GARNET: A Holistic System Approach for Trending Queries in Microblogs*

Christopher Jonathan Amr Magdy Mohamed F. Mokbel Albert Jonathan

Department of Computer Science and Engineering,
University of Minnesota, MN, USA
{cjonathan,amr,mokbel,albert}@cs.umn.edu

***Abstract*—The recent wide popularity of microblogs (e.g., tweets, online comments) has empowered various important applications, including, news delivery, event detection, market analysis, and target advertising. A core module in all these applications is a frequent/trending query processor that aims to find out those topics that are highly frequent or trending in the social media through posted microblogs. Unfortunately current attempts for such core module suffer from several drawbacks. Most importantly, their narrow scope, as they focus only on solving trending queries for a very special case of localized and very recent microblogs. This paper presents GARNET; a holistic system equipped with one-stop efficient and scalable solution for supporting a generic form of context-aware frequent and trending queries on microblogs. GARNET supports both frequent and trending queries, any arbitrary time interval either current, recent, or past, of fixed granularity, and having a set of arbitrary filters over contextual attributes. From a system point of view, GARNET is very appealing and industry-friendly, as one needs to realize it once in the system. Then, a myriad of various forms of trending and frequent queries are immediately supported. Experimental evidence based on a real system prototype of GARNET and billions of real Twitter data show the scalability and efficiency of GARNET for various query types.**

I. Introduction

Microblogs, e.g., tweets, online reviews on Amazon, comments on news websites and Facebook, or check-in's at Foursquare, have recently become very popular among web users [8], [26]. As rich user-generated data, microblogs have been exploited in several applications, e.g., news delivery [23], [25], event detection [1], [14], [21], market analysis [7], studying public opinion [19] and geo-targeted advertising [28]. All such applications rely on the ability to understand what people are talking about in their microblogs, and use this information as an indication of the importance of news, events, and/or people interests. As a result, numerous recent research efforts have focused on supporting frequent and trending queries on microblogs with the form: "*Find top-k* ***frequent/trending*** *keywords within the last T* ***time units*** *within* ***location*** *L*" [4], [24]. Unfortunately, such efforts have a narrow scope by: (a) supporting either **frequent** queries as in [24] or **trending** queries as in [4]. Frequent queries aim to find the keywords that appear the most while trending queries aim to find the keywords that exhibit some burst according to a given trending function. (b) supporting only recent time queries expressed by either the last T time units as in [4] or a recent time window interval as in [24]. (c) supporting only a location filter where the location is either defined as a textual attribute as in [4] or coordinates as in [24]. Very few work (e.g., [16]) have waived the location filter to get trending keywords over all microblogs. Twitter [27] shows **localized** or worldwide **current** trending hashtags on a side panel.

The narrow scope of existing techniques result in the following three main problems that hinder the widespread and applicability of frequent and trending queries: (1) There is no support for context filters beyond the single location context. For example, one may need to know the trending keywords in politics, or the trending keywords among Spanish teenagers in California, and so on. The only way to support such filters within current work is to apply the filter over all microblogs first, followed by executing the trending query algorithm. This is very inefficient and may not be even possible when the size of the intermediate result is large. (2) There is no support for trending queries over a historical time interval, which is needed for historical data analysis, e.g., finding trending keyword(s) during the past US election in 2012. The main reason is that all existing techniques rely only on in-memory algorithms and data structures, which is only enough to store current or very recent incoming data. (3) From a system point of view, existing techniques are not practical. To equip a system with modules for each type of trending queries, one needs to implement an algorithm for localized frequent queries [24], another algorithm for localized trending queries [4], a third algorithm for general trending queries [16], and so on. This is not practical as each technique has its own data structure and storage requirements.

In this paper, we present GARNET; a holistic system equipped with one-stop efficient and scalable solution for supporting a generic form of *context-aware* frequent and trending queries on microblogs: "*Find top-k* ***frequent/trending*** *keywords within an arbitrary* ***current/recent/past*** *time window of fixed granularity according to a certain* ***context***". In that sense, GARNET supports: (a) both frequent and trending queries, (b) any arbitrary time interval either current, recent, or past, of fixed granularity, (e.g., hour, day, month, year) and (c) having a set of arbitrary filters over contextual attributes. In addition to such general form of frequent and trending queries on microblogs, GARNET also supports *reverse* trending queries, which are not supported by any of existing techniques. An example of such queries is: "*Among what age does the*

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

word XBox was trending last month". From a system point of view, such one-stop holistic approach is very appealing, as one needs to realize it once in the system. Then, a myriad of various forms of trending and frequent queries are immediately supported efficiently. Contrast such approach to the case of realizing tens of various algorithms to be able to support various forms of trending queries.

GARNET goes beyond the location context of trending queries in microblogs to the general case of multi-dimensional context attributes. A context could be a combination of location, topic, gender, age, language, or other attributes. An example of a *context-aware* trending query supported by GARNET is: "*Find trending keywords posted last March and related to **health** within **USA***". Such query is a two-dimensional context query with two context attributes: *topic* and *location*, with values of *health* and *USA*, respectively. Another example is: *"Find trending keywords posted during the last week among **Spanish** tweets posted by **teenagers** in **California**"*, which is a three-dimensional context query with three context attributes: *language*, *age*, and *location*, with values of *Spanish*, *teenagers (13 to 19)*, and *California*, respectively. The need for each context is different based on the underlying application. For example, some applications (e.g., market analysis) may need to understand the trend among the age and gender context. Other applications (e.g., education sector) may have more interest in the language context. Others (e.g., news delivery) may have more interest in the topic and location context. It is the job of GARNET to support efficient and scalable execution of trending queries under the preferred context of each application's need.

The main idea behind GARNET is to treat context-aware trending queries in the same way database management systems (DBMSs) treat index structures. Admin users of GARNET can build multi-dimensional index structures to be used for an efficient access of context-aware trending queries on specific dimensions. For example, if it is deemed frequent to issue queries on both the *topic* and *location* context together, then an admin user of GARNET would decide to build a two-dimensional index structure on both the *topic* and *location* context. Once the index is built, any incoming queries on that context would be answered very efficiently and in a scalable way. Meanwhile, if a context-aware query is issued on a non-indexed context, the query will encounter a slow response; same as issuing a query to a DBMS for a non-indexed field. However, building an index for all combination of context is impractical, same like it is impractical to build index structures over all attribute combinations of a relational table. Thus, GARNET allows its users to create and drop indexes depending on their needs. This facilitates a great flexibility to tune the system performance for important query workloads and for different applications within one system.

Each multi-dimensional index structure built by GARNET is basically a *Context-Aware Temporal Index* that consists of two indexing layers: a context layer and a temporal layer. The context layer uses a multi-dimensional grid index, where each dimension corresponds to one context. The temporal layer appears in each grid cell of the context grid index, where it contains a hierarchical temporal index that maintains aggregate information about incoming keywords at different levels of temporal granularity. This is basically a materialization of the answer of trending queries at different temporal granularity. In particular, each node in a temporal tree maintains a concise summary structures to maintain top-k frequent keywords and top-k trending keywords. This information is maintained on multiple levels of temporal granularity to support queries on arbitrary long periods of time. GARNET query processor exploits this aggregate information to answer incoming queries efficiently. An optimized part of the GARNET index is stored in memory for faster execution of current and recent time queries. The majority of the index is stored in the disk storage for historical data analysis.

A real system prototype of GARNET is experimentally evaluated by using a large repository of billions real Twitter data with several types of query workloads. Experimental evidence shows that GARNET is scalable and is able to insert each microblog to eight different context index structures with arrival rates that are $8\times$ faster than a regular Twitter rate [26]. GARNET is also able to provide up to 0.3 miliseconds of query response time in answering both top-k frequent and top-k trending queries of any given point of time.

The rest of the paper is organized as follows: Section II presents preliminaries. The system architecture of GARNET is described in Section III. Sections IV, V, and VI present index creation, management, and flushing to disk, respectively. Section VII discusses query processing. Section VIII gives the experimental evaluation. Related work is discussed in Section IX. Finally, Section X concludes the paper.

II. PRELIMINARIES

This section presents a set of preliminaries for context extraction from microblogs, frequent and trending computations, and problem formulation.

A. Context Extraction from Microblogs

Each microblog comes up with a set of contextual information that is either: (a) explicitly mentioned: For example, Twitter associates the *language* of each tweet as part of its metadata. Also, tweets posted from mobile devices have explicit *location* information expressed as (latitude, longitude) in the tweet metadata. (b) discovered through data mining techniques: For example, discovering the *gender* or the *age* of a person posting a certain microblog may require some data mining techniques going through the friend list and/or prior posts from the same person. Similarly, the *location* context, if not mentioned explicitly, may be discovered from the user location, the textual content of the microblog, or earlier posts. (c) discovered through natural language processing: For example, understanding the *topic* or the *emotions* of the microblog post requires employing natural language processing modules.

The domain values of each context may be either: (a) *categorical*: For example, the *topic* context may have a specific list of categories including politics, sports, entertainment, the *gender* context is either male or female. Also, the *age* context may be designed to have only three possible values as teenagers, adults, and elderly. (b) *continuous*: For example, the *location* context may have any value within the world space boundary.

Inferring the context value for each attribute is beyond the scope of this paper. GARNET assumes that each microblog is already mapped to its set of context attributes. Then, the main

focus of GARNET is on how to efficiently answer context-aware trending queries on microblogs, given that the context values of each microblog is already known.

B. Frequent and Trending Computations

Frequent keyword queries aim to find keywords that have appeared the most (in absolute numbers) during the specified time and contextual constraints. Meanwhile, trending keyword queries aim to find the keywords that are trending over a period of time (under given contextual constraints), which is measured by the growth in frequency over time. To compute trending keywords over time, we use the trending aggregate measure T_{trend}, which measures the growth in frequency of a keyword W along N consecutive time units t_i, $1 \leq i \leq N$, where N is a system parameter. T_{trend} is given per the following equation, which is derived based on the statistical linear regression [13]:

$$T_{trend} = \frac{6 \sum_{i=1}^{N} [i \times (f_i - f_0)]}{N(N+1)(2N+1)} \quad (1)$$

Where f_i, $1 \leq i \leq N$, is the frequency of keyword W at time t_i. Details about this equation is in Appendix A.

C. Problem Definition

A context-aware trending query comes with the following form: "Find top-k *frequent/trending* keywords within context C in time range T (of fixed granularity)", which has three parameters: (1) k as the number of keywords to be returned, (2) Multi-dimensional contextual constraints C, and (3) time range T of fixed granularity; hour, day, month, or year.

Definition 1. Context-aware query: *Given integer k, multi-dimensional contextual constraints C, and time range T of fixed granularity (hour, day, month, or year), a context-aware frequent/trending query returns k keywords that are: (1) top ranked as either frequent or trending, (2) posted within the context C constraints, and (3) posted within time T.*

GARNET also supports *reverse context-aware queries* with the form: "In which values of context C is the keyword W *frequent/trending* over a time range T of fixed granularity", which has three parameters: (1) multi-dimensional context C, (2) keyword W, and (3) time range T of fixed granularity; hour, day, month, or year.

Definition 2. Reverse context-aware query: *Given multi-dimensional contextual constraints C, keyword W, and time range T of fixed granularity, a reverse context-aware frequent/trending query returns a value within C where W is: (1) top ranked as either frequent or trending, (2) posted within the context C, and (3) posted within the time point T.*

III. System Architecture

Figure 1 gives system architecture of GARNET , which is composed of four main modules (Section III-A) and four main data structures (Section III-B).

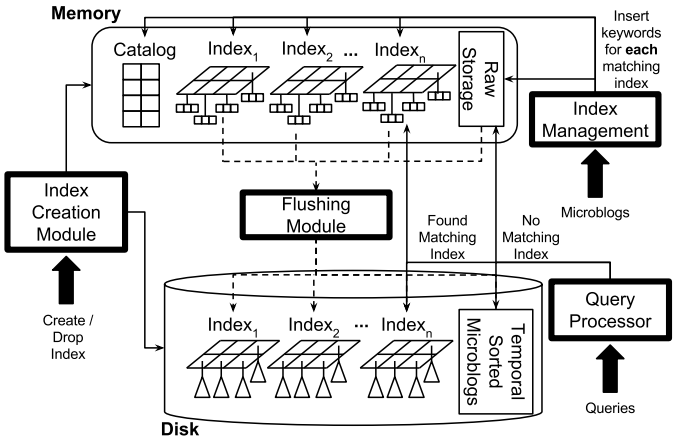


Fig. 1. GARNET System Architecture

A. GARNET Main Modules

GARNET includes four main modules, namely, *index creation*, *index management*, *flushing*, and *query processor*, described briefly below:

Index Creation. This module receives index create/drop requests from GARNET administrator and creates/drops both an in-memory index structure and a corresponding disk-resident one; along with updating an index catalog. Both data structures have a flat grid index for context indexing. Each in-memory grid cell has a flat temporal index, while in-disk index cells have a hierarchical temporal index. Details in Section IV.

Index Management. For each new incoming microblog M, this module performs the following: (1) Extract the microblog M contextual information and set of keywords. This particular step is beyond the scope of this paper, as there are already a plenty of research efforts in this area from both data mining and natural language processing communities, e.g., see [18]. (2) Consult the in-memory catalog to get the set of in-memory index structure(s) $\mathcal{I}$ that M belongs to, according to its extracted context. (3) Insert M (with its keywords) into each index in $\mathcal{I}$ (if any), as well as into an in-memory buffer as a raw store for all incoming microblogs. Details in Section V.

Flushing. This module is triggered either periodically or once the memory becomes full. The objective is to flush part of the in-memory buffer and index structures contents to their corresponding disk structures as a means of giving a room to new incoming microblogs to be inserted in memory. Details in Section VI.

Query processor. This module receives frequent and trending context-aware queries as described in Section II-C. Based on both temporal and contextual attribute constrains in the given query, this module decides on: (1) Which index structure, if any, to consult for the answer, and (2) Whether the answer will be retrieved from the in-memory index and/or the in-disk index. Details in Section VII.

B. GARNET Data Structures

GARNET maintains four main data structures, namely, *in-memory index*, *disk-resident index*, *raw storage*, and *in-memory catalog*, described briefly below:

In-Memory Index. In-memory index is created whenever GARNET administrator submits an *index create* request to the index creation module. An in-memory index is composed of two main layers, namely the *context index layer* and *temporal index layer*. The context index layer is basically a multi-dimensional uniform grid index where each dimension represents one context attribute. The temporal index layer is basically a temporal list of nodes, where each node represents a fixed time granularity. Details in Section IV-B.

Disk-Resident Index. For each in-memory index, there exist a corresponding disk-resident index. Similar to the in-memory index, a disk-resident index is composed of two layers, the *context index layer* and *temporal index layer*. Disk-resident index has the same context index layer as the corresponding in-memory index. However, the temporal index layer of the disk-resident index is a tree of temporal hierarchy that maintains aggregate information about microblogs on different temporal granularity. Details in Section IV-C.

Raw Microblogs Storage. GARNET maintains two raw microblogs stores: one in-memory and one in-disk. Both stores are append-only and have their microblogs stored on their arrival rates. GARNET keeps one hour worth of microblogs inside its memory raw storage before flushing them to the disk raw storage which is triggered every hour. These raw microblogs are used to answer user queries with a non-indexed contextual attributes as well as used to populate newly created index structures with historical microblogs.

In-Memory Catalog. GARNET keeps all its index structures metadata in an in-memory catalog. The in-memory catalog is basically a table with two columns and n rows, where n is the number of index structures that GARNET maintains. The first column contains the name of each index, while the second column contains the indexed contextual attributes.

IV. Index Creation Module

This section discusses the index creation module of Figure 1. We start by the SQL syntax that triggers this module (Section IV-A) to create an index structure that will be used to support frequent and trending queries for specific context. The execution itself results in creating both an in-memory index structure (Section IV-B) and an in-disk index structure (Section IV-C). Whenever a new index is created, GARNET will bulk load the index with historical microblogs (Section IV-D).

A. Create/Drop Index Syntax

GARNET users can create new index structures for a specific context by using the following SQL syntax:

```
CREATE INDEX IndexName ON
CONTEXT Context_1, Context_2, ..., Context_M
GRANULARITY [t]
LIMIT [k]
TREND [N]
```

The above index creation command has five parameters: (1) $IndexName$; the name of the index, (2) One or more $Context$ to indicate the set of context attributes that the index will be built on, (3) t; the finest granularity temporal value that the index must capture (e.g., hour, day, week, or month), (4) k; the number of returned results per the top-k functionality, and (5) N; the number of past time intervals that are taken into account when computing the trending function, per Equation 1. The last three parameters, t, k, and N, are optional and have default values of day, 100, and 3, respectively, if not defined in the Create Index SQL statement.

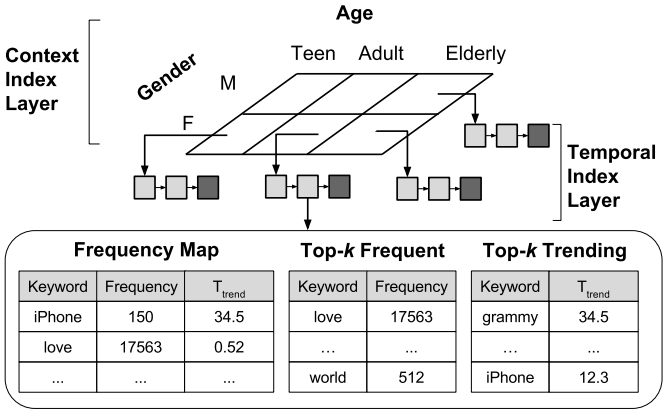


Fig. 2. AgeGender In-Memory Index Organization

The following is an example of creating a two-dimensional index with context of AGE and GENDER:

```
CREATE INDEX AgeGender ON
CONTEXT Age, Gender
GRANULARITY [hour]
LIMIT [100]
TREND [3]
```

Once an index is no longer needed, a simple Drop Index command can be issued to clean all the index footprints from both memory and disk storage.

B. In-Memory Index Structure

Figure 2 depicts the organization of an in-memory Context-Aware temporal index structure (CAT, for short) for a two-dimensional context composed of *Age* and *Gender*. The index is composed of two main layers, namely, *context* layer and *temporal* layer, as follows:

Context Index Layer. The context index layer is basically a multi-dimensional index on the given context. There is already a rich literature on multi-dimensional index structures [10], classified into two categories: *Data-partitioning* index structures (e.g., R-tree [12]) that partition the data over the index and *space-partitioning* index structures (e.g., Quadtrees [22]) that partition the space. In GARNET we decided to go with the Grid Index as an example for *space-partitioning* data structures, for the following reasons: (1) Most of the context domain values are discrete (e.g., the *Gender*, *topic*, *language*), which is more suitable to be expressed within space-partitioning data structures. (2) Microblogs are added to the system in very high arrival rates. Hence, the index structure must be simple and as static as possible in order to avoid expensive index repartitioning. Having said this, GARNET framework can

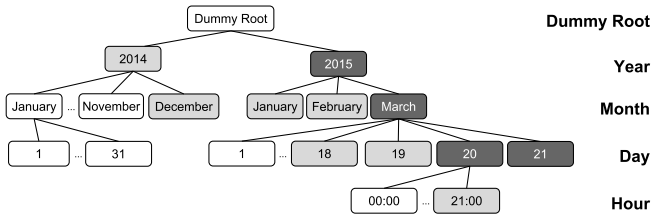


Fig. 3. AgeGender Temporal Tree

accommodate other multi-dimensional index structures as a replacement of our grid index.

Each grid cell in the context layer keeps metadata about its temporal index layer. In particular, the number of nodes and the frequency of incoming microblogs. Figure 2 gives an example of a context index layer as a two-dimensional grid structure for *Age* and *Gender* attributes. The *Age* attribute takes three possible values while the *Gender* attributes takes two possible values, making the whole index composed of six grid cells.

Temporal Index Layer. Each cell in the multi-dimensional context grid points to a list of temporal nodes. Each node has a temporal range of fixed temporal granularity (e.g., one hour) and maintains aggregate information about microblogs that have arrived within: (a) the node temporal range, and (b) the same context of the parent grid cell. Each temporal node includes the following three tables: (1) *Keywords Frequency Map*, which is a table that contains the number of times that each keyword has appeared during the temporal and context domain of that node as well as the T_{Trend} of the keyword which is computed by using the keyword frequency maps of this node and the last N nodes per Equation 1, (2) *Top-k Frequent List*, which is a table that maintains the list of the Top-k keywords that have appeared the most within the node temporal and context domain. This can be computed directly from the keyword frequency map, yet, we are materializing it here for efficiency. (3) *Top-k Trending List*, which maintains the list of the Top-k keywords that are trending.

There are two types of in-memory temporal nodes, *active* and *inactive*. Examples of *active* nodes are the right most temporal nodes in each grid cell in Figure 2 (highlighted in dark gray). An *active* receives new incoming microblogs and will stay active till the end of its temporal domain (e.g., top of the hour). Then it will be considered *inactive*, i.e., its data structure becomes static as no new incoming microblogs will be added to this node anymore. *Inactive* nodes are highlighted with light gray in Figure 2. Once an *active* node becomes *inactive*, a new *active* node will be created to receive the new incoming microblogs.

C. Disk-Resident Index Structure

Each in-memory CAT index has a corresponding disk resident one. The disk-resident index structure has the same context layer as the in memory one. However, the temporal layer is different as it is composed of a temporal tree rather than a temporal list of nodes. Figure 3 gives the structure of the temporal layer in the disk-resident index, where each grid cell in the disk context layer points to the root node of its corresponding temporal tree. The temporal tree is a hierarchical index with multiple temporal levels where each level corresponds to a certain temporal granularity of year, month, day, or hour. The highest level of the tree always corresponds to a year temporal granularity, while the lowest level corresponds to the finest temporal granularity determined in the `CREATE INDEX` statement (Section IV-A). All non-leaf nodes have multiple children nodes that correspond directly to a finer time granularity, e.g., a year node has up to twelve children nodes where each of them represents a month node. A dummy root node that does not maintain any data, is created as a parent for an unlimited number of year nodes. The objective of maintaining nodes over different levels of temporal granularity is to allow answering queries with arbitrarily long time periods while accessing minimal tree nodes.

Excluding the empty dummy root, the temporal tree includes three types of nodes, namely, *active*, *intermediate*, and *archived*, described briefly below:

Active nodes. (highlighted in dark gray). Similar to the in-memory *active* nodes, the temporal domain for the disk *active* nodes is still not concluded as it can receive more incoming microblogs. For example, as of Oct., 15, 2015, we have three *active* nodes for year "2015", month "October", and day "15", as none of these has been concluded yet. If the lowest tree level matches the time granularity of the in-memory CAT index (e.g., the hour level in Figure 3), then, there will be no *active* nodes in that level in the disk-based index. This is because the temporal domain of all hours in that level have definitely been concluded. The only *active* one (i.e., the node for current hour) must be residing in memory at that time. Meanwhile, there could be more than one *active* node in the same temporal level. e.g., the day level in Figure 3. This happens only if the temporal domains of the in-memory index spans over the temporal domain of multiple in-disk temporal nodes of the same level. For example, consider the case where there are five in-memory hourly temporal nodes spanning the time from 10:00 PM on Oct 14 till 3:00 AM on Oct 15. Two of these nodes will need to be later added to the in-disk daily node of Oct 14, while three will be added later to the daily node of Oct 15. This means that both the daily nodes of Oct 14 and Oct 15 are still active. Each in-disk *active* node maintains the same three tables as in-memory *active nodes* (Section IV-B), namely, *key frequency map*, *top-k frequency list*, and *top-k trending list*.

Intermediate nodes. (highlighted in light gray). Similar to the in-memory *inactive* nodes, these nodes have the following three properties: (a) they maintain the same three tables as in active nodes, (b) the values in these three tables are static and would never change, as the temporal domain of these nodes are concluded, and (c) the values in their *key frequency maps* are used to calculate the trending function of current active nodes of the same level per Equation 1. Each temporal level has at most N *intermediate* nodes, where N is a parameter in the CREATE INDEX statement (Section IV-A) and used in the trending in Equation 1. Figure 3 has a temporal tree with N=3. There are two cases where there could be less than N intermediate nodes in a certain temporal level: (1) if there are no available historical data in the system, e.g., the year level in Figure 3, and (2) If the lowest in-disk temporal level matches the level of the in-memory temporal list and there are less than N *inactive* in-memory nodes. For example, in Figure 2, there are only two inactive hourly in-memory nodes,

then, given $N = 3$, there should be one *intermediate* hourly disk-resident node as in Figure 3

Archived nodes. These are the rest of temporal tree nodes that are neither *active* nor *intermediate*. Unlike other nodes, *archived* nodes maintain only two tables; *top-k frequency list* and *top-k trending list* that materialize the answer of frequent and trending queries for the context and temporal domains of these nodes. There is no need to maintain an exhaustive list of keyword frequency map since, unlike intermediate node, it will not be used in any trending value. An *archived* node is far from any *active* node by at least N+1 nodes.

D. Bulk Loading

Once GARNET recieves a CREATE INDEX command, we populate the newly created index with historical raw microblogs by using a bulk loading procedure. The procedure starts by creating an hour level node for each hour of raw microblogs starting from the oldest timestamp of the raw microblogs. During the creation, we will also update the node's three statistical data structures. Whenever a time period of the hour level nodes is concluded, i.e., the end of the day, we create a day level node to be the parent of these hour nodes and aggregate all statistical information from its children. Similarly, for higher temporal levels. This process will continue until all raw microblogs inside the disk-resident raw storage are inserted. Then, we will drop all nodes that have a finer temporal granularity than the lowest temporal granularity of the newly created index. Once the disk-resident index is created, we create the corresponding in-memory index followed by creating *active* nodes for the index. Finally, we populate each *active* node with the microblogs from in-memory raw microblogs storage. When creating the in-memory index, we will also load N previous hour nodes' frequency maps from the disk-resident index in order to minimize disk access for calculating T_{Trend} of each keyword in the in-memory index.

V. Index Management

This section discusses the index management module of Figure 1. The input to this module is a newly incoming microblog M. The first things that GARNET does here is to extract the context C_M from M. As mentioned in Section II-A, context extraction is beyond the scope of this paper. Given the extracted context C_M, GARNET goes through three steps: (1) Inserting M in the in-memory raw storage (Section V-A). (2) Checking the in-memory *Catalog* for the set of index structure(s) that match the context C_M. If this set is null, we proceed with the next incoming microblog. Otherwise, we extract the set of keywords W from M and insert each keyword in W to the index structure(s) that it belongs to (Section V-B). (3) Reorganizing the updated index structure(s), if needed (Section V-C).

A. Step 1: Insertion in Raw Storage

For each incoming microblog, GARNET will keep the microblog inside its in-memory raw storage. This is an append-only heap storage where the microblogs are ordered based on their arrival time. This raw storage serves three purposes: (1) Supporting queries asking for non-indexed context. (2) Even for those microblogs that match an existing index, we would still need to insert them in the raw storage. It may happen later that the index is dropped, so, we would still need to have the microblogs stored in our raw storage. (3) A newly created index will need to be bulk loaded first. The contents of the initial bulk loading will be retrieved from this raw storage, along with its corresponding disk-resident one.

B. Step 2: Updating CAT Index Statistics

We start this step by consulting the in-memory *Catalog* to get the set of index structures $\mathcal{I}$ that match the context C_M of the incoming microblog M. If $\mathcal{I}$ ends up to be an empty set, we simply skip the rest of this step as well as the next step, and just proceed to process the next incoming microblog. If $\mathcal{I}$ is a non-empty set, we insert each keyword w extracted from M into each index in $\mathcal{I}$.

The insertion of a keyword w to an index I goes though the following four steps: (1) We locate the grid cell C in the context layer of I that corresponds to the context values C_M of M. (2) We insert w in the *Keyword Frequency Map* of the active node of C. If there is already an entry for w there, we just increase its counter w_{freq}, otherwise, we add a new entry for w with the counter w_{freq} set to one. (3) We calculate the trending value w_{trend} of w using Equation 1, and store it in the *Keyword Frequency Map*. This will require consulting the N in-memory *inactive* nodes that are just before the current *active* node in C. If there are less than N such in-memory nodes, we will need to encounter a disk access to retrieve *intermediate* node(s) for computing the trending function. This is why GARNET strives to make sure that there are at least N+1 in-memory nodes available for each in-memory index cell. Ensuring so will guarantee that there is no need to have any disk access to compute the trending value for any incoming microblog. (4) We check if the insertion of w needs to update the top-k frequent and trending keywords lists in C. We do so by comparing w_{freq} and w_{trend} with the smallest (i.e., k^{th}) values in the top-k frequent and trending lists, k_f and k_t, respectively. If w_{freq} (w_{trend}) is smaller than k_f (k_t), we do not do any update, otherwise, we remove the k^{th} entry from the frequent (trending) list, and insert w to the frequent (trending) list, ordered by its w_{freq} (w_{trend}) value.

C. Index Reorganization

Recall that all temporal nodes in the temporal index layer are of fixed granularity, e.g., one hour. So, when a new fixed time interval starts, e.g., on the top of the hour, we do the following three reorganization steps for each in-memory CAT index structure: (1) We create a new empty *active* temporal node for each grid cell in the context layer of each in-memory CAT index. (2) We mark all the previously active ones as *inactive* nodes, which means that their temporal domain is concluded and their statistical tables cannot be updated any more. (3) We run a storage optimization technique with the objective of minimizing the size of the *Keyword Frequency Map* without losing much in the ability of answering frequent and trending queries with high accuracy. In particular, we employ the Lossy Counting Algorithm [15] to proactively remove keywords that have low frequency under certain percentile threshold ϵ of the total frequency of the keyword frequency map, where ϵ is a system parameter that has a small positive fraction, e.g., $\epsilon = 0.0001$. With the skewed nature of keywords distribution

in microblogs, we are able to save up to 94% of storage while maintaining 90%+ frequent query accuracy and 85%+ trending query accuracy. Such optimization is very important as it allows us to store more *inactive* nodes in memory, and hence much less need to have disk access either for supporting new incoming microblogs or for answering frequent and trending queries.

VI. Flushing Module

This section discusses the flushing module of Figure 1, which is triggered on two events: (1) *Every hour* to flush the in-memory raw storage and (2) *Once the memory is full* to flush part of the in-memory CAT index structures to their corresponding disk index structures. The hourly flushing process appends the contents of the in-memory raw storage to the in-disk raw storage, and then wipe the in-memory one. The purpose is to leave room for newly incoming microblogs. The reason for doing so on an hourly basis is that raw microblogs consume a large portion of the memory storage. Yet, it would be more effectively having the memory consumed by more CAT index structures. Hence, we limit the size of the raw storage to only the set of microblogs coming within one hour. Meanwhile, we let the CAT index structures grow in memory as much as possible. Once the memory is all consumed, we decide to flush a percentage $B\%$ of the in-memory contents consumed by the index structures, where B is a system parameter. The index flushing process goes through three steps, described below in Sections VI-A to VI-C.

A. Step 1: Selecting Victim Nodes

The objective of this step is select a set of in-memory temporal nodes with a total size of at least $B\%$ to flush to the corresponding disk index. Our choice goes towards keeping N+1 nodes (one *active* and N *inactive*) in each grid cell in each in-memory CAT index, and flush the rest to the disk storage. The rationale here is that new incoming microblog need to consult N *inactive* nodes to compute its trending value. So, having less than N+1 in-memory nodes will result in a disk access for each new incoming microblog. However, if flushing these nodes did not make it to the required $B\%$, we have no options other than going for more valuable nodes that are within the first N+1. In this case, we start flushing from the index structures that have least recent arrived microblogs one by one till we reach the desired ratio of $B\%$.

B. Step 2: Flushing Victim Nodes to Disk-Resident Index

This step flushes the selected nodes from Step 1 to their corresponding disk structures. In general, there are two cases:

Case 1: The finest temporal granularity of the disk index is the same as in-memory index (i.e., one hour). In this case, we will concatenate the flushed nodes in each gird cell of the in-memory CAT index to the finest level of the temporal tree for each corresponding grid cell of the disk-resident CAT index. We would then need to label the disk nodes. Assume that after this flushing, there are still V nodes in memory for a certain grid cell. If V is more than N, then all the disk nodes are marked as *archived*. Otherwise, we mark the first $N + 1 - V$ nodes in disk as *intermediate* as we would need their values to calculate the trending score of new incoming microblogs.

Case 2: The finest temporal granularity of the disk index is higher than the in-memory index. In this case, we aggregate all keyword frequency maps of the victims for each cell. Then, we combine the aggregated values to the leaf in-disk *active* node(s), along with updating the corresponding top-k frequent and trending keywords lists. If the flushed content starts a new *active* node on the leaf level of the disk-resident CAT index, e.g., a new day, we will mark the old *active* node as *intermediate* node as well as updating the N^{th} intermediate node to an *archived* one.

C. Step 3: Proactive Vs. Lazy Update of Disk Parent Nodes

Once we update the finest temporal granularity of the disk index structure, we need to propagate all the changes to the higher levels of the temporal tree. We can do so in a **proactive** mode, which means recursively updating all parent nodes of any newly flushed node, along with all the statistics. This ensures the full accuracy of the in-disk index structure. However, it comes with a high cost of disk updates with every flushing process. As a result, GARNET leans to updating the contents of the parent nodes in a **lazy/on-demand** mode. In this mode, we do not do any updates for the parent nodes during the flushing process. This means that the parent nodes may not be accurate by the flushing time. The information of such inaccurate nodes will be updated either: (1) *Lazy*. When the time period of the right-most parent active node is concluded, we aggregate all the information from its children. (2) *On-demand*. When a query triggers the computation of top-k lists in the node, we compute it in an ad-hoc way, and materialize the answer in the node's tables.

The *proactive* and *lazy/on-demand* modes result in a trade-off between the insertion efficiency (hence, the digestion rate) and the query efficiency. The *lazy* mode is much more efficient in insertion time as we insert in less number of nodes. Meanwhile, the *lazy* mode may encounter extra time to process top-k frequent and trending queries of the parent active nodes that are not updated yet. Yet, such queries will encounter slow response time only on their first time. Since the tables in the parent active node are also updated on-demand, a second issuing of a query on a parent active node would have much better performance.

VII. Query Processing

This section discusses the query processor module of Figure 1. As mentioned earlier, GARNET supports two types of trending queries, namely context-aware trending queries and reverse context-aware trending queries.

A. Context-Aware Trending Query

The query processing module in GARNET receives frequent/trending queries on the form: "*Find top-k frequent / trending keywords within an arbitrary current / recent/ past time window of fixed granularity T according to a certain context C*". Accordingly, the first thing we do is to consult the in-memory Catalog to check if there is already a corresponding CAT index structure for the query context C. With a careful system adminstration of GARNET, we expect that most queries will find a corresponding index. It is the responsibility of GARNET system administrator to decide on what are the set of

context attributes that most queries are asking about. For this set of context attributes, the system administrator should build a corresponding CAT index, as discussed in Section IV.

For those queries with matching index, we will check the temporal constraint of the query T to see if the query answer can be all generated from memory, disk, or both of them. In general, T is of a fixed granularity, e.g., a specific hour, day, month, or year. So, there must be a corresponding temporal node N_T for T that resides in either an in-memory or an in-disk CAT index structure. Based on the status of N_T, we have the following four cases: (1) N_T is in memory, (2) N_T is an *intermediate* disk node, (3) N_T is an *archived* disk node, and (4) N_T is an *active* disk node.

In the first three cases, we just retrieve the materialized answer for the query form either the top-k frequent or top-k trending list of node N_T based on whether the query is a frequent or trending query, respectively. In the fourth case (N_T is an active disk node), we first need to compute the query answer from disk, then, add to it the answer from aggregating all memory contents. The rationale here is that the temporal range covered by any active node must include the current timing, which does exist only in in-memory contents and not flushed yet to the disk storage. Computing part of the answer from disk depends on the update mode we have used in Section VI-C. In case we are deploying a *proactive* flushing mode, getting part of the answer from disk is as simple as retrieving it from the materialized top-k lists. In case we are deploying a *lazy* mode, we would need to aggregate the answer from the children nodes of N_T as it is not materialized there yet. Once computing this part of the answer, we store it in N_T for future queries.

In the unfavorable (and unlikely) case that there is no matching index for context C, we will need to generate the query answer from the raw microblogs sorted in either memory, disk, or both. If T is this hour, we answer the query from memory. If T is greater than the current hour but includes current hour, e.g. today, we will scan both in-memory and disk-resident raw storage to get the answer. Note that the disk-resident storage is ordered temporally in chunks of hours. So, we will only scan the part we need from it. If T is beyond any *current* time slot, we just retrieve the answer only from the disk storage by directly reading the corresponding part of the answer from the raw disk storage.

B. Reverse Context-Aware Trending Query

The query processing module in GARNET receives reverse frequent/trending queries on the form:"*In which context C that a keyword w is frequent / trending within an arbitrary current / recent / past time window of fixed granularity T?*" Similar to answering context-aware trending query in the previous subsection, the first thing we do is to consult the in-memory Catalog to check if there is a corresponding index structure for the query context C.

For those queries with a matching index, we will check the temporal constraint of the query T to see if the query answer can be all generated from memory, disk, or both. This check results in the same four cases as the previous subsection. However, rather than having one corresponding temporal node from one grid cell, we will end up having a set of temporal nodes that match T, as: N_{T1}, N_{T2}, $\cdots$, N_{Tm}, where m is the number of grid cells in the index that match C. We do so as we will need to scan over all the top-k frequent/trending lists for all these nodes to find the node N_{Ti} with highest value for keyword w. The query answer is the context values for node N_{Ti}. There is one caveat here. Since we only keep up to k frequent/trending keywords in each node, the keyword w might not be available within the index. In this case, we will inform the user that the keyword w is not among the top-k frequent/trending keywords anywhere. Then, we ask the user whether he/she wants GARNET to scan the raw microblogs to get at which context w was most frequent/trending even though it is not among the top-k. If needed then, we will have to get the answer in an expensive way by consulting the raw storage as in the case of having no matching index.

In the unlikely case that there is no matching index for the context C, we will need to generate the query answer from the raw microblogs. Similarly, rather than finding the keyword that has the highest frequency/trending, we will check whether the keyword w has appeared in each raw microblog that has the same matching context as the query and stores the combination of the context attributes in a sorted list. Then, GARNET will return the combination of context attributes from the first entry of the sorted list.

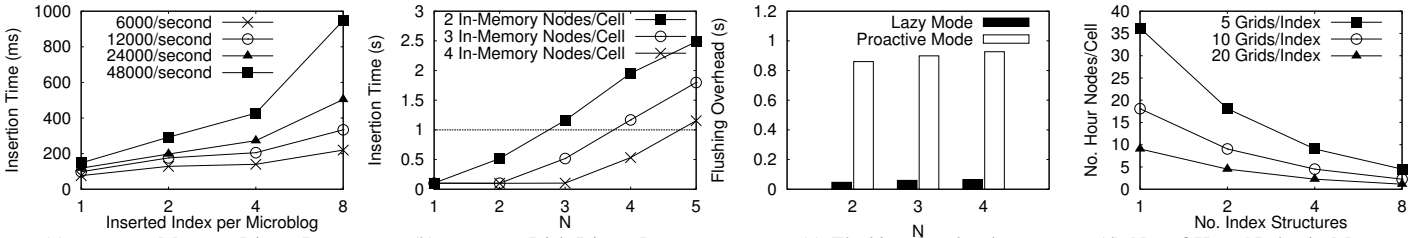
VIII. Experiments

In this section, we evaluate the performance and scalability of GARNET in answering different types of queries. To the best of our knowledge, GARNET is the first system that supports generic context-aware queries on microblogs. Thus, there is no direct competitors to compare its performance with. However, we compare our performance in answering both *location-aware* top-k frequent queries and *reverse frequent queries* to AFIA [24]. We limit our comparison with AFIA to these queries as AFIA does not support answering trending queries. All experiments presented in this section are based on real Twitter data that is collected from Twitter Streaming APIs from January 2015 to May 2015. Our dataset consists of 1.1+ Billion tweets that consumes 1TB of disk storage altogether with each tweet's attributes. Tweets are preprocessed to associate a set of keywords with each data record. We generated a synthetic query workload which consists of values that are randomly selected from the pool of available context. Unless mentioned otherwise, we set the parameters of the indexes: k and N to 100 and 3, respectively. All experiments run on a single machine with Intel Quad-Core with CPU 3.40GHz, one thread per core, and 16GB of RAM running 64-bit Ubuntu 14.04. We will first show the scalability of the index structures in GARNET in Section VIII-B. This includes the index digestion rate of incoming microblogs as well as the flushing performance of both *proactive* and *lazy* modes. We then evaluate the performance of GARNET in answering different types of queries in Section VIII-B.

A. Index Evaluation

To evaluate the scalability of the index structures in GARNET, we use two measurements; *digestion rate* and *number of hour nodes* per grid cell in the in-memory CAT index.

https://dev.twitter.com/streaming/



(a) GARNET Memory Digest Rate (b) GARNET Disk Digest Rate (c) Flushing Overhead (d) No. of Hour Nodes in Memory

Fig. 4. Index Scalability Evaluation

A higher number of *hour nodes* in memory will reduce the number of disk accesses since it allows more queries to be answered from memory. We evaluate the index performance and scalability in GARNET using eight index structures with 20 attribute combination (i.e, grid cells) each.

In Figure 4(a), the number of index structures that a new microblog will be inserted to is varied from 1 to 8. The experiment is run for various microblog arrival rates from 6K to 48K per second (typical Twitter rate is 6K tweets/second). In this experiment, we assume that all required nodes for the insertion are available in the memory (one active node and N inactive nodes). We first buffer one-second-worth of microblogs in the memory, then insert these microblogs into different number of index structures. If GARNET is able to insert these buffered microblogs in less than one second, which includes inserting each keyword to the Keyword Frequency Map and updating both top-k Frequent and Trending Lists, then GARNET is able to handle the digestion rate of that amount of buffered microblogs per second. From the figure, we can see that GARNET is able to handle 8$\times$ the number of the regular Twitter rate by inserting 48,000 microblogs to 8 different index structures in less than one second. This shows the scalability of CAT index digestion rate in GARNET.

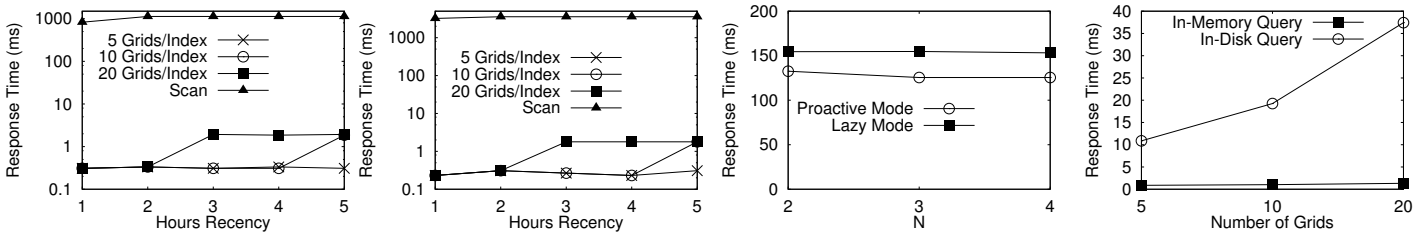
In Figure 4(b), the parameter N that represents that number of prior nodes needed to compute the trending function is varied from 1 to 5. The experiment is run with three different values of m as the number of in-memory available nodes for each gird cell in the context index layer. In this experiment, we use the Twitter rate of 6K microblogs/second with 4 index structures to be inserted per microblog. When $N > m$, GARNET needs to fetch $(N - m + 1)$ node(s) from the in-disk CAT index in order to calculate the T_{Trend} of the incoming microblogs. This results in a slower insertion rate in Figure 4(b) compared to Figure 4(a). However, even with fetching one node in every cell of each index from the in-disk CAT index, GARNET is still able to manage the incoming rate of Twitter's microblogs by inserting those microblogs in less than one second.

Figure 4(c) compares the overhead of using different flushing policies discussed in Section VI-C, i.e., *lazy* and *proactive* modes, for each grid cell of in-memory CAT indexes. In the *lazy mode*, the flushing will only affects the smallest granularity of its index in the disk and lets the updates on the other levels of its temporal tree index be updated in a deferred manner. On the other hand, the *proactive mode* will update all levels of the temporal tree (day, month, and year) when the flushing happens. This results in a higher overhead flushing time for the *proactive mode* compared to the *lazy mode*. The figure also shows that the flushing overhead is relatively stable with different values of N, which is approximately 900 ms for the *proactive mode* and 60 ms for the *lazy mode*. The reason is that GARNET loads the N Keyword Frequency Maps into the memory once for the whole flushing period.

Figure 4(d) shows the number of hour nodes in each cell of the in-memory CAT index at a certain point of time over different numbers of index structures as well as different numbers of grid cells per index. A higher number of hour nodes in each cell of the index will result in a higher hit ratio for the index block residing in memory. Note that the number of hour nodes in the in-memory CAT index structures is also determined by the total number of grid cells from all index structures. With 16GB of memory and $N = 3$, GARNET is able to answer T_{Trend} on four index structures with ten grid cells per index without accessing the disk-resident CAT index structures. From this point onward, we will use four index structures for subsequent experiments, allowing approximately 4.5 most recent hours to be covered in memory with ten grid cells/index.

B. Query Performance Evaluation

We evaluate the query response time of answering frequent/trending queries in GARNET by retrieving the answer directly from either the in-memory or in-disk CAT index. Figure 5(a) compares the query response time of the CAT index built in GARNET with the ad-hoc query evaluation using non-indexed files (Scan). We use four index structures with different number of grid cells per index g (5, 10, or 20 cells per index). With 16GB of memory, $g = 5$ contains approximately nine hour nodes in each cell in memory while $g = 10$ and $g = 20$ contain four and two hour nodes in each cell, respectively. This explains the immediate increase in query response time from 0.3 ms to 2 ms for $g = 10$ when the query time point is more than four hours and $g = 20$ in more than two hours. The increase is a result from the requirement of retrieving the result from the materialized answer inside the in-disk CAT index. With $g = 5$, the increase of response time happens when the time point of the query is more than nine hours, which is not shown in the figure. For the non-supported-index (Scan), only raw microblogs from the most recent hour are stored in the memory and the rest of the microblogs will be stored in the disk storage to have more memory space available for the index structures. Thus, scanning raw microblogs from the most recent hour data to get the top-k frequent keywords within the last hour can be done in-memory within 827 ms without accessing a disk. However, if the query contains a temporal value that is older than the most recent hour, answering the query will introduce



(a) Frequent Query Response Time (b) Trending Query Response Time (c) Flushing Based Query Latency (d) Reverse Query Response Time

Fig. 5. Query Performance Evaluation

an overhead of finding the corresponding *hour block* from the disk before scanning is performed which results in 1,130 ms query response time. Finding the corresponding *hour block* is done using a binary search. This is the main reason of the slight increase in the Scan's response time for queries on data that is older than one hour. In general, using index structures in GARNET significantly reduces the query response time of frequent queries by approximately three orders of magnitude.

Figure 5(b) shows that trending query response time gives a similar conclusion as the frequent query response time. However, it is noticeable that the non-indexed scan results in higher latency in trending queries compared to the frequent queries (approximately 3,000 ms query response time). The reason is that the trending query evaluation requires additional calculations as it checks the growth of a keyword's frequency over time rather than its frequency in a single point of time. This is not the case for the indexed options with GARNET as both frequent and trending keywords are materialized within the index nodes. In general, for both frequent and trending queries, using index structures significantly reduces the query response time for trending queries by up to four orders of magnitude depending on the residence of the data.

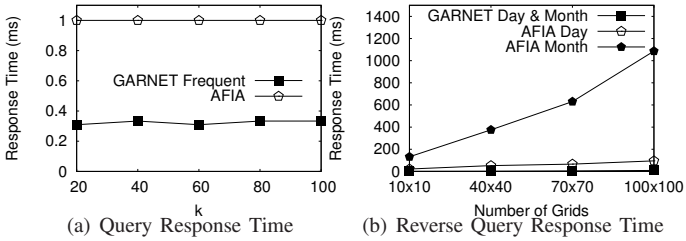
Figure 5(c) shows the trade-off between *proactive* and *lazy* flushing modes in supporting incoming queries. In general, if the incoming query is requesting data that is completely either in memory or in disk, there will be no difference between the two flushing modes as the answer is either already materialized in disk or will need to be computed in-memory for both modes anyway. The only way to show the difference is to trigger a query in which the answer is generated by aggregating both in-memory and disk-resident CAT index data, i.e., a query that asks for top-k frequent/trending keywords **today / this month / this year**. In this case, using *proactive mode* within the flushing module will result in a better query response time compared to the *lazy mode*. For each query, the query response time is roughly 20 ms better when using the *proactive mode* compared to the *lazy mode* for both frequent and trending queries.

We also evaluate the reverse query performance in GARNET with an increasing number of grid cells. Note that, the reverse query performance highly depends on the number of grid cells (context) since the reverse query requires finding a keyword in every cell of the context index layer. An example of a reverse query used in this experiment is: "*In which language and age group is the keyword love the most trending in January 2015*". Figure 5(d) compares the response time in answering a reverse query when the cells are stored in memory to when the cells are stored in a disk storage.

In general, answering a reverse trending query from memory has an average query response time of 1 ms. When the cells are stored in a disk, however, the response time of different number of cells increases significantly due to a high number of I/O cost from disk accesses. With 5, 10, and 20 grid cells, the query response time is 10 ms, 20 ms, and 38 ms, respectively. We omit the response time of answering reverse queries from non-indexed files since the number of grid cells does not affect the performance of the Scan. Answering reverse queries with non-indexed files will result in the same response time for both frequent and trending queries because the non-indexed files require a scan to be performed for all microblogs that correspond on the time constraint.

Figure 6 compares the query response time between GARNET and AFIA [24] for both *location-aware* frequent queries (Figure 6(a)) and reverse *location-aware* frequent queries (Figure 6(b)). For these experiments, we create a location-aware CAT index with different numbers of grid cells which divides the world into $n \times n$ grids with equal size. Both queries are executed from the in-memory index structures since AFIA stores all its content in the memory. For the materialized data that is stored in a disk, we first load the data into the memory and start the evaluation in memory. Thus, we do not include the overhead of loading the data from disk to memory in our evaluation. We limit our comparison to frequent queries due to the limitation of AFIA which is not able to answer trending queries. Figure 6(a) compares the response time between GARNET and AFIA in answering frequent queries with a time granularity of *day* over different numbers of k. GARNET is able to answer such frequent queries in 0.3 ms which is more than 3$\times$ faster than AFIA (1 ms). The main reason is that AFIA extends SpaceSaving algorithm which requires aggregation over a set of summaries when answering frequent queries with high time granularity. On the contrary, GARNET has the materialized answer of frequent keywords available in each grid. Thus, GARNET is able to outperform AFIA in answering location-aware top-k frequent queries. From the figure, we can also see that the query response time does not depend on the value of k for both GARNET and AFIA which is a result of materializing the answer for both techniques.

We also evaluate the performance of reverse queries between GARNET and AFIA in Figure 6(b). As mentioned earlier, to answer reverse queries, we need to scan through all possible grid cells and output the context that is most popular among all locations for the requested keyword. This applies to both GARNET and AFIA. Thus, a higher number of grid cells result in a higher query response time for both techniques. In this experiment, we used queries with different time granularity: day and $month$. The query response time of GARNET is stable



(a) Query Response Time (b) Reverse Query Response Time

Fig. 6. GARNETvs AFIA [24]

	Type	Time Horizon	Query Types
Event Detection	Location	Recent Window	Trending
GeoScope	Location	Recent Window	Geo-Correlated Trending
AFIA	Location	Recent Time Interval	Frequent
GARNET	Arbitrary	Arbitrary Time Point	Frequent & Trending

Fig. 7. Frequent/Trending Items in Microblogs: Event Detection [1], [3], [9], [14], [16], [20], [21], [29]; GeoScope [4]; AFIA [24]

at 0.3-9.4 ms for any time granularity since we materialize the top-k frequent keywords in every level of the temporal tree. Thus, accessing any level of the temporal tree in GARNET requires the same amount of effort. On the contrary, with a coarser temporal granularity, each grid in AFIA contains more summaries. Hence, AFIA encounters more overhead in the combining operations which increases the response time significantly with query response time of 21.75-95 ms for day and 132-1,000 ms for $month$.

IX. Related Work

Discovering trending items has been extensively studied in the literature of streaming data [2], [5], [6], [11], [15], [17], where the main focus was to discover items with highest frequency either since the beginning of the stream [5], [15], [17] or over a time sliding window [2], [6], [11]. However, all these techniques have two main limitations that hinder their applicability to microblogs applications. First, they mainly process *continuous* queries, which is the dominant type of queries in data streams. This is unlike queries on microblogs where snapshot queries are highly important. Second, they mostly focus on finding frequent items rather than trending items, while the notion of trend discovery is important and has several applications in microblogs [1], [14], [21].

To address these two limitations, a whole literature of trend discovery in microblogs have been proposed [3], [4], [16], [24], [30]. Table 7 classifies these techniques based on their temporal horizon, query type, and context type. With the importance of real-time microblogs, all existing techniques are optimized to support recent time horizons, either as a sliding time window of last T time units or as a time interval within the recent time horizon. Regarding query types, existing techniques address either frequent or trending items discovery, based on the supported application. Event detection applications [1], [3], [9], [14], [16], [20], [21], [29], however, are more focused on grouping several trending keywords together to report an event rather than focusing on the scalability and performance of retrieving trending keywords. Thus, these applications are orthogonal to our work in a way that GARNET can use one of these techniques in order to report an event based on the returned trending keywords.

With recent importance of location information on microblogs, recent existing techniques [4], [24] are tailored to support location-aware queries. Geoscope [4] answers a different type of trending queries, called geo-correlated trending queries, which returns top-k trending keywords that are trending only at a certain predefined location and not at other locations. On the other hand, AFIA [24] returns top-k frequent keywords at different locations in recent time interval. However, AFIA is only able to answer frequent keywords queries and is not geared towards answering trending queries. The reason is that AFIA uses extended SpaceSaving algorithm in retrieving their top-k frequent keywords in which the technique will remove keywords other than the top-k frequent keywords. These removed keywords, however, will be used in order to calculate and find the top-k trending keywords in microblogs.

GARNET distinguishes itself from all existing techniques in the four main aspects: (1) GARNET is the first to address queries within arbitrary context on arbitrary attributes. All existing techniques either support no context or only support the location context. (2) GARNET supports queries on arbitrary time points with fixed granularity. Hence, arbitrarily old data can still be queried, for example, to support social media analysis for the past few months. (3) GARNET is the first to support discovering both frequent and trending items in the same system. Existing work cannot be adapted to answer both types of queries simultaneously. (4) Overall, GARNET is a system solution that acts as a one-stop solution for a myriad of various frequent and trending queries on microblogs. It is more appealing to industry and more practical to be realized. Contrast GARNET approach to the case of realizing tens of various algorithms within one system to be able to support various forms of trending queries.

X. Conclusion

In this paper, we presented GARNET; a holistic system equipped with one-stop efficient and scalable solution for supporting a generic form of *context-aware* trending keywords queries on microblogs on the form: "*Find top-k* ***frequent/trending*** *keywords within an arbitrary* ***current/recent/past*** *time window of fixed granularity according to a certain* ***context***". Unlike all previous attempts, GARNET supports both frequent and trending queries, any arbitrary time interval and having a set of arbitrary filters over contextual attributes. This makes GARNET a very appealing solution for industry. From a system point of view, one needs to realize GARNET once in the system, then, a myriad of various forms of trending and frequent queries are immediately supported efficiently. This is in contrast to realizing tens of various algorithms to be able to support various forms of trending queries. GARNET employs in-memory and disk-resident *Context-Aware Temporal* (CAT) index structure as infrastructure to materialize top-k frequent and trending keywords within certain combinations of arbitrary context and time granularity. Similar to database management systems (DBMSs), GARNET allows its user to create/drop CAT indexes on demand to tune its

performance depending on incoming *context-aware* trending queries workloads. Experimental studies are done by deploying a real system prototype of GARNET and using billions of real microblogs data from Twitter. The experiments show that GARNET is able to digest up to 8× the Twitter rate with eight different index structures and is capable to provide up to 0.3 miliseconds of query response time in answering both top-k frequent and top-k trending queries of any given point of time.

REFERENCES

[1] H. Abdelhaq, C. Sengstock, and M. Gertz, "Eventweet: Online localized event detection from twitter," *PVLDB*, vol. 6, no. 12, pp. 1326–1329, 2013.

[2] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *SIGMOD*, 2004, pp. 286–296.

[3] H. Becker, M. Naaman, and L. Gravano, "Beyond trending topics: Real-world event identification on twitter." *ICWSM*, vol. 11, pp. 438–441, 2011.

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

[5] G. Cormode and M. Hadjieleftheriou, "Finding Frequent Items in Data Streams," in *PVLDB*, 2008.

[6] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SICOMP*, vol. 31, no. 6, pp. 1794–1813, 2002.

[7] "Domo," https://www.domo.com/solution/twitter-reporting-dashboard.

[8] "Facebook statistics." https://www.facebook.com/business/power-of-advertising.

[9] 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.

[10] V. Gaede and O. Günther, "Multidimensional Access Methods," *ACM Computing Surveys*, vol. 30, no. 2, pp. 170–231, 1998.

[11] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, "Identifying frequent items in sliding windows over on-line packet streams," in *SIGCOMM on Internet Measurement Conference*, 2003, pp. 173–178.

[12] A. Guttman, *R-tree: a dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[13] J. F. Kenney and E. S. Keeping, *Mathematics of Statistics, Part 1*, 3rd ed. van Nostrand, 1962, ch. 15, pp. 252–285.

[14] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang, "Tedas: A twitter-based event detection and analysis system," in *ICDE*, 2012, pp. 1273–1276.

[15] G. S. Manku and R. Motwani, "Approximate Frequency Counts Over Data Streams," in *PVLDB*, 2002.

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

[17] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *ICDT*. Springer, 2005, pp. 398–412.

[18] A. Mislove, S. Lehmann, Y.-Y. Ahn, J.-P. Onnela, and J. N. Rosenquist, "Understanding the demographics of twitter users." *ICWSM*, vol. 11, p. 5th, 2011.

[19] B. O'Connor, R. Balasubramanyan, B. R. Routledge, and N. A. Smith, "From tweets to polls: Linking text sentiment to public opinion time series." *ICWSM*, vol. 11, no. 122-129, pp. 1–2, 2010.

[20] N. Pervin, F. Fang, A. Datta, K. Dutta, and D. Vandermeer, "Fast, scalable, and context-sensitive detection of trending topics in microblog post streams," *TMIS*, vol. 3, no. 4, Jan. 2013.

[21] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: real-time event detection by social sensors," in *WWW*, 2010, pp. 851–860.

[22] H. Samet and R. E. Webber, "Storing a collection of polygons using quadtrees," *TOG*, vol. 4, no. 3, pp. 182–222, 1985.

[23] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling, "Twitterstand: news in tweets," in *SIGSPATIAL*, 2009, pp. 42–51.

[24] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen, "Scalable Top-k Spatio-Temporal Term Querying," in *ICDE*, 2014.

[25] B. E. Teitler, M. D. Lieberman, D. Panozzo, J. Sankaranarayanan, H. Samet, and J. Sperling, "Newsstand: A new view on news," in *SIGSPATIAL*, 2008, p. 18.

[26] "Twitter statistics." https://about.twitter.com/company.

[27] "Twitter," https://twitter.com/.

[28] "Unifiedsocial," www.unifiedsocial.com/.

[29] M. Walther and M. Kaisser, "Geo-spatial event detection in the twitter stream," in *Advances in Information Retrieval*, ser. Lecture Notes in Computer Science, P. Serdyukov, P. Braslavski, S. Kuznetsov, J. Kamps, S. Rüger, E. Agichtein, I. Segalovich, and E. Yilmaz, Eds. Springer Berlin Heidelberg, 2013, vol. 7814, pp. 356–367. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36973-5_30

[30] J. Weng and B.-S. Lee, "Event detection in twitter." *ICWSM*, vol. 11, pp. 401–408, 2011.

APPENDIX

A. TREND LINES IN GARNET

GARNET uses statistical linear regression slope to measure the trendiness of a certain keyword. The following Lemma derives the equation that determines the trendiness of a keyword:

Lemma 1: *Given a keyword consecutive frequencies vector f = [f_0, f_1,..., f_N], the keyword trend line can be estimated with the following formula:*

$$T_{trend} = \frac{\sum_{i=1}^{N}[i \times (f_i - f_0)]}{N(N+1)(2N+1)} \tag{1}$$

Proof: The simple linear regression slope T_{trend} of x and y is given with the following equation:

$$T_{trend} = \frac{Mean(xy)}{Mean(x^2)} \tag{2}$$

Where $Mean(x)$ is the average value of the vector and xy is a vector that results from value-wise multiplication of the vectors x and y. In GARNET, the vector x values are always constants while the vector y contains the frequencies of a keyword W. Thus values of vector x are always be $[1, 2, 3, ..., N]$ while values of vector y are $[f_1, f_2, f_3, ..., f_N]$. Thus, $Mean(x^2)$ can be simplified as $\frac{(N+1)(2N+1)}{6}$ respectively. On the other hand, $Mean(xy)$ can be calculated as $\frac{\sum_{i=1}^{N} i \times f_i}{N}$. Substitutes both variables to Equation 1:

$$T_{trend} = \frac{\frac{\sum_{i=1}^{N} i \times f_i}{N}}{\frac{(N+1)(2N+1)}{6}} = \frac{6\sum_{i=1}^{N} i \times f_i}{n(n+1)(2n+1)} \tag{3}$$

The equation above assumes that the measurement is used from the start of the stream and each keyword W starts from frequency 0. However, in GARNET, we need to consider the start position of a keyword W by using the previous frequency, namely f_0. Thus, the equation above can be modified to:

$$T_{trend} = \frac{6\sum_{i=1}^{N}[i \times (f_i - f_0)]}{N(N+1)(2N+1)} \tag{4}$$