FAST: *F*requency-*A*ware Indexing for *S*patio-*T*extual Data Streams

Ahmed R. Mahmood [#], Ahmed M. Aly [*], Walid G. Aref [#]

[#] *Purdue University, West Lafayette, IN*
[#] {amahmoo,aref}@cs.purdue.edu

[*] *Google Inc., Mountain View, CA*
[*] aaly@google.com

Abstract—**Many applications need to process massive streams of spatio-textual data in real-time against continuous spatio-textual queries. For example, in location-aware ad targeting publish/subscribe systems, it is required to disseminate millions of ads and promotions to millions of users based on the locations and textual profiles of users. In this paper, we study indexing of continuous spatio-textual queries. There exist several related spatio-textual indexes that typically integrate a spatial index with a textual index. However, these indexes usually have a high demand for main-memory and assume that the entire vocabulary of keywords is known in advance. Also, these indexes do not successfully capture the variations in the frequencies of keywords across different spatial regions and treat frequent and infrequent keywords in the same way. Moreover, existing indexes do not adapt to the changes in workload over space and time. For example, some keywords may be trending at certain times in certain locations and this may change as time passes. This affects the indexing and searching performance of existing indexes significantly. In this paper, we introduce FAST, a *F*requency-*A*ware *S*patio-*T*extual index for continuous spatio-textual queries. FAST is a main-memory index that requires up to one third of the memory needed by the state-of-the-art index. FAST does not assume prior knowledge of the entire vocabulary of indexed objects. FAST adaptively accounts for the difference in the frequencies of keywords within their corresponding spatial regions to automatically choose the best indexing approach that optimizes the insertion and search times. Extensive experimental evaluation using real and synthetic datasets demonstrates that FAST is up to 3x faster in search time and 5x faster in insertion time than the state-of-the-art indexes.**

I. INTRODUCTION

Nowadays, many applications rely on processing and analyzing spatio-textual data. Example applications include social networks (e.g., Facebook), micro-blogs (e.g., Twitter), web search for local places and events, and location-aware ad targeting. These applications process spatio-textual data at a massive scale and in real-time. For example, 500 million tweets [1, 2] and 9 million Foursquare check-ins [3] are being generated and processed daily. These applications require efficient spatio-textual indexing to support this scale of spatio-textual data.

In this paper, we focus on the indexing of continuous spatio-textual filter queries. This type of queries appears in many applications, e.g., location-aware publish/subscribe systems [4], information dissemination [5], and sponsored search [6]. A continuous spatial-keyword filter query consists of a spatial range and an associated set of keywords. For a stream of
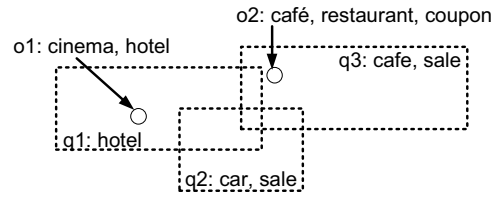

Fig. 1: E-coupon example.

spatio-textual objects, a continuous spatio-textual filter query identifies the objects that fall inside the spatial range of the query and that contain all the keywords of the query.

Example 1: Figure 1 illustrates a sample location-aware e-coupon application in a location-aware publish/subscribe system. Three users show interest in promotions represented by the three continuous spatio-textual queries $q1$, $q2$, and $q3$. Promotion $o1$ matches Query $q1$ because $o1$ is located inside $q1$'s spatial range, and contains all the keywords of $q1$.

Recently, several access-methods have been proposed to handle continuous spatio-textual queries in streaming environments, e.g., [1, 7, 8]. These access methods integrate a spatial index (e.g., a spatial grid, the R-tree [9], or the quad-tree [10]) with a textual index (e.g., the inverted list [11], or the ordered-keyword trie [12]). However, these access-methods do not account for the frequencies and the popularity of some of the keywords within the indexed spatio-textual queries. Consider Figure 2 that illustrates the frequencies of keywords in a set of 50,000 tweets. The frequencies of the keywords follow a Zipfian distribution [13]. This distribution has many infrequent keywords and few frequent keywords. Although the distribution of the frequencies of keywords is Zipfian, the exact ranking and frequencies of keywords may not be known and the frequencies of keywords may change overtime. Also, new keywords get introduced to the vocabulary and it is estimated that 1000 new words are added to the Oxford dictionary every year[1]. Also, some infrequent keywords may become frequent, e.g., Hurricane Irma. Furthermore, the distribution of the frequent keywords is non-uniform across the space.

Existing indexes treat queries with frequent keywords in the same way as they treat queries that contain infrequent keywords. For example, when using inverted lists [11], a query is indexed based on a single keyword. This keyword

---

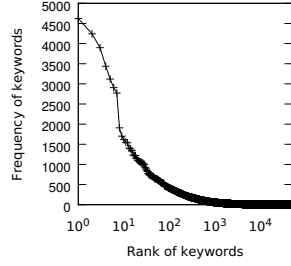[1]http://blog.oxforddictionaries.com/august-2013-update

Fig. 2: Zipfian distribution of query keywords.

is usually the least-frequent keyword. Inverted lists are well-suited for queries with infrequent keywords. However, the inverted list structure has the following two limitations: (1) it suffers from poor performance for queries that only have frequent keywords because the inverted lists associated with these frequent keywords can be very large, and (2) it assumes knowledge of the entire vocabulary of keywords and their frequencies. However, in real scenarios, e.g., when processing tweets, the entire vocabulary and the ranking of the keywords are not known a priori.

Another popular textual index is the ordered-keyword trie [12] that is a variation of the traditional trie structure [14]. The ordered-keyword trie indexes keywords instead of characters in the traditional trie structure. The ordered-keyword trie offers better textual filtering for queries with no infrequent keywords. However, the ordered-keyword trie suffers from the following limitations: (1) it has a large memory-footprint, and (2) it does not quickly prune queries with infrequent keywords unless the indexed keywords have a total order based on their frequencies. Having a total order of keywords based on their frequencies requires prior knowledge of the entire vocabulary of keywords and their frequencies, which may not be feasible. It is challenging to support efficient indexing of continuous spatio-textual queries in a streaming environment due to the following reasons:

- The massive scale of the indexed queries as it is typical to deal with millions of rapidly arriving continuous queries.
- Spatio-textual objects are streamed at a high rate, and it is required to process these objects against millions of indexed queries with minimal latency.
- The locations and frequencies of spatio-textual data and queries are not uniformly distributed. Hence, an efficient index needs to account for the varying distributions of spatial and textual aspects of the indexed queries.
- The assumption of knowing the entire vocabulary of keywords in advance is not valid in many situations, e.g., as in processing social media posts.

To address these challenges, we introduce FAST, a *F*requency-*A*ware *S*patio-*T*extual access method for indexing continuous queries in a streaming environment.

FAST treats the frequencies of keywords and their distribution in space as first-class properties of spatio-textual queries. FAST integrates a variant of the incomplete *spatial pyramid* structure [15] with a new textual index, termed *the adaptive*

keyword index (AKI) to boost the spatial and textual pruning power of FAST. The spatial pyramid is a multi-resolution spatial index that is being adopted in many spatio-textual indexes, e.g., [16, 17]. AKI accounts for the frequencies of the keywords, and automatically distinguishes between frequent and infrequent keywords. AKI allows FAST to quickly prune queries that have infrequent keywords. Queries that have no infrequent keywords are indexed in a more selective way in FAST. Moreover, instead of searching for all the keywords at all the levels of the pyramid, FAST adopts *frequency-aware spatial indexing*, where queries containing infrequent keywords are indexed only at the top level of the spatial pyramid. This reduces the number of keywords being searched while descending the spatial pyramid. Because of this frequency awareness, FAST is 3x faster than the state-of-the-art indexes in terms of search time.

The textual index AKI is designed to reduce the memory footprint of FAST by distinguishing between queries with no infrequent keywords and queries that have some infrequent keywords. FAST requires less memory for queries that have some infrequent keywords by attaching the queries only to the least-frequent keyword and not to all keywords in the query. Also, FAST improves the pruning power for queries with no infrequent keywords by attaching these queries to longer sequences of cascaded keywords that appear in the query. Hence, FAST demands more space only when higher pruning power is needed. When queries span multiple spatial nodes inside FAST's spatial pyramid, FAST adopts a spatial sharing technique to further reduce its memory footprint. These optimizations results in reducing the memory footprint of FAST by up to one-third of that of the state-of-the-art indexes.

FAST does not require prior knowledge of the entire vocabulary of keywords or their frequencies. FAST captures this information dynamically as queries get inserted or deleted. Also, FAST employs a lazy cleaning mechanism that removes the expired queries and updates the index structure to reflect the current frequencies of the query keywords.

The main contributions of this paper are as follows:

- We introduce FAST, a frequency-aware spatio-textual index for continuous spatio-textual filter queries in a streaming environment. FAST is equipped with a new *adaptive keyword index (AKI)* that adaptively accounts for the frequencies of keywords and does not require prior knowledge of the vocabulary of keywords or their frequencies.
- FAST is designed as a light-weight index that uses a *frequency-aware spatial pyramid* and *spatial-sharing of query lists* to improve the performance of the search operation with an optimized memory footprint.
- We propose a light-weight cleaning mechanism that lazily removes the expired queries, and dynamically re-adjusts the structure of the index to account for changes in the frequencies of the keywords of the indexed queries.
- We present a mathematical analysis that aids in tuning the parameters of FAST.

- We conduct an extensive performance study of FAST using real and synthetic datasets. When compared to the state-of-the-art indexes, results demonstrate that FAST is 3x faster in search time, 5x faster in insertion time, and requires up to one-third of the memory needed by the state-of-the-art index.

The rest of this paper proceeds as follows: Section II presents notations used throughout the paper and presents the data structures related to FAST. The structure and the main algorithms of FAST are presented in Section III. The performance evaluation of FAST is presented in Section IV. Section V highlights the related work, and Section VI contains concluding remarks.

II. PRELIMINARIES

In this section, we introduce the problem definition, and describe the data structures relevant to FAST.

A. Problem Definition

A **spatio-textual data object**, say o, is of the form $o = [oid, loc, text]$, where oid is the identifier of the object, loc is the geo-location of the object, and $text$ is the set of keywords associated with the data object.

A **continuous spatio-textual filter query**, say q, is of the form $q = [qid, MBR, text, t_{exp}]$, where qid is the identifier of the query, MBR is the spatial range of the query represented as a minimum bounding rectangle, i.e., $[x_{min}, y_{min}, x_{max}, y_{max}]$, and $text$ is the set of keywords associated with the query. The continuous query q remains registered in the index until Timestamp t_{exp}, where t_{exp} is the expiration timestamp of the query.

For a streamed spatio-textual data object, say o, the objective is to match o with all the continuous queries that have their spatial and textual criteria satisfied by o's location and textual data. The formal definition of spatio-textual matching is as follows:

Definition 1: Spatio-Textual Matching. A spatio-textual data object o matches a continuous spatio-textual query q when the spatial location of the object, i.e., $o.loc$, is located inside the spatial range of the query $q.MBR$, i.e., and when the keywords of the object, i.e., $o.text$, contain all the keywords of the query, i.e., $q.text$.

Problem Statement. In this paper, we study the problem of *matching* an unbounded stream of spatio-textual objects O against a set of continuous spatio-textual queries Q.

Example 2: We use the example given in Figure 3 throughout the rest of the paper. The figure contains the following nine continuous queries $\{q_1, \cdots, q_9\}$. Spatio-textual Object o_1 falls inside the spatial range of Queries q_1, and q_7. However, $o_1.text$ fully contains the keywords of only $\{q_1\}$. Thus, q_1 is reported as the result of matching o_1 against indexed queries.

B. Related Structures

In FAST, we integrate a spatial index with a new textual indexing approach, termed the *adaptive keyword index AKI*. To motivate the need for AKI, we describe existing textual
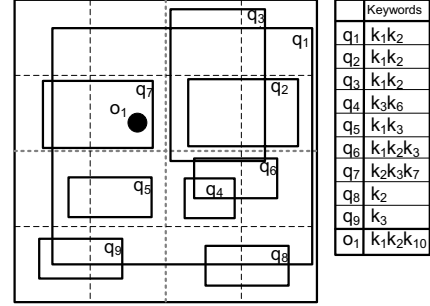

Fig. 3: Running example.

indexing approaches and discuss their limitations. The two most widely adopted textual indexing approaches are: (1) the *ranked-keyword inverted list (RIL)* [11], and (2) the *ordered-keyword trie (OKT)* [12]. In addition to describing related textual indexes, we outline the structure of the AP-tree, the state-of-the-art spatio-textual index [1].

RIL [11] is a data structure for indexing textual items that contain multiple keywords. A spatio-textual filter query, say q, can be regarded as a textual item as it contains a set of keywords, i.e., in $q.text$. In RIL, textual items are usually indexed based on their least-frequent keyword. Every keyword has a *posting list* of textual items attached to this keyword. Figure 4(a) illustrates textual-only indexing of the queries in Example 2 using RIL. The keywords are usually ranked based on prior knowledge of their frequencies. This imposes a limitation on the efficiency of RIL as prior knowledge of the vocabulary of keywords and their frequencies may not be feasible. RIL has low memory requirements and has good search performance for objects indexed on infrequent keywords that have short posting lists, e.g., as in the posting lists attached to keywords k_6 and k_7. The performance of RIL deteriorates when searching for frequent keywords that have long posting lists, e.g., k_1. In Figure 4(a), the posting lists of the dotted keywords are visited when searching for the keywords of O_1 in Example 2. Because textual items are indexed on a single keyword, search in RIL requires an additional verification step to remove queries whose keywords are not fully contained in the search keywords, e.g., when searching for queries that match the keywords of o_1, q_6 is initially retrieved as part of the posting list of k_1. q_6 is removed as $q_6.text \not\subset o_1.text$.

OKT [12] is a variation of the traditional trie structure [14] for indexing textual items. Figure 4(b) illustrates textual-only indexing of the queries in Example 2 using OKT. In this figure, keywords are assumed to be ordered lexicographically. OKT offers better textual filtering than RIL for objects with no infrequent keywords. However, OKT has higher memory requirements than RIL and does not provide early pruning for indexed objects that contain infrequent keywords, e.g., k_6 and k_7. Search in OKT follows the traditional trie search algorithm. For example, in Figure 4(b), the shaded queries attached to dotted keywords are retrieved as the resultset when searching for the keywords of O_1 in Example 2. In contrast to RIL,

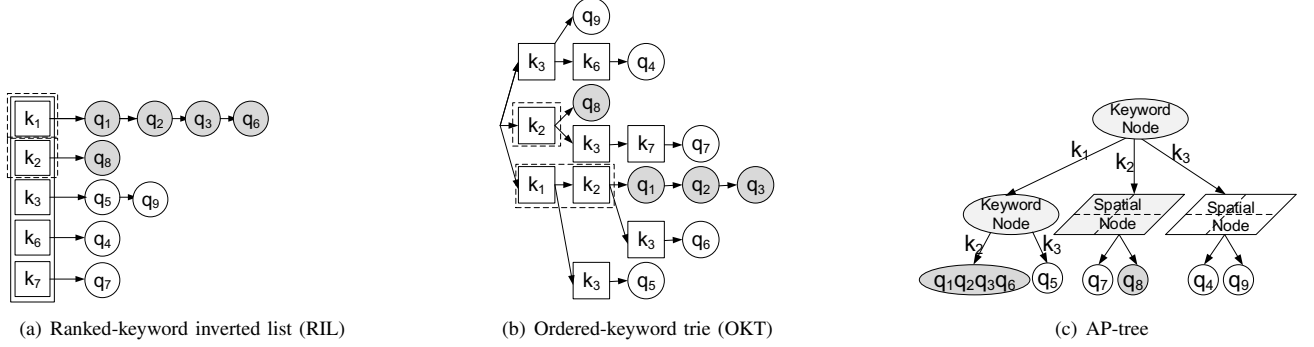(a) Ranked-keyword inverted list (RIL) (b) Ordered-keyword trie (OKT) (c) AP-tree

Fig. 4: Relevant textual indexes and spatio-textual indexes.

no additional verification is required when searching OKT as indexing in OKT is based on all the keywords of the indexed item.

The AP-tree [1] is the current state-of-the-art structure for indexing continuous spatio-textual queries in a streaming environment. When indexing queries, the AP-tree arbitrates between spatial and textual partitioning using an expensive cost function. The AP-tree integrates spatial decomposition using a variation of OKT. The main limitations of the AP-tree are: (1) the AP-tree does not account for the frequencies of keywords to prune queries having infrequent keywords, and (2) the AP-tree is based on the memory intensive OKT, and has a large memory footprint. Figure 4(c) illustrates how the queries in Example 2 are indexed using the AP-tree. Matching in the AP-tree visits all relevant spatial and textual nodes. Figure 4(c) illustrates the spatial and the textual nodes, i.e., the shaded nodes, that are visited when matching O_1 in Example 2. The AP-tree requires a verification step to remove non-relevant queries, e.g., q_6.

III. FAST INDEX DESIGN AND ALGORITHMS

Given the inherent property that the frequencies of keywords follow a Zipfian distribution, an efficient spatio-textual index needs to account for the frequencies of occurrence of the keywords in real-time and to distinguish between the frequent and infrequent ones.

We equip FAST with a new textual index termed, the *adaptive keyword index (AKI)*. AKI is a text-only index and does not have any spatial discrimination abilities. AKI is integrated with a spatial pyramid to distinguish between queries that are indistinguishable textually. Figure 5(b) illustrates an AKI that textually indexes all queries in Example 2.

AKI is designed as a *multi-level **hash map** of **textual nodes*** with keywords as the key to the hash map (see Figure 5). A textual node, say N_t, contains one or both of the following: (1) a list of queries attached to this node, i.e., $N_t.qlist$, and (2) a hash map to children's textual nodes with keywords as the key of the hash map, i.e., $N_t.children$. Textual nodes are identified using a unique ***textual-path*** of keywords, e.g., in Figure 5(b), Query q_5 is attached to Textual Node $[k_1k_3]$, as q_5 is stored under the path k_1, k_3, where Keywords k_1 and k_3 are the keywords in q_5.

Textual nodes in AKI are assigned to levels. A ***top-level*** textual node has no parent textual node and is identified using a textual-path with a single keyword, e.g., in Figure 5(b), Textual Nodes $[k_1]$, $[k_2]$, $[k_3]$, $[k_6]$, and $[k_7]$ are top-level textual nodes. ***Leaf*** textual nodes do not have child nodes, e.g., in Figure 5(b), Textual nodes $[k_7]$, $[k_1k_3]$ are leaf textual nodes. Levels of textual nodes in AKI are incrementally numbered, where the top level is numbered Level 1, as illustrated in Figure 5.

Also, for every keyword, say k_i, the total number of queries having k_i in their textual content, is stored in a hash table termed the ***frequencies map***. For example, in Figure 5(a), the frequencies map indicates that there are five queries containing Keyword k_1, i.e., q_1, q_2, q_3, q_5 and q_6.

In AKI, queries are first indexed to top-level textual nodes using their least-frequent keyword similar to RIL. The least-frequent keyword is identified using the ***frequencies map*** and not using prior ranking of the keywords. A textual node remains infrequent as long as the number of queries that must be attached to this node in the RIL manner, i.e., the queries do not have any other infrequent keywords to be attached to, is less than a specific threshold, termed the *frequent-keyword threshold* θ.

*Definition 2: **Frequent-keyword** threshold θ.* The *frequent-keyword* threshold distinguishes between the infrequent and the frequent textual nodes. Initially, all textual nodes are infrequent and queries are indexed in the RIL manner, i.e., using a single keyword. When the number of queries that must be attached to an infrequent textual node, say N_t, in the RIL manner exceeds θ, N_t is marked as frequent.

For example, in Figure 5(a), assume that the *frequent-keyword* threshold is two. Before inserting q_9, the number of queries attached to all textual nodes is ≤ 2 and all textual nodes are top-level and are infrequent. q_9 has a single keyword q_3 and the Textual Node $[k_3]$ has two queries attached to it. First, we attempt to transfer some of the queries attached to $[k_3]$ to any other infrequent textual node. However, this is infeasible as q_5 and q_6 only contain keywords k_1, k_2, and k_3 and Textual Nodes $[k_1]$, $[k_2]$, and $[k_3]$ have θ queries attached to them. Hence, Textual Node $[k_3]$ is marked as frequent and all queries attached to $[k_3]$ get inserted to frequent textual
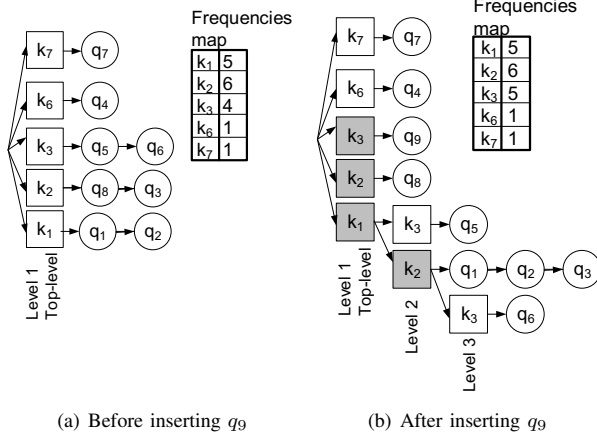
(a) Before inserting q_9 (b) After inserting q_9

Fig. 5: The adaptive keyword index AKI.

nodes using a lexicographic ordering of their keywords. We use the lexicographic ordering of keywords as we assume no prior knowledge of the frequencies of keywords, and we cannot use the *frequencies map* to provide a total order on the keywords because values in the *frequencies map* change over time with the insertion and removal of queries.

This requires marking Textual Nodes k_1 and k_2 as frequent as well. In Figure 5(a), queries attached to textual Nodes $[k_1]$, $[k_2]$, and $[k_3]$ will be re-attached to these textual nodes using the first keyword in their textual content according to a lexicographic ordering of keywords. In Figure 5(b), q_9 is attached to $[k_3]$. Also, q_1, q_2, q_3, q_5,and q_6 should be attached to $[k_1]$. However, the number of queries attached to the *frequent* Textual Node $[k_1]$ exceeds θ. The level of $[k_1]$ is 1 and AKI uses more keywords to distinguish queries to be attached to $[k_1]$ as illustrated in Figure 5(b). Textual Nodes $[k_1k_2]$ and $[k_1k_3]$ are created at Level 2, and Textual Node $[k_1k_2k_3]$ is created at Level 3 to distinguish between q_1, q_2, q_3, q_5, and q_6 textually. $[k_1k_3]$ is marked as infrequent because only q_5 is attached to it. $[k_1k_2]$ is marked as frequent because the textually indistinguishable queries q_1, q_2, and q_3 are attached to it.

Although the number of queries attached to $[k_1k_2]$ exceeds θ, these queries are indistinguishable textually and contain exactly the same keywords. Hence, no further discrimination can be performed by the AKI because AKI is a text-only index. AKI lacks any spatial discrimination power, and if we desire to spatially distinguish between queries attached to $[k_1k_2]$, we need to integrate spatial pruning abilities with AKI. FAST integrates AKI with a spatial pyramid to combine spatial and textual pruning abilities.

Notice that, in contrast to RIL, AKI attempts to restrict the length of the lists of queries attached to textual nodes to prevent long lists of queries. However, AKI may contain long lists of queries that are textually **indistinguishable**. It is a desirable property of textual indexes to group textually indistinguishable queries. Also, AKI has lower space requirements than that of OKT as AKI requires a lower number of index nodes as illustrated in Figures 4(b) and 5(b).

AKI is adaptive and uses the *frequent-keyword* threshold to create more textual nodes when a higher level of textual discrimination is required. The *frequent-keyword* threshold θ is very crucial to the performance of FAST. We discuss the experimental tuning of the value of θ, and compare the performance of AKI against both RIL and OKT in Section IV.
Converting Frequent Textual Nodes to Infrequent Ones. AKI keeps track of the frequencies of the keywords of the indexed queries in the *frequencies map*. Whenever a query is removed, the frequencies of the keywords of the removed query are updated in the *frequencies map*. This maintains the dynamic differentiation between frequent and infrequent keywords. Also, updating the *frequencies map* enables converting frequent textual that are no longer frequent, and marking them as infrequent as explained in Section III-A3.
Frequency-Aware Spatial Indexing Adaptive textual indexing using AKI is insufficient for indexing spatio-textual queries that share the same set of keywords and have different spatial locations. An efficient spatio-textual index needs to adapt to the spatial and textual selectivities of the indexed queries. In Figure 5(b), Queries q_1, q_2, and q_3 are attached to the textual node $[k_1k_2]$ and cannot be distinguished from each other textually. However, these queries are located at different spatial regions, i.e., can be distinguished from each other spatially. In FAST, we integrate the *spatial pyramid* with *AKI* to achieve spatio-textual pruning. The spatial pyramid is a multi-level and a multi-resolution index. Every level in the spatial pyramid contains a spatial grid with a specific granularity. Levels in the spatial pyramid are numbered bottom up and level 0 is the lowest pyramid level.

Definition 3: **Granularity** at pyramid level i *gran(i):* is the number of pyramid nodes per dimension at level i.
The top level of the pyramid has a single pyramid node covering the entire indexed space and has a granularity of one. The second level from the top in the spatial pyramid has a granularity of two and contains four cells that covers the entire space.

Let $gran_{max}$ be the maximum supported granularity in FAST. $gran_{max}$ is the pyramid granularity at $level\ 0$, i.e., the lowest pyramid level. The top level in the spatial pyramid is numbered $log_2(gran_{max})$, e.g., if $gran_{max}$ equals 2, the top level in the spatial pyramid is numbered 1. We discuss the experimental tuning of $gran_{max}$ in Section IV.

We calculate the granularity at level i as follows:
$$gran(i) = \frac{gran_{max}}{2^i i} \tag{1}$$
Definition 4: **SideLen(i)** is the side length of a spatial pyramid node at level i.

We define $SideLen_{min}$ as the smallest possible side length size in the spatial pyramid. $SideLen_{min}$ is the side length of spatial pyramid nodes at $level\ 0$, i.e., the lowest spatial pyramid level. We calculate the side length of spatial pyramid nodes at level i as follows:
$$SideLen(i) = SideLen_{min} \times (2^i) \tag{2}$$
Every spatial pyramid node within any level, say i, has a specific spatial coordinate. To map a spatial location, say
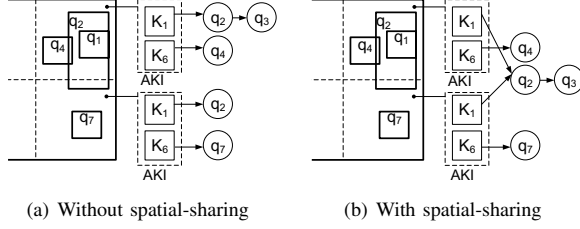
(a) Without spatial-sharing   (b) With spatial-sharing

Fig. 6: Spatial-sharing of query lists.

(x_1, y_1) into the spatial coordinate $(x_c(i), y_c(i))$ of a pyramid node at Level i, we use the following equations:

$$x_c(i) = \lfloor x_1/SideLen(i) \rfloor$$
$$y_c(i) = \lfloor y_1/SideLen(i) \rfloor \qquad (3)$$

To reduce the space required by the spatial pyramid, only spatial pyramid nodes that contain queries are instantiated. Empty spatial pyramid nodes are not instantiated and do not consume any memory, e.g., the shaded spatial pyramid node within Level 0 in Figure 7. To support this space optimization, all spatial pyramid nodes are accessed using a **hash table**. The key to the hash table is the address of the spatial pyramid node. The value is a pointer to the spatial pyramid node. The address of a spatial pyramid node is calculated using a function of the level number i and the grid **coordinates** (x_c, y_c) of the spatial pyramid node as follows:

$$address(i, x_c, y_c) = i \times gran_{max}^2 + y_c \times gran(i) + x_c \quad (4)$$

For example, the address of the spatial pyramid node at Level 0 with grid coordinates $(1, 0) = 0 \times 2^2 + 0 \times 2 + 1 = 1$.

Spatial-Sharing of Query Lists. Each spatial pyramid node contains an AKI instance. To optimize the space required by FAST, we share lists of queries when a query spans multiple spatial pyramid nodes while being attached to infrequent top-level AKI textual nodes. Figure 6(a) illustrates two spatial pyramid nodes with two separate AKI indexes. Notice that Query q_3 spans two spatial nodes as illustrated in Figure 3. In Figure 6(b), we avoid creating two separate lists of queries to be attached to Keyword k_1. We share a single list of queries between two AKI indexes. This reduces the space required for one list of queries. Spatial-sharing of query lists happens at the granularity of keywords. For example, In Figure 6(b), query lists attached to Textual Nodes $[k_6]$ and $[k_7]$ are not shared as these lists do not contain any queries that span more than one spatial pyramid node.

Notice that when a spatial pyramid node, say N_p, employs spatial-sharing of query lists with another spatial pyramid node, say N_p', N_p may also point to a few extra queries that overlap only with Node N_p'. For example, in Figure 6(b), Query q_1 is attached to both spatial pyramid nodes. However, q_1 spans only the top spatial pyramid node. This is acceptable and does not introduce overhead in the matching processing as the length of the shared lists is restricted to the *frequent-keyword* threshold θ. Before marking an AKI textual node as frequent, we check if a spatially-shared query list is attached to the AKI textual node. We separate the spatially-shared lists and remove non-spatially overlapping queries to reduce the length of $qlist$ that is attached to the AKI textual node to
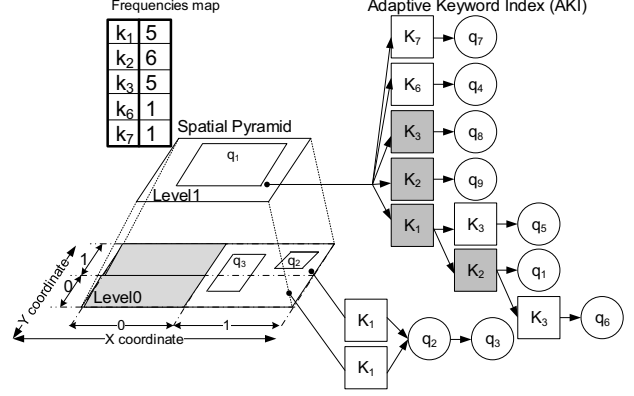

Fig. 7: The structure of FAST.

prevent unnecessary marking of textual nodes as frequent.

Frequency-Aware Spatio-textual Indexing. Initially, all queries are indexed at the top level of the spatial pyramid. When the number of queries attached to a frequent AKI textual node exceeds a specific threshold, e.g., Textual Node $[k_1 k_2]$ in Figure 5(b), queries are partitioned to descend to a lower spatial pyramid level, i.e., to a spatial pyramid level with higher resolution. A spatial pyramid node in any spatial pyramid level, say i, other than Level 0, potentially covers four children spatial nodes in the spatial pyramid level directly below i, i.e., $i - 1$. When the number of queries attached to a frequent textual node exceeds 4θ, the queries are sorted based on their ranges. Queries having area less than the median of the sorted query list descend to a lower pyramid level. Queries with smaller ranges are chosen to descend as these queries have higher probability of joining different spatial nodes at the lower pyramid level. This adaptively captures the difference in frequencies across different spatial regions.

Figure 7 illustrates the hybrid structure of FAST. In this figure, a two-level spatial pyramid is integrated with AKI. Assume that the *frequent-keyword* threshold is two. For the sake of illustration, assume that queries descend when the number of queries attached to a frequent textual node exceed $1 \times \theta$ instead of $4 \times \theta$. In Figure 5(b), q_1, q_2, and q_3 are all attached to Textual Node $[k_1 k_2]$. This calls for a descent of queries to improve spatial discrimination. q_1, q_2, and q_3 are sorted according to the area of their spatial ranges. q_1 remains in Level 1 of the spatial pyramid while q_2 and q_3 descend to Level 0. Also, Queries q_2 and q_3 are shared between two infrequent AKI textual nodes at Level 0 of the spatial pyramid.

Notice that only queries with no infrequent keywords at level i descend to level $i - 1$. For example, q_4 has Keyword $k_6 \in q_4.text$. No other query contains k_6, and k_6 is an infrequent keyword and the number of queries attached to k_6 is less than θ. Hence, all queries containing k_6 remain at Level 1, and there is no need to search for k_6 in Level 0. This improves the matching performance by reducing the number of keywords being searched for as the search goes down the spatial pyramid.

Notice that when queries descend the spatial pyramid, they may be replicated to more than one pyramid node with in-

creased memory overhead. The replication overhead becomes more significant when queries with large spatial ranges descend to lower pyramid-levels with higher resolution because these queries will span multiple pyramid nodes. As a heuristic, to reduce the number of queries with large spatial ranges that descend to lower pyramid levels with higher resolution, we set the lowest spatial pyramid level a query can descend into to be the level having a slide length that is strictly greater than the side length of the query. We refer to the side length of Query q by $q.r$, where $q.r$ is calculated as follows:

$$q.r = max((q.x_{max} - q.x_{min}), (q.y_{max} - q.y_{min})) \quad (5)$$

The lowest level L_{min} of Query q is calculated as follows:

$$L_{min}(q) = \lceil \log_2(\lfloor \frac{q.r}{SideLen_{min}} \rfloor) \rceil \quad (6)$$

We analyze the replication of queries in Appendix A.

A. Algorithms

In this section, we present the indexing, searching, and cleaning algorithms of FAST.

1) Insertion Algorithm: First, we update the frequencies of keywords in the ***frequencies map*** according to the query being inserted. Then, at the top level of the spatial pyramid, we attempt to attach the incoming query to an infrequent AKI textual node using the least-frequent keyword of the query, i.e., $key_{minfreq}$. If the query has more than one infrequent keyword, $key_{minfreq}$ is chosen arbitrarily from the set of keywords with the minimum frequency. If the incoming query cannot be attached to any infrequent AKI textual node, we index the query to frequent AKI nodes according to the lexicographic ordering of the keywords of the query. After attaching the incoming query to a frequent AKI textual node, we check if a descend operation is required, i.e., when the number of queries attached to the frequent AKI textual node exceeds $4 \times \theta$. If a descend operation is required, we identify queries to be descended, and we recursively reinsert queries these queries at the subsequent level of the spatial pyramid.

If the incoming query spans more than one spatial pyramid node and the incoming query is attached to infrequent AKI textual nodes in all relevant spatial pyramid nodes, then we employ *spatial-sharing of query lists* to share the textual index among the spatial nodes to reduce the memory footprint of FAST.

2) Matching Algorithm: When an incoming data object arrives, it needs to be inspected against relevant pyramid and AKI nodes. Matching in FAST consists of the following three steps: (1) Identify relevant pyramid nodes, (2) Search AKIs within the relevant pyramid nodes, and (3) Refine the results to remove the expired queries. Algorithm 1 describes the matching algorithm adopted in FAST. The matching process starts from the highest pyramid-level. For a data object, say o, with point spatial location $o.loc$, at most one pyramid node per level is relevant for matching. The data object that has a point location cannot overlap more than one spatial pyramid node per level because there is no overlap in spatial ranges of pyramid nodes in the same level. We calculate the index of every relevant pyramid node using Equations (2)

Algorithm 1: Match(Data Object o)

1   $keywords \leftarrow o.text$
2   **for** $level = level_{max}; level >= level_{min}; level - -$ **do**
3     $nextLevelKeywords \leftarrow \{\}$
4     $N_p \leftarrow getSpatialPyramidNode(level, o.loc)$
5   **if** N_p *is not Null* **then**
6   **for** $i = 1; i <= |keywords|; i + +$ **do**
7         $N_t \leftarrow N_p.get(keywords[i])$
8   **if** N_t *is infrequent* **then**
9   **foreach** *Query q in* $N_t.qlist$ **do**
10   **if** *q not expired and o.loc inside q.MBR and keywords contains q.text* **then**
11   add q to result
12   **else**
13           $nextLevelKeywords.add(keywords[i])$
14           $SearchFrequent(N_t, i, o, keywords)$
15     $keywords \leftarrow nextLevelKeywords$

Algorithm 2: SearchFrequent($N_t, i, o, keywords$)

1   **if** N_t *is infrequent* **then**
2   **foreach** *Query q in* $N_t.qlist$ **do**
3   **if** *q not expired and o.loc inside q.MBR and keywords contains q.text* **then**
4   add q to result

5   **else**
6   **foreach** *Query q in* $N_t.qList$ **do**
7   **if** *q not expired and o.loc inside q.MBR* **then**
8   add q to result

9   **for** $j = (i + 1); j \le |keywords|; j + +$ **do**
10       $SearchFrequent(kIndex.children.get(keywords[j]),$
11       $j, keywords)$

---

to (4). We assume that keywords of the data objects are sorted lexicographically. We retrieve textual nodes for every keyword in $o.text$, where $o.text$ is the set of keywords of Data object o that is being matched against FAST. If the top-level textual node, say N_t, is infrequent, we verify the spatial and textual criteria of all queries in $N_t.qlist$. If Node N_t is frequent, children of this node are recursively searched as outlined in Algorithm 2. In the matching process, spatial validation of queries verifies that the data object is located inside the spatial range of the query. Textual validation verifies that this data object contains all the keywords of the query.

Notice that queries directly attached to frequent textual nodes do not require textual validation as these queries only contain the keywords that constitute the path of the frequent textual node. For example, in Figure 7, consider Query q_1 that is attached to the frequent textual node $[k_1k_2]$ in Pyramid Level 1. This query has only two keywords k_1 and k_2. If more keywords exist in q_1, then q_1 would have been attached to a child node of $[k_1k_2]$. However, queries attached to infrequent textual nodes require textual validation as these queries may

contain more keywords than the path of the textual node. For example, in Figure 7, Query q_4 has more keywords, i.e., k_3, than the path of the infrequent textual node $[k_6]$ in Level 1, and hence requires additional textual validation at matching time.

Notice that in FAST, Keywords being searched in Pyramid Level i-1 are a subset of keywords being searched in Level i $\subseteq o.text$, where o is the spatio-textual object being matched. Recall that matching in FAST is top-down and the lowest pyramid level in FAST is Level 0. All queries attached to an infrequent top-level textual node in Level i can never descend to be indexed at Level $i - 1$ for the same spatial range. All infrequent top-level textual nodes at a pyramid node, say N_p, within Level i, correspond to a set of keywords, say SU_i. SU_i can never exist at a pyramid node, say N_p', at Level $i - 1$ that shares the same spatial range with N_p. Hence, at matching time, the Set SU_i is not considered for matching at Level $i - 1$.

The final step in the matching process is to remove the expired queries from the resultset and to verify the spatial overlap between the incoming data object and the matched queries.

3) Index Maintenance: Over time, some indexed queries expire, and some new queries get inserted. FAST employs a lazy vacuum-cleaning mechanism that maintains the structure of FAST and updates the frequencies of the keywords of the indexed queries. The vacuum cleaner has the following functionalities: (1) Detect and remove the expired continuous queries, and (2) Reflect the current frequencies of keywords according to the expired and removed queries.

The vacuum cleaner maintains a queue of the pyramid nodes. In every *cleaning interval I*, the vacuum cleaner visits a pyramid node to be cleaned, say N_p, from the top of the cleaning queue. Then, the cleaner iterates over all textual nodes within N_p and scans all the queries attached to the textual nodes within N_p to check for the expired queries, i.e., $q.t_{exp} < current\ time$. The vacuum cleaner updates the frequencies of keywords of a removed query. The vacuum cleaner needs to account for the expired queries that span multiple pyramid nodes to avoid updating the frequencies multiple times. When an expired query is first removed, the vacuum cleaner updates the keyword statistics and marks the expired query to prevent updating the statistics more than once.

When the frequency of a keyword in the *frequencies map* reaches zero, the textual nodes associated with this keyword are removed. When an entire pyramid node becomes empty, the entire node is removed from FAST. Notice that in the lazy cleaning approach, expired continuous queries are not removed instantaneously. Instead, they may remain indexed until the vacuum cleaner touches them. However, this does not affect the correctness of matching in FAST because the matching algorithm in FAST has a refinement step that removes expired queries from the matching result.

Indexing Queries with General Boolean Expressions. FAST supports matching data objects against queries whose keywords are fully contained in the keywords of the incoming data object. Also, FAST is able to support queries with general boolean expressions on their keywords. For example, the textual condition of a query, say q, is to be matched against all data objects that either contains (k_1 and k_2) or (k_3 and k_4). This textual condition is a boolean expression in the disjunctive normal form (DNF). We address boolean expressions in DNF because boolean expressions represented in the conjunctive normal form (CNF) can be converted to DNF [18]. To support queries in DNF, we instantiate a sub-query per conjunction. For example, q is split into two sub-queries q_1 and q_2, where $q_1.text$ is $\{k1, k2\}$ and $q2.text$ is $\{k_3, k_4\}$. Sub-queries q_1 and q_2 have pointers to q. Then, q_1 and q_2 are indexed using the insertion algorithm of FAST. If a sub-query, e.g., q_1, appears in the matching resultset, the original query, i.e., q, is reported in the final resultset. To avoid duplicate results when more than one sub-query qualifies in the matching process, a flag is added to the original query when it is first added to the matching resultset. This flag is cleared at the end of the matching process.

Matching Objects with Rectangular Spatial Ranges. FAST supports matching data objects with point location. Also, FAST is able to support the matching of data objects with rectangular spatial locations. Matching of rectangle data objects starts from the top level of the spatial pyramid. When the matching algorithm descends the spatial pyramid, the matching algorithm visits all the nodes of the spatial pyramid that overlap the rectangular range of the incoming data object. When a query, say q, spans multiple spatial nodes that overlap the rectangular spatial location of an object being matched, q may appear multiple times in the matching resultset. The matching algorithm prevents duplicate results by adding a flag to mark queries added to the matching resultset. This flag is cleared at the end of the matching process.

B. Analysis

In this section, we analyze the matching time for FAST. AKI within FAST is proposed to address the limitations of existing textual indexes, i.e., the deterioration in the matching performance in RIL, and the large memory requirements of OKT. RIL's matching performance deteriorates due to the existence of long posting lists of indexed objects that have no infrequent keywords. OKT has an advantage over RIL in the matching performance as OKT uses multi-level indexing that uses all the keywords of an indexed object. However, this increases the memory footprint of OKT. AKI uses the *frequent-keyword* threshold θ to restrict the number of queries attached to textual nodes. This creates balance between the memory requirements and the matching performance in AKI. We measure the matching performance of an index by the number of index nodes visited during matching.

To analyze the matching performance of AKI, we first study the matching performance (MP, for short) of RIL for a set of keywords S. The total number of textual items visited when

matching S against the indexed objects is

$$MP_{RIL}(S) = \sum_{j=1}^{|S|} |RIL[s_i]| \qquad (7)$$

where $|S|$ is the number of keywords being searched in S, and $|RIL[s_i]|$ is the number of indexed textual items attached to the Keyword s_i.

OKT is a multi-level keyword index that is illustrated in Figure 4(b). The matching process of the keyword Set S at level i in OKT iterates over the keywords in S to find a subset of matching keywords to proceed to level $i + 1$ in OKT (Notice that, in OKT, level numbers increase as we descend the index). OKT assumes a total order of the indexed keywords. For a matched keyword, say s_j, at level i of OKT, the search proceeds to level $i + 1$ with the keyword set $[S-\{s_1, s_2, \ldots, s_j\}]$. Hence, at level i, the matching time for OKT can be expressed recursively as follows [12]:
$$MP_{OKT}(i, S)=$$
$$|S| + \sum_{j=1}^{|S|} \alpha_{ij} \times MP_{OKT}(i+1, S - \{s_1, \ldots, s_j\}) \qquad (8)$$

where α_{ij} is the probability of having Keyword S_j indexed at level i. α_{ij} depends on the frequencies of the indexed keywords and their probabilities of co-occurrence. The recursion in Equation 8 terminates at the deepest level of OKT, i.e., the largest $|q.text|$ for any indexed query q. Notice that Equation 8 is a recurrence relation and is not in closed form. However, for datasets with known probabilities of keyword co-occurrence and a bounded $|q.text|$ for any indexed query q, a closed formula can be devised. For textual items with infrequent keywords, AKI has a similar behavior to that of RIL, yet with a restricted length of posting lists, i.e., θ. For textual items with no infrequent keywords, AKI has a similar behavior to that of OKT. From Equations 7 and 8, we estimate the matching performance of AKI as follows:
$$MP_{AKI}(i, S)=$$
$$\begin{cases} |S| \times \theta, infrequent \\ |S| + \sum_{j=1}^{|S|} \alpha_{ij} MP_{AKI}(i+1, S - \{s_1, \ldots, s_j\}), frequent \end{cases}$$
$$(9)$$
Similar to Equation 8, the recursion in Equation 9 terminates at the deepest level of AKI.
Estimation of The *Frequent-Keyword* Threshold. We use Equation 9 to estimate an upper bound on θ. The matching performance of infrequent AKI nodes should not exceed the matching performance of frequent AKI nodes. In the worst case, frequent AKI nodes resemble an OKT index.
$$\theta \leq \frac{MP_{OKT}}{|S|} \qquad (10)$$
From Equation 9, the worst-case matching in FAST requires AKI matching at every level of the spatial pyramid. The matching performance in FAST can be estimated as follows:
$$MP_{FAST}(S) = log(gran_{max}) \times MP_{AKI}(0, S) \qquad (11)$$
where $log(gran_{max})$ represents the height of the spatial pyramid.

TABLE I: The datasets used in the experiments.

Dataset	Tweets	Synthetic	Places
Number of entries	30M	30M	12.9M
Vocabulary size	804K	804K	854k
Avg num of keywords/Entry	4	4	9

TABLE II: The values of the parameters used in the experimental evaluation.

Parameter	Value
Number of queries (million)	1,2.5, **5**,7.5,10,20
Number of query keywords	1, 2, **3**, 5, 7
Spatial side-length of a query	.01%,.05%,.1%,.5%,**1%**,5%,10%

IV. EXPERIMENTAL EVALUATION

In this section, we compare the performance of FAST against the performance of the state-of-the-art index, the AP-tree [1].

A. Experimental Setup

Datasets. Two real datasets, namely, *Tweets* and *Places*, and three synthetic datasets, namely, *SpatialUni*, *SpatialSkew*, and *TextUni*, are used in the experimental evaluation. The Tweets dataset consists of 30 million geo-tagged tweets located inside the United States. These tweets are collected over the period from January 2014 to March 2015. The Places [19] dataset contains 12.9 million public places inside the United States. Each entry in the Places dataset includes the geo-location and the set of keywords describing a specific place represented by the entry. Frequencies of keywords in both datasets follow a Zipfian distribution, as illustrated in Figure 2. Table I summarizes the details of the real datasets used in the experiments.

The SpatialUni and the SpatialSkew synthetic datasets change the spatial location of entries in the Tweets dataset to follow a uniform and a skewed Gaussian distribution, respectively. The TextUni dataset uses the spatial locations of entries in the Tweets dataset. However, keywords in the TextUni dataset are chosen uniformly from the vocabulary of the Tweets dataset, i.e., the frequencies of keywords follow a uniform distribution. We use the TextUni dataset to study the performance of FAST under a textual distribution that is not Zipfian to demonsrate that FAST is able to maintain its performance under various textual distributions.
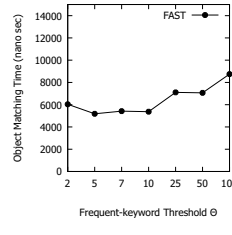
Query Workload. Entries in datasets are used to construct spatial-keyword filter queries. The geo-location of a dataset entry is used as the center of the spatial range of a query. Table II summarizes the query workload used in the experimental evaluation.

Object Workload. The AP-tree requires a training phase. We use 100K random dataset entries as historical training data. To measure the average matching time, we stream 100k data objects generated from the dataset entries against the indexed spatial-keyword queries. In the SpatialSkew dataset, we generate two synthetic object workloads, namely, SpatialSkewL and SpatialSkewO, where the spatial locations of objects in SpatialSkewL follow the same Guassian distribution as the one for the indexed queries. The spatial locations of objects in SpatialSkewO are skewed away from the spatial locations of the indexed queries.
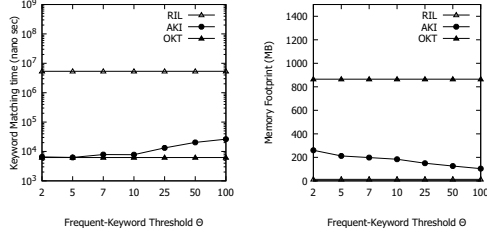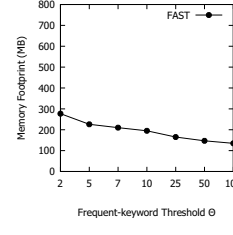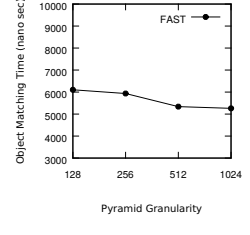
| (a) Matching time AKI | (b) Memory footprint AKI | (c) Matching time FAST | (d) Memory footprint FAST | (a) FAST Granularity |

Fig. 8: *Frequent-keyword* threshold θ Fig. 9: Granularity

All implementations are in Java 8. All experiments are conducted on a 64-bit virtual machine running Ubuntu Linux 16.04. This virtual machine is allocated 16 cores each clocked at 2.6MHz. The total memory of the virtual machine is 49GB. The source code of the AP-tree has been provided by the authors of the AP-tree index. We set the parameters of the AP-tree according to the default values recommended by the authors of the AP-tree index [1].

B. Index Tuning

In this section, we describe how to set the parameters of FAST. The main parameter in FAST is the *frequent-keyword* threshold θ. In this experiment, we study the performance of AKI and FAST under various *frequent-keyword* thresholds.

Performance of Textual Indexes. In Figure 8, we study the effect of varying the *frequent-keyword* threshold θ on the performance of AKI. We compare both the matching time and the memory footprint of AKI against both RIL and OKT. In Figure 8(a), notice that AKI achieves keyword matching time that is comparable to that of OKT when $\theta \leq 10$ while having a memory footprint that is up to one third of that required by OKT. Notice that the performance of OKT and RIL is not affected by varying θ as both RIL and OKT do not have the *frequent-keyword* threshold parameter. Increasing θ increases the matching time and reduces the memory footprint of AKI. The reason is that as θ increases, the number of textual items attached to infrequent textual nodes increases. Matching textual items attached to infrequent textual nodes requires further validation. This validation increases the overall matching time.

Performance of FAST. In Figure 8(c), observe that as we increase the *frequent-keyword* threshold θ, the matching time of FAST deteriorates. The reason is that the higher the *frequent-keyword* threshold the longer the list of queries attached to the infrequent textual nodes, as in Figure 8(a). This increases the textual validation time needed to verify the containment of query keywords within the keywords of the streamed data objects. Figure 8(d) demonstrates that the smaller the *frequent-keyword* threshold the higher the memory requirements of FAST.

The reason is that having a small *frequent-keyword* threshold results in marking more textual nodes as frequent, and demanding more memory for the splitting of their attached lists of queries. Figure 8 illustrates that using a *frequent-keyword*
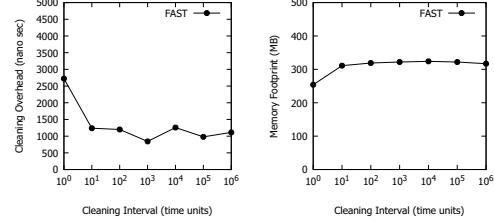
| (a) Cleaning overhead | (b) Memory footprint |

Fig. 10: The effect of varying the cleaning interval I

threshold between 5 and 10 results in good matching time with moderate memory requirements in FAST. We set the *frequent-keyword* threshold to 5 thourghout the rest of the experiments. The formula of Equation 10 estimates that the worst case value of θ is 13.6 that conforms with the simulation results in Figure 8.

The Effect of Varying the Pyramid Granularity. Figure 9 illustrates the matching time of FAST while varying the finest granularity of the spatial pyramid. From the figure, increasing the granularity of the pyramid within FAST improves the matching time initially. Then, increasing the granularity further does not offer further improvement. Because of this observation, we set the granularity of FAST to 512 as increasing the finest granularity beyond 512 does not improve the matching performance.

The Cleaning Overhead. In order to remove the expired queries in FAST, cells of FAST are visited periodically to be cleaned, i.e., every I time units a cell is visited to be cleaned, as described in Section III-A3. Figure 10 illustrates the effect of varying the cleaning interval I on the memory footprint of FAST and the average cleaning overhead, i.e., the average time spent in cleaning. Figure 10(a) illustrates that the cleaning overhead decreases as the cleaning interval increases. Having a very small cleaning interval results in redundant visits to cells that have been recently cleaned. Figure 10(b) illustrates that the memory footprint of FAST increases as the cleaning interval I increases. In our experiments, we set the cleaning interval to 1000 time units as it achieves balance between the cleaning overhead and the memory footprint of FAST.

C. Performance Evaluation

In this section, we study the performance of FAST under various query workloads.

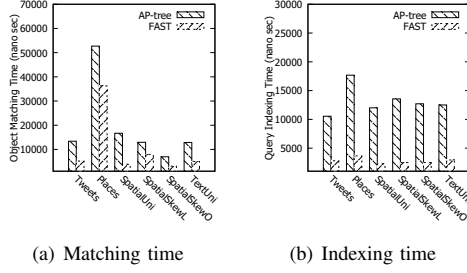**Performance using the Various Datasets.** We compare the

(a) Matching time   (b) Indexing time   (c) Memory footprint

Fig. 11: Performance under different datasets

(a) Matching time   (b) Indexing time

Fig. 12: The effect of the spatial range

(a) Matching time   (b) Indexing time

Fig. 13: The effect of the number of keywords

(a) Matching time   (b) Indexing time   (c) Memory footprint
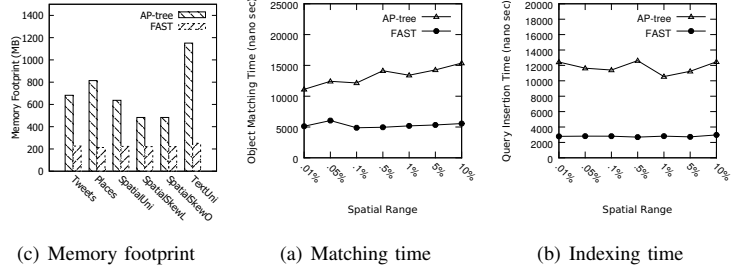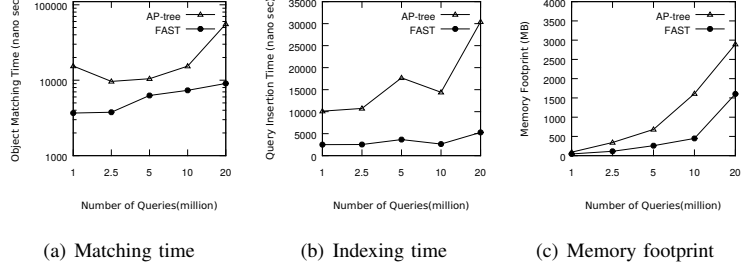
Fig. 14: Performance under varying number of indexed queries

performance of FAST against that of the AP-tree using both the real and the synthetic datasets. Figure 11(a) illustrates that FAST is up to 3x faster than the AP-tree in terms of object matching time. The reason is that FAST accounts for spatial and textual selectivities at the keyword level, as described in Section III. Figure 11(b) illustrates that FAST is up to 5x faster than the AP-tree in terms of query indexing time. The reason is that FAST benefits from the *frequent-keyword* threshold to account for the spatial and textual selectivities of data. However, the AP-tree uses an expensive cost formula to arbitrate between the spatial and textual indexing. In terms of the memory footprint, Figure 11(c) illustrates that FAST requires up to 3x less memory than that of the AP-tree. The reason is that FAST integrates AKI with spatial-cell sharing to reduce the size of the textual indexes, and to limit the replication of queries. However, the AP-tree is based on the memory intensive OKT and does not impose any restrictions on the replication of the indexed queries among the index cells. Notice that FAST maintains its performance advantages over the AP-tree under different synthetic distributions of the spatial and textual aspects of the data objects and queries.

The Effect of Varying the Spatial Range. In this experiment, we vary the spatial ranges of the queries from .01% to 10% of the entire spatial range. Figure 12 illustrates the object matching time and the query indexing time for both the AP-tree and FAST. From the figure, observe that FAST maintains its performance advantage against the AP-tree for both the object matching and the query indexing (insertion) times.

The Effect of Varying the Number of Keywords. In this experiment, we measure the *object matching time* and *the query indexing time* when changing the number of keywords in the indexed queries from 1 to 7. Figure 13 illustrates that FAST remains up to 3x faster than the AP-tree in terms of the

object matching time and up to 5x faster in terms of the query indexing time.

The Scalability of FAST. In this experiment, we demonstrate the scalability of FAST against that of the AP-tree when increasing the number of indexed queries from 1 million to 20 million. Figure 14 illustrates that FAST maintains its performance advantage against the AP-tree. When increasing the number of indexed queries, FAST remains 3x faster than the AP-tree in object matching time, 5x faster than the AP-tree in the query indexing time, and requires one third of the main-memory required by the AP-tree.

V. Related Work

We classify the related work into the following categories: (1) spatio-textual indexing, (2) publish/subscribe systems, and (3) superset containment search.

Spatio-Textual Indexing. Recently, several spatio-textual indexes have been proposed to answer snap-shot queries over spatio-textual data. Examples of these queries include the filter, top-k, and collective group queries. Chen et al. [20] surveys spatio-textual indexes and benchmarks their performance under various spatio-textual queries. The most relevant indexes are the IQ-tree [21] and the R^t-tree [8]. These indexes are mainly disk-based and have been outperformed by the AP-tree [1].

Publish/Subscribe Systems. One main use case of FAST is in location-aware publish/subscribe systems. Publish/subscribe systems maintain subscriptions for long durations and match incoming messages against stored subscriptions. Publish/subscribe systems can be categorized according to their matching approach into the following categories: (1) content-based [22], (2) TopK-similarity-based [23], and (3) location-aware [24]. These publish/subscribe systems do not simultaneously account for the spatial and textual properties of

subscriptions and messages. Recently, several spatio-textual publish/subscribe systems [1, 25] have been proposed. To the best of our knowledge, the AP-tree [1] is the most relevant work for indexing continuous spatio-textual queries in a streaming environment.

Superset Containment Search. AKI addresses the problem of superset containment search, where it is required to retrieve indexed items with keywords that are fully contained in the search keywords. Several indexes have been proposed to address the superset containment problem, e.g., [5, 11, 26, 27]. OKT [5] and RIL [11] are the most adopted structures for superset containment search [12]. Terrovitis et al. [26, 27] present two structures for superset containment search. However, these structures are mainly disk-based and require knowing the frequencies of the entire vocabulary. AKI is a main-memory index and does not assume prior knowledge of the frequencies of keywords.

VI. Conclusion

In this paper, we introduce FAST; a Frequency-Aware Spatio-Textual access method for indexing continuous spatio-textual filter queries in a streaming environment. FAST automatically accounts for both the spatial and textual selectivities of indexed queries to improve the indexing and searching performance. FAST integrates the spatial pyramid with a new textual index, and supports a cell-sharing technique that reduces the memory required by the index. FAST uses a light-weight lazy-cleaning mechanism to remove the expired queries and to reflect changes in the frequencies of the keywords of the indexed queries. Extensive experimental evaluation using real and synthetic datasets demonstrates that FAST is up to 3x faster in search time and 5x faster in indexing time than the state-of-the-art index. Also, FAST requires up to 3x less memory than the state-of-the-art index.

References

[1] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang, "Ap-tree: Efficiently support continuous spatial-keyword queries over stream," in *ICDE*, 2015, pp. 1107–1118.

[2] "Geotagged tweets," http://www.futurity.org/tweets-give-info-location/, 2017.

[3] "Foursquare," https://foursquare.com/about, 2017.

[4] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. Basalamah, "Tornado: A distributed spatio-textual stream processing system," *PVLDB*, vol. 8, no. 12, pp. 2020–2023, 2015.

[5] T. W. Yan and H. García-Molina, "Index structures for selective dissemination of information under the boolean model," *TODS*, vol. 19, no. 2, pp. 332–364, 1994.

[6] A. C. König, K. Church, and M. Markov, "A data structure for sponsored search," in *ICDE*, 2009, pp. 90–101.

[7] L. Chen, G. Cong, and X. Cao, "An efficient query indexing mechanism for filtering geo-textual data," in *SIGMOD*, 2013, pp. 749–760.

[8] G. Li, Y. Wang, T. Wang, and J. Feng, "Location-aware publish/subscribe," in *SIGKDD*, 2013, pp. 802–810.

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

[10] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[11] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM computing surveys*, vol. 38, no. 2, p. 6, 2006.

[12] Z. Hmedeh, H. Kourdounakis, V. Christophides, C. Du Mouza, M. Scholl, and N. Travers, "Subscription indexes for web syndication systems," in *EDBT*. ACM, 2012, pp. 312–323.

[13] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, 1998, pp. 151–160.

[14] D. E. Knuth, *The art of computer programming*. Addison-Wesley, 1968, vol. 3.

[15] W. G. Aref and H. Samet, "Efficient processing of window queries in the pyramid data structure," in *PODS*. ACM, 1990, pp. 265–272.

[16] A. Magdy and M. F. Mokbel, "Towards a microblogs data management system," in *MDM*, vol. 1, 2015, pp. 271–278.

[17] T. Lee, J.-w. Park, S. Lee, S.-W. Hwang, S. Elnikety, and Y. He, "Processing and optimizing main memory spatial-keyword queries," *PVLDB*, vol. 9, no. 3, pp. 132–143, 2015.

[18] H. Enderton and H. B. Enderton, *A mathematical introduction to logic*. Academic press, 2001.

[19] "Places dataset," https://archive.org/details/2011-08-SimpleGeo-CC0-Public-Spaces, 2017.

[20] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: An experimental evaluation," in *VLDB*, vol. 6, no. 3, 2013, pp. 217–228.

[21] Y.-Y. Chen, T. Suel, and A. Markowetz, "Efficient query processing in geographic web search engines," in *SIGMOD*. ACM, 2006, pp. 277–288.

[22] D. Zhang, C.-Y. Chan, and K.-L. Tan, "An efficient publish/subscribe index for e-commerce databases," *PVLDB*, vol. 7, no. 8, pp. 613–624, 2014.

[23] A. Shraer, M. Gurevich, M. Fontoura, and V. Josifovski, "Top-k publish-subscribe for social annotation of news," *PVLDB*, vol. 6, no. 6, pp. 385–396, 2013.

[24] J. Bao, M. F. Mokbel, and C.-Y. Chow, "Geofeed: A location aware news feed system," in *ICDE*. IEEE, 2012, pp. 54–65.

[25] H. Hu, Y. Liu, G. Li, J. Feng, and K.-L. Tan, "A location-aware publish/subscribe framework for parameterized spatio-textual subscriptions," in *ICDE*. IEEE, 2015, pp. 711–722.

[26] M. Terrovitis, P. Bouros, P. Vassiliadis, T. Sellis, and N. Mamoulis, "Efficient answering of set containment queries for skewed item distributions," in *EDBT*, 2011, pp. 225–236.

[27] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis, "A combination of trie-trees and inverted files for the indexing of set-valued attributes," in *CIKM*, 2006, pp. 728–737.

[28] A. R. Mahmood, W. G. Aref, and A. M. Aly, "Fast: Frequency-aware spatio-textual indexing for in-memory continuous filter query processing," *arXiv preprint arXiv:1709.02529*, 2017.

Appendix

In this section, we estimate the expected replication of queries indexed in FAST. First, we estimate the expected replication of queries when indexed at their lowest pyramid level, i.e., $E_{rep}(L_{min}(q))$.

$$E_{rep}(L_{min}(q)) = 2 \int_{.5}^{1} (1+r)^2 \ dr \qquad (12)$$

From Equation 12, $E_{rep}(L_{min}(q)) = 3.08$ that is less than the worst case replication of 4. The overall expected replication E_{rep} is:

$$E_{rep} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{2}{2^{2i}} \int_{.5}^{1} (2^i + r)^2 \ dr \qquad (13)$$

Proofs are omitted for space limitation, and are detailed in our technical report on FAST [28]. From Equation 13, E_{rep} is equal to 1.27 when the number of levels n is 9. The average query replication measured experimentally is 1.08 that is very close to the estimated query replication.