Spatial-Keyword Skyline Publish/Subscribe Query Processing Over Distributed Sliding Window Streaming Data

Ze Deng, Yue Wang, Tao Liu, Schahram Dustdar, *Fellow, IEEE*, Rajiv Ranjan, Albert Zomaya, *Fellow, IEEE*, Yizhi Liu, and Lizhe Wang, *Fellow, IEEE*

Abstract—Current spatial-keyword publish/subscribe systems need to handle spatial-keyword skyline queries over geo-textual streams to continuously obtain good results. The skyline queries in such systems face two main problems: (1) query problems, because the powerful query capability is required for the strict limit of the response time and the large number of items concerned by the users, and (2) scalability issue, because millions of active users are maintained simultaneously with many network-connected machines. Unfortunately, the current approach is towards static data. Thus, this paper first proposes a distributed skyline query processing framework. Then, we optimize the skyline computing by introducing MF-R^t-tree, which is an update-efficient and space-saving indexing structure and a fast approach for processing a continuous spatial-keyword skyline query called $eager^*$. Finally, a spatial and textual signature-based communication optimization method is proposed to support scalability. The experimental results indicate that (1) MF-R^t-tree can significantly reduce update costs, while maintaining a low storage cost, and a query performance comparable to IL-Quadtree, (2) $eager^*$ can averagely accelerate 79.72 $\times$ faster than the method based on BNL, (3) the communication optimization method significantly reduces the communication cost, and (4) the distributed framework can efficiently support large-scale skyline queries.

Index Terms—Publish/subscribe systems, spatial-keyword skyline query, geo-textual streaming data, indexing structure, communication cost

1 Introduction

A massive amount of geo-textual data that contain both textual information and geographical location information are being generated at an unprecedented speed due to the proliferation of GPS-equipped mobile devices, embedded systems [1], and social-media services. For instance, millions of social-media users are uploading photos to Instagram with both location and textual tags, posting geo-tagged tweets on Twitter, and creating location-aware events on Facebook using their smart phones [2]. These geo-textual data have been generated in a stream fashion and contain valuable information for users. Moreover, users may focus on events in particular regions, and they hope to receive up-to-date geo-textual data related to such events.

The spatial-keyword publish/subscribe systems (e.g, [3], [4], [5]) have provided the basic primitives with which to support the above-mentioned information processing paradigms. However, most of them are geared towards Boolean matching query [6] such that too many results satisfy these two constraints for users in terms of locations and keywords. Thus, users would prefer to gain "better" results by using preference queries. Top-k and skyline queries are two kinds of preference queries [7]. Recently, two methods of top-k queries [3], [6] in spatial-keyword publish/subscribe systems have been proposed. However, according to the description in [7], users cannot get a measuring standard to obtain the top-k answers, because the weights of the dimensions of top-k queries may be unknown. In this case, skyline queries can help users become aware of all the good results.

The challenges associated with the skyline queries in spatial-keyword publish/subscribe systems over streams fall into two categories:

1) The query problem. The geo-textual data are generated in a stream fashion. Thus, multiple skyline queries lie in the continuous query processing. The strict limit of the response time and the large number

- *Ze Deng and Lizhe Wang are with the School of Computer Science, China University of Geosciences, Wuhan 430078, China, and also with Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China. E-mail: deng_ze@163.com, lizhe.wang@gmail.com.*
- *Yue Wang, Tao Liu, and Yizhi Liu are with the School of Computer Science, China University of Geosciences, Wuhan 430074, China. E-mail: {yuewang, taoliu}@cug.edu.cn, 123654784@qq.com.*
- *Schahram Dustdar is with the Technische Universität Wien, 1040 Vienna, Austria. E-mail: dustdar@dsg.tuwien.ac.at.*
- *Rajiv Ranjan is with the School of Computing, Newcastle University, Tyne NE1 7RU, U.K. E-mail: rranjans@gmail.com.*
- *Albert Zomaya is with the School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia. E-mail: albert.zomaya@sydney.edu.au.*

Manuscript received 8 December 2020; revised 20 August 2021; accepted 2 January 2022. Date of publication 6 January 2022; date of current version 8 September 2022.
This work was supported in part by the National Natural Science Foundation of China under Grants U21A2013, 41925007, and 62076224 and in part by Open Research Project of The Hubei Key Laboratory of Intelligent Geo-Information Processing under Grant KLIGIP-2019B14.
(Corresponding author: Lizhe Wang.)
Recommended for acceptance by H. Jiang.
Digital Object Identifier no. 10.1109/TC.2022.3140884

of items that concern the users (i.e., the size of the sliding window in our setting) make the query more challenging and demand more powerful query capability.

2) The scalability problem. The limited computational resources in a single machine often become a bottleneck. Thus, the spatial-keyword publish/subscribe systems often are very large scale, with many network-connected machines, and millions of active users consequently need to be maintained simultaneously. Thus, the main challenge here lies in how to achieve a small communication cost among these machines.

Unfortunately, only [7] and [8] have discussed how to deal with spatial-keyword skyline queries over static geo-textual data. There are no existing solutions to our problems. Therefore, in this study, we first present a distributed skyline query processing framework for the spatial-keyword publish/subscribe systems. Furthermore, we propose optimization methods for the performance and scalability of our framework. The principal contributions of the paper are as follows:

1) We proposed a distributed skyline query processing framework for large-scale spatial-keyword publish/subscribe systems by modifying a distributed top-k spatial-keyword query processing framework [6].
2) We proposed an update-efficient and space-saving indexing structure Memo-and-Filter-based R^t-tree (MF-R^t-tree) to efficiently index geo-textual streaming data tuples in the framework, by fusing and improving the features of R^t-tree [9], cuckoo filter [10], and RUM-tree [11].
3) We proposed a fast processing approach for a continuous spatial-keyword skyline query to suit our streaming setting, by modifying a skyline computing scheme for streaming data [12] based on a spatial-keyword skyline computing algorithm for static data [7].
4) We proposed a novel communication optimization method to significantly reduce the communication cost of our distributed framework, by providing a spatial cuckoo filter based on the spatial coding technique for space pruning and by employing a one-permutation min-wise signature method for keyword pruning.

To the best of our knowledge, the proposed skyline query method is the first spatial-keyword skyline query processing method for distributed publish/subscribe systems.

The remainder of this paper is organized as follows: Section 2 presents the related research work. Section 3 proposes the skyline query processing framework and optimization methods. Section 4 proposes the method of optimizing the communication cost of the framework. Section 5 presents the experiments and results. Section 6 concludes this paper with a summary.

2 Related Work

This section describes the most salient studies of skyline computing over streaming data, continuous spatial-keyword query, indexing geo-textual objects, and distributed skyline computing.

2.1 Skyline Over Streaming Data

Computing the skyline over data streams is a substantial challenge, due to the unbounded stream length [13]. The *sliding window* has been widely employed to process skyline over a data stream. For instance, in [14] the skyline was computed over a count-based *window* of the last n received tuples. Subsequently, a time-based window method was used for continuous skyline queries in [15], with the algorithm *LookOut*. In [12], the authors proposed the lazy and eager sequential algorithms for computing the skyline over streams with a time-based *sliding window*. In [16], authors presented an energy-efficient continuous skyline query method over a sensor data stream in WSNs, called EECS, to reduce communication cost on processing skyline query. The above-mentioned methods aim to compute the skyline over the precise streaming data. Due to the fact that streaming data values are inherently uncertain and imprecise [17], some approaches for computing skyline over uncertain streaming data have been proposed. In [18], Zhang *et al.* proposed efficient techniques for finding probabilistic skyline objects based on sliding windows. Unlike the work in [18], which proposed techniques only for the case in which each object has a single instance, Ding *et al.* [19] proposed a skyline computing method for dealing with the case in which each uncertain object contains multiple instances. However, these skyline query methods were not considered over geo-textual streaming data. In this paper, we employed the sliding window to process skyline queries over geo-textual streaming data.

2.2 Continuous Spatial-Keyword Queries

Continuous spatial-keyword queries are issued once and then logically execute continuously to retrieve objects which satisfy both spatial and keyword constraints over geo-textual streams. The queries can be roughly classified into two categories [7], i.e., Boolean matching and preference matching. The representative methods based on Boolean matching include IQ-tree [20] and AP-tree [5]. However, Boolean matching query methods may incur too many results to satisfy the two constraints for users in terms of locations and keywords. Thus, preference queries (e.g., top-k query and skyline query) can be used to gain "better" results. Currently, preference matching focuses on top-k query. For instance, the authors of [3] proposed an efficient solution to process a large number of top-k queries over a stream of geo-textual objects. In [21], to deal with continuous top-k spatial-keyword queries on road networks, the authors proposed two methods that can monitor such moving queries in an incremental manner and reduce repetitive traversing of network edges for better performance. In [6], the authors investigated top-k spatial-keyword publish/subscribe over a sliding window in both a single machine and a distributed cluster. In this paper, we focus on skyline queries.

2.3 Indexing Geo-Textual Objects

To support fast spatial-keyword query, extensive efforts have been made to index geo-textual objects. The representatives

of conventional indexes includes IR-tree [22], Inverted-KD tree [7], I^3 [23] that is an integrated inverted index, which adopts the quadtree to hierarchically partition the data space into keyword cells, and IL-Quadtree [24] that is an inverted linear quadtree. These indexing methods can directly applied to our setting. However, the high update cost is an issue in the case of streams, because they are toward static geo-textual objects. Meanwhile, most of the conventional indexing structures may incur high space costs due to holding numerous keywords.

In recent years, the indexing structures mainly focus on supporting fast complex spatial-keyword queries, e.g., collective spatial-keyword ones and sematic spatial-keyword ones. For example, LIR-tree [25], which is extended from IR-tree, is designed for level-aware collective spatial-keyword queries. In [26], authors provided a two-layer hybrid index structure called Quad-cluster Dual-filtering R-Tree (QDR-Tree) to support Attributes-Aware Spatial-Keyword Query (ASKQ). In [27], LHQ-tree was proposed to support sematic-aware top-k Spatial-keyword queries. LHQ-tree is a three layered hybrid indexing structure that first uses a quadtree to index objects based on their locations, then creates a set of n-gram inverted list for textual information and holds topic-based sematic information with LSH, respectively, for every quadtree leaf node. As we can see, compared with conventional indexing structures, the indexing structures for complex spatial-keyword queries are more complex, resulting in worse update performances, in the case of streams.

Moreover, to support dynamic geo-textual objects, recently, a gird-based index [28] was proposed to manage dynamic both top-k spatial-keyword queries and dynamic spatial-keyword objects. However, the gird-based index only bounds the maximum and minimum weights of objects' textual attributes related to queries, instead of indexing the keyword sets of objects. When computing skyline, we need employ an indexing structure to quickly match keywords between geo-textual objects. Therefore, this indexing method is not suitable for our setting.

2.4 Distributed Skyline Computing

Because the limited computational resources in a single machine often become a bottleneck when processing large-scale skyline queries, some studies of distributed skyline computing have been conducted. For instance, in [29], the authors proposed a collaboration approach for continuous skyline computing in a two-tier streaming environment with a server as query interface and multiple data sites. In [30], the authors proposed a distributed parallel framework to address the parallel skyline query problem over uncertain data streams with a sliding window streaming model. In [31], a two-level distributed skyline query processing method was presented over uncertain data streams.

Additionally, the MapReduce framework has been widely used to process skyline queries (e.g, MR-GPMRS [32], MR-BNL [33], PPF-PGPS [34], SKY-MR [35] and SKY-MR$^+$ [36]). The above-mentioned skyline computing methods are not designed for the publish/subscribe case. In [37], the MapReduce framework was used to deal with continuous skyline queries in dynamically weighted road networks. However, the method is not towards spatial-keyword skyline queries.

In contrast to existing skyline computing methods, this paper targets continuous spatial-keyword skyline query processing over geo-textual streaming data in distributed publish/subscribe systems. To the best our knowledge, only [7] and [8] have proposed how to deal with spatial-keyword skyline queries. However, the method about spatial-keyword skyline query in [7] is toward static geo-textual objects, while in [8] the authors proposed secure spatial-keyword skyline query approaches for static geo-textual objects as well, through an encrypted IR-tree. There are no existing solutions to our problem.

3 The Distributed Spatial-Keyword Skyline Query Processing Framework

This section first formulates this problem. It then describes the details of distributed spatial-keyword skyline query processing framework.

3.1 Problem Formulation

We assume that the space is a 2D space. Then, we formulate the problem.

Definition 1(a geo-textual streaming data tuple or a message[1]) *Is described as $dt = (d, w, l, t_a, t_e)$, where d is a geo-textual object related to the tuple, w is a set of keywords, $dt.l$ is a spatial point with longitude and latitude, t_a is the time when the tuple arrives at the sliding window, and t_e is its expiration time.*

Definition 2(sliding Window). *Defined as W, is a time-based window over a stream in time order and covers $|W|$ geo-textual data tuples. That means the expiration time is $dt.t_e = dt.t_a + |W|$ for a streaming data tuple dt.*

Definition 3 (a continuous spatial-keyword query or a subscription[2]). *Is denoted as $q = (w, r)$, where $q.w$ is a set of query keywords and $q.r$ is a spatial range in the form of a rectangle. Meanwhile, we denote $q.r.c$ as the centroid of the range $q.r$, and $q.r.d$ as half the length of the diagonal line of $q.r$.*

Definition 4 (Text Relevance). *For a continuous spatial-keyword query q and a geo-textual streaming data tuple dt, the text relevance between q and dt is denoted as $R_T(q, dt)$. It is the set resemblance between $q.w$ and $dt.w$ and can be computed as*

$$R_T(q, dt) = \frac{|q.w \cap dt.w|}{|q.w \cup dt.w|}. \quad (1)$$

Definition 5(Spatial Relevance). *For a continuous spatial-keyword query q and a geo-textual streaming data tuple dt, the spatial relevance between q and dt is denoted as $R_S(q, dt)$. It can be estimated as*

$$R_S(q, dt) = \begin{cases} 0 & if(dt.l \ is \ not \ in \ q.r) \\ 1 - \frac{eDis(dt.l,\ q.r.c)}{q.r.d} & otherwise \end{cases}. \quad (2)$$

where $eDis(dt.l, q.r.c)$ is the distance between $dt.l$ and $q.r.c$.

1. In publish/subscribe systems, a message is a streaming data tuple.

2. In publish/subscribe systems, a subscription is a continuous query.

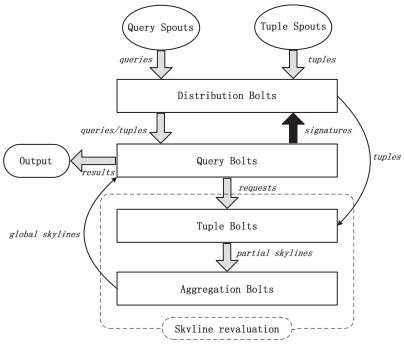


Fig. 1. The framework for processing spatial-keyword skyline queries in the distributed publish/subscribe system.

Definition 6 (Coincidence With Respect to q). *Given a spatial-keyword query q and two geo-textual streaming data tuples $dt1$ and $dt2$, $dt1$ is coincident with $dt2$ with respect to q, denoted as $dt1 \equiv dt2|q$,* **iff :** $(R_T(q, dt1) = R_T(q, dt2) \wedge R_S(q, dt1) = R_S(q, dt2)) \wedge (R_T(q, dti) > 0 \wedge R_S(q, dti) > 0 \mid i = 1, 2)$. *Otherwise, $dt1$ is not coincident with $dt2$ with respected to q, denoted as $dt1 \not\equiv dt2|q$.*

Definition 7 (Dominance With Respect to q). *Given a spatial-keyword query q and two geo-textual streaming data tuples $dt1$ and $dt2$, $dt1$ dominates $dt2$ with respected to q, denoted as $dt1 \prec dt2 \mid q$,* **iff :** $R_S(q, dt1) \geq R_S(q, dt2) \wedge R_T(q, dt1) \geq R_T(q, dt2) \wedge dt1 \not\equiv dt2|q$.

Definition 8 (Skyline With Respect to q). *Given a continuous spatial-keyword query q and a set of geo-textual streaming data tuples P, the skyline over P with respect to q, denoted as $SKY(P|q)$, is the set of data tuples. $SKY(P|q) = \{p \in P \mid \nexists k \in P: k \prec p \mid q\}$.*

For a set of continuous spatial-keyword queries Q, the objective in this paper is to efficiently compute each $SKY(P_i|q_i)$ $(q_i \in Q)$ over W in a distributed manner.

3.2 The Framework

We first describe the framework, then present an indexing approach for streaming data tuples, and finally propose a method for skyline processing over geo-textual streaming data.

3.2.1 The Architecture of Framework

Like a distributed top-k spatial-keyword query processing framework DSkype [6], our framework (illustrated in Fig. 1) is based on Apache Storm [38], and it also consists of six components: 1) query spouts, 2) tuple spouts, 3) distribution bolts, 4) query bolts, 5) tuple bolts, and 6) aggregation bolts.

The *query spouts* are responsible for receiving new query requests, whereas the *tuple spouts* input geo-textual streaming data tuples from external data sources, such as Twitter API.

The *distribution bolts* have three functions:

Function 1:
Distributing queries from *query spouts* to *query bolts*.
Function 2:
Navigating streaming data tuples from *tuple spouts* to *query bolts*.

Function 3:
Routing the new tuples remaining after pruning by spatial signatures and textual signatures to *tuple bolts* to ensure an up-to-date sliding window.

Furthermore, the *query bolts* have four functions:

Function 1:
Establishing an index of queries based on a Quadtree structure with inverted files for filtering incoming tuples.

Function 2:
Receiving and outputting skylines from *aggregation bolts* and maintaining the result buffers of skyline queries.

Function 3:
Sending the requests of skyline revaluation to tuple bolts when tuples are expired in result buffers.

Function 4:
Creating spatial signatures and textual signatures from queries and sending them to *distribution bolts*.

Note that, unlike the *subscription bolts* in [6], our *query bolts* have one new function (i.e., function 4). Function 4 can generate spatial signatures and textual signatures of queries. *Query bolts* send the two kinds of signatures to *distribution bolts* to significantly reduce the communication costs between them. The details are proposed in Section 4.

The *tuple bolts* and *aggregation bolts* cooperatively execute skyline computing once *query bolts* send requests as new tuples arrive or old tuples expire. Concretely, the *tuple bolts* maintain the sliding window in a distributed manner. Each *tuple bolt* maintains a geo-textual index over the local *sliding window* and generates a partial skyline result over the local *sliding window*. Then, the *aggregation bolts* are responsible for computing the global skyline based on partial skyline results from all *tuple bolts*, and they send the final result to the *query bolts*.

3.2.2 Indexing Streaming Data Tuples

Both *tuple bolts* and *aggregation bolts* need efficiently process skyline queries over sliding window. For fast skyline processing, an efficient indexing structure over sliding window is an essential block.

In [6], the authors used IR-tree [22] to index streaming data tuples. Nevertheless, IR-tree is designed for static data and unsuitable to streaming data. On the other hand, IR-tree has to hold a number of keywords to incur a high space cost, which may degrade its performance in terms of both queries and updates. Therefore, we propose an update-efficient and space-saving indexing structure called MF-R^t-tree to index geo-textual streaming data tuples by fusing the advantages of the RUM-tree [11], cuckoo filter [10] and R^t-tree [9]. The indexing structure is illustrated in Fig. 2. First, we employ R^t-tree to index geo-textual tuples in one sliding window. This means that the tuples are indexed by a R-tree based on their locations. Then, each R-tree node stores the keywords of its descendant tuples for the purpose of textual

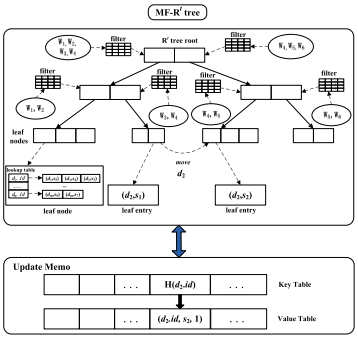


Fig. 2. The tuple data index.

filtering. To improve the update performance of R^t-tree over the streaming data tuples, we incorporate an update memo in [11] into R^t-tree. We call it M-R^t-tree. The primary feature of M-R^t-tree is that the old entry in the tree is allowed to coexist with newer entries before it is removed later, rather than deleting it when updating tuples. In M-R^t-tree, each index entry in one leaf is assigned a stamp by the timestamp of a tuple when the tuple is inserted into the tree. To distinguish the latest entries from the obsolete entries, an update-memo structure is used as in [11]. Specially, the update-memo structure is a hash table, which contains multiple <key, value> pairs. Each entry is formed with <H($d.id$), V>, where $d.id$ is the identifier of a geo-textual object related to one tuple, H($d.id$) is the hash value of $d.id$, and V = ($d.id$, S_{latest}, N_{old}) represents one entry in the update memo, in which S_{latest} is the stamp of the latest entry of the geo-textual object and N_{old} stands for the maximum number of obsolete entries. The deletion of a tuple in the M-R^t-tree is equivalent to marking the latest entry of the geo-textual object as obsolete. The deletion does not actually go through the R^t-tree. It affects only the update memo entry for the data tuple to be deleted, by changing S_{latest} to the next value assigned by the stamp counter, and incrementing N_{old} by 1.

Furthermore, to achieve a good query performance, we employ a lookup table to organize all tuples including old tuples in one leaf node. The lookup table is essentially a map whose key is the identifier of one geo-textual object, and the value is a stack list for storing index entries for the object in descending order of stamp. A new tuple can be pushed into the top of a stack list with an object identifier. The lookup table can improve the query performance of M-R^t-tree because we only need to check the top entry in a stack list, since the top entry is the latest one in the list. This method can avoid scanning numerous obsolete entries. For example, in Fig. 2, in the stack list for d_1, we only check the entry (d_1, s_5) and ignore (d_1, s_4) and (d_1, s_3). Meanwhile, this map structure also can improve the update performance of our indexing structure by removing the whole list when the top entry in a stack list is obsolete, instead of removing each entry one by one.

However, like IR-tree, M-R^t-tree has the issue of high space cost. M-R^t-tree has to store the large number of keywords in indexing nodes, especially for nodes in higher levels due to the aggregation of keywords. To solve this problem, we employed a space-saving data structure to represent keywords in M-R^t-tree, following the idea in [39]. In [39], the authors proposed a Bloom-filter-based R-tree (BR-tree), which integrates counting Bloom filters [40] into R-tree nodes. A counting Bloom filter is used to represent the item set in each BR-tree node to facilitate fast approximate membership query and dynamic update of items. The parent node can compute its counting Bloom filter by taking the union operation of the Bloom filters in its children. However, a counter consists of four or more bits such that a counting Bloom filter requires four times more space than a standard Bloom filter. Thus, we incorporate a more space-efficient data structure cuckoo filter [10] into M-R^t-tree. The cuckoo filter takes ≤ 1 time space cost than a standard Bloom filter with deletion support. Furthermore it only uses two hashing functions and can achieves a query performance with $O(1)$ time complexity. Concretely speaking, a cuckoo filter is a cuckoo hash table, which consists of an array of buckets, where a bucket can have multiple entries. Each entry stores the fingerprint of one item x. Each item x has two candidate buckets determined by hash function $h_1(x) = hash(x)$, and $h_2(x) = h_1(x) \oplus hash(x'sfingerprint)$. With $h_1(x)$ and $h_2(x)$, the cuckoo filter can add, delete and lookup items dynamically.

We refer the cuckoo filter-based M-R^t-tree as the MF-R^t-tree. As Fig. 2 shows, we used a cuckoo filter (i.e., a cuckoo hash table) to represent the set of keywords in each MF-R^t-tree node. The lookup and maintenance operations over cuckoo filters (see the details in [10]) can be employed to search and maintain keywords.

In [11], garbage cleaners were used to remove old index entries in RUM-tree lazily. The garbage cleaners can work in two ways: cleaning in batches in idle time and cleaning upon touch whenever a leaf node is accessed during an update operation. Because our study addresses streaming processing, we apply the clean-upon-touch scheme for MF-R^t-tree. However, the original clean-upon-touch scheme in [11] is easy to incur a high update cost for frequent update operations. Therefore, we optimized the cleaning scheme with lookup tables in leaf nodes (see Fig. 2). The main idea is that a garbage cleaner only runs deletion operations over some obsolete stack lists whose lengths are greater than a cleaning threshold T_{clean}, instead of obsolete tuples in all stack lists, when the lookup table in a leaf node is accessed by an update operation. Thus, we can avoid numerous deletion operations.

However, how to gain T_{clean} is an issue. In our setting, the update access frequency of a leaf node is mainly related to two factors: the arrival rate of incoming streaming tuples (denoted as $arrR$) and the height of MF-R^t-tree (denoted as h). If h is fixed, the higher $arrR$ will cause a leaf node to be accessed by more update operations. Therefore, a larger T_{clean} is needed. On the other hand, supposed that $arrR$ is fixed, the larger h will lead to less update operations over a leaf node. Thus, a smaller T_{clean} can be used. Therefore, T_{clean} is defined as:

$$T_{clean} = \left[\left(\beta \times \frac{arrR}{MAX_{arrR}} + (1-\beta) \times (1 - \frac{h}{MAX_h})\right) \times C\right] + r. \tag{3}$$

Where, MAX_{arrR} is the maximum arrival rate in the system and MAX_h is the maximum height of MF-R^t-tree. β ($\in$ [0, 1]) is a parameter used to balance arrival rate and the

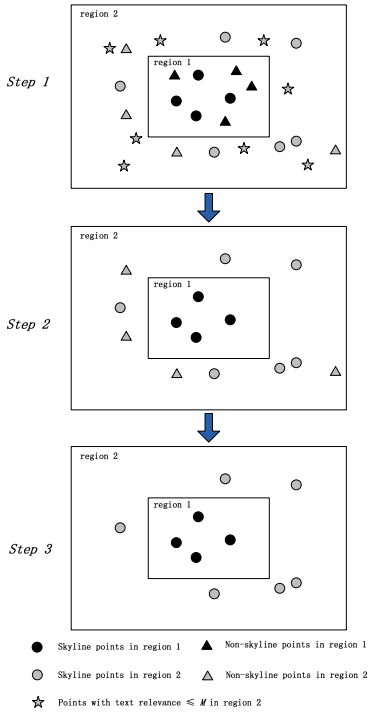


Fig. 3. The skyline processing method.

height of indexing tree. C means the capacity of one leaf node. To ensure the update performance, the value of T_{clean} is at least r that is a positive integer (≥ 2), when $arrR \rightarrow 0 \wedge h \rightarrow MAX_h$.

3.2.3 The Skyline Processing Method

Based on the proposed MF-R^t-tree, we propose a spatial-keyword skyline processing method by combining the skyline operation for static geo-textual data proposed in [7] and a skyline computation method over a sliding window proposed in [12]. Given a query q and a set of streaming data tuples DT whose positions fall in $q.r$, the skyline processing method is illustrated in Fig. 3. The skyline computing procedure follows three steps:

Step 1: The space of q (i.e., $q.r$) is partitioned into two regions, that is, the inner region denoted as **region1** and exterior one denoted as **region2**, where **region1** $\cup$ **region2** = $q.r$ $\wedge$ **region1** $\cap$ **region2** = $\emptyset$. To achieve the partition of $q.r$, let d be half the length of the diagonal line of **region1**, and c be the centroid of the **region1**. **region1** can be drawn by setting its c as $q.r.c$ (see Definition 3 for $q.r.c$) and its d as $\gamma \times q.r.d$ (see Definition 3 for $q.r.d$), where the value of γ should be around 0.2 according to the description in [7]. Thus, **region2** is the difference part between $q.r$ and **region1**. Finally, the data tuples in DT are divided into the two regions.

Step 2: A set of skyline points are first obtained, denoted as S1, labelled by black circles in **region1**. The maximum text relevance among S1 is referred to M, and all the points are then removed with their text relevances $\leq M$ in **region2**, labelled by gray pentagrams, because it is not possible for these points to become skyline points.

Step 3: Finally, the skyline points are gained, denoted as S2, labelled by gray circles, among the rest of the points in **region2**. The final skyline points = S1 $\cup$ S2.

However, the *block nested loops* (BNL) algorithm for static data was used for skyline computation of S1 and S2 in [7]. To suit the streaming data environment, we proposed a skyline processing algorithm over streaming data called $eager^*$ based on the $eager$ algorithm proposed in [12], instead of BNL algorithm. In [12], the eager algorithm has two main behavior features: 1) minimizing the memory consumption by keeping only tuples that are or may become part of the skyline in the future, and 2) reducing the cost of the maintenance of the skyline result by performing addition work in the preprocessing procedure (i.e, establishing and maintaining an event list). Concrete speaking, the $eager$ algorithm depends on the following concept:

Definition 9 ($skyline\ influence\ time$)**.** *Given a steam data tuple p over a sliding window W, the skyline influence time of p, denoted as SIT_P, is the expiring time of the youngest data tuple r who dominates $p \in W$. The data tuple r is called the critical dominator of p, denoted as CD_p^r.*

The $eager$ algorithm maintains an event list EL. Two types of events are supported:

Event 1: Skytime event, denoted as $skytime(p,t)$, reflects that the data tuple p will enter the skyline at time t, where time t is determined by the following formula

$$t = \begin{cases} SIT_P & if(CD_p^r\ exists) \\ Current\ time & otherwise \end{cases}. \quad (4)$$

Event 2: Expire event, denoted as $expire(p,t)$, reflects the exit of the data tuple p from the skyline at time t, where $t = p.t_e$ (i.e., the expiration time of p).

Nevertheless, the $eager$ algorithm ignores the dynamic nature of the data's properties. In our environment, one data tuple records the two properties (i.e., geo-location and textual string) of a geo-textual object. Because the values of the two properties may change over time, streaming data tuples in W may be updated before they expires. The updates may lead to a change of skyline results. Therefore, we enhance the $eager$ algorithm by appending a new type of event called the update event. The update event is presented as follows:

Event 3: Update event, denoted as $update(p,t_u)$, reflects that the data tuple p is updated at time t_u. Because an update can be treated as a combination of a separate deletion and a separate insertion, we first triggers an event $expire(p,t_u)$, where t_u = the update time of p, and then treat the update point of p, defined by p', as a new incoming point.

We refer to the enhanced $eager$ as $eager^*$, which is shown in algorithm 1. The algorithm first inputs all points in W into a queue Q (line 4), and then respectively initializes a

MF-R^t-tree T (line 6) and a skyline result set (line 7) through inserting a point $CurP$ from the front of Q. Furthermore, an event list EL is initialized with an $expire$ event concerning $CurP$ (line 8). After initialization, the procedure of checking EL does not stop until $EL = \emptyset$ (lines 9-26). During this checking procedure, the front point of Q is continuously popped. Each popped point is used to prune all points dominated by the popped point from the set sky (lines 10-11), denoted as F. Then, we delete the points in F from the result set sky (line 12) and index tree T (line 13), respectively. The events related to the points in F are removed as well (line 14). Then, we insert a skytime event (line 17) or an expire event (line 20) into EL. Subsequently, we deal with an update event (lines 22-23), expire event (lines 24), or skytime event (lines 25) when these events happen.

Algorithm 1. The $eager^*$ Algorithm

```
1  ComputingSyline_Procedure(W)
   /* Initialization */
2  Initialize a skyline result set sky= ∅
3  Initialize an event list EL = ∅
4  Initialize a queue Q ← W
5  CurP ← Q.pop()
6  Initialize a MF-R^t-tree T by inserting CurP
7  sky ← CurP
8  Add an event expire(CurP, CurP.t_e) into EL
/* Checking the event list */
9  while EL ≠ ∅ do
10   CurP ← Q.pop()
     /* Pruning Phase */
11   F ← points ≻ CurP in sky with T
12   sky ← sky/F
13   Delete the points in F from T
14   Clear events related to points in F from EL
     /* Insertion Phase */
15   Insert CurP into T
     /* Search Phase */
16   if CD^r_CurP ≠ ∅ then
17     Insert an event skytime(CurP, SIT_CurP) into EL
18   end
19   else
20     Insert an event expire(Curp, CurP.t_e) into EL
21   end
     /* Process update events */
22   if a point p is updated in W then Insert an event expire(p,
     p.t_u)
23   append p' into W and Q.push(p')
     /* Process expire events */
24   if An event expire(p, t) happens then Delete p from Q, W,
     sky and T, respectively
     /* Process skytime events */
25   if An event skytime(p, t) happens then Add p into sky
26 end
27 return sky
```

4 Reducing the Framework's Communication Cost

We first describe the basic idea of our method, and then the two optimization schemes are proposed based on spatial and textual signatures, respectively.

TABLE 1
Summary of Four Distribution Schemes in [6] for our Setting

Distribution scheme	Replication ratio	Communication cost
Hashing-based	1	N_{qb}
Location-based	δ	1
Keyword-based	$\mid m.\psi \mid$	$\min(\mid m.\psi \mid, N_{qb})$
Prefix-based	$\leq \mid m.\psi \mid$	$\min(\mid m.\psi \mid, N_{qb})$

4.1 The Basic Idea

Like DSkype [6], our framework suffers from the high communication cost between distribution bolts and query bolts as the number of query bolts increases. DSkype alleviates this problem by using four distribution methods, namely hashing-based, location-based, keyword-based, and prefix-based methods. However, these methods are designed for top-k queries rather than skyline queries. Thus, there exist differences between DSkype and our setting. To explain the differences, we first present an observation:

Observation 1. *Given a query q and a tuple dt, dt cannot become one skyline member related to q if $dt.l$ does not fall in $q.r$ or $\mid q.w \cap dt.w \mid = 0$.*

Proof of **Observation1 :** *According to Definition 5, the spatial relevance between q and dt, $R_S(q, dt) = 0$, if $dt.l$ is not in $q.r$. On the other hand, the textual relevance between q and dt, $R_T(q, dt) = 0$, if $\mid q.w \cap dt.w \mid = 0$, based on Definition 4. Thus, dt cannot become a dominator w.r.t q if $R_S(q, dt) = 0$ or $R_T(q, dt) = 0$, according to Definition 6. Therefore, dt cannot become one skyline member related to q.*

Given that N_{qb} refers to the number of query bolts, $\mid m.\psi \mid$ is the number of keywords in a geo-textual streaming data tuple, and δ is the average number of query bolts whose regions overlap one query. Table 1 summarizes the four schemes in Dskype [6] for our setting. For the location-based mechanism, a streaming data tuple in our setting cannot become one skyline if its geo-location does not fall in the region of any query according to Observation 1. Thus, a data tuple needs to be forward to only one query bolt whose spatial region contains the spatial region of the tuple. Therefore, the average $communication\ cost$ of each message is one rather than N_{qb} (see Table 2 in [6]) when the location-based mechanism is applied for our environment. In addition, the $replication\ ratio$ of subscriptions is δ rather than one (see Table 2 in [6]), because the range region of a query may overlap the regions of multiple query bolts.

Although, according to Table 1, the location-based scheme can be optimal in terms of the communication cost for our setting (i.e., the communication cost is one for each incoming message), the communication cost is still an issue in the streaming data environment, due to the continuous forwarding of messages from distribution bolts to query bolts. In fact, it is evident that some incoming tuples over distributing bolts cannot become skyline candidates according to Observation 1. We can avoid forwarding such tuples to query bolts to reduce the communication cost.

However, a top-k spatial-keyword query cannot meet similar results as Observation 1. To explain this case, we recall the definition of top-k query in DSkype [6].

TABLE 2
Dataset

Set name	# of objects	Vocabulary	Average # of keywords in objects
TWEETS	12.7M	1.7M	9

Definition 10 (**atop** − **kquery**). *Is defined as $q' = (w, l, k, \alpha)$, where $q'.w$ is a set of keywords, $q'.k$ is the maximum number of streaming data tuples that q' is willing to receive, and $q'.\alpha$ is the preference parameter used in score function.*

To measure the relevance between a query q' and a streaming data tuple dt, a score function is defined as

$$Score(q', dt) = q'.\alpha \times SSim(q'.l, dt.l) + (1 - q'.\alpha) \times TSim(q'.w, dt.w). \quad (5)$$

In formula (5), $SSim(q'.l, dt.l)$ is spatial proximity and is computed as $1 - \frac{eDist(q'.l, dt.l)}{MaxDist}$, where $eDist(q'.l, dt.l)$ is the euclidean distance and $MaxDist$ is maximum distance in the space. $TSim(q'.w, dt.w)$ is textual proximity and computed by the cosine similarity.

According to Definition 10 and formula (5), a streaming tuple dt may still become a top-k candidate with respect to q' even although $SSim(q'.l, dt.l)$ or $TSim(q'.w, dt.w)$ is a lower value, since the score function is based on the combination of $SSim(q'.l, dt.l)$ and $TSim(q'.w, dt.w)$. Therefore, DSkype for top-k queries cannot in advance filter out some incoming tuples over distributing bolts like our framework for skyline queries based on Observation 1.

Next, we propose a novel approach for optimizing the communication overheads of our framework based on Observation 1. The main idea of our method is that it individually employs spatial and textual signatures to compactly represent the spatial and textual information of queries over each query bolt. Then, these spatial and textual signatures are deployed on distribution bolts. Thus, one distribution bolt can use the two types of signatures to prune incoming tuples that cannot become skyline candidates to reduce unnecessary forwarding of messages to query bolts. The location-based scheme combined with spatial and textual signatures enables the achievement of a much lower communication cost than the original location-based scheme. Fig. 4 illustrates our approach. In this example, there are a distribution bolt and a query bolt. A query set $\{q_1, q_2\}$ has been deployed in advance on the query bolt using the location-based distribution scheme. Then, a spatial signature with spatial cuckoo filter (see Section 4.2) and a textual signature with min-wise hashing (see Section 4.3) are created over $\{q_1, q_2\}$. Both signatures are sent back to the distribution bolt. Subsequently, we suppose that the distribution bolt receives two tuples t_1 and t_2 in turn from tuple spouts. According to Fig. 4a, both locations of t_1 and t_2 fall in the spatial region of the query bolt U_s. Thus, both t_1 and t_2 should have been forwarded to the query bolt. But we can avoid these forwards with signatures. As Fig. 4b shows, we can first prune t_1 with the spatial signature, because the spatial signature compactly represents the region information of q_1 and q_2 and neither q_1 or q_2 contains t_1 in space,

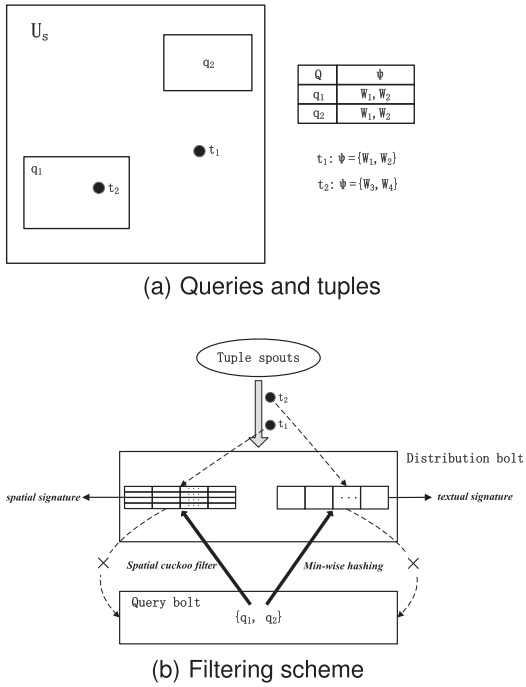


Fig. 4. An example of the proposed method for optimizing communication costs.

even though the keywords of t_1 match those of q_1 and q_2 (see Fig. 4a). On the other hand, although q_1 contains t_2 in space (see Fig. 4a), we are still able to prevent the forwarding of t_2 to the query bolt, because the textual signature can detect that the set of keywords of t_2 (i.e, $\{W_3, W_4\}$) has no intersection with that of q_1 (i.e, $\{W_1, W_2\}$).

More generally, there are a distribution bolt (denoted as D_Bolt) and m query bolts (denoted as $Q\_Bolt_1, \ldots, Q\_Bolt_m$). Each query bolt Q_Bolt_i periodically sends a <spatial signature, textual signature> pair (denoted as $<ss_i, ts_i>$) to D_Bolt according to its local query workload. Thus, the D_Bolt locally holds m $<ss_i, ts_i>$ pairs. Let tup be an incoming tuple, the optimization scheme is shown in algorithm 2. The D_Bolt first checks whether the location of an incoming tuple can hit anyone in m spatial signatures (line 2). If it hits a spatial signature ss_i, then we further observe whether the set of keywords of the tuple also hit the textual signature ts_i (line 3). If yes, the tuple is forwarded to the Q_Bolt_i (line 4). For other cases, the tuple is discarded.

Algorithm 2. The Optimization Scheme of Communication

```
1  CommOptimization_Procedure(tup)
   // Check the tuple with spatial signatures
2  if ∃ ss_i where tup.l hits ss_i then
   // Check the tuple with the textual signature
3    if tup.w hits ts_i then
4      forward tup to the Q_Bolt_i and return
5    end
6    else
7      discard tup and return
8    end
9  end
10 discard tup and return
```

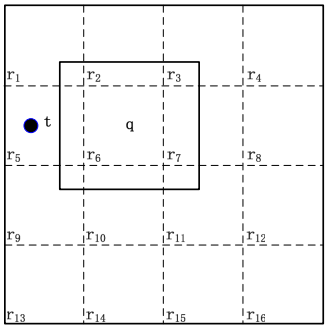


Fig. 5. An example of spatial representation.

In the following two subsections, we will provide the details of generating spatial and textual signatures and filtering streaming data tuples with them.

4.2 The Optimization Scheme Based on Spatial Signatures

A Bloom filter (BF) is a data structure that represents a set of elements in a space-efficient manner [41]. The BF can support membership queries on the originating set without knowledge of the set itself. The BF always determines positively if an element is in the set, whereas it generally determines elements outside the set negatively with a probabilistic false positive error [41]. The BF is an effective signature approach for compressing a set and querying membership in a set. Nevertheless, the original BF does not consider the spatial objects. Our setting requires a BF that can represent a set of spatial objects[3] from a set of queries. Furthermore, the BF can be utilized to determine whether the location point in a tuple is within the spatial region of any query in the query set. In [42], a spatial Bloom filter (SBF) was proposed for spatial objects. However, the SBF is based on the BF, so it cannot support deletion operations. As a result, the SBF is not suitable for dynamic updates of queries in our setting. In Section 3.2.2, we employed the cuckoo filter [10] to index streaming data tuples due to its advantages of taking only ≤ 1 time space cost than a standard Bloom filter and supporting deletion. Therefore, we proposed a spatial cuckoo filter (SCF) for generating and maintaining spatial signatures. We obtain the SCF by replacing the Bloom filter in the SBF with a cuckoo filter.

To enable SCFs to store location information, like the SBF, we first divide the entire 2D space into a set of distinct regions denoted as ε. The dimension of such regions can be set to an arbitrarily small size so that there is no loss in terms of location precision. Thus, a spatial object (i.e., a rectangle) concerning one query can be represented as a subset of ε, denoted as $\triangle$, where each region in the subset $\triangle$ overlaps the rectangle. Meanwhile, the geo-location of one tuple can be identified with the region containing it. Fig. 5 illustrates the set-based spatial representation of a query and a tuple. In Fig. 5, the universe space consists of 16 regions (i.e., $\varepsilon = \{r_i \mid 1 \leq i \leq 16\}$). Thus, for the query q, the corresponding $\triangle = \{r_1, r_2, r_3, r_5, r_6, r_7\}$, while the tuple t is identified by the region r_5.

3. Each spatial object in this paper is a 2D rectangle.

Once a rectangle and a geo-location can be identified as elements of a subset of ε, a SCF can be utilized to encode the spatial information. Formally, for the region set $o.\triangle = \{r_1,...,r_n\}$ of a spatial object o, a spatial cuckoo filter $SC(o.\triangle)$ is a cuckoo hash table that is denoted as HT for the spatial object o, and it is defined as

$$SC(o.\Delta) = HT = \bigcup\nolimits_{r \in o.\Delta, h \in H} < h(r), fringer(r) > . \tag{6}$$

In formula (6), $H = \{h_1, h_2\}$ is a set of two partial-key cuckoo hashing functions. The first hashing function $h_1 = \text{hash}(r)$, where hash is a uniform hashing function. h_1 takes the binary string of the identifier of one region r as input to output the index of one bucket in HT. Moreover, $h_2 = h_1 \oplus \text{hash}(finger(r))$, where $\oplus$ is the XOR operation and $finger(r)$ is the fingerprint of r. For h_1 and h_2, the $finger(r)$ is inserted into bucket[h_1] and bucket[h_2], respectively.

Note that in [42], the spatial representation of a spatial object is identified as a set of areas of interest $\{\triangle_1,...,\triangle_s\}$ with different priorities. However, in our case, we are concerned only about whether one tuple falls in one area of interest. Therefore, the spatial representation of a spatial object by an area of interest $\triangle$.

Suppose that r_t is one region ($\in \varepsilon$) that contains the geo-location of an incoming tuple and $f(r_t)$ is the fingerprint of r_t. Having constructed a SCF for one region set $\triangle$, denoted as HT, we can determine whether $r_t \in \triangle$ by formula (7).

$$\begin{array}{l} i = h_1(r_t),\ j = h_2(i, f(r_t)) \\ HT.bucket[i]\ has\ f(r_t)\ or\ HT.bucket[j]\ has\ f(r_t) \end{array}. \tag{7}$$

There exist false positive errors when applying the SCF for spatial pruning. The errors arise from two aspects. The first aspect is that a tuple may be falsely verified to be in the query range with the SCF, because its region is a member of the region set of the query. The problem is illustrated in Fig. 5, where the tuple t out of query q is verified to hit q due to $r_5 \in \varepsilon$ corresponding q when the SCF is used. The other aspect is the potential collision of hashes. Therefore, the false positive probability denoted as p is computed as

$$p = p_s + p_h. \tag{8}$$

where, p_s and p_h are the false positive probabilities from the above-mentioned first and second aspects, respectively. $p_s \approx \frac{num_{border}}{|\varepsilon|}$, where num_{border} is the number of border regions ($\in o.\triangle$) that intersect the region that contains the geo-location of a data tuple o and $|\varepsilon|$ is the number of regions in ε. The value of p_h can be configured according to [10].

4.3 The Optimization Scheme Based on Textual Signatures

According to formula (1), the text relevance of a query q and a data tuple t can be determined by the set similarity between $q.w$ and $t.w$. Thus, assuming that a set of queries $Q = \{q_1,...,q_n\}$ on one query bolt and one incoming tuple t arrives at one distributing bolt, we can copy the keyword sets of Q from the query bolt to the distribution bolt and then check whether the formula (9) holds.

$$\forall q \in Q, R_T(q,t) = \frac{|q.w \cap t.w|}{|q.w \cup t.w|} = 0. \quad (9)$$

If it holds, the tuple t can be pruned by the distribution bolt. However, transferring all sets of keywords is expensive in terms of both transmission and storage overheads. Therefore, we provide a space-saving solution that utilizes one $min-wise$ signature to compactly represent a set of keywords. These signatures are deployed on the distribution bolt instead of sets of keywords. As a result, both transmission and storage overheads can be significantly reduced. Moreover, we can estimate the similarity of two sets with a high accuracy to filter tuples as well, through comparing two $min-wise$ signatures.

Concretely, according to [43], given a family of $min-wise$ independent permutations F, for a set X and any element $x \in X$, when π is chosen at random from F, the following formula holds

$$\Pr(\min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}. \quad (10)$$

where $\pi(X)$ is a permutation of X and $\pi(x)$ is the location value of x in the resulted permutation, and $\min\{\pi(X)\} = \min\{\pi(x)|x \in X\}$. With ℓ $min-wise$ independent permutations from F, the $min-wise$ signature of X is defined as

$$S(X) = \{\min\{\pi_1(X)\}, \min\{\pi_2(X)\}, \ldots, \min\{\pi_\ell(X)\}\}. \quad (11)$$

Note that the expensive preprocessing cost may be incurred when the number of independent permutations ℓ is large. Therefore, we employed the one-permutation min-wise hashing method proposed in [44] to generate signatures instead of ℓ permutations. The one-permutation method breaks the space evenly into ℓ bins, and stores the smallest nonzero in each bin. Thus, we can gain a speedup of ℓ times for generating a signature.

Assume that two keyword sets A, $B \subseteq U$ where U is the set of all keyword, Thus, the set resemblance of A and B defined as $\rho(A,B)$ can be estimated by the similarity of their $min-wise$ signatures $S(A)$ and $S(B)$ defined as $\hat{\rho}(S(A),S(B))$. This means that $\rho(A,B)$ can be computed by the following formula.

$$\rho(A,B) = \hat{\rho}(S(A),S(B)) = \frac{|\{i|\min\{\pi_i(A)\} = \min\{\pi_i(B)\}\}|}{\ell}. \quad (12)$$

Additionally, for k sets of keywords A_1,...,A_k, the $min-wise$ signature of union of A_1,...,A_k can be computed by combining the $min-wise$ signature of individual sets (see Formula (13)).

$$S(A_1 \cup ... \cup A_k)[i] = \min\{S(A_1)[i], \ldots, S(A_k)[i]\}. \quad (13)$$

Based on function 1 of query bolts, queries on each query bolt are indexed by a Quadtree with inverted files. Each leaf node maintains an inverted file that contains keywords of all queries in the leaf node. We can employ formula (11) to generate one min-wise signature for presenting a set of keywords in each leaf node. Thus, one query bolt can send m min-wise signatures, where m is the number of leaf nodes in its Quadtree, to the distribution bolt for textual filtering. However, the size of min-wise signatures increases linearly with the number of leaf nodes. The large number of min-wise signatures is an issue in terms of communication and storage costs. Therefore, we propose a method for generating constant-size signatures. Our method is based on the idea of repeatedly calling formula (13) to merge signatures in indexing tree nodes in a bottom-up manner until the number of signatures is reduced to one predefined threshold.

After generating a set of min-wise signatures V, for an incoming tuple t, we can decide to prune it if formula (14), where $\lambda \in [0, 1)$ is a threshold, holds.

$$\forall v \in V, \hat{\rho}(v, S(t.w)) \leq \lambda, where\ \lambda \in [0,1). \quad (14)$$

Note that in formula (14) we use λ as a threshold instead of zero, due to the estimating error of $\hat{\rho}(.)$. According to the description in [45], the upper bound of error of one min-wise independent permutation in the worst case, denoted as $UB(e)$.

$$UB(e) \leq O\left(\frac{1}{\sqrt{\ell}}\right). \quad (15)$$

where ℓ is the number of bins for the one-permutation min-wise hashing method. Therefore, we set $\lambda = \frac{1}{\sqrt{\ell}}$.

5 Performance Evaluation

First, we introduce the experimental setup, and then present the experimental results of evaluating the proposed index structure, the skyline computing method, the scheme of communication optimization, and the distributed skyline processing performance, respectively.

5.1 Experimental Setup

5.1.1 Dataset

We employed a real-life dataset TWEETS[4] [9]. The dataset contains 12 million tweets with geo-textual information from 2008 to 2011. The statistics of the dataset are summarized in Table 2.

5.1.2 Query Workload

For query workload, we generate spatial-keyword queries based on the TWEETS dataset. 5M geo-textual messages are randomly selected. For each selected object, η terms are randomly picked as query keywords, and η is a random number between 1 and 5. The query region is set to a rectangle, and the region size is uniformly chosen between 0.01% and 1% of the universe data space.

5.1.3 Experimental Environment

All the experiments are conducted on a cluster with six homogenous nodes (one nimubus and five supervisors). A single-node Zookeeper server is deployed for coordination between nimubus and supervisors. These six nodes are connected via a 1Gbps's Ethernet. Table 3 lists the major configurations of each node in the cluster. Each supervisor can run at most four workers at the same time, and each worker can run multiple spouts/bolts concurrently.

4. https://github.com/lt-cug/TWEETS-data-set

TABLE 3
Configurations of Each Node

Hardware	Feature
CPU	quad-core CPU (i7-7700, 3.6GHz)
Memory	32GB DDR4
Disc storage	1TB
Software	**Feature**
OS	Windows 10
Storm	1.2.1

5.2 Evaluating the Data Indexing Structure

In the following experiments, we measured MF-R^t-tree on a single node in the cluster, because each node needs to use its MF-R^t-tree for skyline computing. We investigated update performance, space cost, and query performance of MF-R^t-tree for geo-textual streaming data tuples. Like the configures about time-based sliding windows in [12], the sliding window length ranges from $|W_1|$ = 800, to $|W_2|$ = 1.6K and $|W_3|$ = 3.2K seconds, and the arrival rate is set to a low rate with arr1 = 10 tuples/second and a high rate with arr2 = 1000 tuples/second, respectively. The parameters of the cuckoo filter (shown in Table 4) follow the configuration in [10] to achieve the false positive probability $p_h \leq 1\%$. Additionally, the number of buckets m in one cuckoo filter depends on the number of different keywords in the incoming tuples in the sliding window W.

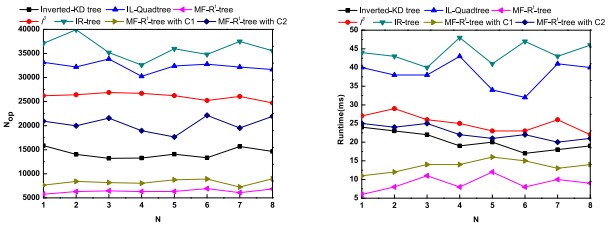
For comparison, the following indexing structures are constructed in main memory to index streaming data tuples as well.

1) *Inverted-KD tree* is a KD-tree whose leaves hold inverted files proposed in [7].
2) I^3 is an integrated inverted index [23], which adopts the quadtree to hierarchically partition the data space into keyword cells.
3) *IL-Quadtree* is an inverted linear quadtree proposed in [24].
4) *IR-tree* is a R-tree with inverted files proposed in [22].

5.2.1 Update Cost

First, we filled one sliding window with size = 3.2K seconds and constructed the Inverted-KD tree, I^3, IL-Quadtree, IR-tree, and MF-R^t-tree, respectively. Then, we continuously observed the operation number and runtime incurred by updating the these indexing structures at intervals of 20 seconds.

As shown in Fig. 6a, N on the x-axis represents the N-th 20 seconds, and N_{op} on the y-axis is the number of operations. The N_{op} of MF-R^t-tree is about 17.4%, 19.7%, 24.5%, and 44.7% of one of IR-tree, IL-Quadtree, I^3, and Inverted-KD tree, respectively. The reason for the operation saving of MF-R^t-tree lies in its ability to avoid numerous deletion operations with the update memo structure.



(a) The number of operations (b) Runtime

Fig. 6. The update cost of MF-R^t-tree under arr1.

Meanwhile, we also investigated the impact of our cleaning scheme on the update performance of MF-R^t-tree. For comparison, we implemented the original clean-upon-touch scheme in [11] denoted as C2. Our cleaning scheme is denoted as C1. For our cleaning scheme, according to the formula (3), the cleaning threshold T_{clean} is set with β = 0.8, MAX_{arrR} = 10K tuples/second, MAX_h = 8, C = 40, and r = 5. Fig. 6a shows that on average the MF-R^t-tree with C1 increases the operations by 29.7%, due to incurring the deletion operations compared with MF-R^t-tree. However, the update performance of MF-R^t-tree with C1 still outperforms other indexing structures. The N_{op} of MF-R^t-tree with C1 is about 22.5%, 25.6%, 31.8%, and 58.0% of one of IR-tree, IL-Quadtree, I^3, and Inverted-KD tree, respectively. On the other hand, as we can see, our cleaning scheme C1 can significantly reduce the operation number by 59.1 %, compared with C2. This result indicates C2 is not well suitable for the streaming data setting. Moreover, the experimental results for runtime evaluation shown in Fig. 6b are similar to the ones in Fig. 6a. Furthermore, in the case of high arrival rate shown in Fig. 7, we can see the similar experimental results as well.

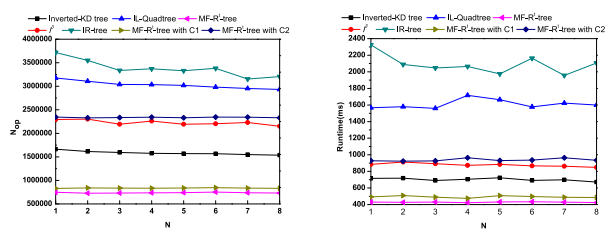
5.2.2 Space Cost

In the following experiment, we observed the space cost of MF-R^t-tree against Inverted-KD tree, I^3, IL-Quadtree, and IR-tree. For I^3, we followed the parameter setting in [23], one keyword is dense if the number of tuples containing the keyword exceeds 128.

As shown in Fig. 8a, MF-R^t-tree can on average save the space cost by 51.4%, 51.9%, and 55.0%, respectively, compared with Inverted-KD tree, IL-Quadtree, and IR-tree. The excellent performance gains result from MF-R^t-tree's use of the space-saving data structure cuckoo filters to store

TABLE 4
Parameters of Cuckoo Filter

Parameter	Value
Number of hashing functions k	2
Target false positive probability p_h	$\leq 1\%$
Number of entries per bucket b	4
Load factor α	95%
Fingerprint length in bits f	$\lceil \log_2(1/p_h) + \log_2(2b) \rceil = 10$



(a) The number of operations (b) Runtime

Fig. 7. The update cost of MF-R^t-tree under arr2.

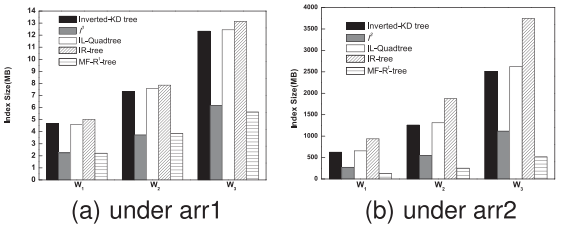


Fig. 8. The space cost of MF-R^t-tree.

keywords in indexing nodes. Meanwhile, MF-R^t-tree slightly outperforms I^3 by 2.1% space saving on average. I^3 is also space-efficient, because it only builds spatial indexing trees for dense keywords. In our experiments, dense keywords in I^3 are about 30.4% of all keywords on average. According to Fig. 8b, the space saving of MF-R^t-tree is more obvious under a high arrival rate because more keywords can be compactly represented with cuckoo filters.

However, the size of MF-R^t-tree will gradually expand over time as the number of old tuples increases over time. Thus, we investigated the impact of the proposed cleaning scheme on the size of MF-R^t-tree. We run the same cleaning procedure described in Fig. 6. Fig. 9 shows that the space cost of MF-R^t-tree exhibits a continuous increase over time, whereas that of MF-R^t-tree with the cleaning scheme can consistently maintain a stable level. The experimental results indicate that the cleaning scheme can effectively alleviate MF-R^t-tree's issue of space cost.

5.2.3 Query Cost

In this experiment, we randomly selected 1000 queries from the query workload. We measured the average runtime of per query over Inverted-KD tree, IR-tree, I^3, IL-Quadtree, MF-R^t-tree, and MF-R^t-tree after cleaning. As Fig. 10a illustrates, MF-R^t-tree can averagely accelerate 14.56×, 12.35×, 4.17×, individually, relative to Inverted-KD tree, IR-tree, and I^3. The performance gain of MF-R^t-tree mainly benefits from the query performance of $O(1)$ through cuckoo filters for textual pruning, while Inverted-KD tree cannot in advance terminate queries due to no textual pruning in non-leaf nodes, and IR-tree needs to match keywords in indexing nodes with time complexity of $O(N)$. Meanwhile, I^3 has to run multiple linear scans over lists, where each list holds tuples containing a non-dense keyword, if one query contains non-dense keywords. Furthermore, we observed that IL-Quadtree is 0.25 milliseconds faster than MF-R^t-tree on average per query. The reason for the good query performance of IL-Quadtree is that one indexing tree is established for each keyword.

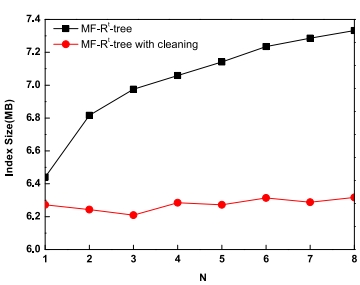


Fig. 9. The space cost of MF-R^t-tree with cleaning.

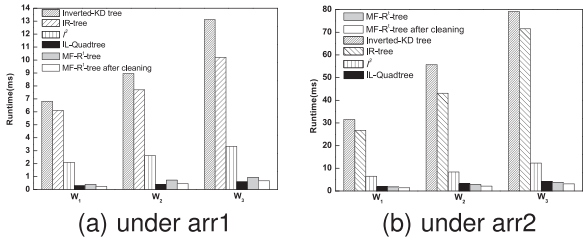


Fig. 10. The query cost of MF-R^t-tree.

Thus, IL-Quadtree can achieve a fast spatial pruning for one query keyword. However, IL-Quadtree has a much higher storage cost (see Fig. 8) and update cost (see Fig. 6) compared with MF-R^t-tree. Meanwhile, MF-R^t-tree after cleaning only is 0.09 milliseconds slower than IL-Quadtree on average per query. Moreover, in Fig. 10b, MF-R^t-tree and MF-R^t-tree after cleaning on average outperform IL-Quadtree by about 8% and 25%, respectively. These results indicate that our indexing structure has a query performance comparable to IL-Quadtree.

5.3 Evaluating the Skyline Computing Method

We inspect the processing time of the proposed skyline computing method over geo-textual streaming data on a single node. For convenience, we denote the eager-based and eager*-based spatial-keyword skyline computing method as, eager and eager* ,respectively, whereas the BNL-based spatial-keyword skyline computing method is termed BNL. We respectively observe the runtime speedups of eager and eager* compared with BNL. We randomly choose 10 queries to continuously observe the average runtime of three methods for skyline computing at intervals of 20 seconds. The window has been filled when the experiment starts, and the tuples in window also have been indexed by MF-R^t-tree and have been preprocessed for both eager and eager*. According to Fig. 11a, compared to BNL, eager can on average gain speedups of 43.22×, 63.9×, and 82.45× over time, for W_1, W_2, and W_3, respectively. This is because BNL has to repeatedly travel the tuples in the windows to recompute the skylines as new tuples arrive, whereas eager can continuously monitor the incoming tuples and maintain the skyline incrementally with pruning schemes. Furthermore, eager* can accelerate 1.235×, 1.248×, and 1.25× relative to eager for W_1, W_2, and W_3. Meanwhile, in the case of high arrival rate shown in Fig. 11b, we can see the similar experimental results. The performance gain lies in the ability of eager* to update the skyline results as early as possible when update events happen compared with eager.

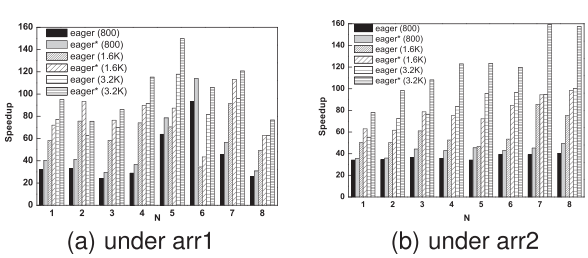
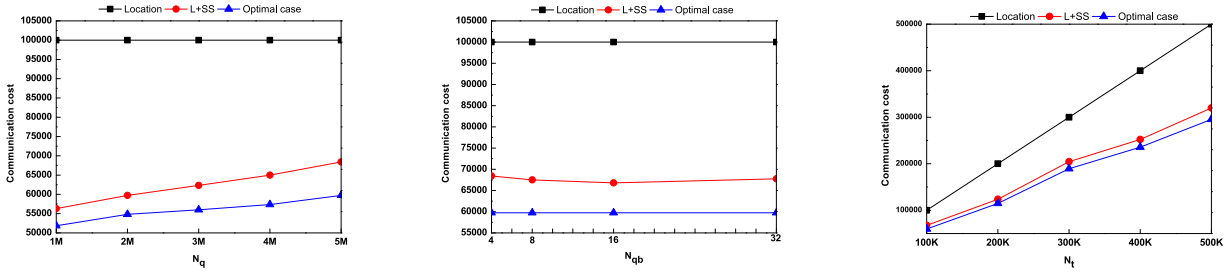


Fig. 11. The computing cost of spatial-keyword skylines.



(a) The number of queries (b) The number of query bolts (c) The number of tuples

Fig. 12. The communication cost with spatial signatures.

5.4 Evaluating the Communication Optimization Schemes

In this section, we evaluate the proposed communication optimization schemes for our framework in a cluster. Like DSkype [6], the communication cost between distribution bolts and query bolts is dominant for our distributed framework. Thus, the communication overhead between one distribution bolt and multiple query bolts is used to measure the performance of the proposed schemes. In the following experiments, we first evaluated the location-based distribution scheme with spatial signatures, and we then measured the location-based distribution scheme with both spatial and textual signatures.

5.4.1 Evaluating the Location-Based Distribution Scheme With Spatial Signatures

We observe the effects of spatial signatures on the communication cost under the different numbers of queries, query bolts, and inputting streaming data tuples, respectively.

In the first experiment, the number of query bolts is fixed to N_{qb} = 4 and the number of inputting data tuples N_t is set to 100K. The number of queries N_q ranges from 1M to 2M, 3M, 4M, and 5M.

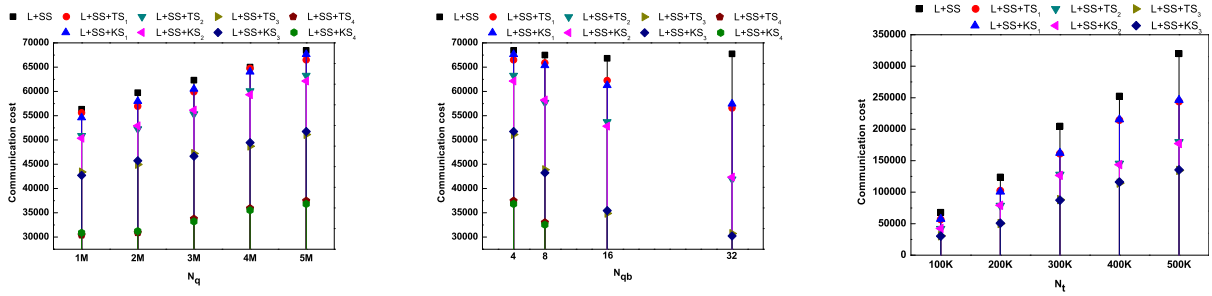
Based on the standard geographic coordination system (i.e., latitude and longitude), the minimum global space containing all spatial objects (i.e., rectangles) in the query set is first evenly partitioned into N_{qb} disjoint domains. Each query bolt owns one domain only. Then, each query from the query set is distributed to the query bolts whose domains overlap the query range. Furthermore, to generate spatial signatures, we adopted the means of spatial representation in [42]. The space is represented by the standard geographic coordination system. Thus, the whole space is divided into a set of distinct regions. The precision of the sides (in meters) of a region is fixed to 0.001^o. Therefore, the number of regions N_r = 2000 × 2000. Subsequently, each query bolt generates a set of regions based on the spatial objects from local queries and the spatial representation. Furthermore, each query bolt creates one spatial signature with SCF and sends it to the distribution bolt. The parameter configuration of the cuckoo filter follows Table 4. Meanwhile, the number of buckets in one cuckoo filter m is equal to the number of distinct regions owned by one query bolt (i.e., $\lceil N_r/N_{qb} \rceil$). Thus, the size of one SCF, denoted as $|SCF|$, can be computed as follows.

$$|SCF| = 1/\alpha \times b \times f \times \lceil N_r/N_{qb} \rceil. \quad (16)$$

The location-based distribution scheme has been used for comparison, because it can achieve the optimal communication cost (see Table 1) in our setting. Moreover, we also provided the optimal communication cost without the false positive. For convenience, the location-based distribution scheme, the location-based distribution scheme with spatial signatures, and the optimal communication cost without the false positive are denoted as $Location$, $L + SS$, $Optimal\ case$, respectively.

The experimental results presented in Fig. 12a show that the communication cost of $Location$ remains constant (i.e., 100K), because each incoming tuple must be forwarded to one query bolt whose domain contains its location. Compared with $Location$, $L + SS$ can reduce the communication cost by 43.7%, 40.3%, 37.7%, 35%, and 31.6% with N_q ranging from 1M to 5M, respectively. This demonstrates the pruning power of the spatial signatures. We also observed that the pruning power of the spatial signatures decreases as the size of N_q increases. This occurs because the union of all query regions increases with the size of queries increasing in a given global space, so that the possibility of filtering incoming tuples with spatial signatures decreases. On the other hand, $L + SS$ holds the false positive rate of 4.5%, 4.9%, 6.3%, 7.6%, and 8.7% for various sizes of queries (from 1M to 5M), respectively, compared to $Optimal\ case$. The reason for the increasing trend of false positive rate is partially that the probability value p_s in formula (8) will increase for more queries and partially that the possibility of the hashing collision of the cuckoo filter will increase when one SCF stores more query regions. Additionally, in terms of the storage cost, the distribution bolt holds four spatial signatures. According to formula (16), the size of one spatial signature ≈ 5MB. Thus, the total space cost of spatial signatures is 4 × 5MB ≈ 20MB.

In the second experiment, the number of queries is fixed to N_q = 5M and the size of tuples remains 100K. The number of query bolts N_{qb} ranges from 4 to 8, 16, and 32. According to the results presented in Fig. 12b, $L + SS$ can reduce the communication cost by 31.6%, 32.5%, 33.1%, and 32.3% for N_{qb} = 4, 8, 16, and 32, respectively, compared to $Location$. The experimental results indicate that the filtering effects have no obvious difference for various numbers of query bolts, because the query set is fixed so that SCFs generated from 4, 8, 16, and 32 query bolts have similar pruning power. The same filtering results for different N_{qb} under $Optimal\ case$ also demonstrate this point. Furthermore, the space costs incurred by SCFs on the distributed bolt are 20MB as well based on formula (16) and N_{qb}.



(a) The number of queries (b) The number of query bolts (c) The number of tuples

Fig. 13. The communication cost with spatial and textual signatures.

In the last experiment, the number of queries N_q is fixed to 5M and N_{qb} = 32, whereas the number of incoming tuples N_t increases from 100K to 200K, 300K, 400K, and 500K. Fig. 12c shows that the communication overhead of $L + SS$ is on average 64.99% of that of $Location$ for various numbers of tuples where the positive false rate $\leq$ 8%. The results indicate that the pruning power of spatial signatures can remain stable when the spatial signatures are fixed.

5.4.2 Evaluating the Location-Based Distribution Scheme With Both Spatial and Textual Signatures

We further investigate the filtering effect of the textual signatures over the tuples remaining after spatial pruning. In our setting, the indexing tree is the Quadtree with inverted files. Thus, the maximum number of textual signatures, denoted as MAX_T, in the distribution bolt is equal to $N_{qb} \times 4^{L-1}$, where L is the level number of the local indexing tree. Thus, in the case that MAX_T is set to 512, for N_{qb} = 4 and 8, each query bolt generates textual signatures based on at most the first four levels in the local indexing tree, whereas the first three levels are used for textual signatures for N_{qb} = 16 and 32. The number of bins of one-permutation min-wise hashing ℓ is set to 2,500 so that the estimation error is bounded by 2.0% (see formula (15)). We also provide the pruning results with spatial signatures and keyword sets for observing the estimating error of min-wise hashing. For convenience, the location-based distribution scheme with both spatial and textual signatures, where the textual signatures are based on the $i-th$ level of the indexing tree, is denoted as $L + SS + TS_i$. The location-based distribution scheme with the spatial signatures and keyword sets, where the keyword sets are based on the $i-th$ level of the indexing tree, is denoted as $L + SS + KS_i$.

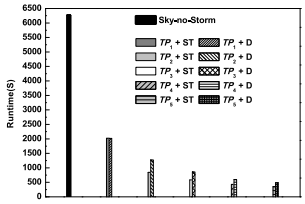
Corresponding to the case of the first experiment in Fig. 12a, for N_q = 1M, Fig. 13a shows that $L + SS + TS_i$ ($i \in$ [1,4]) can further reduce the communication cost of 0.5%, 5.4%, 12.5%, and 25.9%, respectively, compared with $L + SS$. The filtering effect of $L + SS + TS_1$ is not obvious, because one textual signature approximately represents all keywords in the indexing tree of one query bolt so that most tuples overlap the textual signature. Nevertheless, $L + SS + TS_2$ has a better pruning effect than $L + SS + TS_1$. The reason is straightforward: $L + SS + TS_2$ has 16 textual signatures, whereas $L + SS + TS_1$ has only four textual signatures. Moreover, one textual signature of $L + SS + TS_2$ represents a smaller subset of keywords than the one of $L + SS + TS_1$. Thus, $L + SS + TS_2$ is more possible to filter the no-match tuples than $L + SS + TS_1$. The reason is the same for the case in which $L + SS + TS_i$ is more efficient than $L + SS + TS_{i-1}$, where i =3, 4. The experiment results are similar for N_q = 2M, 3M, 4M, and 5M. Additionally, compared to $L + SS + KS_i$, the estimating error of textual signatures can be bounded by 1.83%, because we choose the larger value of ℓ (see formula (15)).

Fig. 13b shows that the pruning powers of $L + SS + TS_i$ (i = 1, 2, 3, 4) rise as N_{qb} increases. For instance, $L + SS + TS_3$ can further reduce the communication cost of 17.3%, 23.6%, 31.9%, and 37%, for N_{qb} = 4, 8, 16, and 32, respectively. The reason for these experimental results is that more textual signatures can be generated as N_{qb} increases and each textual signature can represent a smaller-size keyword set. In terms of storage costs of textual signatures, each textual signature consists of a pair of coordinates (representing a subregion of the local indexing tree) and $\ell = 2500$ location values of one permutation. Here, one coordinate occupies 2 bytes while one location value occupies 8 bytes. Thus, the size of one textual signature $\approx$ 19.54KB. The maximum storage cost of textual signatures is 10MB, because the maximum number of textual signatures is 512 in our experiments. Furthermore, Fig. 13c illustrates that textual signatures can maintain a stable pruning effect on various number of tuples. For example, $L + SS + TS_2$ can further filter 25.8%, 22.6%, 25.5%, 26.7%, and 28%, respectively, in terms of the communication cost. The reason for these results is similar to that for the experimental results presented in Fig. 12c.

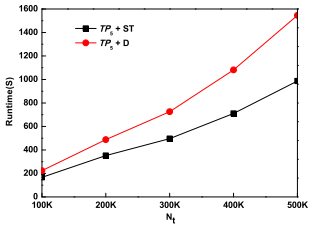
5.5 Evaluating the Distributed Skyline Query Processing

We conduct experiments to verify the performance of skyline query processing of the proposed distributed framework under various numbers of supervisors. We established the proposed distributed framework in a cluster described in Section 5.1.3 "Experimental environment". Concretely speaking, let N_{sup}(= 1, 2, 3, 4, or 5) be the number of supervisors used for the framework, we constructed five Storm topologies (denoted as $TP_{N_{sup}}$) on the top of N_{sup}(= 1, 2, 3, 4, or 5) supervisors, respectively. Each supervisor runs one worker since one supervisor only has one quad-core CPU in our cluster. Based on the configuration for Storm topology in [6], one $TP_{N_{sup}}$ consists of one tuple spout, one query spout, one distribution bolt, 4 $\times$ N_{sup} query bolts[5], three tuple bolts, and one aggregation bolt. We

5. The number of query bolts is set to 4 $\times$ N_{sup} because the CPU in each supervisor holds four cores.



(a) Various Storm topologies



(b) Various numbers of tuples

Fig. 14. Evaluating the distributed skyline query processing.

measured the runtime of each Storm topology with randomly selected 2M queries and 200K incoming tuples. The arrival rate is set to 5000 tuples/second and the size of sliding window is 2K seconds. For comparison, we also measured our skyline computing method (see Section 3.2.3) without Storm on a single node, denoted as Sky-no-Storm. Meanwhile, to distinguish skyline queries over our framework and DSkype, we implemented skyline queries over DSkype (i.e., the location-based distribution scheme without spatial and textual signatures) as well. The skyline queries over our framework and DSkype under five Storm topologies are denoted as TP_i+ST and TP_i+D where i $\in$ [1, 5], respectively. For fairness, we also built query index and data index for both Sky-no-Storm and TP_i+D.

Initially, the whole query set is evenly distributed into query bolts. To achieve the workload balance, we first index all queries with a Quadtree. Then, each query bolt is responsible for the queries in $[N_{leaf}/N_{qb}]$ neighbor leaves of indexing tree, where N_{leaf} be the number of leaves of Quadtree and N_{qb} be the number of query bolts. After that, N_{qb} <spatial signature, textual signatures> pairs are generated based on local queries over query bolts, and are deployed on the distribution bolt. Thus, the distribution bolt will forward the incoming tuples to the corresponding query bolts with spatial and textual signatures.

The experimental results in Fig. 14a show that our distributed framework can gain a speedup of 3.1×, 7.41×, 10.7×, 14.6×, and 17.8×, respectively for TP_1 + ST, TP_2 + ST, TP_3 + ST, TP_4 + ST, and TP_5 + ST compared with Sky-no-Storm. These performance gains result from having more query bolts (i.e., more CPU threads from workers) to process 2M skyline queries in parallel, as the number of supervisors (i.e, the number of workers) increases. Furthermore, TP_1 + ST and TP_1 + D have almost the same time overhead. The reason is that the Storm topology TP_1 only contains one supervisor, so that the data transmission between the distribution bolt and query bolts is a local transmission that does not pass through the network. Therefore, the benefits of TP_1 + ST in terms of network overhead are not reflected compared with TP_1 + D. Nevertheless, TP_i + ST can improve the query processing performance by 33.7%, 32.1%, 28.3%, and 27.8%, respectively, compared with TP_i + D as the number of supervisors i increases from 2, 3, 4, to 5. The reason for such experimental results is that our framework can employ both spatial and textual signatures to reduce the network transmission time of streaming data tuples from the distribution bolt to query bolts in a network connected distributed environment, compared with DSkype. Noted that, the performance gain of our framework decreases as the number of supervisors i increases because we fixed the number of incoming tuples to 200K in this experiment. Thus, the average number of tuples reaching one query bolt decreases when a Storm topology contains more supervisors. In another experiment, given a Storm topology TP_5 and 2M queries, we increase the number of incoming tuples N_t from 100K to 200K, 300K, 400K, and 500K. Fig. 14b shows that TP_5 + ST can improve the query processing performance by 25.5%, 27.8%, 31.6%, 34.3%, and 36.1%, respectively, compared with TP_5 + D. Obviously, the performance gain of our framework increases as the number of incoming tuples increases compared with DSkype when the number of supervisors is fixed, since the distribution bolt can prune more tuples by spatial and textual signatures as the number of incoming tuples increases.

6 Conclusion

In this paper, we proposed a distributed skyline query processing framework for large-scale spatial-keyword publish/subscribe systems. We also introduced an update-efficient and space-saving indexing structure for geo-textual streaming data. Moreover, an efficient computing method for continuous spatial-keyword skyline queries has been presented to optimize skyline computing. Finally, we provided a spatial and textual signature-based communication optimization method to support the scalability of the proposed distributed framework. The experimental results indicate that (1) MF-R^t-tree can significantly reduce update costs, while maintaining a nearly equal storage cost with I^3, and a query performance comparable to IL-Quadtree, (2) $eager^*$ can averagely accelerate 79.72 × faster than the method based on BNL, (3) the communication optimization method significantly reduces the communication cost of the proposed distributed framework, and (4) the distributed framework can efficiently support large-scale spatial-keyword skyline queries.

References

[1] J. Zhou *et al.*, "Thermal-aware correlated two-level scheduling of real-time tasks with reduced processor energy on heterogeneous MPSoCs," *J. Syst. Archit.*, vol. 82, pp. 1–11, Jan. 2018.

[2] Z. Chen, G. Cong, Z. Zhang, T. Z. Fu, and L. Chen, "Distributed publish/subscribe query processing on the spatio-textual data stream," in *Proc. Int. Conf. Data Eng.*, 2017, pp. 1097–1106.

[3] L. Chen, G. Cong, X. Cao, and K. Tan, "Temporal spatial-keyword top-k publish/subscribe," in *Proc. Int. Conf. Data Eng.*, 2016, pp. 255–266.

[4] H. Hu, Y. Liu, G. Li, J. Feng, and K. Tan, "A location-aware publish/subscribe framework for parameterized spatio-textual subscriptions," in *Proc. Int. Conf. Data Eng.*, 2015, pp. 711–722.

[5] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang, "Ap-tree: Efficiently support continuous spatial-keyword queries over stream," in *Proc. Int. Conf. Data Eng.*, 2015, pp. 1107–1118.

[6] X. Wang, W. Zhang, Y. Zhang, X. Lin, and Z. Huang, "Top-k spatial-keyword publish/subscribe over sliding window," *Clustering Valid. Very Large Databases J.*, vol. 26, pp. 301–326, 2017.

[7] J. Li, H. Wang, J. Li, and H. Gao, "Skyline for geo-textual data," *Geoinformatica*, vol. 20, pp. 453–469, 2016.
[8] Y. Teng, D. Liu, X. Liu, W. Zhao, H. Liu, and C. Fan, "Secure spatio-textual skyline queries on cloud platform," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl. Big Data Cloud Comput. Sustain. Comput. Commun. Soc. Comput. Netw.*, 2020, pp. 251–259.
[9] G. Li, Y. Wang, T. Wang, and J. Feng, "Location-aware publish/subscribe," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2013, pp. 802–810.
[10] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. Int. Conf. Emerg. Netw. Experiments Technol.*, 2014, pp. 75–87.
[11] Y. N. Silva, X. Xiong, and W. G. Aref, "The rum-tree: Supporting frequent updates in r-trees using memos," *Very Large Data Bases J.*, vol. 231, pp. 719–738, 2008.
[12] Y. Tao and D. Papadias, "Maintaining sliding window skylines on data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 3, pp. 377–391, Mar. 2006.
[13] T. D. Matteis, S. D. Girolamo, and G. Mencagli, "A multicore parallelization of continuous skyline queries on data streams," *Lecture Notes Comput. Sci.*, vol. 9233, pp. 402–413, 2015.
[14] X. Lin, Y. Yuan, W. Wang, and H. Lu, "Stabbing the sky: Efficient skyline computation over sliding windows," in *Proc. Int. Conf. Data Eng.*, 2005, pp. 502–513.
[15] M. Morse, J. Patel, and W. Grosky, "Efficient continuous skyline computation," in *Proc. Int. Conf. Data Eng.*, 2006, p. 108.
[16] Y. Xiao, X. Jiao, H. Wang, C.-H. Hsu, L. Liu, and W. Zheng, "Efficient continuous skyline query processing in wireless sensor networks," *Sensors*, vol. 19, no. 13, pp. 1–12, 2019.
[17] T. Sjayram, A. McGregor, S. Muthukrishan, and E. Vee, "Estimating statistical aggregates on probabilistic data streams," in *Proc. 26th ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, 2007, pp. 243–252.
[18] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu, "Probabilistic skyline operator over sliding windows," in *Proc. Int. Conf. Data Eng.*, 2009, pp. 1060–1071.
[19] X. Ding, X. Lian, L. Chen, and H. Jin, "Continuous monitoring of skylines over uncertain data streams," *Inf. Sci.*, vol. 184, pp. 196–214, 2012.
[20] L. Chen, G. Cong, and X. Cao, "An efficient query indexing mechanism for filtering geo-textual data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 749–760.
[21] L. Guo, J. Shao, H. H. Aung, and K.-L. Tan, "Efficient continuous top-k spatial keyword queries on road networks," *Geoinformatica*, vol. 19, pp. 29–60, 2015.
[22] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *Very Large Data Bases Endowment*, vol. 2, pp. 337–348, 2009.
[23] D. Zhang, K.-L. Tan, and A. K. H. Tung, "Scalable top-k spatial keyword search," in *Proc. Int. Conf. Extending Database Technol.*, 2013, pp. 359–370.
[24] C. Zhang, Y. Zhang, W. Zhang, and X. Lin, "Inverted linear quadtree: Efficient top k spatial keyword search," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 7, pp. 1706–1721, Jul. 2016.
[25] P. Zhang, H. Lin, B. Yao, and D. Lu, "Level-aware collective spatial keyword queries," *Inf. Sci.*, vol. 347, pp. 1706–1721, 2017.
[26] X. Zang, P. Hao, X. Gao, B. Yao, and G. Chen, "Qdr-tree: An efficient index scheme for complex spatial keyword query," *Lab. Natural Simulated Cogn.*, vol. 11029, pp. 390–404, 2018.
[27] Z. Qian, J. Xu, K. Zheng, P. Zhao, and X. Zhou, "Semantic-aware top-k spatial keyword queries," *World Wide Web*, vol. 21, pp. 573–594, 2018.
[28] Y. Dong, H. Chen, and H. Kitagawa, "Continuous search on dynamic spatial keyword objects," in *Proc. Int. Conf. Data Eng.*, 2019, pp. 1578–1581.
[29] H. Lu, Y. Zhou, and J. Haustad, "Efficient and scalable continuous skyline monitoring in two-tier streaming settings," *Inf. Syst.*, vol. 38, pp. 68–81, 2013.
[30] X. Li, Y. Wang, X. Li, and Y. Wang, "Parallelizing skyline queries over uncertain data streams with sliding window partitioning and grid index," *Knowl. Inf. Syst.*, vol. 41, pp. 277–309, 2014.
[31] Y. Zeng, Z. Yang, W. Zhang, and C. Li, "Application of processing technology based on skyline query in computer network," *Neural Comput. Appl.*, pp. 1–11, 2021.
[32] K. Mullesgaard, J. Pederseny, H. Lu, and Y. Zhou, "Efficient skyline computation in MapReduce," in *Proc. Int. Conf. Extending Database Technol.*, 2014, pp. 37–48.
[33] B. Zhang, S. Zhou, and J. Guan, "Adapting skyline computation to the MapReduce framework: Algorithms and experiments," *Lecture Notes Comput. Sci.*, vol. 6637, pp. 403–414, 2011.
[34] J. Zhang, X. Jiang, W. Ku, and X. Qin, "Efficient parallel skyline evaluation using MapReduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, pp. 1996–2009, 2016.
[35] Y. Park, J.-K. Min, and K. Shim, "Parallel computation of skyline and reverse skyline queries using MapReduce," in *Proc. Very Large Data Bases Endowment*, vol. 6, pp. 2002–2013, 2013.
[36] Y. Park, J.-K. Min, and K. Shim, "Efficient processing of skyline queries using MapReduce," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 5, pp. 1031–1044, May 2017.
[37] Y. TangandS. Chen, "Supporting continuous skyline queries in dynamically weighted road networks," *Math. Probl. Eng.*, vol. 2018, no. PT.11, pp. 1–14, 2018.
[38] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic, "Scalable distributed event detection for twitter," in *Proc. IEEE Int. Conf. Big Data*, 2013, pp. 543–549.
[39] Y. Hua, B. Xiao, and J. Wang, "Br-tree: A scalable prototype for supporting multiple queries of multidimensional data," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1585–1598, Dec. 2009.
[40] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
[41] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, 1970.
[42] L. Calderoni, P. Palmieri, and D. Maio, "Location privacy without mutual trust: The spatial bloom filter," *Comput. Commun.*, vol. 68, pp. 4–16, 2015.
[43] F. Li, B. Yao, M. Tang, and M. Hadjieleftheriou, "Spatial approximate string search," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 6, pp. 1394–1409, Jun. 2013.
[44] P. Li, A. Owen, and C. Zhang, "One permutation hashing," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 3113–3121.
[45] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, pp. 630–659, 2000.