Evaluating Spatial-Keyword Queries on Streaming Data

Abdulaziz Almaslukh[a,b] Amr Magdy[a,b]

[a]Department of Computer Science and Engineering, [b]Center for Geospatial Sciences
University of California, Riverside
aalma021@ucr.edu,amr@cs.ucr.edu

ABSTRACT

This paper provides an extensive experimental evaluation for different spatial-keyword index structures on streaming data. We extend existing snapshot spatial-keyword queries with the temporal dimension to effectively serve streaming data applications. Then, the major index structures are equipped with efficient query processing techniques and evaluated to process the extended queries. The evaluation is oriented towards a system building perspective to provide system builders with insights on supporting scalable spatial-keyword queries on fast data streams, e.g., social media streams and news streams. In particular, we have taken existing spatial-keyword index structures apart into four major building blocks that are commonly supported at a system-level. Ten different index structures are then composed as combinations of these four building blocks. The ten indexes are wholly residents in main-memory, and they are evaluated on real datasets and query locations. The index performance is measured in terms of data digestion rate in real time, main-memory footprint, and query latency. The results show the relative performance gains of both basic and hybrid index structures with abundant insights from a system point of view.

CCS CONCEPTS

• **Information systems → Data streams**;

KEYWORDS

Spatial, Real-time, Geo-texual, Temporal Query Processing

ACM Reference Format:
Abdulaziz Almaslukh[a, b] Amr Magdy[a, b]. 2018. Evaluating Spatial-Keyword Queries on Streaming Data. In *26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18), November 6–9, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3274895.3274936

1 INTRODUCTION

Spatial-keyword queries have drawn tremendous attention by the research community over the past years [9, 10, 12–14, 16–18, 28, 30, 32, 48, 49, 51]. Such queries find objects that satisfy predicates on both spatial and keyword attributes. They are widely applicable in different applications such as points of interests (POIs) search [18], news search [43], route planning [7], and several applications of social media search [5, 24, 25, 47]. As a result to their wide popularity, spatial-keyword queries have recently made it to the system-level support [19, 37, 41] to allow application developers to build applications on top of them.

Spatial-keyword queries have been extensively studied in two settings: snapshot queries on relatively-stable datasets that do not encounter extensive updates, e.g., POIs, and continuous queries on streaming data (as detailed in Section 2). However, several applications still post traditional snapshot queries on streaming data, such as using social media in rescue [15, 24, 27], finding real-time local news [4], and real-time recommendations [46]. In fact, several work on snapshot queries actually use Twitter data in their evaluation [13, 21, 28, 42] ignoring its streaming nature. In addition, Twitter's *Advanced Search* allows snapshot spatial-keyword queries on streaming data due to its importance, yet, with pre-defined list of cities with no support for arbitrary regions, e.g., Downtown Boston. Generally, all top-notch big data systems [1, 20, 45] optimize their performance for snapshot querying of fast data in real time, including systems that support spatial-keyword queries [19, 41]. However, none of existing work has extensively studied the performance of different spatial-keyword indexing techniques on streaming data, which is an essential step towards efficient processing and optimization for such queries at a system-level.

In this paper, we provide an extensive experimental study for different spatial-keyword index structures on streaming data that come with high arrival rates. The study has two main objectives:

(1) Understanding the relative benefits of separate versus hybrid index structures on spatial-keyword queries. From a system perspective, it is preferred to support two separate index structures, one for spatial attributes and one for keyword attributes, as provided in existing systems [19, 34]. This gives the system flexibility to support a variety of queries using a limited number of assets. Moreover, it optimizes system resources such as disk storage requirements and main-memory utilization. However, this comes with performance penalties that affect the scalability of certain query families. This study reveals the performance gains and losses of supporting spatial-keyword queries using separate versus hybrid index structures.

(2) Understanding the relative benefits of spatial-keyword index structures on temporally-extended queries in real-time environments. Snapshot queries on streaming data always incorporate the time aspect in query definitions [6, 50] for two reasons. First, the time-sensitivity of streaming data so recent data is more important

than old data. Second, the plethora of data objects, so it is overwhelming to return all objects that satisfy the query predicates. Thus, we extend spatial-keyword queries on streaming data with the temporal dimension and provide query processing techniques for the extended queries. Recent work has considered the temporal aspect on relatively static datasets [11, 23]. On the contrary, our study reveals the performance trade-offs in real-time environments.

Our study is performed based on two popular types of spatial keyword queries: Boolean range query [16] and Boolean kNN query [10]. We extend both queries with temporal ranking to fit streaming data applications. The two queries are chosen based on appropriateness to streaming environments compared to other queries that add significant overhead as discussed in Section 7. The following examples illustrate the two evaluated queries:

- **Example 1: Temporal Boolean Range Query (TBRQ)**: "Find *the most recent k* tweets, k=20, that mention *save* or *help*, and spatially located *within Jacksonville, Florida*". The query includes both textual and spatial parts of the predicate as Boolean filters while the temporal part is used in ranking to find the most recent objects relative to the query time.
- **Example 2: Temporal Boolean kNN Query (TBKQ)**: "Find *recent k* tweets, k=3, that mention *Boston marathon* or *explosion*, and spatially *near to 500 Boylston Street*". This query includes the textual part of the predicate as a Boolean filter while spatial and temporal parts are used to rank the objects relative to the query location and time, respectively.

We evaluate ten different index structures, composed of four building blocks of spatial and textual index structures that are used in the literature. All index structures are wholly residents in main-memory to scale for digesting streaming data and provide low query latency. For spatial indexing, three building blocks are used: spatial grid, quadtree, and R-tree index structures. In addition, the inverted index structure is used for textual indexing as the fourth building block. The ten evaluated index structures are classified into two categories: pure spatial and textual indexes that index a single-dimension and hybrid indexes that index two dimensions. Each index structure is evaluated with respect to data digestion rate, main-memory consumption, and query latency. We adjust tightly-coupled hybrid indexes ([13]) to be loosely-coupled as they put less overhead on system resources and hence more preferable from a system perspective. Evaluation data and queries are based on real query locations and a real Twitter dataset, a prime example of spatial-keyword streaming data that is widely used in different applications.

The evaluation yielded abundant results. Clearly, pure indexes outperform hybrid indexes in terms of data digestion rates and memory consumption. On another hand, pure indexes suffer from relatively high query latency compared to hybrid indexes. More importantly, the administrative configuration settings play an essential role in the overall index performance. Configuring the right index cell size, query weighting parameters, and number of Boolean filters changes the performance drastically. Such insights are important for system administrators for tuning query performance based on the underlying index(es). Our contributions in this paper can be summarized as follows:

- Our work is the first to evaluate different spatial-keyword index structures on streaming data.
- Our work is the first to temporally extended and address snapshot spatial-keyword queries in real-time environments.
- Our study reveals the relative benefits of separate versus hybrid index structures on spatial-keyword queries to give insights for system builders and administrators on efficient processing and optimization of these queries.

The rest of this paper is organized as follows. Section 2 highlights the related work. Section 3 presents the problem definition. Sections 4 and 5 present overview about the evaluated indexing and query processing techniques. Section 6 gives an extensive experimental evaluation and Section 7 highlights the result implications on the system level. Finally, Section 8 concludes the paper.

2 RELATED WORK

Querying geo-textual data is studied under different categories of spatial-keyword queries. The major such categories are: (1) Snapshot queries [9, 10, 12–14, 16–18, 28, 30, 32, 49] that are posted and get their answers one-time from the current snapshot of the data. These queries evaluate each data object separately to check whether or not it satisfies the spatial and keyword predicates. (2) Continuous queries [12, 17, 29, 38, 49], e.g., in publish/subscribe systems, that register in the system and notify users with incremental answers on receiving new data objects that satisfy the query predicates. (3) Group queries [8, 9, 21, 31, 39] are another type of snapshot queries that evaluate the answer as a group of objects rather than individual objects, so the group of objects collectively satisfy the query predicates. An existing experimental study [13] has evaluated different techniques for the first category on traditional non-streaming datasets where disk-based storage and indexing techniques are used. Our work studies existing techniques for the first category of queries as well, yet, with two major differences compared to [13]. First, our study focuses on streaming data in real-time applications. This implies temporal extensions for queries definitions, main-memory storage and indexing, and hence query processing techniques are adjusted for the new environment. Although main-memory has been used in the literature to speed up snapshot spatial-keyword queries [28], streaming data environments have not been considered, where several fundamental differences apply including the query signatures and processing optimizations. Second, our study provides a system-oriented perspective to draw conclusions on supporting scalable spatial-keyword queries in big data systems that optimize their performance for fast data, e.g., [19, 37, 40, 41]. This study is complementary to our prior work [32, 41] on providing holistic real-time data management environments for spatial data.

Recently, time-awareness is studied in the context of spatial-keyword queries in both snapshot [11, 23] and continuous [14] queries. However, both temporal techniques on snapshot queries [11, 23] consider non-streaming data where storage, indexing, and query processing are not optimized for fast data. Consequently, they encounter significant overhead in real-time environments.

3 PROBLEM DEFINITION

Spatial-keyword queries are evaluated on a geo-textual streaming dataset D that consists of a set of geo-textual objects. Each object

ID	Keywords	Timestamp
*o*1	FIFA, Final, Ceremony	06-12-2018 20:18:30
*o*2	World, Cup, Openning	06-12-2018 20:18:27
*o*3	NBA, Lakers, Final	06-12-2018 20:18:23
*o*4	French, Open, R.Nadal, D.Thiem	06-12-2018 20:18:19
*o*5	Russia, Moscow, Ceremony	06-12-2018 20:18:17
*o*6	FIFA, Russia	06-12-2018 20:18:14
*o*7	Brazil, FIFA, Argentina, Game	06-12-2018 20:18:09
*o*8	NBA, LeBron, Injury	06-12-2018 20:18:06

Table 1: Content of Objects in Figure 1a

$o \in D$ is represented with a triple $(loc, kw, time)$, where loc is a point location (latitude/longitude coordinates), kw is a set of keywords, and $time$ is a timestamp. D_{t_1} is a snapshot of the dataset D at time t_1, so every object $o \in D_{t_1}$ has $o.time \leq t_1$. Table 1 gives an example of a dataset that consists of eight objects, $o1$ to $o8$, each is associated with a set of keywords, a timestamp, and located in the 2D space as shown in Figure 1a.

We evaluate two spatial keyword queries that are common in the literature [13] extended with the temporal dimension, so they are appropriate for applications on streaming data that is sensitive to time. A third major query in the literature is omitted due to its high-cost in streaming systems as discussed in Section 7. We give the query definitions followed by an illustration of their extensions compared to existing queries in the literature.

Temporal Boolean Range Query (TBRQ): Given a TBRQ $q = (w, r, k, t)$, where $q.w$ is a set of keywords, $q.r$ is a spatial region, $q.k$ is an integer, and $q.t$ is a timestamp, q finds k objects $o_i \in D_t$, $1 \leq i \leq k$, such that: (1) $o_i.kw \cap q.w \neq \phi$, (2) $o_i.loc \in q.r$, and (3) o_is are the most recent k objects in D_t. So, q retrieves k objects from the dataset snapshot D_t that corresponds to the query timestamp t. Each object lies in the query spatial range and contains one or more of the query keywords. In addition, the k objects are ranked based on time to retrieve the most recent objects in D_t.

Temporal Boolean kNN Query (TBKQ): Given a TBKQ $q = (w, p, k, t)$, where $q.w$ is a set of keywords, $q.p$ is a spatial point location, $q.k$ is an integer, and $q.t$ is a timestamp, q finds k objects $o_i \in D_t$, $1 \leq i \leq k$, such that: (1) $o_i.kw \cap q.w \neq \phi$, and (2) o_is are the closest k objects in D_t according to a spatio-temporal distance F_α:

$$F_\alpha(o, q) = \alpha \times SpatialScore(o, q) + (1 - \alpha) \times TemporalScore(o, q)$$

Where α is a weighting parameter, $0 \leq \alpha \leq 1$, that weights the relative importance of spatial and temporal scores in the object proximity. $SpatialScore$ and $TemporalScore$ are defined as follows:

$$SpatialScore(o, q) = \frac{distance(o.loc, q.p)}{R_{Max}}$$

$$TemporalScore(o, q) = \frac{q.t - o.time}{T_{Max}}$$

Where R_{Max} and T_{Max} are the maximum allowed spatial and temporal ranges for any object, and $distance$ is the spatial distance between object and query locations in the Euclidean space.

Both queries are extended in their temporal and top-k parts compared to the existing spatial keyword queries in the literature [13]. Time is essential in streaming data as recent data is significantly more important than older data. In addition, streaming data comes with an excessive number of objects. Thus, limiting answer size to k objects prevents overwhelming end users with excessive answer that is hard to interpret and render. This approach is accommodated by all major web services such web search engines, e.g., Google, and social media providers, e.g., Twitter, where their search by default show the most important 10 results to maximize the user utility, then users can choose to retrieve more pages of results. Twitter, in specific, uses pure temporal recency as one of its major ranking functions as social media posts represent a prime example of streaming data where latest posts have prime importance.

4 MAIN-MEMORY INDEXING

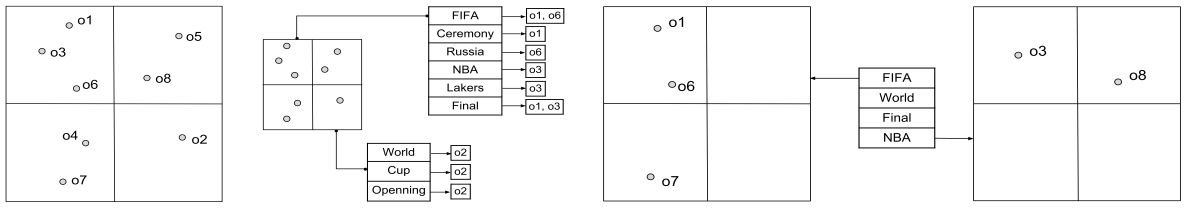
This section presents the different index structures that are evaluated to index and query streaming geo-textual objects efficiently. All indexes are wholly residents in main-memory for scalable data digestion and high-throughput query processing. We decompose the existing indexes in the literature [13] into four major building blocks. Different combinations of these building blocks are used to compose different indexes. Only loosely-coupled combinations are used as tightly-coupled indexes put much overhead on system resources. Each building block is equipped with efficient insertion and query processing techniques to scale for streaming data. Data expiration (deletion) is performed with existing techniques as in [33].

We choose to use indexing building blocks to gear the evaluation to a system-oriented perspective. Big data management systems provide common utilities that are used to support a variety of queries with minimal system overhead. Thus, they support building blocks in both indexing and querying to compose different combinations in different scenarios. Our goal is to understand the relative benefits of different building blocks as well as their combinations.

4.1 Indexing Building Blocks

Spatial-keyword indexes consists of two indexing components: a spatial component and a keyword component [13]. Thus, our building blocks are the major spatial and keyword indexing components in the literature that are appropriate for real-time data indexing. For spatial indexing, we use the three major building blocks: spatial grid, quadtree, and R-tree. For keyword indexing, we use one building block, the inverted index. The literature presents another keyword component, bitmaps, that uses bit flags to indicate presence or absence of each keyword in the index cell. We omit evaluating bitmaps indexes as they are not appropriate for streaming data. In specific, bitmaps assume representing all keywords in all index cells with bit flags. This puts significant overhead on real-time index updates and main-memory storage with the excessive amount (hundreds of millions of data objects) and dynamic distributions of data cross different index cells. The rest of this section presents the four building blocks, illustrating adapting them for real-time data indexing.

As streaming data comes with high-arrival rates, scalable data digestion is one of the crucial requirements for efficient indexing. So, our indexes are equipped with a *batch insertion* module that buffers incoming data objects and group them based on the indexed attribute. Typically a data batch include thousands of data objects that correspond to streaming data that arrive in few seconds. For the inverted keyword index, buffered data is grouped based on



(a) Pure grid index (PG) **(b) Grid-inverted index (GI)** **(c) Inverted-grid index (IG)**

Figure 1: Structure of grid-based indexes

keywords. For spatial indexes, buffered data is grouped based on the underlying index structure, typically with an minimum bounding rectangle (MBR) that represents spatial boundaries of the buffered data. Then, the index is navigated efficiently using batch data groups instead of individual objects as follows.

(1) Spatial grid. We use a single-level spatial grid index [48] similar to the one depicted in Figure 1a. The index divides the space into fixed number of equal-area disjoint cells. In real time, newly arriving data are buffered to be inserted in batches. The buffered data are grouped using a buffer grid that has the same cell size as the underlying grid, yet it is much lighter due to small size of buffered data compared to actual indexed data. Then, the buffer is merged with the underlying grid index through one-to-one cell merging, where all data in a buffer cell is appended to the corresponding cell in the index cell. This enables the grid to digest higher rates of data.

(2) Quadtree. We use a partial quadtree index [2] adapted for scalable batch insertions as in [35] to provide low-cost index structuring under highly-skewed data in real time. The quadtree starts with a single root node where data is inserted in batches. Once the node reaches a maximum capacity, it is split into four quadrants and data is distributed over children nodes, and so recursively. Thus, the index is being shaped dynamically and handles highly-skewed data adaptively.

(3) R-tree. We use a regular R-tree as a representative for the R-tree family variations, e.g., R+ [44] and R* [3] trees, due to its low indexing overhead. Incoming data batches are inserted in the corresponding leaf nodes. When a leaf node reaches its maximum capacity, it is split into two nodes and its objects are distributed over the children nodes. Several heuristics exist to split R-tree nodes [3, 22, 26]. As our work adjusts the index for scalable real-time data digestion, we adapt Guttman’s QuadraticSplit [22] due to its low cost especially with point locations.

(4) Inverted index. We realize an in-memory version of the inverted file index [52], that maps a keyword w to a list of data objects that contain w, using a hashtable structure. As illustrated above, the buffered data batch are grouped based on keywords in a buffer hashtable. Then, data of each buffer hash entry is appended to the corresponding index hash entry, which enables scalable data digestion in real time.

It is noteworthy that index nodes of grid, quadtree, and inverted index store data objects ordered by their arrival timestamp. Such ordering does not add any overhead on real-time indexing as it is a natural order for streaming data that arrives in append-only fashion. This is exploited by the query processing (Section 5) to provide efficient query response. Only R-tree is not guaranteed to preserve this temporal order due to node split algorithm that might reorder data based on spatial proximity rather than timestamp.

Index Name	1st-level	2nd-level
Grid-inverted (GI) [48]	Spatial grid	Inverted index
Inverted-grid (IG) [48]	Inverted index	Spatial grid
Quadtree-inverted (QI)	Quadtree	Inverted index
Inverted-quadtree (IQ)	Inverted index	Quadtree
Rtree-inverted (RI) [51]	R-tree	Inverted index
Inverted-Rtree (IR) [42, 51]	Inverted index	R-tree

Table 2: Structure of hybrid indexes with two-level indexing

4.2 Real-time Spatial-Keyword Index Structures

Using the indexing building blocks presented in Section 4.1, we compose ten different index structures, most of them correspond to existing indexes in the literature [13]. The ten indexes fall in two categories: four *pure indexes* and six *hybrid indexes*. The four pure indexes organize data based on a single dimension, either spatial or keyword, and they correspond to the four building blocks (Section 4.1). Thus, our four pure indexes are Pure Grid index (*PG*), Pure Quadtree (*PQ*), Pure R-tree (*PR*), and Pure Inverted index (*PI*) and they correspond to the spatial grid, quadtree, R-tree, and inverted index, respectively. The six hybrid indexes organize data based on both spatial and keyword dimensions simultaneously, so each of them uses exactly two building blocks, one spatial index and one keyword index. The structure and data insertion of each hybrid index is briefly outlined below.

Table 2 shows structures of the six hybrid indexes with their correspondence in the literature. Each hybrid index uses two levels of indexing, spatial then textual or vice versa. Each index node of the first-level index stores an second-level index for its data objects instead of a simple list of objects. Incoming data batches are first inserted in the first-level index, then data of each first-level index node is considered a data batch for the second-level index and inserted accordingly. All insertions are done as described in Section 4.1. All hybrid indexes are loosely-coupled as tightly-coupled indexes ([13]) significantly increase the system overhead of index storage and maintenance in real time. Thus, the spatial component does not store keyword pruning information and vice versa. The following example gives further illustration.

Example. Figure 1 shows the structure of both pure and hybrid grid-based indexes, *PG*, *GI*, and *IG*. Figure 1a shows a pure grid index that divides the space into four equal-area cells. The index organizes eight objects, $o1$ to $o8$, based on their spatial locations. Each cell stores its objects in a list ordered by arrival time. Figure 1b shows the same spatial grid equipped with an inverted index in each cell to form a grid-inverted index (*GI*) for the eight objects. Instead of storing a list of objects, the cell organizes objects based on keywords. Figure 1c shows a reciprocal structure of the inverted-grid index (*IG*). The index organizes the eight objects based on keywords in an inverted index. Each keyword entry stores a spatial grid that stores only the objects that correspond to this keyword.

5 QUERY PROCESSING

This section presents processing TBRQ and TBKQ queries, that are defined in Section 3, using the ten indexes that are presented in Section 4. Both queries are temporally extended versions to common ones in the literature, so their processing is extended with the temporal dimension. Section 5.1 presents processing TBRQ query and Section 5.2 presents processing TBKQ query.

5.1 TBRQ Query

A key ingredient in processing TBRQ query on most indexes is using temporal pruning to speed up traversal of individual index nodes. We first introduce traversing an index node with temporal pruning. Then, we present processing TBRQ query in all indexes.

Temporal pruning. Traversal with temporal pruning assumes: (1) Index node N that stores a list of objects ordered by timestamp from most to least recent, which is the case in grid, quadtree, and inverted indexes. (2) A list *Ans* of k objects that represents an initial query answer. The goal of this traversal is to refine *Ans* to include objects in N that outperform the current k objects. To this end, we calculate a temporal upper bound T_u that equals the timestamp of the k^{th}, i.e., worst, object in *Ans*. Then, we traverse objects $o_i \in N$, one by one in time order. If $o_i.time > T_u$, i.e., o_i is more recent than the oldest object in *Ans*, we compute o_i ranking score, check if it outperforms the worst *Ans*'s object, update T_u, and move to the next object. If $o_i.time \leq T_u$, we terminate processing this cell as all following objects are worse than all *Ans* objects. This enables pruning a lot of objects and provides efficient query processing.

Processing TBRQ. *PG*, *PQ*, and *PI* indexes share a similar TBRQ query processor that consists of three steps. First, index nodes that intersect with the query range (or correspond to query keywords) are enqueued. Second, nodes are traversed, in queuing order, to get an initial answer list *Ans* that has k objects contain query keywords and within the spatial range. Finally, rest of nodes are traversed with temporal pruning to refine *Ans* and get the final result. *PR* index shares the first two steps, however, the second step continues to scan all nodes without temporal pruning to get the final answer. The reason that R-tree maintains objects unordered on time because its node split algorithm does not preserve the temporal order.

The hybrid indexes *GI*, *QI*, and *RI* use the same three-step framework as *PG*, *PQ*, and *PR*, respectively, with a modification to the first step. In particular, instead of enqueuing all content of the index node, we use the node inverted index to filter out data that correspond to the query keywords. Then, lists of objects are enqueued to be processed by the second and third steps.

The hybrid indexes *IG*, *IQ*, and *IR* share a three-step query processor that works as follows. First, inverted index nodes that correspond to query keywords are enqueued. Second, for each node in *IG*, *IQ*, and *IR*, the spatial index is queried the same as *PG*, *PQ*, and *PR*, respectively, to get top-k objects in each node. Finally, in case of multiple keywords, top-k objects from different nodes are merged to get the final top-k objects.

5.2 TBKQ Query

TBKQ query uses a spatio-temporal ranking function that scores objects based on both spatial and temporal proximities. Thus, the query processor prunes search space on both spatial and temporal dimensions. We first introduce spatial and temporal pruning procedures. Then, we present processing TBKQ query in all index structures.

Spatial pruning. Traversal with spatial pruning assumes: (1) A set S of index nodes ordered by spatial distance from a query point location $q.p$. (2) A list *Ans* of k objects that represents an initial query answer. The goal of this traversal is to refine *Ans* to include objects in S that outperform the current k objects. To this end, we calculate a spatial range upper bound R_u based on the spatio-temporal ranking function F_α assuming zero temporal score. Thus, $R_u = \frac{F_{\alpha,k}}{\alpha} \times R_{max}$ based on notations used in Section 3, where $F_{\alpha,k}$ is the value of F_α for the k^{th} object in *Ans*. Then, we traverse nodes $N \in S$, one by one in distance order. If $distance(N, q.p) < R_u$, we process N to refine *Ans* list, update R_u, and move to the next node. If $distance(N, q.p) \geq R_u$, we terminate processing all remaining S nodes as all following nodes do not outperform any object in *Ans*.

Temporal pruning. Temporal pruning for TBKQ is the same as the one presented in Section 5.1 with the exception of calculating the temporal upper bound T_u. Instead of using only timestamp, T_u is calculated based on the spatio-temporal ranking function F_α assuming zero spatial score. Thus, $T_u = q.t - \frac{F_{\alpha,k}}{1-\alpha} \times T_{max}$, where $F_{\alpha,k}$ is the value of F_α for the k^{th} object in *Ans*, $q.t$ is query time.

Processing TBKQ. *PG* and *PQ* indexes share a similar TBKQ query processor of three steps. First, we compute an initial list *Ans* of k objects that contain query keywords and are retrieved from the query point index node, and its neighbor nodes if it contains less than k objects. Second, we compose a set S of index nodes that intersect with the spatial range $R = \frac{F_{\alpha,k}}{\alpha} \times R_{max}$ ordered by distance from the query point. Finally, nodes in S are traversed with spatial and temporal pruning to get the final *Ans* list. *PR* index shares the exact first two steps. However, its third step uses only spatial pruning to get the final *Ans* list. *PI* index has a three-step query processor that works as follows. First, it retrieves index nodes that correspond to query keywords. Second, it gets initial k objects calculating their F_α scores. Finally, index nodes are traversed with temporal pruning to get the final top-k objects.

The hybrid indexes *GI*, *QI*, and *RI* use the same three-step framework as *PG*, *PQ*, and *PR*, respectively, with a modification to the third step. Instead of traversing all content of S nodes, the inverted index in each node is used to filter out data that correspond to the query keywords. Then, lists of objects are traversed with the pruning procedures.

The hybrid indexes *IG*, *IQ*, and *IR* process the query on three steps. First, inverted index nodes that correspond to query keywords are enqueued. Second, the nodes are traversed, in queuing order, and the spatial index of each node in *IG*, *IQ*, and *IR* is queried the same as *PG*, *PQ*, and *PR*, respectively, to get top-k objects in each node. Finally, in case of multiple keywords, top-k objects from different nodes are merged to get the final top-k objects.

6 EXPERIMENTAL EVALUATION

This section provides experimental evaluation of the different indexing and query processing techniques that are discussed in the previous sections. Section 6.1 presents the experimental setup. Sections 6.2-6.4 present index digestion scalability, memory consumption, and query processing scalability, respectively.

6.1 Experimental Setup

We evaluate ten indexes that are illustrated in Section 4. Indexes are denoted as *PG* for pure grid, *PQ* for pure quadtree, *PR* for pure R-tree, and *PI* for pure inverted index. Hybrid indexes that combine both spatial and keyword components are denoted by two corresponding letters in order, *IG* for inverted-grid, *GI* for grid-inverted, *IQ* for inverted-quadtree, *QI* for quadtree-inverted, *IR* for inverted-Rtree, and *RI* for Rtree-inverted. Our parameters include index node size, dataset size, query answer size *k*, query range, number of keywords in a query, and the space-time weighting parameter α. Unless mentioned otherwise, we use a default node size of 400 for both quadtree and R-tree, grid is divided into 30x30 cells, dataset of 60 millions objects, *k* value of 20, 2 keywords per query, 50 km query range, and α value of 0.2. Our performance measures include index data digestion rate (the average number of indexed objects per second), index memory footprint, and query latency.

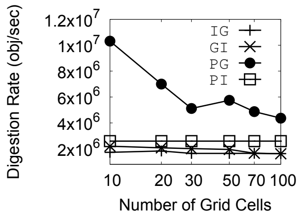
All experiments are based on Java 8 implementations for the evaluated indexes and their query processing and using an Intel Xeon(R) server with CPU E5-2637 v4 (3.50 GHz) and 128GB RAM running Ubuntu 16.04 (64 bit). The evaluation datasets and query workloads are described below.

Datasets. We have collected 6+ billions geotagged tweets from public Twitter Streaming APIs over the course of four years. All tweets are geotagged with either a precise latitude/longitude coordinates or a coarser granularity place, e.g., a city or a landmark. All non-point locations are represented with their centroid points. A random word/hashtag from the tweet text is associated as a keyword. Then, four datasets are composed with 30, 60, 90, and 120 millions tweets, which fit in the available main-memory and used to evaluate the scalability with growing data sizes. Each dataset is used to simulate a fast stream of data objects.

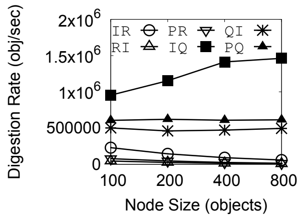
Query workloads. Each of our queries has one point location and 1-6 keywords. Thus, we use six query workloads; each set consists of a hundred queries and has the same number of keywords from 1 to 6. All query point locations are sampled from real location queries of Bing Mobile users, that are used in our prior work [35, 36]. In order to generate non-empty keyword queries, we use each query location and run a range query to retrieve objects within 50 km radius. Then, we filter out keywords that are associated with 15 or more objects to be used in our query workloads.

6.2 Index Scalability

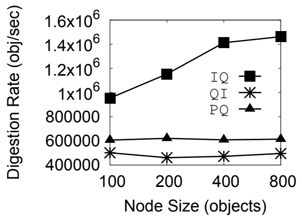
This section evaluates the scalability of different indexes in terms of real-time data digestion rate. Figure 2 shows the digestion rate with different index node sizes for all indexes. Figure 2a shows the digestion rates of *PG*, *IG*, *GI*, and *PI* with different numbers of grid cells, where the larger number of cells in the grid indicates a finer granularity of each cell (as the whole covered region is fixed). Hybrid indexes, *IG* and *GI*, have relatively stable digestion rates



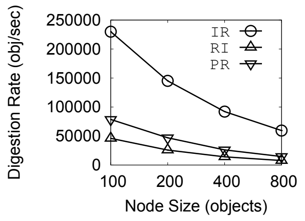
(a) Gird and inverted indexes



(b) Quadtree and Rtree indexes



(c) Quadtree indexes

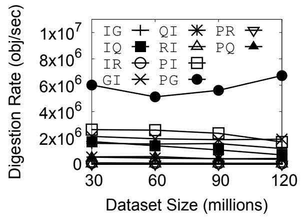


(d) Rtree indexes

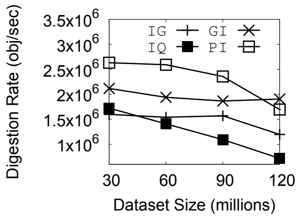
Figure 2: Digestion rate with varying node size

for different number of cells. *IG* has the smallest digestion rate with 1.65 million objects/second, while *GI* has a slightly higher rate with 2 million objects/second. *PI* digestion rate is higher than both with 2.6 millions object/second, fixed for all number of cells, which is self-explanatory as *PI* has no grid component. *PG* digestion is significantly higher than all other alternatives, even though it encounters a decreasing digestion with increasing number of grid cells. *PG*'s digestion rate starts at 10.3 millions objects/second at 10x10 grid and degrades to 5.5 millions objects/second at 100x100 grid. The four alternatives in this figure show that spatial grid index is always digesting more data compared to inverted index due to the small number of hash entries in the underlying hashtable which makes it cheaply accessible. When both grid and inverted index are coupled together, in *IG* and *GI*, the grid-first option (*GI*) still digests more data. However, the digestion of spatial grid is sensitive to the number of index cells as shown by the decreasing rate of *PG*. With the increasing number of cells, accessing the underlying hashtable becomes more expensive due to larger number of hash entires. Although the inverted index is also realized with a hashtable, its number of entries is much higher than grid, order of millions keywords, which makes it less sensitive. So, tuning the grid size parameter is crucial when using a pure spatial grid for streaming data. On the other hand, hybrid indexes, *IG* and *GI*, still show a stable digestion, which suggests that the inverted index component always stabilizes the digestion rate performance.

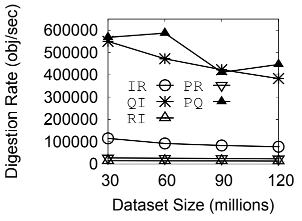
Figures 2b, 2c, and 2d show the quadtree-based and Rtree-based indexes with different node sizes. Figure 2b shows all the six alternatives. Obviously, all alternatives has significantly lower rate compared to *PI* and grid indexes, and all quadtree-based indexes outperform all Rtree-based indexes. *PI* and grid, as hash-based, are cheaply accessible compared to navigating multiple levels in hierarchical trees. Figures 2c and 2d focus on quadtree-based and Rtree-based alternatives, respectively. In Figure 2c, *IQ* has significantly better digestion rate, 1-1.4 million objects/second, compared to *PQ* and *QI*. This is interpreted by the inverted-first component as the inverted index has higher digestion than quadtrees. In addition, quadtree size in each inverted index entry of *IQ* is much smaller



(a) All indexes

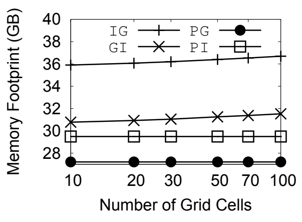


(b) Indexes with ≥ 1M obj/sec

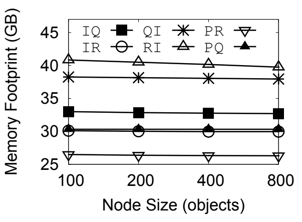


(c) Indexes with < 1M obj/sec

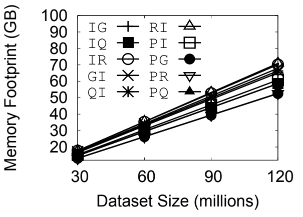
Figure 3: Digestion rate with varying dataset size



(a) Varying no. of grid cells



(b) Varying node size



(c) Varying dataset size

Figure 4: Memory consumption

than *PQ* and *QI*, which makes its insertion more efficient. Insertion in such small trees is also improved with increasing node size, due to less number of tree levels, which improves *IQ* digestion in turn. Both *PQ* and *QI* have relatively stable digestion with different node size. In fact, with 60 millions of indexed objects, changing the node size from 100 to 800 does not significantly affect the number of tree levels, and thus the insertion efficiency remains stable. Unlike quadtree, all Rtree-based indexes in Figure 2d encounter decreasing digestion rates with increasing node size due to the overhead encountered in restructuring the R-tree nodes as a data-partitioning index. This overhead significantly affects the real-time digestion and favors quadtrees over Rtrees. Still the inverted-first index *IR* has the best real-time digestion among the Rtree-based family.

Figure 3 shows the digestion rates of all indexes with varying dataset sizes. *PG* still significantly outperforms all other alternatives, as shown in Figure 3a, for the same reasons discussed in Figure 2. Figure 3b focuses on *PI*, *GI*, *IG*, and *IQ*, which digest more than 1 million objects/second. All of them encounter decreasing digestion rates with increasing data size. This is interpreted with the inverted index component that is common in all of them. With more objects indexed, the number of hash entries in the inverted index increase and makes insertion more expensive. Even though, it is noticeable that both *PI* and *IQ* are more sensitive to this change than the grid-based indexes *GI* and *IG*. In fact, the number of hash entries in a spatial grid is relatively stable, while insertion is done in append-only fashion, so increasing data size slightly affect the insertion performance. Figure 3c focuses on *PQ*, *QI*, *IR*, *PR*, and *RI*. Still all alternatives encounter decreasing digestion rates with increasing the dataset size due to the increasing tree hight which induces slower insertion. Yet, quadtree-based indexes *PQ* and *QI* encounter a higher relative degradation than Rtree-based indexes due to their better digestion performance. In all experiment, Rtree-based indexes have shown to 2-20 times lower digestion rates than other alternatives, which is a significant degradation. This is mainly caused by the high restructuring cost of the R-tree nodes in real time, which makes it less efficient to digest streaming data.

6.3 Memory Consumption

Figure 4 shows the memory consumption of different indexes with varying number of grid cells (Figure 4a), index node size (Figure 4b), and dataset size (Figure 4c). Varying both number of grid cells and node size do not significantly affect the memory consumption while increasing dataset size linearly increase the memory consumption of all indexes. This shows the minimal effect of increasing number of index cells on memory consumption compared to the huge amount of data objects that are kept in main-memory. With millions of streaming data objects managed in main-memory, multiplying the number of index cells by an order of magnitude does not really add much overhead to the system.

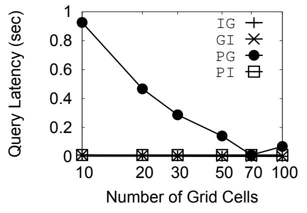
The pure indexes, *PG*, *PQ*, *PR*, and *PI*, consume less memory compared to all hybrid indexes. This is self-explanatory by the additional data structures in the hybrid indexes. The worthy note is the relatively slight increase in the memory consumption of hybrid indexes compared to the pure indexes. The increase falls in the range 1-8 GB which represents ∼15% increase on the average for 60 millions data objects for all indexes except *RI* that consumes the largest memory of ∼41 GB because different R-tree nodes replicate inverted index entries for the same keyword. Such repetition significantly increases memory storage due to excessive number of keywords. So, the major memory consumption still come from data objects storage even in hybrid indexes that use additional indexing data structures.

6.4 Query Evaluation

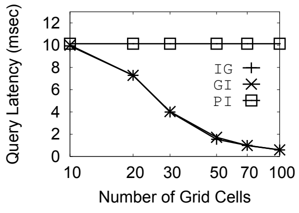
This section evaluates the performance of different indexes on the two queries that are defined in Section 3: Temporal Boolean Range Query (TBRQ) in Section 6.4.1 and Temporal Boolean kNN Query (TBKQ) in Section 6.4.2.

6.4.1 ***TBRQ Query.*** This section evaluates Temporal Boolean Range Query (TBRQ), abbreviated as *range query*. We first discuss the effect of index parameters on the query performance. Then, we discuss the effect of different query parameters and dataset size.

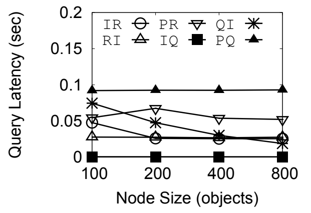
Effect of index parameters. Figure 5 shows the effect of index node size on range query latency. Figure 5a shows grid-based and



(a) Gird and inverted indexes

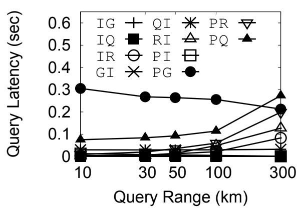


(b) *IG, GI,* and *PI* indexes

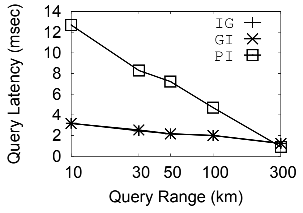


(c) Quadtree and Rtree indexes

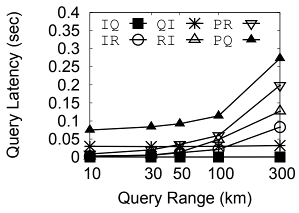
Figure 5: Range query latency with varying index node size



(a) All indexes

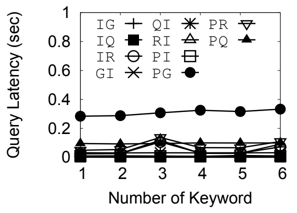


(b) Gird and inverted indexes

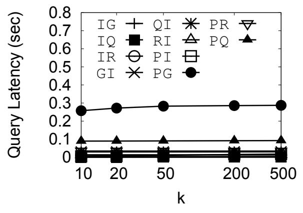


(c) Quadtree and Rtree indexes

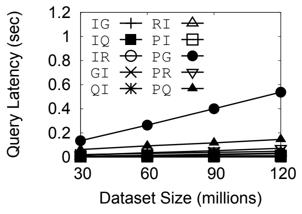
Figure 6: Range query latency with varying query range



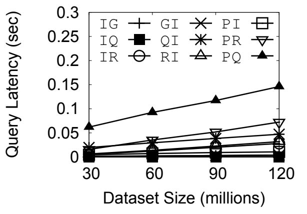
(a) Varying no. of keywords



(b) Varying *k*



(c) Varying dataset size



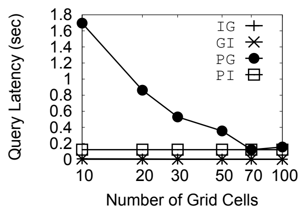
(d) Figure 7c excluding *PG*

Figure 7: Range query latency with different parameters

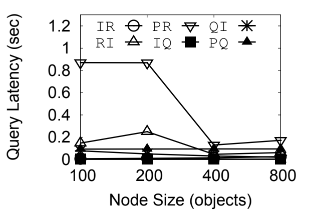
inverted indexes. Obviously, *PG* dominates all other alternatives with ∼0.9 sec latency for 10x10 grid, which decreases with increasing number of grid cells to reach 69 milli-sec for 100x100 grid. Such decrease is caused by dividing the same amount of data into smaller chunks with increasing number of grid cells in the query range. As a result, the temporal pruning eliminates much more objects, which leads to significant improvement in query latency. This is also the pattern with hybrid grid-based indexes *IG* and *GI* as shown in Figure 5b. In this figure, both *IG* and *GI* start with 10 milli-sec and drop to below 2 milli-sec. *PI* has a constant latency of 10 milli-sec as it does not depend on spatial grid cells. Figure 5c shows quadtree-based and Rtree-based indexes. Pure indexes *PQ* and *PR* has the highest consistent latency while hybrid indexes cut the latency at least to half. *PQ* has consistently the highest latency of 92 milli-seconds as it does not provide any keyword pruning, while *IQ* has consistently the lowest latency of 1 milli-second as it prunes by keyword, then spatial, then time, respectively. Although *QI* uses the same three pruning stages, however, their different order, spatial then keyword then time, adds significant overhead for the query processing which shows the power of keyword pruning at the first stage. Rtree-based indexes, *PR*, *RI*, and *IR*, do not employ temporal pruning at all which leads to considerable latency that ranges from 53 milli-seconds to 27 milli-seconds for different node sizes. Yet, *PR* still has better latency compared to *PQ* due to its balanced structure.

Effect of query parameters and dataset size. Figure 6 shows the range query latency with query range varied from 10 to 300 km. Figure 6a shows that *PG* still has the highest latency, ∼0.3 sec, however, it slowly decreases with increasing query range, which is a pattern for all grid-based and inverted indexes. Figure 6b shows this pattern. *PI* starts at 13 milli-sec while *IG* and *GI* start at 3 milli-sec and all of them decreases to 1 milli-sec with increasing range. This decrease is counter-intuitive as more data comes with wider ranges. However, the decrease is a result of fixing the answer size k and the low overhead of index nodes retrieval. With increasing range, getting the most recent k objects from more cells takes less time while the rest of objects are actually pruned by the temporal pruning procedure. In addition, the cost of retrieving more cells from grid and inverted indexes is low enough not to suppress this pruning power. So, combining both of them, the query latency decreases. Even though, this does not happen in quadtree-based and Rtree-based indexes as shown in Figure 6c. In these structures, retrieving more nodes adds an overhead, which is roughly equivalent to the temporal pruning savings so *IQ* and *QI* has a stable performance with increasing query range. Rtree-based indexes, *PR*, *IR*, *RI*, do not employ temporal pruning and so they encounter significant increase in latency with increasing range. Yet, this still of lower latency compared to *PQ* due to their balanced structures.

Figure 7 shows range query latency with varying number of keywords (Figure 7a), value of k (Figure 7b), and dataset size (Figures 7c and 7d). Different indexes have stable performance with varying number of keywords and k, while query latency increases with increasing data size. *PG* dominates other indexes with latency grows from 135 milli-sec with 30 millions objects dataset to 538 milli-sec with 120 millions objects. This is almost linear increase compared to a slow sub-linear increase for other indexes as shown in Figure 7d.

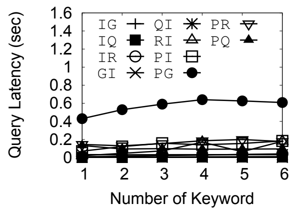


(a) Gird and inverted indexes

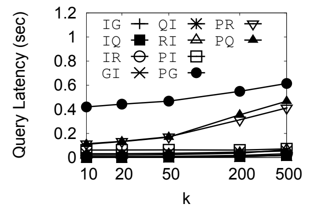


(b) Quadtree and Rtree indexes

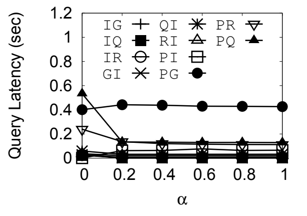
Figure 8: kNN query latency with varying index node size



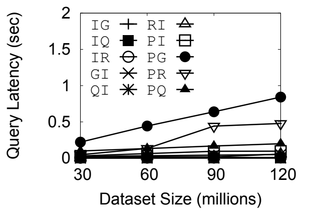
(a) Varying no. of keywords



(b) Varying k



(c) Varying α



(d) Varying dataset size

Figure 9: kNN query latency with different parameters

6.4.2 ***TBKQ Query.*** This section evaluates Temporal Boolean kNN Query (TBKQ), abbreviated as *kNN query*, versus index parameters, query parameters, and dataset size.

Effect of index parameters. Figure 8 shows kNN query latency with different index node sizes. Figure 8a shows constant latency for *PI* at 122 milli-sec and decreasing latency with increasing number of cells for grid-based indexes, *PG* from 1.7 to 0.15 sec, *IG* from 7.5 to 2.7 milli-sec, and *GI* from 5.5 to 1.8 mill-sec. The decreasing latency of grid-based indexes has the same interpretation as discussed for range query, while kNN query adds the power of spatial pruning to temporal pruning as well. However, it is noticeable that *PI* has significantly worse latency for kNN query compared to range query. This reflects the effect of the more complex spatio-temporal ranking function in kNN compared to pure temporal ranking in range query. Such additional ranking term, along with weighting parameter α, leads to processing much more objects to get the final top-k answer and increases the query latency significantly. Figure 8b shows superior performance for quadtree-based indexes over Rtree-based indexes. *PR* has the highest latency followed by *RI* then *PQ*. This is unlike range query where *PQ* had the highest latency due to its skewed structure. In kNN query, the spatio-temporal ranking, with potential spatial and temporal pruning, gives an advantage for quadtrees even in its pure spatial structure *PQ* compared to the hybrid structure *RI*. However, *RI* outperforms *PQ* when node size increases beyond 400 objects/node. The lowest latency is encountered by *IQ* at 3 milli-sec where the full powers of all pruning stages, keyword, spatial, and temporal, are exploited.

Effect of query parameters and dataset size. Figure 9 shows the effect of number of keywords (Figure 9a), value of k (Figure 9b), value of α (Figure 9c), and dataset size (Figure 9d) on kNN query latency. Figure 9a shows an increasing latency with increasing number of keywords for all indexes. This pattern is more observable in *PG* due to its high latency. However, changing number of keywords from one to six leads to ~50% increase in latency, which is a reasonable increase given that real applications mostly have less than six keywords. Figure 9b shows an increasing latency with increasing value of k. Yet, the relative increase is highly variable for different indexes. For example, *PI* has 15% increase, *PG* has 50% increase, while *PR* and *PQ* have up to 300% increase. Such increase is highly affected by the temporal pruning, which is more effective with the inverted index. However, in kNN query, hybrid indexes provide much lower latency due to its superior pruning and complexity of ranking objects in pure indexes.

Figure 9c shows a relatively high latency for smaller values of α for all indexes. Increasing α leads to decreasing latency with almost stable performance beyond α=0.2. At small values of α, spatial distance is dominating the temporal distance in the ranking function, and hence the spatial pruning is more important. High latency in such cases shows that spatial pruning is less effective than temporal pruning that leads to lower latency at higher α values. Unlike other indexes, *PG* maintains a relatively stable performance which shows that gird cells is more sensitive to spatial pruning compared to quadtree and R-tree. *PI* still has the best latency among all pure indexes. Figure 9d shows an increasing latency with increasing dataset size for all indexes, where more data naturally puts an overhead on query processing. However, *PG* and *PR* has much more increase compared to *PQ*, *PI*, and hybrid indexes.

7 DISCUSSION

Based on our evaluation, we provide system builders with insights and suggestions to support scalable spatial-keyword queries on streaming data. We first discuss query signatures to be supported in the system. Then, we discuss system index structures.

Query signatures. The evaluated queries, TBRQ and TBKQ, are temporal extensions for the two Boolean spatial-keyword queries as defined in the literature [13]. We choose to omit extending the third common spatial-keyword query, named *Topk kNN query* [13]. This query ranks result objects based on both spatial and textual similarity. Extending this with time means the ranking function would employ three ranking terms, spatial, temporal, and keyword. This is expected to significantly slow down the query latency. It is actually shown in our evaluation that TBKQ, that uses two-terms ranking function is significantly slower than TBRQ that uses only one term. Given that, keywords should be kept as Boolean filters on streaming data queries rather than incorporating them in the ranking for better system scalability.

Index structures. The evaluation shows the relative performance gains of different pure and hybrid indexes. Intuitively, pure indexes encounter higher query latency compared to hybrid indexes. However, from a system builder perspective, pure indexes have better scalability in terms of digestion rate, consume less memory resources, and have better flexibility in supporting a variety of query signatures, e.g., pure spatial queries. Given that, the evaluation shows that quadtree outperforms R-tree in TBKQ while the opposite is true for TBRQ. Spatial grid has higher latency than both of them in both queries. In fact, the inverted index outperforms all pure spatial indexes on both queries. This shows the power of keyword pruning and the benefit of limiting keywords to remain Boolean filters rather than a ranking term. For hybrid indexes, their performance gains increase with more complex ranking functions, such as in TBKQ. Inverted-quadtree index has shown superior performance in terms of digestion rate and query latency compared to other tree indexes. Nevertheless, grid-inverted index performs better and more consistent in terms of query latency and slightly better in digestion rate compared to tree indexes.

8 CONCLUSION

This paper provided an experimental evaluation for spatial-keyword queries on streaming data. We extended the common queries in the literature with the temporal dimension to effectively serve streaming applications. Existing query processing techniques are consequently extended with temporal pruning to effectively prune old objects. Ten major in-memory index structures, that are combinations of four indexing building blocks, are evaluated to process the extended queries. The evaluation provides system builders with insights on supporting scalable spatial-keyword queries with different trade-offs between system resources and query efficiency. The results have shown the superiority of keyword pruning and inverted index as a basic structure that is favorable at a system-level. For hybrid indexes, inverted-quadtree index has shown superior performance in terms of system resources while grid-inverted index has shown consistently low query latency.

REFERENCES

[1] Apache Spark. https://spark.apache.org/, 2017.
[2] W. G. Aref and H. Samet. Efficient Processing of Window Queries in the Pyramid Data Structure. In *PODS*, 1990.
[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In *Acm Sigmod Record*, volume 19, 1990.
[4] After Boston Explosions, People Rush to Twitter for Breaking News. http://www.latimes.com/business/technology/la-fi-tn-after-boston-explosions-people-rush-to-twitter-for-breaking-news-20130415,0,3729783.story, 2013.
[5] C. Budak, T. Georgiou, D. Agrawal, and A. El Abbadi. Geoscope: Online Detection of Geo-correlated Information Trends in Social Networks. *PVLDB*, 7(4), 2013.
[6] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-Time Search at Twitter. In *ICDE*, 2012.
[7] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware Optimal Route Search. *PVLDB*, 5(11), 2012.
[8] X. Cao, G. Cong, T. Guo, C. S. Jensen, and B. C. Ooi. Efficient Processing of Spatial Group Keyword Queries. *TODS*, 40(2), 2015.
[9] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective Spatial Keyword Querying. In *SIGMOD*, 2011.
[10] A. Cary, O. Wolfson, and N. Rishe. Efficient and Scalable Method for Processing Top-k Spatial Boolean Queries. In *SSDBM*, 2010.
[11] G. Chen, J. Zhao, Y. Gao, L. Chen, and R. Chen. Time-Aware Boolean Spatial Keyword Queries. *TKDE*, 29(11), 2017.
[12] L. Chen, G. Cong, and X. Cao. An Efficient Query Indexing Mechanism for Filtering Geo-textual Data. In *SIGMOD*, 2013.
[13] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial Keyword Query Processing: an Experimental Evaluation. In *PVLDB*, volume 6, 2013.
[14] L. Chen, Y. Cui, G. Cong, and X. Cao. SOPS: A System for Efficient Processing of Spatial-keyword Publish/Subscribe. *PVLDB*, 7(13), 2014.
[15] How Twitter, Facebook, WhatsApp And Other Social Networks Are Saving Lives During Disasters. http://www.huffingtonpost.in/2017/01/31/how-twitter-facebook-whatsapp-and-other-social-networks-are-sa_a_21703026/, 2017.
[16] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. Space: Efficient Geo-search Query Processing. In *CIKM*, 2011.
[17] G. Cong and C. S. Jensen. Querying Geo-textual Data: Spatial Keyword Queries and Beyond. In *SIGMOD*, 2016.
[18] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *PVLDB*, 2(1), 2009.
[19] S. A. et. al. AsterixDB: A Scalable, Open Source BDMS. *PVLDB*, 7(14), 2014.
[20] R. Grover and M. Carey. Data Ingestion in AsterixDB. In *EDBT*, 2015.
[21] T. Guo, X. Cao, and G. Cong. Efficient Algorithms for Answering the M-closest Keywords Query. In *SIGMOD*, 2015.
[22] A. Guttman. *R-trees: A Dynamic Index Structure for Spatial Searching*, volume 14. 1984.
[23] T.-A. Hoang-Vu, H. T. Vo, and J. Freire. A Unified Index for Spatio-Temporal Keyword Queries. In *CIKM*, 2016.
[24] Hurricane Harvey Victims Turn to Twitter and Facebook. http://time.com/4921961/hurricane-harvey-twitter-facebook-social-media/, 2017.
[25] In Irma, Emergency Responders' New Tools: Twitter and Facebook. https://www.wsj.com/articles/for-hurricane-irma-information-officials-post-on-social-media-1505149661, 2017.
[26] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. Technical report, 1993.
[27] Embrace of Social Media Aids Flood Victims in Kashmir. https://www.nytimes.com/2014/09/13/world/asia/embrace-of-social-media-aids-flood-victims-in-kashmir.html, 2014.
[28] T. Lee, J.-w. Park, S. Lee, S.-w. Hwang, S. Elnikety, and Y. He. Processing and Optimizing Main Memory Spatial-Keyword Queries. *PVLDB*, 9(3), 2015.
[29] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware Publish/Subscribe. In *KDD*, 2013.
[30] Z. Li, K. C. Lee, B. Zheng, W.-C. Lee, D. Lee, and X. Wang. IR-tree: An Efficient Index for Geographic Document Search. *TKDE*, 23(4), 2011.
[31] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective Spatial Keyword Queries: a Distance Owner-driven Approach. In *SIGMOD*, 2013.
[32] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. M. Ghanem, S. Ghani, and M. F. Mokbel. Taghreed: a System for Querying, Analyzing, and Visualizing Geotagged Microblogs. In *SIGSPATIAL*, 2014.
[33] A. Magdy, R. Alghamdi, and M. F. Mokbel. On Main-memory Flushing in Microblogs Data Management Systems. In *ICDE*, 2016.
[34] A. Magdy and M. Mokbel. Demonstration of Kite: A Scalable System for Microblogs Data Management. In *ICDE*, 2017.
[35] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Mercury: A Memory-constrained Spatio-temporal Real-time Search on Microblogs. In *ICDE*, 2014.
[36] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Venus: Scalable Real-time Spatial Queries on Microblogs with Adaptive Load Shedding. *TKDE*, 2016.
[37] 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. 8(12), 2015.
[38] A. R. Mahmood, W. G. Aref, and A. M. Aly. FAST: Frequency-Aware Indexing for Spatio-Textual Data Streams. In *ICDE*, 2018.
[39] A. R. Mahmood, W. G. Aref, A. M. Aly, and M. Tang. Atlas: On the Expression of Spatial-keyword Group Queries Using Extended Relational Constructs. In *SIGSPATIAL*, 2016.
[40] A. R. Mahmood, A. Daghistani, A. M. Aly, W. G. Aref, M. Tang, S. M. Basalamah, and S. Prabhakar. Adaptive Processing of Spatial-Keyword Data Over a Distributed Streaming Cluster. *CoRR*, abs/1709.02533, 2017.
[41] M. Mokbel and A. Magdy. System and Method for Microblogs Data Management, U.S. Patent and Trademark Office on August 31, 2015, Application number: 14/841299. http://appft1.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PG01&p=1&u=/netahtml/PTO/srchnum.html&r=1&f=G&l=50&s1=20160070754.PGNR., 08 2015.
[42] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient Processing of Top-k Spatial Keyword Queries. In *SSTD*, 2011.
[43] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. TwitterStand: News in Tweets. In *SIGSPATIAL*, 2009.
[44] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. Technical report, 1987.
[45] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36(2), 2013.
[46] J. Subercaze, C. Gravier, and F. Laforest. Real-time, Scalable, Content-based Twitter Users Recommendation. In *Companion of WWW*, 2018.
[47] Health Department Use of Social Media to Identify Foodborne Illness - Chicago, Illinois, 2013-2014. https://www.cdc.gov/mmwr/preview/mmwrhtml/mm6332a1.htm, 2014.
[48] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual Indexing for Geographical Search on the Web. In *SSTD*, 2005.
[49] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: Efficiently Support Continuous Spatial-keyword Queries over Stream. In *ICDE*, 2015.
[50] L. Wu, W. Lin, X. Xiao, and Y. Xu. LSII: An Indexing Structure for Exact Real-Time Search on Microblogs. In *ICDE*, 2013.
[51] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid Index Structures for Location-based Web Search. In *CIKM*, 2005.
[52] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACS*, 38(2), 2006.