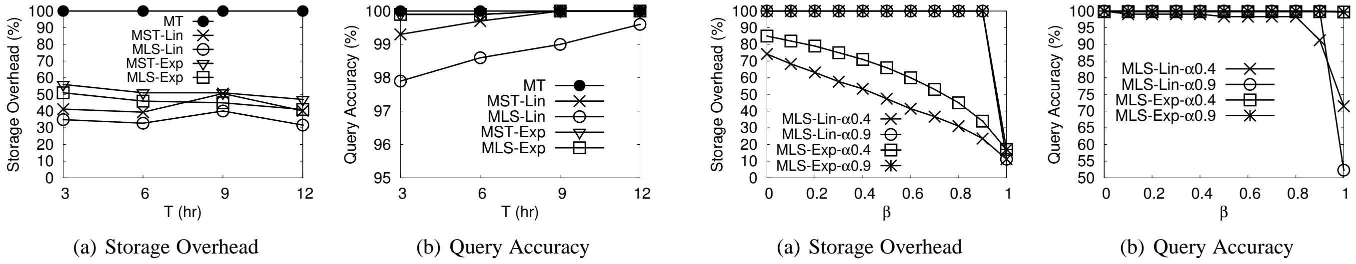
9.3.2 Ranking Effect on Spatio-temporal Query Pruning

Figure 10 compares the performance of *Venus* query processor employing either only spatial pruning, denoted as *PR*, temporal pruning, denoted as *PT*, or spatio-temporal pruning, denoted as *P*, for both *F-Lin* and *F-Exp*. In Figure 10(a), query latency of all alternatives of *F-Exp* are bounded between *PT-Lin* and *P-Lin*, except for large values of R (> 64) where *P-Exp* has a lower latency than *P-Lin*. This behavior can be interpreted by discussing two contradicting factors: (1) The computation cost, and (2) the pruning effectiveness of each ranking function. First, the cost of computing exponential score by *F-Exp* is higher than the linear score by *F-Lin* due to the higher mathematical complexity. As



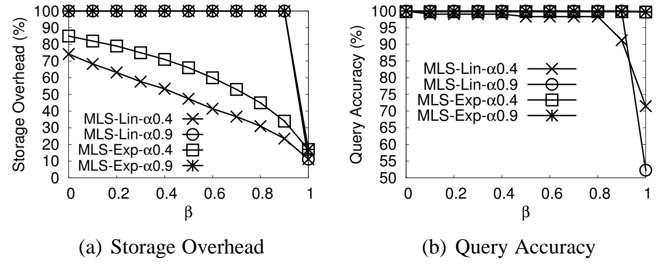
(a) Varying arrival rate (b) Varying k (c) Varying α (d) Varying R

Fig. 6. Index insertion time.



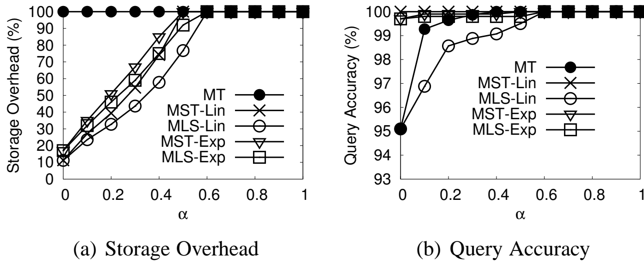
(a) Storage Overhead (b) Query Accuracy

Fig. 7. Ranking effect on storage vs. accuracy varying T



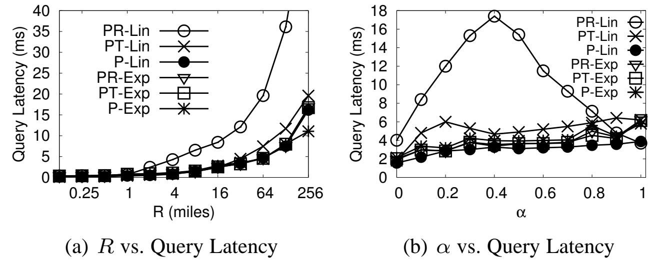
(a) Storage Overhead (b) Query Accuracy

Fig. 9. Ranking effect on storage vs. accuracy varying β



(a) Storage Overhead (b) Query Accuracy

Fig. 8. Ranking effect on storage vs. accuracy varying α



(a) R vs. Query Latency (b) α vs. Query Latency

Fig. 10. Ranking effect on query pruning

this operation repeats for every single microblog, its cost is not negligible. Second, *F-Exp* is much more powerful in pruning the search space. For the same increase in either spatial or temporal distance, the exponential score is demoted rapidly and thus the search can quit much earlier than the linear score. Consequently, in Figure 10(a), for R values ≤ 64, the expensive computation cost of *F-Exp* makes all its alternatives have higher latency than *P-Lin* while its pruning power make them better than both *PT-Lin* and *PR-Lin*. For larger values of R (> 64), when many cells are involved in the query, the pruning power of *F-Exp* makes more difference and gives *P-Exp* a latency of 11 ms for $R = 256$ compared to 16 ms for *P-Lin*. Consistently, both *PR-Exp* and *PT-Exp* have query latency as low as *P-Lin* which shows two conclusions: (a) Pruning a single dimension using the exponential score gives the same latency as pruning both dimensions using the linear score. (b) Unlike *F-Lin*, all *F-Exp* alternatives have query latency within a small margin which shows that pruning either spatial or temporal dimension has the same effectiveness, on the contrary to *F-Lin* in which the temporal pruning is much more effective than the spatial pruning (see [6] or Appendix C.5 for full analysis).

Figure 10(b) shows the query latency varying α. In this figure, the computation cost of *F-Exp* dominates the pruning power (as default R value is 30 miles) and so all alternatives of *F-Exp* have slightly higher latency than *P-Lin* but still lower than *PR-Lin* and *PT-Lin*. The figure also shows the effectiveness of both spatial and temporal pruning using *F-Exp*.

10 Conclusion

We have presented *Venus*; a system for real-time support of spatio-temporal queries on microblogs, where users request a set of recent k microblogs near their locations. *Venus* works under a challenging environment, where microblogs arrive with very high arrival rates. *Venus* 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. In addition, effective load shedding modules are employed to smartly shed the useless data while providing almost perfect query accuracy.

References

[1] "Twitter Statistics," http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/, 2013.

[2] "Twitter Data Grants, 2014," https://blog.twitter.com/2014/introducing-twitter-data-grants.

[3] "Facebook Statistics," https://www.facebook.com/business/power-of-advertising, 2012.

[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] W. G. Aref and H. Samet, "Efficient Processing of Window Queries in the Pyramid Data Structure," in *PODS*, 1990.

[6] 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.

[7] G. Lee, J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy, "The Unified Logging Infrastructure for Data Analytics at Twitter," *PVLDB*, vol. 5, no. 12, pp. 1771–1780, 2012.

[8] J. Lin and A. Kolcz, "Large-scale machine learning at twitter," in *SIGMOD*, 2012.

[9] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-Time Search at Twitter," in *ICDE*, 2012.
[10] C. Chen, F. Li, B. C. Ooi, and S. Wu, "TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets," in *SIGMOD*, 2011, pp. 649–660.
[11] L. Wu, W. Lin, X. Xiao, and Y. Xu, "LSII: An Indexing Structure for Exact Real-Time Search on Microblogs," in *ICDE*, 2013.
[12] J. Yao, B. Cui, Z. Xue, and Q. Liu, "Provenance-based Indexing Support in Micro-blog Platforms," in *ICDE*, 2012.
[13] 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.
[14] A. Bermingham and A. F. Smeaton, "Classifying Sentiment in Microblogs: Is Brevity an Advantage?" in *CIKM*, 2010.
[15] E. Meij, W. Weerkamp, and M. de Rijke, "Adding semantics to microblog posts," in *WSDM*, 2012.
[16] G. Mishne and J. Lin, "Twanchor Text: A Preliminary Study of the Value of Tweets as Anchor Text," in *SIGIR*, 2012.
[17] 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.
[18] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling, "TwitterStand: News in Tweets," in *GIS*, 2009.
[19] H. Abdelhaq, C. Sengstock, and M. Gertz, "EvenTweet: Online Localized Event Detection from Twitter," in *VLDB*, 2013.
[20] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang, "TEDAS: A Twitter-based Event Detection and Analysis System," in *ICDE*, 2012.
[21] M. Mathioudakis and N. Koudas, "TwitterMonitor: Trend Detection over the Twitter Stream," in *SIGMOD*, 2010.
[22] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: Real-time event detection by social sensors," in *WWW*, 2010.
[23] V. K. Singh, M. Gao, and R. Jain, "Situation Detection and Control using Spatio-temporal Analysis of Microblogs," in *WWW*, 2010.
[24] J. Lin and G. Mishne, "A Study of "Churn" in Tweets and Real-Time Search Queries," in *ICWSM*, 2012.
[25] D. Ramage, S. T. Dumais, and D. J. Liebling, "Characterizing Microblogs with Topic Models," in *ICWSM*, 2010, pp. 130–137.
[26] 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.
[27] I. Uysal and W. B. Croft, "User Oriented Tweet Ranking: A Filtering Approach to Microblogs," in *CIKM*, 2011.
[28] J. Hannon, M. Bennett, and B. Smyth, "Recommending twitter users to follow using content and collaborative filtering approaches," in *RecSys*, 2010.
[29] O. Phelan, K. McCarthy, and B. Smyth, "Using twitter to recommend real-time topical news," in *RecSys*, 2009.
[30] 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.
[31] 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.
[32] ——, "Processing and Visualizing the Data in Tweets," *SIGMOD Record*, vol. 40, no. 4, 2012.
[33] L. Hong, A. Ahmed, S. Gurumurthy, A. J. Smola, and K. Tsioutsiouliklis, "Discovering Geographical Topics In The Twitter Stream," in *WWW*, 2012.
[34] C. Budak, T. Georgiou, D. Agrawal, and A. E. Abbadi, "GeoScope: Online Detection of Geo-Correlated Information Trends in Social Networks," in *VLDB*, 2014.
[35] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen, "Scalable Top-k Spatio-temporal Term Querying," in *ICDE*, 2014, pp. 148–159.
[36] Y.-Y. Chen, T. Suel, and A. Markowetz, "Efficient Query Processing in Geographic Web Search Engines," in *SIGMOD*, 2006.
[37] G. Cong, C. S. Jensen, and D. Wu, "Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects," *PVLDB*, vol. 2, no. 1, 2009.
[38] 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*, vol. 23, no. 4, 2011.
[39] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint Top-K Spatial Keyword Query Processing," *TKDE*, vol. 24, no. 10, 2012.
[40] 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.
[41] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial Keyword Query Processing: An Experimental Evaluation," in *VLDB*, 2013.
[42] S. J. Kazemitabar, U. Demiryurek, M. H. Ali, A. Akdogan, and C. Shahabi, "Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight," *PVLDB*, vol. 3, no. 2, 2010.
[43] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing, "Discovering Spatio-temporal Causal Interactions in Traffic Data Streams," in *KDD*, 2011.
[44] E. Meskovic, Z. Galic, and M. Baranovic, "Managing Moving Objects in Spatio-temporal Data Streams," in *MDM*, 2011.
[45] M. F. Mokbel and W. G. Aref, "SOLE: Scalable On-Line Execution of Continuous Queries on Spatio-temporal Data Streams," *VLDB Journal*, vol. 17, no. 5, 2008.
[46] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger, "Temporal and Spatio-temporal Aggregations over Data Streams using Multiple Time Granularities," *Information Systems*, vol. 28, no. 1-2, 2003.
[47] 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.
[48] S. Shekhar and S. Chawla, *Spatial Databases: A Tour*. Prentice Hall, 2003.
[49] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," *ACTA*, vol. 4, no. 1, 1974.

Amr Magdy is a Ph.D. candidate at the Department of Computer Science and Engineering, University of Minnesota - Twin Cities. He received his M.Sc. at the same department in 2013. His research interests include big data management, social data management, and spatial data management. His current research focus is managing microblogs data. His research work has been incubated by Bing GeoSpatial team and has been selected among best papers in ICDE 2014. He has been selected a finalist for Microsoft Research PhD Fellowship 2014-2016.

Mohamed F. Mokbel (Ph.D., Purdue University, USA, MS, B.Sc., Alexandria University, Egypt) is an associate professor at University of Minnesota. His current research interests focus on providing database and platform support for spatio-temporal data, location-based services 2.0, personalization, and recommender systems. His research work has been recognized by four best paper awards at IEEE MASS 2008, IEEE MDM 2009, SSTD 2011, and ACM MobiGIS Workshop 2012, and by the NSF CAREER award 2010. Mohamed is/was general co-chair of SSTD 2011, program co-chair of ACM SIGSPAITAL GIS 2008-2010, and MDM 2014, 2011. He has served in the editorial board of ACM Transactions on Spatial Algorithms and Systems, IEEE Data Engineering Bulletin, Distributed and Parallel Databases Journal, and Journal of Spatial Information Science. Mohamed has held various visiting positions at Microsoft Research, USA, Hong Kong Polytechnic University, and Umm Al-Qura University, Saudi Arabia. Mohamed is an ACM Senior and IEEE Senior member and a founding member of ACM SIGSPATIAL. He is currently serving as an elected chair of ACM SIGSPATIAL. For more information, please visit: www.cs.umn.edu/∼mokbel

Sameh Elnikety is a researcher at Microsoft Research in Redmond, Washington. He received his Ph.D. from the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland, and M.S. from Rice University in Houston, Texas. His research interests include distributed server systems, and database systems. Samehs work on database replication received the best paper award at Eurosys 2007.

Suman Nath is a senior researcher at Microsoft Research in Redmond, Washington. He received his M.S. and Ph.D. from Carnegie Mellon University (CMU). His research interests include sensor/time-series data management, data privacy and security, and flash memory. His research work has been recognized by best paper awards at BaseNets Workshop 2004, NSDI 2006, ICDE 2008, SSTD 2011, Grace Hopper 2012, and MobiSys 2012.

Yuxiong He is a researcher at Microsoft Research in Redmond, Washington. She received her Ph.D. in Computer Science from Singapore-MIT Alliance in 2008. Her research interests include resource management, algorithms, modeling and performance evaluation of parallel and distributed systems. Her research work has been selected among best papers in ICDE 2014.

APPENDIX A
INDEX SIZE TUNING TIME HORIZON

This appendix 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 as discussed in Section 6. Per the following Lemma, T_c is computed based on the default values of k, R, T, and α, and uses the microblog arrival rate λ_c for each cell C assuming we are using the linear scoring functions. 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 λ_c

Lemma 1: *Given query parameters k, R, T, and α, and the average arrival rate of microblogs in cell C, λ_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 λ_R as the expected arrival rate of microblogs to query area R, among the microblogs in cell C with arrival rate λ_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 λ_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 kth (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: $F(M_k)$ = α + $(1-\alpha)\frac{k}{\lambda_R T}$. Accordingly, to satisfy the condition that $F(M) < F(M_k)$, the following should hold:

$$(1-\alpha)\frac{T_c}{T} < \alpha + (1-\alpha)\frac{k}{\lambda_R T} \quad (7)$$

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 λ_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:

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

■

In case of exponential ranking function that is presented in Section 3.3, the equation of T_c is given as follows:

$$T_c = Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^w - 1) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\times\lambda_c T}}]\right) \quad (8)$$

This equation can be derived using exactly the same steps as in the case of linear ranking function. The proof is given as following.

Proof: The proof is composed of exactly the same three steps as the previous one. The first and second steps are independent of the ranking function and gives:

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

$$T_k = \frac{k}{\lambda_R}$$

Then, 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 the exponential 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 kth (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., $F(M) = (1-\alpha)e^{w\times\frac{T_c}{T}}$ 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: $F(M_k) = \alpha + (1-\alpha)e^{w\times\frac{k}{\lambda_R T}}$. Accordingly, to satisfy the condition that $F(M) < F(M_k)$, the following should hold:

$$(1-\alpha)e^{w\times\frac{T_c}{T}} < \alpha + (1-\alpha)e^{w\times\frac{k}{\lambda_R T}} \quad (9)$$

By separating the two sides and substituting the value of λ_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:

$$T_c = Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^w - 1) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\times\lambda_c T}}]\right) \quad (10)$$

■

APPENDIX B
LOAD SHEDDING TIME HORIZON AND ACCURACY LOSS

This appendix aims to find the value $T_{c,\beta}$ for each cell C, such that only those microblogs that have arrived in C in the last $T_{c,\beta}$ time units are kept in memory, and analyze the accuracy loss of the load shedding module as discussed in Section 7. We first derive the value of $T_{c,\beta}$ in B.1 then we analyze the accuracy loss in B.2.

B.1 Storage Saving

Building on the derivation of T_c in Appendix A, and assuming we use the linear scoring functions, we will relax the very conservative assumption of having a query location exactly on the location a microblog M, and hence Equation 7 will be reformulated as:

$$\alpha\beta + (1-\alpha)\frac{T_{c,\beta}}{T} < \alpha + (1-\alpha)\frac{k}{\lambda_R T} \quad (11)$$

Then, 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 λ_R, and bounding $T_{c,\beta}$ by the value of T, the maximum value of $T_{c,\beta}$ would be:

$$T_{c,\beta} = Min\left(T, \frac{\alpha(1-\beta)}{1-\alpha}T + \frac{k}{Min\left(\frac{Area(R)}{Area(C)}, 1\right) \times \lambda_c}\right) \quad (12)$$

Following the same steps, we can derive $T_{c,\beta}$ for the exponential scoring functions to be:

$$T_{c,\beta} = Min\left(T, \frac{T}{w}ln[\frac{\alpha}{1-\alpha}(e^w - e^{w\beta}) + e^{\frac{wk}{Min\left(\frac{Area(R)}{Area(C)},1\right)\lambda_c T}}]\right) \quad (13)$$

B.2 Accuracy Loss

Given the less conservative assumption in Equation 11, 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 $T_{c,\beta}$. A microblog M in area A_x satisfies two conditions: (1) The spatial score of M is less than β, 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:

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

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

This means that the temporal ratio is bounded by β.

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 β while a microblog at distance *Radius(R)* has spatial score of 1, then *Radius(A_x)* = β *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:

$$AccuracyLoss_{\beta} = T_{ratio} \times R_{ratio} \leq \beta^3 \qquad (14)$$

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

For the exponential scoring function, the area A_x in the spatio-temporal space would be an area bounded by two exponential curves and hence its area can be calculated using integration under the bounded area. However, roughly, expelling exponentially scored microblogs would lead to much less accuracy loss as a slight increase in either spatial or temporal distances would lead to exponential decay in the relevance score. The experimental evaluation clearly verifies this observation.

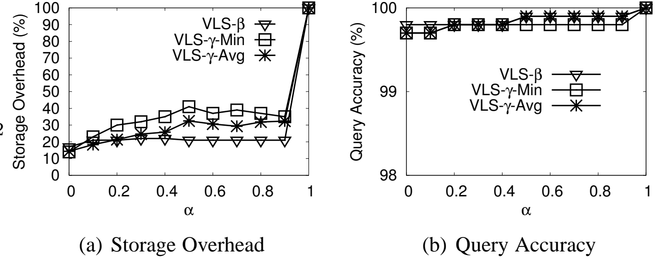
APPENDIX C
ADDITIONAL EXPERIMENTAL RESULTS

In this appendix, we provide additional experimental results that do not fit in the main paper contents due to space limitations.

C.1 Adaptive Load Shedding Additional Experiments

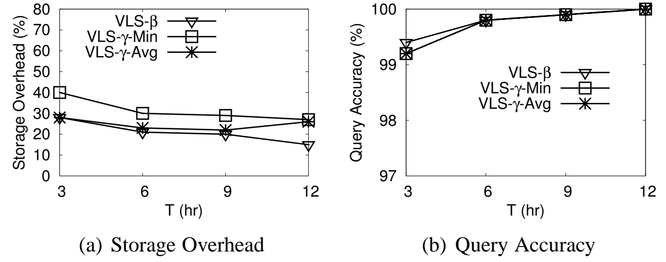
In this section, we present the results of *Venus* adaptive γ-load shedding with *average* estimation method (denoted as *VLS-γ-Avg*) compared to adaptive β-load shedding (denoted as *VLS-β*) and adaptive γ-load shedding with *min* estimation method (denoted as *VLS-γ-Min*) that are presented in Section 9.2.1. Generally, *VLS-γ-Avg* behavior is similar to *VLS-γ-Min* with minor differences in actual numbers of storage overhead and query accuracy. Particularly, Figure 11 shows the effect of *Venus* adaptive load shedding on both storage overhead and query accuracy with varying α. For α ranges from 0 to 0.9, Figure 11(a) shows that all *Venus* alternatives of adaptive load shedding techniques are able to save 59-86% of storage overhead, which is a significant improvement, while keeping almost perfect query accuracy as shown in Figure 11(b). In this range of α, *VLS-γ-Avg* consistently behaves at least as good as *VLS-γ-Min* and at most as good as *VLS-β*. For storage overhead, Figure 11(a) shows that *VLS-γ-Avg* encounter slightly higher storage overhead than *VLS-β* and slightly lower than *VLS-γ-Min* for all values of α. This is mainly because *VLS-γ-Min* is more conservative in estimating the value of load shedding parameter γ and hence stores more microblogs and then encounter higher storage overhead. For query accuracy, Figure 11(b) shows that performance of all alternatives for all α changes in a very narrow range that is very close the perfect accuracy. Yet, *VLS-γ-Avg* and *VLS-γ-Min* give similar accuracy for α ranges from 0 to 0.4 while *VLS-γ-Avg* and *VLS-β* give similar accuracy for $\alpha > 0.4$.

Figure 12 shows the effect of varying T on storage and accuracy. For small values of T, Figure 12(a) shows that *VLS-γ-Avg* encounter storage similar to *VLS-β* which is 10% less than *VLS-γ-Min*. With enlarging T (at T =12 hours), *VLS-γ-Avg* and *VLS-γ-Min* behave the same with 40% storage overhead (which means 60% storage saving). Figure 12(b) shows that for all values of T, all alternatives still come with $>$ 99% query accuracy even for small T values (at T=3 hours). The accuracy increases with increasing T value to reach almost 100% at T =12 hours.



(a) Storage Overhead (b) Query Accuracy

Fig. 11. Effect of α on storage vs. accuracy - average estimation.



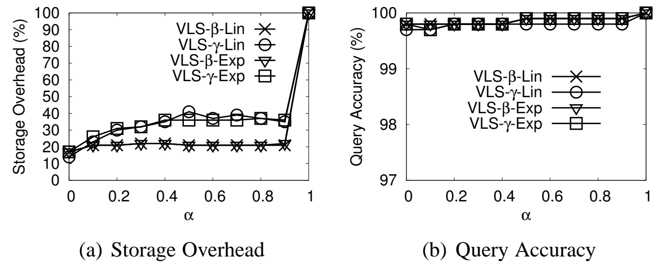
(a) Storage Overhead (b) Query Accuracy

Fig. 12. Effect of T on storage vs. accuracy - average estimation.

C.2 Top-k Ranking Additional Experiments

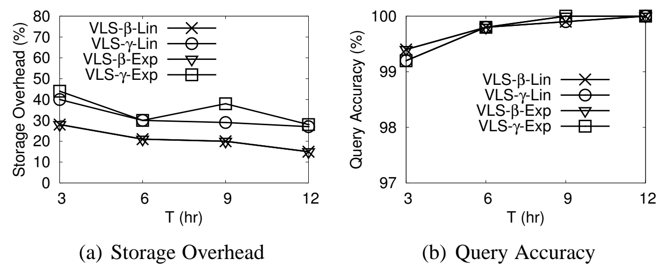
This section presents the effect of employing exponential ranking function combined with *Venus* adaptive γ-load shedding (with *min* estimation method) and adaptive β-load shedding, denoted as *VLS-β* and *VLS-γ*, respectively. This represents an extension for the results presented in Section 9.3 evaluating the effect of employing exponential ranking function on different components of the system.

As mentioned in Section 9.3.1, employing *VLS-β* and *VLS-γ* with both linear and exponential ranking functions give pretty similar performance. This is clearly shown in Figures 13 and 14. Figure 13 shows *VLS-β* and *VLS-γ* with employing both linear and exponential ranking function (denoted with suffix *Lin* and *Exp*, respectively) with different values of α. The figure shows the same performance for both ranking functions in both storage overhead (in Figure 13(a)) and query accuracy (in Figure 13(b)) for different



(a) Storage Overhead (b) Query Accuracy

Fig. 13. Effect of α on storage vs. accuracy - exponential ranking.



(a) Storage Overhead (b) Query Accuracy

Fig. 14. Effect of T on storage vs. accuracy - exponential ranking.

values of α. This basically all the analysis and conclusions drawn in Section 9.2 for *VLS*-β and *VLS*-γ with linear ranking apply for the exponential ranking as well. This performance similarity applies also for different values of T that are shown in Figure 14.

C.3 Time-based Partitioning

In this section, we show the effect of applying temporal partitioning to our index. Specifically, the idea is to partition incoming data based on their timestamp in n partitions, where each partition indexes data of $\frac{T}{n}$ time units. Each temporal partition employs a spatial index just like the one described throughout the paper. Incoming data are continuously digested in the partition that indexes the most recent data. Every $\frac{T}{n}$ time units, the currently active partition is concluded and a new empty partition is introduced to digest the new data. A partition is completely wiped from the memory when its most recent microblog is T time units old.

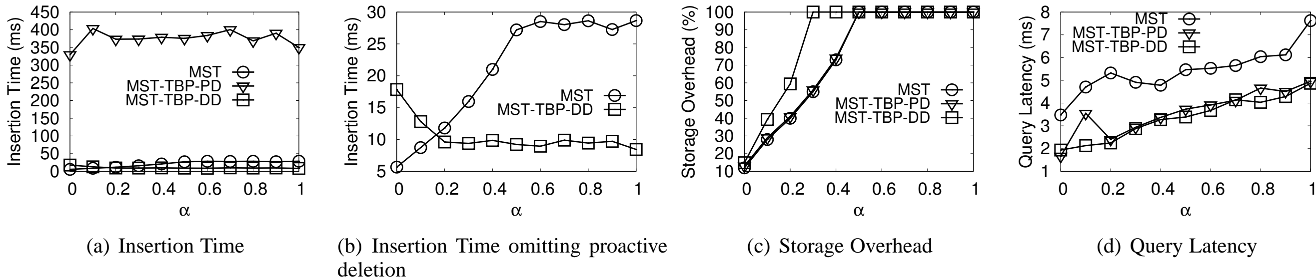
With the described temporal partitioning to our index, our insertion techniques remain the same and would be applied to currently active partition that digest incoming data. However, our deletion technique would need a small adaptation. Specifically, our deletion depends on removing useless microblogs from a certain index cell while visiting this cell to insert new data. The oldness of removed microblogs is varying from cell to another based on the density of microblogs in the spatial locality of the cell. To apply this technique to the temporally partitioned index, we need to perform two steps. First, we apply this deletion criteria to the active partition on inserting new data. Second, checking corresponding cells in older partitions to check existence of microblogs to remove. As the second step is expected to put a significant overhead, we experiment in this section two alternatives. The first alternative represents the temporally partitioned index with the deletion process employs the two previous steps, we call this *proactive deletion* as it proactively deletes any microblog that can be kicked out. The second alternative represents the temporally partitioned index with the deletion process employs only the first step to get rid of useless microblogs only in the active partition and defer deleting older microblogs to the periodic cleanup process that wipe out a complete partition every $\frac{T}{n}$ time units, we call this *deferred deletion*. We next experiment the temporally partitioned index (setting n=4) with both proactive and deferred deletion comparing it to our current technique that use only one spatial index (i.e., setting n =1). Throughout this section, our spatial index is denoted as *MST*, while the temporally partitioned index with proactive deletion and deferred deletion are denoted as *MST-TBP-PD* and *MST-TBP-DD*, respectively.

Figure 15 shows the effect of changing α on both indexes in terms of insertion time, storage overhead, and query latency. Figure 15(a) shows insertion time with different values of α. The figure shows a significant insertion overhead for *MST-TBP-PD* that leads to an order of magnitude higher insertion time due to the expensive piggybacked deletion that accesses multiple indexes to get rid of the useless microblogs. This overhead is consistent with all values of α and dominates both *MST* and *MST-TBP-DD*. Figure 15(b) omits the dominating insertion time of *MST-TBP-PD* and shows only *MST* and *MST-TBP-DD*. For all values of $\alpha \geq 0.2$, insertion overhead of *MST-TBP-DD* is less than *MST* because the insertion is performed in an index that carries only one quarter of data and hence it becomes more efficient, due to less number of index levels to be navigated. In addition, while *MST* insertion time is increasing with increasing α (as the index accumulates more microblogs and hence encounter more levels and higher insertion overhead), the insertion time of *MST-TBP-DD* decreases for α ranges from 0-0.2 and then it saturates for $\alpha \geq 0.2$. The reason of that behavior is that for small values of α, many recent microblogs are deleted and hence a lot of deletion operations are performed on the most recent partition that is accessed by *MST-TBP-DD*. However, with increasing α, more recent microblogs are kept and hence almost no deletions performed on insertion and all deletions are deferred to the periodic cleanup, which reduce the insertion overhead. This comes with a cost in storage overhead as shown in Figure 15(c). The figure shows that storage overhead of *MST-TBP-PD* is almost equals the storage overhead of *MST*, with a non-noticeable increase due to the overhead of multiple indexes storage. However, *MST-TBP-DD* comes with significant storage overhead increase that ranges from 10-50%, depending on value of α. This shows a significant overhead increase that is saved through employing one index as in *MST*. Analyzing both insertion overhead (where *MST-TBP-PD* is dominant) and storage overhead (where *MST-TBP-DD* gives significant higher values) shows that *MST* is a smart compromise that achieve good performance in both insertion overhead and storage overhead. However, Figure 15(d) shows an advantage for *MST-TBP-PD* and *MST-TBP-DD* over *MST* which is a lower average query latency. This is mainly caused by searching a lighter index segments and hence process less data to retrieve the final answer. Table 2 shows the 90, 95, and 99 percentiles of query latency of the three alternatives. We can see that all *MST-TBP-PD* and *MST-TBP-DD* percentiles are under 30 ms while all *MST* percentiles are under 50 ms.

		Query Latency (ms)		
		90%	95%	99%
$\alpha = 0$	MST	6	20	34
	MST-TBP-PD	2.2	2.9	11
	MST-TBP-DD	2.3	3.5	15.3
$\alpha = 0.2$	MST	12.3	28.8	40.8
	MST-TBP-PD	3.3	4.2	10
	MST-TBP-DD	3.3	4.3	11.5
$\alpha = 0.4$	MST	12.8	25.8	35.4
	MST-TBP-PD	5.3	7.1	18.6
	MST-TBP-DD	5	7.2	16
$\alpha = 1$	MST	19.4	33.9	47
	MST-TBP-PD	9.7	14.7	29.6
	MST-TBP-DD	10	14.5	29.3

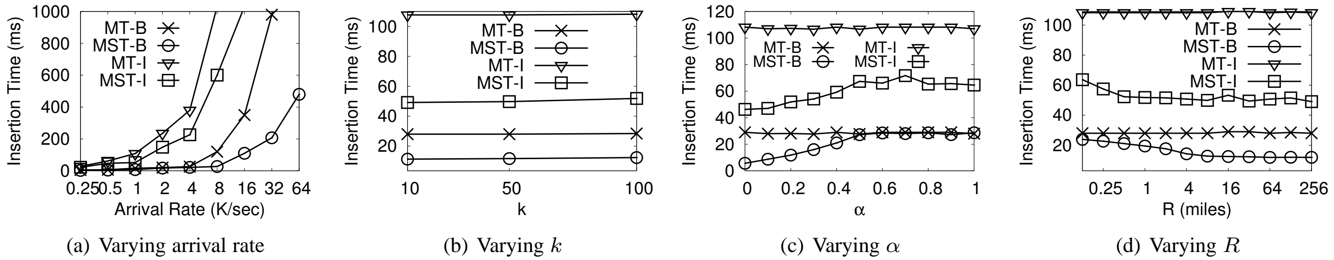
TABLE 2
Query Latency Percentiles varying α

Figure 16 shows the effect of changing k on both indexes in terms of insertion time, storage overhead, and query latency. Figure 16(a) shows insertion time with different values of k. The figure again shows a significant insertion overhead for *MST-TBP-PD* confirming the previous findings. This overhead is also consistent and dominant for all values of k. This insertion overhead comes with a storage saving as it proactively removes all useless microblogs. Figure 16(b) shows that storage overhead of *MST-TBP-PD* is almost equals the storage overhead of *MST*. However, the efficient insertion of *MST-TBP-DD* comes with significant storage overhead increase that is 100% for all values of k. This confirms that *MST* is a smart compromise that achieve good performance in both insertion overhead and storage overhead. Figure 16(c), yet, shows that *MST-TBP-PD* and *MST-TBP-DD* have lower average query latency than *MST*, for searching a lighter index segments. Table 3 shows the 90, 95, and 99 percentiles of query latency of the three alternatives. The table shows that all *MST-TBP-PD* and *MST-TBP-DD* percentiles are around 10 ms



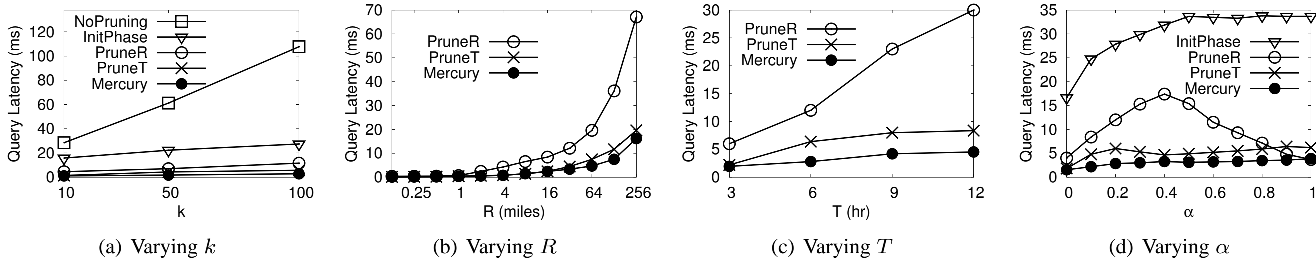
(a) Insertion Time (b) Insertion Time omitting proactive deletion (c) Storage Overhead (d) Query Latency

Fig. 15. Effect of α on temporally partitioned index.



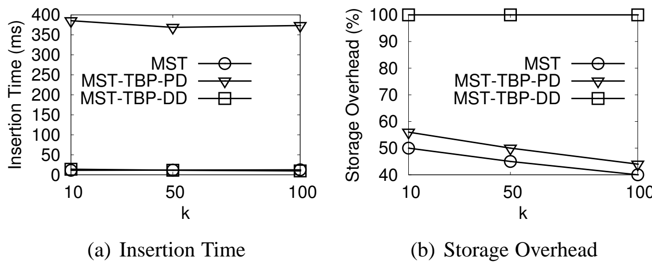
(a) Varying arrival rate (b) Varying k (c) Varying α (d) Varying R

Fig. 17. Real-time Insertion Scalability.

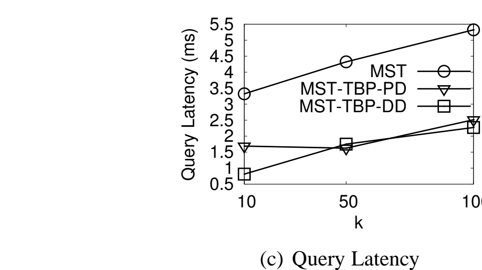


(a) Varying k (b) Varying R (c) Varying T (d) Varying α

Fig. 18. Average query latency.



(a) Insertion Time (b) Storage Overhead



(c) Query Latency

Fig. 16. Effect of k on temporally partitioned index.

		Query Latency (ms)		
		90%	95%	99%
$k = 10$	MST	4.5	12.9	19.2
	MST-TBP-PD	1.3	1.8	6
	MST-TBP-DD	1.2	1.7	6.1
$k = 50$	MST	7.7	20.2	30.5
	MST-TBP-PD	2.5	3.4	9.5
	MST-TBP-DD	2.6	3.6	8.4
$k = 100$	MST	10.5	25.3	37.8
	MST-TBP-PD	3.5	4.5	11
	MST-TBP-DD	3.6	4.4	11.4

TABLE 3
Query Latency Percentiles varying k

while all *MST* percentiles are under 40 ms for different k values.

C.4 Real-time Insertion Scalability

In this section, we show the scalability of index insertion techniques that are proposed and applied in both *Mercury* [6] and *Venus*. We show that by showing the performance of applying our bulk insertion techniques, along with lazy split/merge criteria, versus employing a non-bulk insertion technique that inserts microblogs one by one in the index. Figure 17 shows the insertion time of our bulk insertion (denoted with suffix *B*) versus inserting microblogs individually in the index (denoted with suffix *I*) for both *MT* and *MST* indexing alternatives that stores all microblogs for the whole last T time units and employ *Mercury* index size tuning module, respectively.

Figure 17(a) shows insertion time for all alternatives with varying tweet arrival rate per second. The bulk insertion techniques, *MT-B* and *MST-B*, show a significant performance boost which is four times faster insertion compared to inserting one tweet at a time through *MT-I* and *MST-I*. Specifically, *MT-B* can digest up to 32,000 tweet/second while *MT-I* cannot sustain for 8,000 tweet/second and can only handle few hundreds less than this number. Similarly, *MST-B* can digest 64,000 tweet/second in a half second while *MST-I* can sustain up to few hundreds less than 16,000 tweet/second. This shows clearly the effectiveness of *Mercury* bulk insertion techniques that reduce the amortized

insertion time per microblog and so can sustain for much higher than arrival rates.

Figures 17(b), 17(c), and 17(d) show the insertion time with varying k, α and R, respectively. In all these figures, and for different parameters values, *MT-B* and *MST-B* show a superior performance over and *MT-I* and *MST-I* with three times faster insertion time in most of the case. This supports the findings of Figure 17(a) and takes it a step further to show that *Mercury* can handles much larger amount of microblogs per second whatever the system parameters setting. This shows robustness of *Mercury* and its successor *Venus* for different query workloads.

C.5 Query Evaluation

In this section, we recall the analysis of *Mercury* query processing techniques from [6], where we contrast *Mercury* query processing with spatio-temporal pruning against: (a) *NoPruning*, where all microblogs within R and T are processed, (b) *InitPhase*, where only the *initialization* phase of *Mercury* is employed, (c) *PruneR*, where only spatial pruning is employed, and (d) *PruneT*, where only temporal pruning is employed. Figure 18(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 18(b) and 18(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 α=0.2 that favors the temporal dimension.

Figure 18(d) gives the effect of varying α 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 α, the temporal boundary of *PruneR* increases and hence it visits more microblogs inside each cell. For low values of α ($<$ 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 α. In all cases, *Mercury* takes advantage of both spatial and temporal pruning to achieve its overall performance of around 4 msec.

Appendix D
Concurrency Control

The system is adopting Single-Writer-Multiple-Reader concurrency model, where always a single thread is modifying the index while multiple threads can query simultaneously. In this appendix, we elaborate on the multi-thread contention in *Venus* for different index operations.

Insertion and deletion. While in the middle of insertion and deletion operations, new queries may arrive to *Venus*. Similarly, while a query is processed, new microblogs may be inserted or deleted. For such concurrent actions, *Venus* opt not to support transactions, but its concurrent update/insert/delete operations preserve the integrity of the index. No update is lost. However a query concurrent with multiple insert operations may read some of the new microblogs and miss some of them. After the update operations complete, any new query observes their effects completely. The rational here is that there is nothing much to lose from these concurrent operations, where the worst scenario would be that some microblogs may not make it promptly to the query answer. So, the side effect is that it may take few (milli)seconds for a microblog M to be available for search. For deletion operations, it may end up that an incoming query considers microblogs that should be deleted during the process. The worst case is that a microblog appears in the result while it is deleted. This is due to a very minor milliseconds time margin, which makes it very unlikely that a deleted microblog would score high enough to make it worthy reporting in the query answer. In general, the effect of having such concurrent operations is minimal and does not warrant employing any special concurrency control here, e.g., locking.

Splitting and merging. As described in the paper, the query processing module employs a priority queue data structure H to enqueue and dequeue pyramid cells in order. Insertion and deletion from enqueued cells in H do not pose any problems here as discussed above. However, splitting and merging in these calls may cause severe problems. For example, if a cell C is merged while in H, we will not be able to locate it when getting it out from H. To avoid such cases, *Venus* prevents any cell from being split or merged if it is in the priority queue structure of any current query. This is done through a simple pin counting technique that is incremented and decremented with every enqueue and dequeue operation from H. The side effect here is that cells may not be split or merged immediately once they are due. However, this does not cause problems as cells do not stay long in any priority queue data structure.