# GeoTrend: Spatial Trending Queries on Real-time Microblogs

Amr Magdy* Ahmed M. Aly† Mohamed F. Mokbel* Sameh Elnikety‡
Yuxiong He‡ Suman Nath‡ Walid G. Aref§

*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
†Google Inc., Mountain View, CA ‡Microsoft Research, Redmond, WA
§Department of Computer Science, Purdue University, West Lafayette, IN

*{amr, mokbel}@cs.umn.edu, †aaly@google.com,
‡{samehe, yuxhe, sumann}@microsoft.com, §aref@cs.purdue.edu

## ABSTRACT

This paper presents *GeoTrend*; a system for scalable support of spatial trend discovery on recent microblogs, e.g., tweets and online reviews, that come in real time. *GeoTrend* is distinguished from existing techniques in three aspects: (1) It discovers trends in arbitrary spatial regions, e.g., city blocks. (2) It supports trending measures that effectively capture trending items under a variety of definitions that suit different applications. (3) It promotes recent microblogs as first-class citizens and optimizes its system components to digest a continuous flow of fast data in main-memory while removing old data efficiently. *GeoTrend* queries are top-$k$ queries that discover the most trending $k$ keywords that are posted within an arbitrary spatial region and during the last $T$ time units. To support its queries efficiently, *GeoTrend* employs an in-memory spatial index that is able to efficiently digest incoming data and expire data that is beyond the last $T$ time units. The index also materializes top-$k$ keywords in different spatial regions so that incoming queries can be processed with low latency. In case of peak times, a main-memory optimization technique is employed to shed less important data, so that the system still sustains high query accuracy with limited memory resources. Experimental results based on real Twitter feed and Bing Mobile spatial search queries show the scalability of *GeoTrend* to support arrival rates of up to 50,000 microblog/second, average query latency of 3 milli-seconds, and at least 90+% query accuracy even under limited memory resources.

## Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

## Keywords

Microblogs, Trend, Spatial, Real-time, Indexing, Query Processing

This work has been supported by National Science Foundation grants, Mohamed F. Mokbel's research under Grant Numbers IIS-0952977, IIS-1218168, IIS-1525953, CNS-1512877, and Walid G. Aref's research under Grant Number III-1117766. The work has started while the first two authors are interns at Microsoft Research.



*SIGSPATIAL'16, October 31-November 03, 2016, Burlingame, CA, USA*

DOI: http://dx.doi.org/10.1145/2996913.2996986

## 1. INTRODUCTION

Timely discovering and understanding localized trending events from online microblogs, e.g., tweets, comments, and check-ins, have become a reality. In fact, news agencies and people have referred to Twitter (a prime microblogging service) to get to know timely news about various events, e.g., Michael Jackson death [35], Boston explosions [5], tracking health issues [18], and China floods [10]. This is so popular that it outstrips TV as a news source for young people [4]. As a result, Twitter has released its own feature of localized trending hashtags [39], which shows current trending hashtags in a country or a city. Following the needs and importance of such a feature, various research efforts were dedicated to online local event discovery from microblogs [1, 7, 14, 27, 36]. Unfortunately, current efforts are tailored to finding events in pre-defined areas, where one needs to first specify the areas of interest, e.g., Minneapolis, then start to detect the events in these areas. In order to have worldwide high resolution coverage of such feature, there is a real need for an event detection technique that: (a) covers arbitrary ad-hoc areas that are not pre-specified to the system, and (b) covers high resolution areas, e.g., finding events within part of the city, or events at the street level.

Up to our knowledge, there are two main attempts to support localized trend discovery with *arbitrary* spatial regions [22, 36]. However, one of these techniques ([36]) is built on two simplistic assumptions: (1) It assumes a very simplistic definition of "trending" queries as "frequent" queries, which can be computed through simple counting techniques, and (2) It assumes that the underlying system has unlimited memory. Hence, it does not account for expiring data from memory, which is crucial to ensure the accuracy of trending queries on recent data. Meanwhile, the second approach ([22]) is designed in a generic way to support trending queries for various contexts, where *location* can be considered as a context. Due to its generic nature, it has two main drawbacks: (1) It does not take advantage of the distinguishing characteristics of the spatial dimension, and (2) It is mainly designed to handle queries on arbitrarily large historical time periods, which makes it poor in handling queries on recent data in terms of both query performance and memory consumption, while recent data is the most important in discovering timely trends.

In this paper, we present *GeoTrend*; a microblogging system that supports *online trending* queries for *arbitrary ad-hoc* areas with *limited memory* resources. *GeoTrend* abstracts localized trending queries to be in the form: *"Find the top-$k$ trending keywords in the*

*last $T$ time units in area $R$"*, where $R$ is an arbitrary ad-hoc area and the *keyword* search is a proxy for trending events. *GeoTrend* adopts a wide definition of *trending* keywords that goes beyond the simple counting assumption (i.e., *frequent* keywords) to consider trending as the growth in number of appearances over the query period $T$. It is likely that trending keywords are not among the frequent ones. For example, the keyword "*love*" is consistently frequent in Twitter, and it appears much more frequent than the keyword "*elections*", while the latter is considered trending over the election week. This particular property along with the focus on supporting recent trending queries (i.e., last $T$ time units) are the main distinctions of *GeoTrend* over its main competitors [22, 36].

*GeoTrend* employs an in-memory incomplete pyramid structure [3] that is able to digest incoming real-time microblogs with high arrival rates. The incomplete pyramid hierarchy divides the entire space into a set of multi-layers cells, where cells in each layer are non-overlapping. To accommodate incoming data in limited memory resources, each index cell is equipped with a novel and efficient count aggregation technique that maintains count-based measures over the last $T$ time units and expires data that is outside $T$. Injecting the concept of *expiration* in our aggregation is a key to *GeoTrend* success, as it ensures discovering trends from only recent data and ensures continuous digestion of fresh microblogs in the limited memory. *GeoTrend* count aggregation technique distinguishes itself from all previous sliding-window counting techniques (e.g., [2, 12, 17, 25]) by its simple and efficient structure that uses low-overhead update techniques to digest/expire microblogs with high rates; up to an order of magnitude higher than Twitter rate. In particular, it uses a constant memory per keyword regardless of the length of time span $T$. This is in contrast to existing techniques that have memory overhead proportional to $T$. This enables *GeoTrend* to support arbitrarily large time spans with millions of keywords while using much less memory.

For scalable query processing, each *GeoTrend* index cell maintains a materialized list of top-$k$ trending keywords that appear within the cell spatial boundaries. Then, incoming queries with arbitrary spatial regions efficiently merge the materialized top-$k$ lists to come up with a final top-$k$ list. In system peak times with more keywords arriving within the query time $T$, *GeoTrend* employs a memory optimization technique that exploits the nature of user-generated data to smartly select and shed less important keywords that are unlikely to contribute to any incoming query.

*GeoTrend* is experimentally evaluated based on a real system deployment with a real-time feed of US tweets (collected from a Twitter firehose archive) and locations of Bing Mobile search queries. Our experiments show that *GeoTrend* digests microblogs in high rates of up to 50K microblog/second, provides average query latency of 3 milli-seconds, and achieves much less memory consumption than its competitors with 90+% query accuracy.

In the rest of this paper, Section 2 highlights related work, Section 3 introduces our trending measures, and Section 4 gives an overview of *GeoTrend*. The *GeoTrend* indexing, memory optimization, and query processing are discussed in Sections 5, 6, and 7, respectively. Section 8 presents the experimental evaluation and Section 9 concludes the paper.

## 2. RELATED WORK

Related work to *GeoTrend* spans various areas, which include: trending items in data streams, spatial queries on microblogs, and spatial aggregate queries.

**Trending items in data streams**. Discovering trending items in data streams [6, 9, 21, 31] is a well-studied topic. However, the main focus of existing techniques on the entire data stream, i.e., no support for old data expiration. Furthermore, there is no support for the spatial aspect of incoming data streams. This renders all techniques in this category not applicable for the problems addressed in *GeoTrend*, which are: spatial querying, expiring old data as a necessity for digesting new microblogs, and promoting recent data that encounter a high fraction of queries.

**Spatial queries on microblogs**. Microblog locations are recently exploited for either *visualization*, where microblogs are plotted on the map [35, 40], *geotagging*, where geotags are extracted from the microblog contents [20, 26], *modeling*, where a model is built between users, locations, and topics [19], or real-time query processing [7, 29, 36]. The last category is the most related to our work. However, none of them address discovering trending items on recent microblogs. In particular, Mercury [29] searches individual microblogs and does not support any aggregate query. GeoScope [7] addresses an interesting, yet orthogonal, problem of finding correlated <location, topic> pairs. Using GeoScope, we can support neither getting top-$k$ trending keywords as no ranking is employed nor handling arbitrary query regions as the locations are considered as a predefined discrete set, e.g., cities. Finally, AFIA [36] supports getting the top-$k$ *frequent* keywords on real-time data within arbitrary spatial regions. However, AFIA [36] techniques cannot be extended to discover trending items as they keep *only* top-$k$ frequent keywords in their index with no other information about any other keywords.

**Spatial aggregate queries**. There exists a lot of work in spatial aggregate queries, e.g., see [24, 28, 34, 37, 42], where the main focus is on building spatial index structures for disk-resident data. Aggregate information is precomputed and maintained for easy retrieval. Data is infrequently updated, and hence it is acceptable to use traditional spatial index structures without additional features for high arrival rates. Unfortunately, none of these works can support fast microblogs streams where high rates of digestion and expiration are cores issues to address.

## 3. TRENDING MEASURES

Discovering trending items in microblogs currently depends on keyword count [7, 31, 38], within a limited time period, due to its simple computations that scales for massive numbers of microblogs. However, absolute count measure does not capture *trending* items effectively. In fact, it promotes keywords that are immortally top frequent ones, e.g., *job* and *love*, while ignoring other keywords that encounter considerable increasing count over time but they are not among the top frequent ones. For example, consider two keywords *love* and *elections*. Taking their count in hourly basis, over the last three hours, *love* has appeared 1000, 1150, and 950 times, while *elections* has appeared 200, 400, and 600 times. While *love* is the most frequent, it is clear that *elections* is a trending one. Yet, depending on absolute count does not capture this.

To overcome such limitation, trending items in the broader context of streaming data [9, 21] are detected based on changes in items behavior over time. This correctly detects rising keywords even if they are not top frequent. However, existing popular measures usually include expensive computations, e.g., Singular Value Decomposition, which is not efficient to maintain incrementally. In fact, efficient incremental computations is crucial for microblogs environments scalability, so that measures are not recomputed with new arrivals of keywords that come in fast rates. For this, *GeoTrend* uses an efficient and effective measure that is based on the keyword *rate of count increase over time*. Count is easy to compute and maintain incrementally over time. So, measures that depend on count are suitable to scale in microblogs environments. *GeoTrend* can adapt several trending measures as long as each of them is based

on counting. Thus, *GeoTrend* is equipped with two measures: either *rate of count increase over time*, or *weighted count over recent time period*, introduced in Sections 3.1 and 3.2, respectively.

## 3.1 Rate of Increase Measure

Rate of count increase over time is measured using a trend line slope that is computed based on the statistical linear regression [23]. Assume the last $T$ time units are divided into $N$ equal time intervals, trend line slope gauges the increase in keyword count in recent intervals compared to the oldest interval as follows:

$$Trend_{reg} = \frac{6\sum_{i=1}^{N-1}[i \times (c_i - c_0)]}{N(N+1)(2N+1)} \quad (1)$$

Where $N$ is the number of time intervals on which the count change is gauged over the last $T$ time units. $c_i$, $0 \leq i < N$, is the count at time interval $i$, and for all $i > j$, interval $i$ is more recent than interval $j$, so that $c_0$ is the oldest counter and $c_{N-1}$ is the most recent counter. The detailed derivation of Equation 1 based on linear regression slope is shown in [22].

The value of $N$ controls the accuracy of discovering trending items, as it represents the number of counts for which the regression slope is calculated. The higher the value of $N$, the more accurate the regression output. Setting $N$=$T$ gives the highest accuracy, yet it is the most expensive computationally and memory-wise. On the contrary, setting $N$=2 is the least expensive option that divides the whole $T$ time units into two intervals, yet it provides the least accuracy and might miss the actually rising keywords.

$Trend_{reg}$ measure that is presented in Equation 1 is also efficiently maintainable in an incremental way on the arrival of new appearances of the keyword. As a new keyword appearance increase the count of just the most recent counter $c_{N-1}$, the only affected term in $Trend_{reg}$ would be $(N-1) \times (c_{N-1} - c_0)$. With increasing $c_{N-1}$ by one, this term is increased by $(N-1)$ and thus the whole $Trend_{reg}$ value is increased by $\frac{6(N-1)}{N(N+1)(2N+1)}$ (per Equation 1). In case $N$ value is fixed through the processing of a microblog stream, which is the realistic case, the increase in $Trend_{reg}$ is a constant value that guarantees efficient incremental maintenance of $Trend_{reg}$ in real-time environments.

## 3.2 Weighted Count Measure

As an extensible framework for any count-based aggregate measure, *GeoTrend* can employ *weighted count over recent time period* to detect *frequent* keywords in different spatial regions. Assume the last $T$ time units are divided into $N$ equal time intervals, keyword weighted count can be measured as follows:

$$Trend_{freq} = \sum_{i=0}^{N-1} c_i \times w^{N-1-i} \quad (2)$$

Where $0 < w \leq 1$ is a weighting parameter, and $N$ is the number of time intervals on which the count is gauged over the last $T$ time units. $c_i$, $0 \leq i < N$, is the count at time interval $i$, and for all $i > j$, interval $i$ is more recent than interval $j$, so that $c_0$ is the oldest counter and $c_{N-1}$ is the most recent counter.

$Trend_{freq}$ is an exponentially weighted sum of the $N$ counters, where recent keyword counts have higher weight than older ones. The weight of counter $c_i$ is $w^i$, where $i = (N-1)$ is the most recent time period that has the highest weight $w^0 = 1$, regardless the value of $w$. Smaller $w$ gives lower weight to older counts, and setting $w$ to 1 gives equal weights to all counts and produces total count over the last $T$ time units. Similar to $Trend_{reg}$, the value of $Trend_{freq}$ is also efficiently maintainable in an incremental way where each new instance of a keyword simply adds one to both $c_{N-1}$ and $Trend_{freq}$ values.

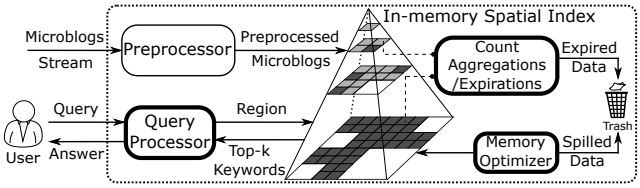



**Figure 1: *GeoTrend* Architecture.**

For presentation simplicity, we assume to maintain a single trending measure $Trend_{reg}$ (Equation 1). However, *GeoTrend* can easily maintain more than one measure simultaneously to support queries that get either recently rising keywords or absolute frequent keywords using the same indexing data structures.

# 4. *GeoTrend* OVERVIEW

This section gives *GeoTrend* system architecture (Section 4.1) and query formulation (Section 4.2).

## 4.1 System Architecture

Figure 1 gives the architecture of *GeoTrend*, which consists of a *preprocessor* and three main components: an in-memory spatial index structure that embeds *count aggregation and expiration* module, a *memory optimizer* module, and a *query processor* module.

**Preprocessor**. Each incoming microblog first goes through a preprocessor that extracts its timestamp, location, and keywords. A microblog location could be the latitude and longitude coordinates associated with the incoming microblog, if available, or be the location associated with the user profile who issued the microblog. Keywords are taken from hashtags associated with microblogs, if present, or a random word of its text.

**In-memory Index**. The preprocessed microblogs are digested, with high arrival rates, in the in-memory spatial index. The index divides the space into multiple levels, each level consists of a set of non-overlapping cells. Each index cell is equipped with efficient *count aggregation and expiration* module that maintains trending measures for the cell's keywords over the last $T$ time units. So, any data that is older than $T$ is expired and thrown out of memory. Details of indexing are presented in Section 5.

**Memory Optimizer**. In case of scarce memory resources, the *memory optimizer* module is invoked on all index cells to shed keywords that are less likely to contribute to query answers. This saves significant memory space while keep highly accurate queries. Details of memory optimization are presented in Section 6.

**Query Processor**. Users post their queries to the *query processor* module, that efficiently exploits the index materialized aggregate measures to return query answers to the users. Details of query processing are presented in Section 7.

## 4.2 Query Formulation

*GeoTrend* users can post queries in the form "*Find the most trending keywords within a spatial region R*". Internally, the system beefs up this query with three parameters: (1) $k$; the number of keywords to be returned, (2) a time span $T$; where the trending keywords should be posted within the last $T$ time units, and (3) a trending measure $Trend$; where the returned $k$ keywords should be highest ranked based on $Trend$. The query answer is then retrieved based on precise point locations that are extracted from microblogs through a pre-processing step (as highlighted in Section 4.1). Formally, *GeoTrend* query is defined as follows:

**Query Definition:** *Given an arbitrary spatial region $R$, an integer $k$, a time span $T$ time units, and a trending measure $Trend$,*

GeoTrend *finds $k$ keywords such that: (1) The $k$ keywords are posted within the region $R$. (2) The $k$ keywords are posted within the last $T$ time units. (3) The $k$ keywords are the highest ranked based on $Trend$ measure among keywords that are posted within $R$ and $T$.*

Our query limits its answer size to $k$ as a natural consequence for the plethora of keywords that come with microblogs, which calls to selectively provide end users with the most relevant results (top-$k$ items) based on a certain ranking function. In fact, for the same reason, all research efforts on microblogs are limiting their answer size to $k$ [7, 8, 29, 36, 41] to be useful for end users. Furthermore, our query retrieves its answer from only *recent* keywords that are posted within the last $T$ time units. This basically promotes real-time nature of microblogs as a first-class citizen, which is a distinguishing property for nowadays microblogging services, to discover trends that are happening *now* on social media websites.

Upon initialization, a system administrator sets default values for parameters $k$, $T$, and $Trend$. Users may still change the default values of $k$ and $T$, yet a query may have less performance if the new values consider larger search space than the default values. Optimizing the index performance for a pre-set parameter values is a common design choice for major web services. For example, Twitter gives the most recent $k$ tweets to a user, where $k$=10, and so in a keyword search result. If a user would like to get more than $k$ results, an extra query response time will be paid on demand.

# 5. *GeoTrend* INDEXING

*GeoTrend* employs a spatial pyramid index [3] to efficiently support queries in arbitrary spatial regions. The index divides the space into multi-layers cells of different spatial granularity, where each layer consists of a set of non-overlapping cells. For each incoming microblog in real time, *GeoTrend* stores *only* its keywords and their aggregate information rather than the microblog itself. To support fast digestion of microblogs fast streams and low query latency, the index is wholly resident in main-memory. However, main-memory resources are limited and cannot accommodate microblogs aggregate information for infinite time. Consequently, *GeoTrend* limits its index contents to aggregate information of data that arrived only in the last $T$ time units, where old data that is outside the time span $T$ is expired. The length of window $T$ depends on the available memory resources, and typical values ranges from several hour to few days of microblogs data. Indexing data of multiple days, that have hundreds of millions of microblogs, is feasible as the index does not store individual data records, but stores only aggregate information. Yet, such fast data rates impose scalability challenges on both index insertion and expiration.

Insertion and expiration in microblogs environments are so challenging that residing in main-memory is not enough to scale for microblogs high arrival rates. In particular, inserting keywords can be performed in a traditional way [3], where each new keyword is traversing the pyramid structure from its root cell passing by all intermediate cells reaching the leaf cell that includes its point location, to update cells contents. However, this is expensive given the large number of keywords that arrive every second in microblogs. To overcome this, *GeoTrend* employs a bulk insertion technique that reduces the insertion cost so that it scales for digesting high arrival rates. Similarly, expiring contents from *GeoTrend* index should be ideally performed in similar rates like insertion so that the index storage is stable in the system steady state. With large number of cells and high data rates, proactive expiration that iterates all index cells is very expensive and put significant overhead on the index performance. To scale, *GeoTrend* employs a lazy expiration that dramatically reduces the expiration cost.

The rest of the section presents the index structure (Section 5.1), insertion (Section 5.2), and expiration (Section 5.3).

## 5.1 Index Structure

*GeoTrend* pyramid index structure is similar to a partial quad tree and consists of a single root cell that represents the entire geographic area, level 1 partitions the space into four equi-area disjoint cells, and so forth. As a partial tree structure, any index level could have both leaf and intermediate cells. Figure 2 depicts an instance of *GeoTrend* pyramid index. The figure shows a partial pyramid that divides the space into three levels, where light gray cells indicate intermediate cells, dark cells indicate leaf cells, and white cells replace areas that are not actually maintained at that level. The pyramid shape is determined based on the spatial distribution of microblogs, through its *index shaping* process. Then, *index real-time operation* is started in which the index continuously digests incoming real-time data.

**Index shaping**. This is a one-time process that determines how pyramid cells are divided to cover the space at different levels of granularity. Areas with dense data distribution are divided into smaller cells at deeper levels of the pyramid. On the contrary, areas with sparse data are divided into large cells that span only one or two pyramid levels. To determine the pyramid shape, we insert a sample of one day microblogs so that any cell stores maximum number of microblogs, called cell's $Capacity$. $Capacity$ has been chosen experimentally to range from 1000 to 2000 for fine granular space division. Any cell contains more than $Capacity$ is divided further into four disjoint children cells. Once there are no more cell divisions, all the individual microblogs are wiped and the shaping process is concluded.

**Index real-time operation**. After its shaping, the index then starts to continuously receive real-time data and stores only aggregate information about incoming keywords rather than storing individual microblogs. Keyword aggregate information are stored in both leaf and intermediate cells, so that information of the same keyword are aggregated at different levels of spatial granularity. Each index cell $C$, both leaf and non-leaf cells, stores four data structures: a hash table $H$, a sorted list $TopK$, a rotating pointer $p$, and a timestamp $t_{last}$, described below:

**Hash table $H$.** Each hash entry $h \in H$ represents a single keyword arrived to cell $C$ in the last $T$ time units. With each hash entry $h$, we maintain the following:

1. A set of $N$ counters, $c_0$ to $c_{N-1}$. The $N$ counters divide the time window $T$ into a **set of equal temporal intervals**, each of $\frac{T}{N}$ time units. Each counter maintains the number of times that the hash entry $h$ has appeared in its corresponding $\frac{T}{N}$ time units. $N$ is a system parameter that trades query accuracy with computation efficiency as discussed in Section 3. A larger value of $N$ gives more accurate results, yet, it comes with processing and storage overhead in maintaining more counters.
2. A trending value $Trend$ that is calculated based on hash entry $h$ counters $c_i$'s according to Equation 1.

**List $TopK$.** A sorted list of size $k$ that maintains the top-$k$ trending keywords in this cell ranked based on the trending value $Trend$. This is mainly to materialize the top-$k$ answer of this cell to speed up the query processing significantly.

**Rotating pointer $p$.** An integer value, in the range of 0 to $(N-1)$, that points to the current (i.e., most recent) counter. Thus, the most recent counter is $c_p$ and the oldest counter is $c_{(p-1)\%N}$. Maintaining $p$ saves huge efforts in shifting the counter values and expiring old counters every $\frac{T}{N}$ time units as discussed in Section 5.3.

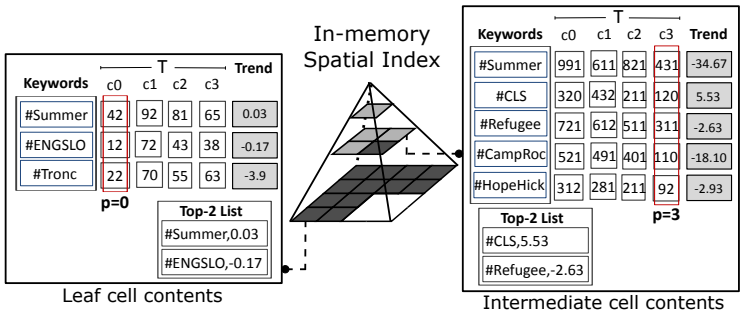
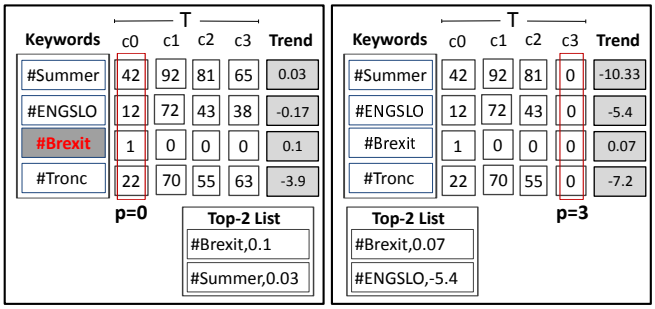



**Figure 2: *GeoTrend* index structure and cell contents.**




(a) Index Insertion (b) Index Expiration

**Figure 3: Example of *GeoTrend* index insertion and expiration.**

**Timestamp $t_{last}$.** The starting timestamp of the time interval of the last expiration of $C$ contents, where it is used to decide which counters need to be expired in the following expiration cycle.

Figure 2 shows the contents of two index cells, one intermediate cell and one leaf cell. Both cells encloses exactly the same data structures. The intermediate cell encounter more keyword arrivals as it lies one level higher than the leaf cell, and so it covers four times larger space area. The intermediate cell in Figure 2 contains five hashtags, *Summer*, *CLS*, *Refugee*, *CampRoc*, and *HopeHick*, each maintains four counters, $N$=4, and $Trend$ value. It also maintains a top-2 list sorted based on $Trend$ value and an integer pointer $p$=3. The leaf cell in Figure 2 contains three hashtags, *Summer*, *ENGSLO*, and *Tronc*, which also maintain four counters per hashtag and a top-2 list. So, $N$ and $k$ values are fixed for all index cells. Yet, its integer pointer $p$=0, which is a different value than the other cell, as $p$ value is updated on data expiration, which happens at different time based on the cell, as discussed in Section 5.3.

## 5.2 Index Insertion

To reduce the index update cost and scale for digesting high arrival rates, *GeoTrend* spatial index employs an efficient bulk insertion technique that saves thousands of comparison operations for keyword locations with spatial cell boundaries compared to the traditional way of inserting individual data records. The bulk insertion process consists of two steps: (1) traversing pyramid index cells with batches of keywords, and (2) while traversing, keywords are inserted in their corresponding cells. Each step is described below.

**Pyramid traversal**. To reduce the pyramid traversal cost, the incoming keywords are buffered for $t$ seconds before being inserted in bulk. $t$ represents a trade-off between the insertion overhead and the delay between a microblog arrival and being available to search results. Typical values of $t$ is 1-2 seconds which is an acceptable delay for real-time applications, and still sufficient to collect several thousands of keywords to insert as a batch. For example, Twitter receives around 12,000 tweets every 2 seconds, which is a reasonable batch size that saves significant insertion cost. During the buffering, a spatial minimum bounding rectangle (MBR) is maintained around point locations that are associated with the keywords. We then traverse the pyramid levels through comparing the MBR boundaries, instead of locations of individual microblogs, and insert keywords in their corresponding cells.

The buffered keywords are first inserted in the root cell $C$, as shown in *cell insertion* below. If $C$ is not a leaf cell, the new keywords are recursively inserted in $C$'s children cells. The new keywords are divided based on their locations into four MBRs, each MBR encloses a subset of the keywords that corresponds to one of the children cells. Then, the same *cell insertion* process is applied to each of the children cells. This leads to replicating all keywords across all index levels. Such replication significantly reduces the query latency for large query areas as it minimizes number of processed cells for large query regions. On another hand, it increases both index insertion time and memory consumption. Our experiments study the impact of this replication on indexing overhead, query processing, and memory consumption.

**Cell insertion**. On the arrival of new keywords to any cell $C$, two steps are performed: (1) inserting the new keywords in the hashtable $C.H$, and (2) updating the list $C.TopK$ that maintains $C$'s top-$k$ keywords.

(1) *Insertion in hashtable $C.H$.* For each newly arrived keyword, if there is no corresponding hash entry in the hashtable $C.H$, it is added to $C.H$ with zero-initialized $N$ counters and $Trend$ value. Then, regardless of whether there was a prior hash entry or not, its most recent counter $c_p$ is incremented by one, which leads its $Trend$ value to be incremented by $\frac{6(N-1)}{N(N+1)(2N+1)}$ (per Equation 1). Such constant increment to $Trend$ value makes it very efficient to maintain it incrementally as discussed in Section 3.

(2) *Updating list $C.TopK$.* For each new keyword inserted in $C.H$, we check its $Trend$ value to update $C.TopK$ list, if needed, so that it keeps maintaining the most trending $k$ keywords in $C$. If $C.TopK$ has less than $k$ keywords, the new keyword is inserted in $C.TopK$ directly. Once $C.TopK$ has $k$ keywords, the $Trend$ value of each new keyword is compared to $Trend_{min}$: the lowest trending value in $C.TopK$. If the new keyword's $Trend$ is larger than $Trend_{min}$, then it is inserted in $C.TopK$ replacing the keyword that corresponds to $Trend_{min}$.

**Example**: Figure 3(a) shows an example for index insertion. The figure shows the content of the leaf cell shown in Figure 2 after inserting hashtag *Brexit*. As the hashtag is not previously present in the cell, a new entry is added to the hashtable $H$ with zero-initialized counters. Then, the most recent counter, $c_0$, is incremented and $Trend$ value is computed. As the new $Trend$ value is eligible for the top-2 list, the hashtag *Brexit* is inserted into the list.

## 5.3 Data Expiration

As *GeoTrend* index limits its contents to data of the last $T$ time units, it needs to periodically expire old data that is outside the time span $T$. Thus, every $\frac{T}{N}$ time units, *GeoTrend* should hold on inserting new data, iterate over *all index cells*, and expire the old contents. However, this causes a significant interruption for index real-time insertion and terribly reduces its digestion rates. To prevent such interruption, *GeoTrend* skips such an expensive expiration that expires *all cells at once* and employs a lazy expiration technique that postpones expiring any index cell contents until: (1) either an insertion occurs in this cell, or (2) a query comes to this cell and hence an expiration is necessary so as not to consider old data in the query answer. In both cases, expiration is necessary, and performed, only in a *single* cell. This minimally interrupts real-time insertion of *GeoTrend* index as it expires *only one cell at a time*, and even consumes no index traversal cost as it piggybacks this cost on either insertion or query processing. The effect of putting this overhead on query response is minimal as expiration is performed once and it pays off for all incoming queries. However, this lazy expiration does not guarantee to expire all old contents. In fact, cells that encounter neither insertions nor queries during the $T$ time units, e.g., low dense spatial regions like suburbs, would keep very old contents. To overcome this, *GeoTrend* runs an additional cleaning process, every $T$ time units, that is very light and efficient, so that it does not put an overhead on the index performance. Both lazy expiration and periodic light cleaning are described below.

**Lazy expiration**. The contents of a cell $C$ is expired only if it is last expired more than a complete period of $\frac{T}{N}$ time units ago. This is checked through $C.t_{last}$ timestamp, that is the starting timestamp of the period when $C.H$ is last expired. If $n_c = \left\lfloor \frac{NOW - t_{last}}{T/N} \right\rfloor \geq 1$, then the oldest $n_c$ counters need to be expired and $C.t_{last}$ is updated to be $t_{last}$=$t_{last}$+$n_c \times \frac{T}{N}$. For presentation simplicity, assumes $n_c$=1, i.e., we expire only the oldest counter. Then, the oldest counter $c_{(p-1)\%N}$ should expire for *all entries* in the hashtable $C.H$. This requires to set the value of $c_{(p-1)\%N}$ to zero, the value of pointer $p$ is decremented to be $p = (p-1)\%N$, and the aggregate $Trend$ value is recomputed. This is repeated $n_c$ times when $n_c$ >1.

Maintaining $p$ saves huge efforts in expiration. A traditional way is to shift the counter values for each hash entry. With $p$, we keep all counter values intact in their positions, and we just shift left (i.e., rotate) the value of $p$ to replace the oldest expiring counter with a new one. With this, it is always the case that counter $c_p$ represents the current $\frac{T}{N}$ time units while counter $c_{(p-1)\%N}$ represents the oldest $\frac{T}{N}$ time units within the time span $T$.

Expiring the contents of hashtable $C.H$ leads to invalidating the contents of $C.TopK$ list. Thus, $C.TopK$ is recomputed with each expiration of $C.H$ contents. However, recomputing $C.TopK$ list comes with a very little overhead on the lazy expiration process. While updating $Trend$ value of each hash entry $h$, $h$ is considered as a potential candidate for $C.TopK$. If $C.TopK$ has less than $k$ keywords, then $h$ is inserted in $C.TopK$ right away. If $C.TopK$ has $k$ keywords, then $h.Trend$ is compared to $Trend_{min}$: the lowest trending value in $C.TopK$. If $h.Trend$ is larger than $Trend_{min}$, then it is inserted in $C.TopK$ replacing the keyword that corresponds to $Trend_{min}$. This repeats for each hash entry $h$ while its counters are updated.

**Example**: Figure 3(b) gives the contents of Figure 3(a) cell after $c_0$'s time period expires. In this case, (a) $c_0$ is concluded, (b) the oldest counter $c_3$ would expire its old values and reset to zero for all keywords, (c) the current pointer $p$ becomes 3 as $c_3$ becomes the current active counter, and (d) $Trend$ values are recomputed based on the new counter positions, where $c_2$ is the oldest counter. Meanwhile, the top-2 list is recomputed, based on the new $Trend$ values, to include *Brexit* and *ENGSLO* keywords.

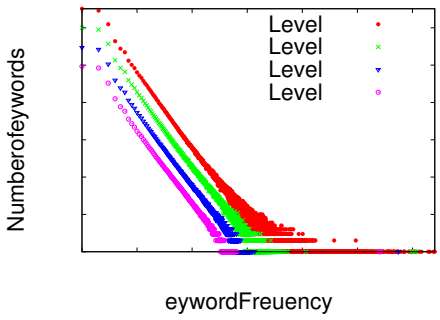



**Figure 4: Zipf distribution of Twitter keywords at different spatial levels.**

**Light cleaning**. To account for sparse cells that rarely encounter insertions and queries, and hence do not encounter any lazy expiration, we run a light periodic cleaning. Every $T$ time units, a light expiration process is traversing *all* index cells. If the cell is last expired older than $T$ time units ago, then all cell contents are wiped, otherwise, nothing is done. This process intentionally overlook contents that is within the last $T$ time units but still old enough to be expired, i.e., older than $\frac{T}{N}$ time units ago. This is intended to make it very light and efficient, while this contents are left for the next lazy or periodic expiration in the cell. Although some cells would contain unneeded contents for $T$ time units, practically this does not cause much overhead as they are very sparse cells. As the light cleaning process wipe all cells contents, so no $TopK$ update is needed as $TopK$ is wiped as well.

# 6. MEMORY OPTIMIZATION

As *GeoTrend* index is wholly resident in main-memory, it might be the case that during peak times, e.g., local events in major cities, available memory resources are limited to store the vast amount of incoming data. In that case, some applications are willing to remove a portion of memory contents that minimally affects query accuracy, still sustain the system real-time performance in peak times. Thus, *GeoTrend* employs a main-memory optimization technique, called *TrendMem* that reduces memory footprint significantly while keep query answers highly accurate. *TrendMem* is based on a key observation that identifies a very interesting spatial property for microblogs data. Such property is used to smartly identify victim data to expel from main-memory without sacrificing the query accuracy. In the rest of this section, Section 6.1 presents the key observation and key idea behind *TrendMem*. Then, Section 6.2 presents the details of *TrendMem* realization inside *GeoTrend* index.

## 6.1 *TrendMem* Key Ideas

**Key observation**. Memory optimization in *GeoTrend* takes advantage of the observation that keywords popularity in microblogs follows a Zipf distribution [11, 16, 32, 33], i.e., small percentage of keywords appear with high frequency while the majority of keywords appear very few times. Interestingly, Zipf distribution holds not only for the entire microblogs collection over the entire world, but also over those appearing in smaller spatial regions. We demonstrate such interesting property in Figure 4 that shows the frequency

distribution of millions of real tweets at four different levels of spatial granularity (Level 1 is the entire USA, Level 2 is the four quarters of the USA, and so on). The figure shows that majority of keywords in the Twitter stream are infrequent across all levels of spatial granularity. Such majority of infrequent keywords consume large percentage of the memory for their counters. Yet, our *TrendMem* technique exploits the existence of such infrequent keywords in a smart way to identify a subset of them that are very unlikely to contribute to trending query answers. This subset is shed from main-memory without hurting the accuracy of query answers.

**Key idea**. The key idea of our *TrendMem* technique that some keywords with low frequency are unlikely to be trending ones. Those keywords must satisfy a crucial condition: they must encounter low frequency *in all sub-intervals of the last $T$ time units*. This condition is sufficiently working as it judges count change over time, which is the same as our trending measures (Section 3). To elaborate, if we decide on a keyword importance only through its total count during the last $T$ time units, it might be the case that a keyword encounter low total count, yet, its count is rising significantly over time. Thus, we may end up removing trending keywords from main-memory. However, if we ensure that the keyword count is low in all the sub-interval of the last $T$ time units, then it is very unlikely that *growth of count* of this keyword makes it a potential trending one. Then, it is unlikely to contribute to query results and it can be removed without affecting the query accuracy.

## 6.2 *TrendMem* Technique

**Main idea**. In each cell $C$ in *GeoTrend* index, *TrendMem* periodically removes keywords that are $\epsilon$-infrequent in *all the $N$ time intervals of the last $T$ time units*. $\epsilon$-infrequent keyword is a keyword that has count less than $\epsilon \cdot n$, where $\epsilon$ is a small fraction, e.g., 0.001, and $n$ is the total number of keyword arrivals in cell $C$ in the corresponding time interval. For example, if $C$ received total of $n_i$ keyword arrivals during time interval $i$, $0 \leq i \leq (N-1)$, then a keyword $W$ is considered $\epsilon$-infrequent if its counter $c_i < \epsilon \cdot n_i$, for all $0 \leq i \leq (N-1)$. Removing infrequent items from a cell $C$ is invoked every $\frac{1}{\epsilon}$ insertion cycles in $C$. This ensures to limit the size of the hashtable $C.H$ to $O(\frac{1}{\epsilon}\log(\epsilon \cdot n))$ entries (inspired by the same ideas presented in LossyCounting algorithm [30]). Also, any keyword with total count $> (\epsilon \cdot n)$ at any sub-interval of $T$ is guaranteed to be maintained. In fact, checking a keyword to be infrequent in each of the $N$ sub-intervals independently ensures the consistency of the keyword infrequency along the whole time window $T$ and thus guarantees not to expel any possibly trending keywords as discussed in Section 6.1. In addition, employing a percentile threshold $\epsilon$, which means keyword importance is identified based on a percentage of frequencies of its neighbor keywords within the spatial locality. This guarantees that dense spatial areas do not affect suburb areas and leads to maintain an accurate top-$k$ keyword list in each spatial locality. This makes *TrendMem* provides highly accurate query answers.

**Impact on the index**. To realize *TrendMem* inside *GeoTrend* index, two main operations are added to the *index insertion*: (1) periodic cleaning of infrequent keywords inside each cell every $\frac{1}{\epsilon}$ insertion cycles in the cell, and (2) checking on $\epsilon$-infrequent keywords in each sub-interval to decide on removing which keywords. To this end, each index cell maintains an insertion cycles counter that is initialized to zero. The counter is incremented by one with every insertion in the cell. Once it reaches $\frac{1}{\epsilon}$, the cleaning procedure is triggered and the counter is reset to zero. The cleaning procedure goes through a complete scan for all hash entries in hashtable $H$ and removes any keyword that is consistently infrequent during all the $N$ intervals. To check for the keyword infrequency in each sub-interval independently, each *cell* maintains additional $N$ counters $n_i$, $0 \leq i \leq (N-1)$, that keep the total number of keyword arrivals in each of the $N$ sub-intervals of the time window $T$. Thus, with each insertion to the cell, the counter of the current interval is incremented by the number of new keywords. Using this, the infrequency check is then performed very cheap by comparing $\epsilon \cdot c_i$ of each keyword counters to the counter $n_i$, for all $0 \leq i \leq (N-1)$.

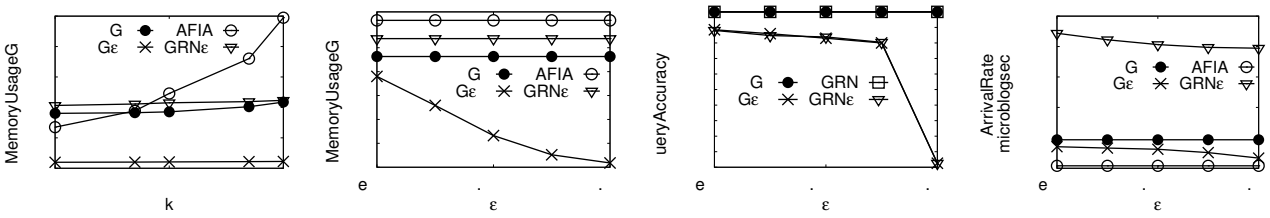
A typical value of $\epsilon$ would be around 0.001, which is considered large enough to limit the memory footprint without really affecting the accuracy of the query result. Although introducing $\epsilon$ saves significant storage, apparently, executing the periodic cleaning procedure incurs additional computational overhead during the index insertion operation. Since we adjust the triggering of our cleaning procedure to be every $\frac{1}{\epsilon}$ insertions, a lower value of $\epsilon$ implies less frequent cleaning, i.e., less insertion overhead and less storage saving, but higher query accuracy. For example, when $\epsilon$ is 0.01, we run the cleaning procedure every 100 insertions. Yet, when $\epsilon$ is 0.0001, we perform the cleaning every 10,000 insertions, which is cheaper in computation cost, achieves higher query accuracy, but consumes more memory. Our experimental evaluation studies in details the effect of varying $\epsilon$ on the insertion overhead, storage saving, and query accuracy, showing that *GeoTrend* can provide a reasonable compromise that achieves excellent performance for all them.

# 7. QUERY PROCESSING

This section discusses query processing in *GeoTrend*. As *GeoTrend* index already materializes top-$k$ items in each spatial cell, processing top-$k$ queries is simple, efficient, and provides low response time. In fact, *GeoTrend* query processing depends on getting top-$k$ keywords in the query region $R$ by manipulating *only* the top-$k$ lists that are maintained in the index cells that overlap with $R$. Our hypothesis is that it is highly unlikely that a keyword that did not make it to any of the top-$k$ lists in any cell would make it to the final answer. The main reason is that our trending measures are additive (per Equations 1 and 2), which means the trending value of a certain keyword $W$ over an arbitrary region $R$ equals the summation of $W$'s trending values in all index cells that overlap with $R$. Thus, top-$k$ items within each cell have much better chances to be the global top-$k$ items in $R$. This hypothesis is supported empirically by our experimental results, where the vast majority of queries can get the true top-$k$ trending keywords in $R$ from the ones that appear in any top-$k$ list.

*GeoTrend* query processing is composed of two main steps. In the first step, *GeoTrend* finds a set of pyramid index cells $S$ that cover the query spatial region $R$ in a way that minimizes the number of cells in $S$ while maximizes the coverage ratio with $R$. In the second step, it finds the top-$k$ keywords in $R$ by aggregating the values from only top-$k$ lists that are maintained in $S$ cells. Details of the two steps are described below.

Step 1 takes the query spatial region $R$ and the root cell of the spatial pyramid index as input and outputs a set of cells $S$ that completely cover $R$, such that: (a) the number of cells in $S$ is minimal, which reduces the aggregation cost in Step 2, and (b) the cells in $S$ have the highest overlap ratio with $R$, which maximizes the accuracy of the retrieved results. We define the overlap ratio between a cell $C$ and the query region $R$ as the area of the part of $C$ that is inside $R$ divided by the area of $C$, i.e., $\frac{C \cap R}{C}$. Starting at the pyramid root cell, we recursively visit the children overlapping with $R$. A cell $C$ is added to $S$ if *one* of the following two conditions is satisfied: (1) $C$ is a leaf cell, or (2) $C$ is completely inside $R$, i.e., overlap ratio of 100%. In both cases, we know that $C$ has the best covering area which is the same coverage we can get from $C$'s children. So, to minimize the number of cells in $S$, we just add $C$, and




(a) Memory usage vs. k (b) Memory usage vs. $\epsilon$ (c) Query accuracy (d) Supported arrival rate

**Figure 5: Impact of memory optimization module.**

skip all its children. Otherwise, we visit children cells applying the same procedure.

Step 2 takes the set of cells $S$ from Step 1 as input and produces the final answer of the top-$k$ keywords that appear in $S$. In this step, we only consider keywords that have appeared in at least one top-$k$ list of all the cells in $S$. Following the spirit of Fagin's TA algorithm [13], the main idea of this step is to employ a max-heap priority queue, initiated by the top item in each list in $S$. The key of the priority queue is the trending value. Then, we keep extracting items from the queue one by one. For each extracted item $Top$, we do the following: (1) We compute the total trending value of $Top$ as the sum of its values in all cells in $S$. (2) If the total value of $Top$ is among the highest $k$ found so far, we update our final answer accordingly. (3) We replace $Top$ in the priority queue by the next item in the top-$k$ list of its cell, if any. This is repeated until either exhausting all top-$k$ lists in $S$ or the maximum possible total value for any remaining keyword is less than the $k^{th}$ entry in the current final answer. This maximum value is upper bounded by the summation of the existing keys in the max-heap.

# 8. EXPERIMENTAL EVALUATION

This section evaluates *GeoTrend* experimentally. We compare *GeoTrend* with AFIA [36] and GARNET [22], which are the state-of-the-art and the closest to our work in the literature. Our AFIA implementation uses two spatial grid levels of granularity of $1km \times 1km$ and $10km \times 10km$, and four levels of temporal resolution, hour, day, week, and month. GARNET [22] is primarily proposed for queries of any generic context, where we instantiated context as *location* to use a one-level spatial grid index of resolution $10km \times 10km$ per cell. We use GARNET memory components and limit our evaluation to its in-memory performance, which is the main focus of *GeoTrend* queries and system components. With our comparison to competitor systems, we also evaluate different design choices and modules of *GeoTrend*, including memory optimization technique, replicating keywords across index level, and materializing top-$k$ list at indexing time.

The rest of this section organized as follow. Section 8.1 presents experimental setup. Section 8.2 evaluates memory overhead of different alternatives and its effect on index scalability. Finally, Section 8.3 evaluates query processing.

## 8.1 Experimental Setup

Our experiments are based on a real *GeoTrend* system prototype implemented in C# as a multi-threaded server that uses latches for concurrency control. *GeoTrend* is deployed on a server running Windows Server 2012 on Intel 2.40GHZ Core i7 CPU with 64GB RAM. We use 152 million geotagged tweets obtained from the Twitter Firehose archive. The tweets are used to simulate an incoming stream of microblogs with high arrival rates. Each tweet is associated with a point location (latitude and longitude). For keywords, we use hashtags (if present) or select a random word from the tweet text. For queries, we use a query log from Bing Mobile containing actual point locations (latitudes and longitudes) of user search queries on Bing. This query log is used to compose a default query load of 1000 MBR queries (centered around the point locations) with different area sizes that range from $4mi^2$ to $400Kmi^2$, containing 15% with large areas ($40Kmi^2$ to $400Kmi^2$). Unless mentioned otherwise, the default value of $k$ is 100, $N$ is 8 counters per hash entry, $T$ is 24 hours, and $\epsilon$ is 0.001.

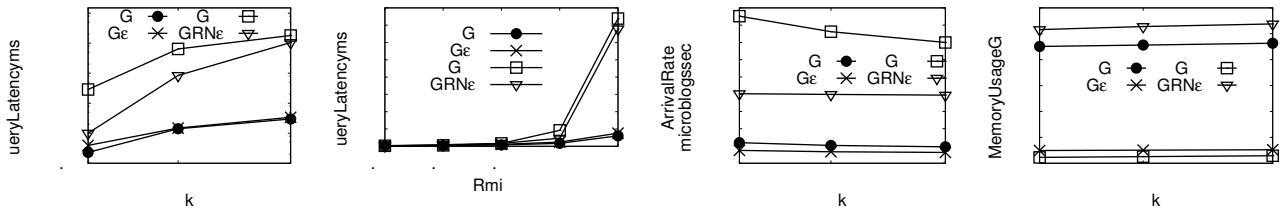
All experimental results are collected during steady state after running the system for at least $T$ time units. All measurements are done in real time, i.e., while the tweet stream is flowing. Our main performance metrics are the supported microblogs arrival rate, memory overhead, query latency, and query result accuracy. Accuracy is calculated as the percentage of entries in the received result that are included in the correct top-$k$ answer computed with infinite resources.

## 8.2 Memory Consumption

Figure 5 shows the memory usage of different techniques studying the impact of memory optimization module on both index scalability and query accuracy. We evaluate the *GeoTrend* pyramid index with and without employing the memory optimization module (denoted as *GT*-$\epsilon$ and *GT*, respectively). We also compare with AFIA [36] (denoted as *AFIA*) and GARNET [22] with and without employing its $\epsilon$ memory cleaning process (denoted as *GRN*-$\epsilon$ and *GRN*, respectively). GARNET $\epsilon$-cleaning process is similar to *GeoTrend* $\epsilon$-cleaning.

Figures 5(a) and 5(b) depict the memory usage for different values of $k$ and $\epsilon$, respectively. For different values of $k$ (Figure 5(a)), only *AFIA* memory usage increase significantly while the rest of technique encounter relatively stable memory usage. The highest *AFIA* memory usage (at $k$=1000) is around 24GB excluding programming language overhead. Such large overhead comes for two reasons. With increasing $k$, the number of items in archived dynamic summaries are increasing significantly and hence it consumes more memory. In addition, such dynamic summaries are replicated in multi-resolution over both spatial and temporal dimensions per its index structure. This even amplifies the effect of increasing $k$ and encounter high memory consumption. Both *GT*-$\epsilon$ and *GRN*-$\epsilon$ encounter nearly 40% of *AFIA* memory. Yet, *GT*-$\epsilon$ can significantly improves this and consumes less than 10% of *AFIA* memory. The amount of memory saving is actually changing with different $\epsilon$ values as Figure 5(b) shows. This figure shows memory usage of *GT*-$\epsilon$ is reducing dramatically with increasing $\epsilon$ as more keywords are removed from all index cells. However, *GRN*-$\epsilon$ consumes relatively high memory due to the large number of cells it maintain. Also, $\epsilon$ value does not have significant effect on its memory overhead as its spatial cell size is much smaller, then each cell receives much less keywords and so $\epsilon$ removes relatively stable amount of keywords.

The effect of reducing memory overhead is shown in Figures 5(c)




(a) Query Latency vs. k (b) Query Latency vs. R (c) Supported Arrival Rate (d) Memory Overhead

**Figure 6: Impact of keyword replication across pyramid index levels.**

and 5(d) on query accuracy and supported arrival rates of incoming microblogs. *AFIA* is not included in query accuracy as it support only top-$k$ frequent queries and cannot adapt our trending measure. For different values of $\epsilon > 0.01$, query accuracy exceeds 90% for both *GT*-$\epsilon$ and *GRN*-$\epsilon$. In Figure 5(d), *GRN*-$\epsilon$ supports the highest arrival rate due to its simple index structure (one-level grid index) while *AFIA* supports the lowest arrival rates due to its cell replication over both spatial and temporal dimensions. *GeoTrend* alternatives come in the middle of both and still can support up to 50K microblog/second which is an order of magnitude higher than current Twitter rate.

## 8.3 Query Processing

This section evaluates *GeoTrend* index design decisions that affects query processing. Section 8.3.1 evaluates the effect of replicating keywords across index levels. Section 8.3.2 evaluates the effect of maintaining top-$k$ list inside each index cell.

### *8.3.1 Keyword Replication*

In this section, we evaluate the replication of keywords in all pyramid index levels. To this end, we compare the pyramid index with a partial quad-tree index [15] that has similar cell structure to the pyramid, yet, keywords are maintained only in leaf cells (denoted as *GT-QT*). The two indexing structures favor different objectives: (1) The pyramid index maintains keywords aggregates in *all* leaf and non-leaf cells, increasing both memory and insertion overhead, but its query processor accesses far fewer cells, from higher levels, to compute the final answer. (2) The quad-tree index maintains keywords aggregates *only* in leaf cells, reducing both memory and insertion overhead, but increasing the query latency as the query processor accesses many cells to compute the final answer. The experiments results show that the quad-tree would not be able to provide low query latency although it has much lower memory and insertion overhead.

Figure 6 denotes the pyramid index, with and without $\epsilon$-cleaning, as *GT* and *GT*-$\epsilon$, quad tree as *GT-QT*, and GARNET as *GRN*-$\epsilon$, excluding AFIA from query evaluation due to its different aggregate measure. Figures 6(a) and 6(b) show one to three orders of magnitude better query latency for *GT* and *GT*-$\epsilon$ than *GT-QT* and *GRN*-$\epsilon$ with varying answer size $k$ and query region area $R$, respectively. *GT* and *GT*-$\epsilon$ consistently outperform both *GT-QT* and *GRN*-$\epsilon$ for different $k$ values. However, with changing values of area $R$, the improvement ratio changes: For small values of $R$, all indexes have almost the same average query latency as the number of processed cells is similar or close. When $R$ increases, *GT* and *GT*-$\epsilon$ use far fewer cells than both *GT-QT* and *GRN*-$\epsilon$, as they have a chance to use larger non-leaf cells contained in $R$, and therefore they give much lower query latency.

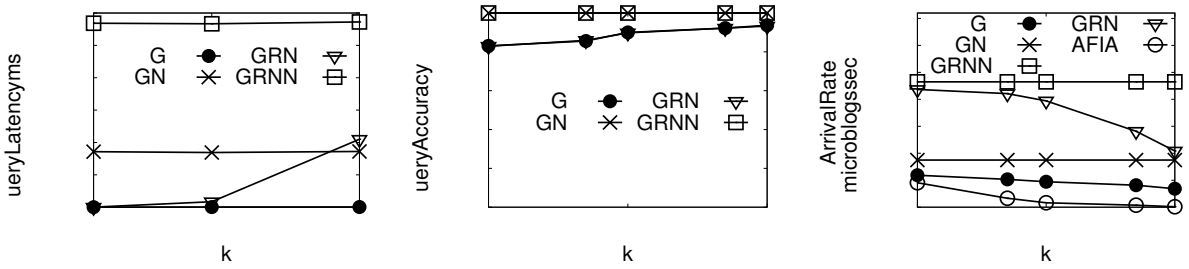
This lower query latency comes with the cost of higher insertion overhead and larger memory footprint than *GT-QT*. Figures 6(c) and 6(d) show that this is a favorable trade-off with affordable indexing overhead and memory footprint. For different values of $k$, *GT* and *GT*-$\epsilon$ still support up to an order of magnitude higher arrival rate than Twitter rate. Furthermore, *GT*-$\epsilon$ incurs only around three times memory overhead compared to *GT-QT*. On the contrary, *GRN*-$\epsilon$ still encounter high memory footprint due to the large number of cells in a fine-divided grid index with high resolution. This shows the effectiveness of *GeoTrend* design decisions to provide an excellent compromise in both memory overhead and query latency.

### *8.3.2 Materializing Top-k Lists*

The query answer can be computed either by using all keywords within the cell, which are expected to be huge with many keywords, or by exploiting only top-$k$ items in each cell as introduced in Section 7. We show that maintaining these lists reduces query latency significantly at the cost of acceptable overhead to store and maintain the sorted lists while continuously inserting new keywords and deleting old information and acceptable reduction in query accuracy. In this section we evaluate the effect of maintaining top-$k$ lists on query latency, query accuracy, and insertion overhead, excluding memory overhead effect as the storage of top-$k$ is negligible compared to the cell all keywords storage. The experimental results show two orders of magnitude improvement in query latency with sublinear increase in insertion overhead.

Figure 7 compares *GeoTrend* (denoted as *GT*), AFIA (denoted as *AFIA*), and GARNET (denoted as *GRN*), with and without maintaining top-$k$ lists (denoted as suffix *K* and *NK*, respectively). Note that AFIA has only top-$k$ option as this is the only maintained data structure in its index cell. It is also excluded from query measures as it support only top-$k$ frequent queries and cannot adapt our trending measure. Figure 7(a) depicts the query latency of all alternatives for different $k$ values. We observe that maintaining the top-$k$ lists reduces query latency of *GeoTrend* alternatives from 850 msec for all values of $k$ to between 1 and 3 msec, which is two orders of magnitude reduction. *GRN* query latency is consistently much higher than *GeoTrend* for two reasons. First, the large number of cells processed from its fine-divided grid index compared to cells of high levels of *GeoTrend* index which is much smaller in number. Such inefficient division for the space is a result for *GRN* generic framework for any context, so it is not tailored for location queries and thus cannot make maximum use of the spatial properties of the data. Second, *GRN* computes its aggregate measures from different temporal cells, as it is originally proposed and optimized for arbitrary temporal periods, which increase the aggregation time.

Figure 7(b) shows that for $k \geq 100$, aggregating from top-$k$ lists provides at least $90\%$ accuracy, providing an empirical evidence for the effectiveness of using top-$k$ lists with an acceptable accuracy loss. Figures 7(c) show the overhead of maintaining the top-$k$ lists on index insertion. AFIA still encounter the lowest arrival rate for the same reason detailed before. For *GeoTrend* and GAR-




(a) Query Latency (b) Query Accuracy (c) Supported Arrival Rate

**Figure 7: Impact of maintaining top-$k$ lists.**

NET, the significant reduction in query latency comes at the cost of 50% reduction in the supported arrival rate. For the worst case ($k$=1000) in Figure 7(c), *GeoTrend* index supports at least 40,000 microblog/sec which is seven times the current Twitter rate.

# 9. CONCLUSION

In this paper, we presented *GeoTrend*; a scalable system that supports spatial trending queries on recent microblogs. *GeoTrend* supports a variety of trending measures that suit different applications. It also supports queries on arbitrary spatial regions using data that has recently arrived in the last $T$ time units. For this, it employs an efficient main-memory spatial index that digests and expires data with high rates. In peak times, where main-memory is overwhelmed, it employs a smart memory optimizer that sheds less important data while keep highly accurate query answers. The experimental evaluation shows the scalability of *GeoTrend* to digest up to 50K microblog/sec, while providing average query latency of 3 msec and sustaining high performance with limited memory.

# 10. REFERENCES


[1] H. Abdelhaq, C. Sengstock, and M. Gertz. EvenTweet: Online Localized Event Detection from Twitter. In *VLDB*, 2013.

[2] A. Arasu and G. S. Manku. Approximate Counts and Quantiles over Sliding Windows. In *PODS*, 2004.

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

[4] Social media 'outstrips TV' as news source for young people. http://www.bbc.com/news/uk-36528256, 2016.

[5] 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.

[6] C. Budak, D. Agrawal, and A. El Abbadi. Structural Trend Analysis for Online Social Networks. *PVLDB*, 4(10):646–656, 2011.

[7] C. Budak, T. Georgiou, D. Agrawal, and A. E. Abbadi. GeoScope: Online Detection of Geo-Correlated Information Trends in Social Networks. In *VLDB*, 2014.

[8] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-Time Search at Twitter. In *ICDE*, 2012.

[9] Y. Chi, B. L. Tseng, and J. Tatemura. Eigen-Trend: Trend Analysis in the Blogosphere Based on Singular Value Decompositions. In *CIKM*, pages 68–77, 2006.

[10] Sina Weibo, China Twitter, comes to rescue amid flooding in Beijing. http://thenextweb.com/asia/2012/07/23/sina-weibo-chinas-twitter-comes-to-rescue-amid-flooding-in-beijing/, 2012.

[11] E. Cunha, G. Magno, G. Comarela, V. Almeida, M. A. Gonçalves, and F. Benevenuto. Analyzing the Dynamic Evolution of Hashtags on Twitter: a Language-Based Approach. In *Proceedings of the Workshop on Languages in Social Media*, pages 58–65, 2011.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows (extended abstract). In *SODA*, 2002.

[13] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, pages 102–113, 2001.

[14] W. Feng, J. Han, J. Wang, C. Aggarwal, and J. Huang. STREAMCUBE: Hierarchical Spatio-temporal Hashtag Clustering for Event Exploration Over the Twitter Stream. In *ICDE*, 2015.

[15] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *ACTA*, 4(1), 1974.

[16] H. Gao, J. Tang, and H. Liu. Exploring Social-Historical Ties on Location-Based Social Networks. In *The 6th Intl. AAAI Conf. on Weblogs and Social Media*, 2012.

[17] L. Golab, D. DeHaan, E. D. Demaine, A. López-Ortiz, and J. I. Munro. Identifying Frequent Items in Sliding Windows over On-line Packet Streams. In *Internet Measurement Comference*, 2003.

[18] Us department of health and human services disease tracking. https://nowtrending.hhs.gov.

[19] L. Hong, A. Ahmed, S. Gurumurthy, A. J. Smola, and K. Tsioutsiouliklis. Discovering Geographical Topics In The Twitter Stream. In *WWW*, 2012.

[20] Y. Ikawa, M. Enoki, and M. Tatsubori. Location Inference Using Microblog Messages. In *WWW*, 2012.

[21] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying Representative Trends in Massive Time Series Data Sets Using Sketches. In *VLDB*, pages 363–372, 2000.

[22] C. Jonathan, A. Magdy, M. Mokbel, and A. Jonathan. GARNET: A Holistic System Approach for Trending Queries in Microblogs. In *ICDE*, 2016.

[23] J. F. Kenney and E. S. Keeping. *Mathematics of Statistics, Part 1*, chapter 15, pages 252–285. van Nostrand, 3rd edition, 1962.

[24] I. Lazaridis and S. Mehrotra. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *SIGMOD*, pages 401–412, 2001.

[25] L.-K. Lee and H. F. Ting. A Simpler and More Efficient Deterministic Scheme for Finding Frequent Items over Sliding Windows. In *PODS*, 2006.

[26] G. Li, J. Hu, J. Feng, and K. Tan. Effective Location Identification from Microblogs. In *ICDE*, 2014.

[27] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang. TEDAS: A Twitter-based Event Detection and Analysis System. In *ICDE*, 2012.

[28] I. F. V. López, R. T. Snodgrass, and B. Moon. Spatiotemporal Aggregate Computation: A Survey. *TKDE*, 17(2):271–286, 2005.

[29] 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.

[30] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *VLDB*, 2002.

[31] M. Mathioudakis and N. Koudas. TwitterMonitor: Trend Detection over the Twitter Stream. In *SIGMOD*, 2010.

[32] S. Nath, F. Lin, L. Ravindranath, , and J. Padhye. SmartAds: Bringing Contextual Ads to Mobile Apps. In *ACM MobiSys*, 2013.

[33] K. Nguyen and D. A. Tran. An analysis of activities in Facebook. In *IEEE Consumer Communications and Networking Conference (CCNC)*, 2011.

[34] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *SSTD*, pages 443–459, 2001.

[35] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. TwitterStand: News in Tweets. In *GIS*, 2009.

[36] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen. Scalable Top-k Spatio-temporal Term Querying. In *ICDE*, pages 148–159, 2014.

[37] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-Temporal Aggregation Using Sketches. In *ICDE*, pages 214–225, 2004.

[38] Trends 24. http://trends24.in.

[39] Twitter Location Trends. https://support.twitter.com/articles/101125#Trend_Location.

[40] I. Weber and V. R. K. Garimella. Visualizing User-Defined, Discriminative Geo-Temporal Twitter Activity. In *ICWSM*, 2014.

[41] L. Wu, W. Lin, X. Xiao, and Y. Xu. LSII: An Indexing Structure for Exact Real-Time Search on Microblogs. In *ICDE*, 2013.

[42] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient Aggregation over Objects with Extent. In *PODS*, pages 121–132, 2002.