Spatio-Temporal Keyword Query Processing Based on Key-Value Stores

Ruiyuan Li[1,2], Xiang He[1,2], Yingying Sun[1,2], Jun Jiang[3,2], You Shang[1,2], Guanyao Li[4], Chao Chen[1*]

[1]College of Computer Science, Chongqing University, Chongqing, China.
[2]Start Lab, Chongqing University, Chongqing, China.
[3]Chongqing City Vocational College, Chongqing, China.
[4]Guangzhou Urban Planning and Design Survey Research Institute, Guangzhou, China.

*Corresponding author(s). E-mail(s): cschaochen@cqu.edu.cn;
Contributing authors: ruiyuan.li@cqu.edu.cn; cquhx@cqu.edu.cn; Yingying.Sun@cqu.edu.cn; iris.jun.jiang@outlook.com; you.shang@stu.cqu.edu.cn; gyli@gzpi.com.cn;

Abstract

With the popularity of mobile devices and the development of location technology, there are increasing amount of text data with spatial and temporal tags generated. Querying with spatial, temporal, and keyword constraints on such data, known as spatio-temporal keyword query (STK query), is of great significance. However, most existing STK query solutions rely on tree-based indexes designed for stand-alone architectures, which struggle to scale for big data. Key-value stores, with the keys as their indexes, are designed for big data scenarios. On one hand, key-value stores can only support one-dimensional indexes initially, which makes them unsuitable for multi-dimensional STK queries. On the other hand, key-value stores put their indexes out of the memory, making it inevitable to trigger many unnecessary disk I/Os and slow down the query efficiency. To this end, based on key-value stores, we provide the first attempt by combining the in-memory index with on-disk index to efficiently support STK queries. Specifically, we design two-layer filters as the in-memory index, which enormously prunes unqualified spatio-temporal keyword combinations. An eviction policy is employed for the in-memory index, allowing it to support an infinite amount of data with limited memory usage. We deploy our solution on both HBase and Redis, conducting extensive experiments with two real and one synthetic datasets. The experimental results demonstrate that our solution achieves approximately

twice the query efficiency of the state-of-the-art key-value based solutions, and is much more scalable than the tree-based competitor.

Keywords: spatio-temporal data management, key-value database, spatio-temporal keyword query, bloom filter

1 Introduction

With the rapid proliferation of mobile devices and continuous advancements in Internet technology, users have shifted their information retrieval preferences from traditional personal computer platforms to mobile devices. This trend not only alters users' information access habits but also provides significant opportunities for social research. A notable example is the surge in social media platforms like Twitter [1] and Instagram [2] which contain blogs with spatial, temporal and keyword information (a.k.a., STK data). They not only enhance user engagement but also offer valuable data insights for businesses, politics, society and so on. Researchers can leverage the social media data in a specific time and spatial range to analyze public sentiment.

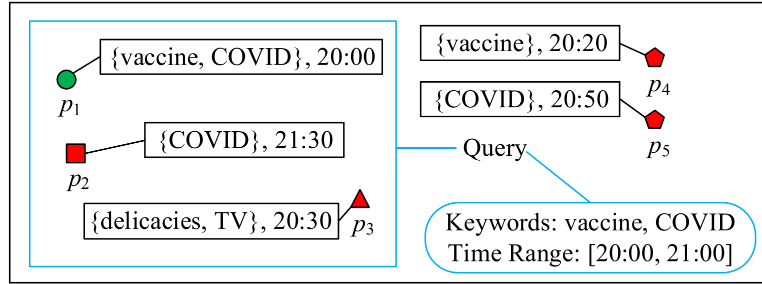


Fig. 1: An Example of STK Query.

Spatio-temporal keyword (STK) queries can meet such requirements by considering not only keyword matching but also spatial and temporal constraints. Fig. 1 illustrates an example of an STK query. Each tweet expresses the user's thought at a certain timestamp. For example, p_1 is posted at 20:00, containing two keywords 'vaccine' and 'COVID'. Suppose the workers in a community want to survey the latest attitudes of residents towards the coronavirus vaccine, based on which they can adjust their promotion strategies. They search for the tweets mentioned "vaccine" and/or "COVID" within the time range of 20:00-21:00 in the community. The pentagonal dots (i.e., p_4 and p_5), square dots (i.e., p_2), and triangular dots (i.e., p_3) represent the tweets that do not meet the spatial constraints, temporal constraints, and keyword constraints, respectively, so they will not be included in the result set. Finally, only p_1 is considered as a match for the query. Please note that STK query here handles

[1]https://twitter.com/
[2]https://www.instagram.com/

spatial and temporal constraints as range constraints, consistent with the definition proposed by Chen et al. [1].

To efficiently support STK queries, two problems need to be addressed. 1) **Big Data Problem**. In platforms like Twitter, where at least 500 million tweets are posted every day [2], or Yelp, which has 178 million active users every month [3], significant quantities of data with spatio-temporal tags are generated and exchanged. In this scenario, the sheer volume of data underscores the complexity of STK queries. 2) **Frequent Insertion Problem**. Spatio-temporal keyword data are generated constantly. These frequently generated data should be inserted into the system in a timely manner, enabling the latest data to be queried for real-time analysis. Once being inserted, the data in Twitter and Yelp would rarely be updated. To this end, this paper does not consider the modification of inserted data.

Most recent works on STK queries mainly focus on the temporal and keyword extensions of in-memory spatial trees [4–12], such as R-tree [13], quad tree [14] and so on. They work well when the data volume is small. However, as the data volume increases, the limited memory capacity often struggles to support the increasingly large tree-based structures. Additionally, these tree-based indexes may encounter troubles in dealing with frequent STK data insertion, since they need to adjust their tree structures frequently to ensure good performance. Further, some of the tree-based extensions [15, 16] can only support one keyword semantic (e.g., containing only one specified keyword in the results). To enable multiple keyword semantics with any boolean logical combinations, these tree-based indexes need to be tailored and modified; otherwise, they would fall into low efficiency.

Key-value stores, e.g., HBase [1], are designed for the management of big data with high throughput. Due to the relatively simple structure, key-value stores are easily scalable to distributed environments to support large volumes of data. However, researches on STK queries based on key-value stores are still scarce. The only work [1] proposes an STK query model based on HBase. It stores data in the same spatio-temporal region (e.g., data within a 1-hour timeframe in Beijing city) in the same row of HBase, where each record is stored in specific columns of the row. For each row, it maintains a Bloom Filter [17] to accelerate queries. However, it comes with the following shortcomings: 1) **Hotspot Issue**. It utilizes the time information as the prefix of the row key, leading to data concentration on a small number of cluster nodes as recently inserted data usually have close timestamps. This significant unbalance in load distribution can greatly affect the throughput. 2) **Additional I/O Overheads**. Storing all filters in HBase may result in additional network and disk I/O overheads when accessed by applications, severely impacting system performance of both insertion and querying. When inserting new data, it needs to update the Bloom Filter on the disks simultaneously. When performing a query, it also requires to first retrieve the Bloom Filter from the disks, and then further determine whether to access the stored data based on the checked result of Bloom Filter.

To address the aforementioned issues for higher efficiency while ensuring high scalability, we propose Key-Value store based Spatial-Temporal Keyword query framework (KV-STK), which combines the high-performance efficiency of memory and the massive storage capability of key-value stores. As far as we know, we are the first to

combine in-memory index with on-disk index based on key-value stores to efficiently support STK queries. KV-STK has several notable characteristics: 1) **Efficient**. We enhance the query efficiency by employing a two-layer index in memory, which prunes unqualified spatio-temporal keyword combinations enormously. Simultaneously, we improve the write efficiency through meticulously designed on-disk index (i.e., keys in key-value stores). 2) **Scalable**. We employ an eviction strategy for the in-memory index, ensuring the memory usage would not exceed a specified threshold. Moreover, all the data records are stored in scalable key-value stores. Therefore, in theory, KV-STK can support an infinite amount of data. 3) **High Throughput**. Compared to tree-based indexes, KV-STK does not require frequent structural adjustment, making it more suitable for scenarios with high-speed data insertions. Besides, we adopt the shard technique to distribute the inserted data across different nodes, thereby avoiding the possible hotspot issue. 4) **Various Keyword Semantics Support**. Through the in-memory index, KV-STK can perform pruning on various keyword semantics with any boolean logical combinations, such as containing all keywords in the query or just one of them.

Overall, the contributions of this paper are summarized as follows:

(1) We are the first to propose a hybrid index that combines in-memory index with on-disk index to answer STK queries based on key-value stores. The hybrid index effectively ensures querying and insertion efficiency.

(2) We design a framework for managing in-memory filters to support the acquisition and eviction of filters within a given memory threshold. The framework ensures excellent scalability of KV-STK.

(3) We organize the filters in memory and store the data in key-value stores, so KV-STK can support high-speed data insertion and query processing.

(4) We deploy our solution on both HBase and Redis, and conduct extensive experiments on two real and one synthetic datasets. The experimental results demonstrate that our solution achieves approximately twice the query efficiency of the state-of-the-art key-value based solutions. At the same time, compared with tree-based indexes, KV-STK has much better scalability.

The remainder of this paper is organized as follows. In Section 2, we summarize the related works. In Section 3, we give some necessary preliminaries. We present the system overview in Section 4, and detail the two main modules of Data Insertion and Query Processing in Section 5 and Section 6, respectively. The experimental results are shown in Section 7. Finally, we conclude this paper in Section 8.

2 Related Works

In this section, we review the related works from three aspects: 1) spatial keyword queries, 2) temporal keyword queries, and 3) spatio-temporal keyword queries.

Spatial Keyword Queries. To support spatial keyword queries, an intuitive idea for indexing spatial keyword objects is using two separate index structures, one for spatial information while another for keywords. Although this type of method enables the system to be flexible and efficient in system resources, it can incur relatively high

Table 1: Methods for Spatio-Temporal Keyword Queries.

Type	Scalability	Architecture	Efficiency	Throughput	Index	Works
Tree-Memory	Low	Stand-alone	Very Fast	Low	Tree-based	[4–11, 32–36]
Tree-Hybrid	Low	Stand-alone	Medium	Low	Quad-tree	[15, 16]
KV-Disk	High	Distributed	Slow	Low	KV+Filter	[1]
KV-Hybrid	High	Distributed	Fast	High	KV+Filter	Our Work

query latency. Therefore, researchers propose hybrid indexes [18–21] that combine R-tree [13], quad-tree [14] or grid-index [22] for spatial indexing with inverted files [23] for keyword indexing. For example, IR-Tree [21] extends R-tree with an inverted file, where each leaf node points to an inverted file containing the keywords of the objects stored in the node. IQ-Tree [24] merges quad-tree and inverted files to efficiently deal with spatial-keyword queries. The index based on grids [20] first partitions the space into grids, and then links each grid to an inverted list of keywords for efficient query processing. However, these sophisticated hybrid indexes, despite their advanced query processing capabilities, often face the challenges of increasing storage requirements and computational complexity. Learned indexes [25–27] for spatial keyword queries utilize the historical query distribution to further optimize the index structures for future query processing [28, 29]. For instance, LIST [27] uses machine learning model to learn spatial and textual relevance. WISK [26] proposes a query-aware learned index considering spatial and textual attributes simultaneously. LSTI [25] enhances query efficiency by employing a random forest regression model. However, a common drawback of these learned indexes is the increasing complexity in index construction and maintenance, especially with dynamic data updates.

Temporal Keyword Queries. The deluge of quickly outdated information like websites, newspapers, and blogs makes temporal relevance as critical as content, necessitating the use of temporal constraints alongside keyword constraints. So researches for temporal keyword queries emerge. For instance, the work [30] proposes a partition selection strategy to manage a temporally partitioned inverted index, significantly reducing redundant data access for long-range temporal queries. Xia et al. [31] utilize a temporal inverted index optimized with a two-tier structure to improve query performance. However, these works would increase the complexity in index maintenance and storage overheads. Besides, they only consider textual and temporal information, but ignore the spatial proximity of objects. Therefore, they cannot handle spatio-temporal keyword queries directly.

Spatio-Temporal Keyword Queries. With the surge of spatio-temporal keyword data, the spatio-temporal keyword queries have attracted widespread attention. Using separate indexes for spatial, temporal, and textual information incurs the overhead of storing multiple indexes and concatenating their results. Therefore, existing studies usually adopt composite indexes. The previous works on this problem can be divided into four types: 1) Tree-Memory Methods, 2) Tree-Hybrid Methods, 3) KV-Disk Methods and 4) KV-Hybrid Methods, as shown in Table 1.

The first type of solutions is called Tree-Memory Methods, which means that they are based on tree indexes, and the entire tree structures are accommodated in memory. As shown in the first row of Table 1, they usually have stand-alone architectures [4–11, 32–36]. For example, based on G-Tree, Zhao et al. [4] design TG index that can prune the search space with both spatio-temporal and textual information simultaneously. STILT [37] uses a binary trie-based index interleaving text, location, and time information in a space-efficient manner. When dealing with small datasets, memory-based indexes can significantly accelerate query processing. However, in the case of big data and real-time systems, it is usually impossible to store the entire tree structure in main memory. When a system needs to process more data, simply adding more servers is not necessarily effective in sharing the memory and computing load, as data distribution and synchronization issues may introduce additional complexity and overhead. It is also difficult for memory-based systems to scale horizontally, which incurs problems when intensive insertion is needed. Therefore, their query performance deteriorates in our study case.

The second type is called Tree-Hybrid Methods, which is usually based on the stand-alone architecture as well, as shown in the second row of Table 1. For example, the index structures in the works [15, 16] are divided into two parts, where the in-memory part is a quad tree and the on-disk component stores a set of lists. During runtime, the whole quad tree structure still needs to be retained in memory. Therefore, their scalability is limited.

Chen et al. [1] propose a model for spatio-temporal keyword queries based on HBase, which we call KV-Disk Methods, as shown in the third row in Table 1. It avoids the problem of low scalability by storing the index in a distributed key-value store instead of memory. To achieve efficient query processing on massive data, it includes row keys for indexing spatio-temporal dimensions and Bloom Filters for quickly detecting the existence of query keywords. However, both of the index structures and Bloom Filters are stored on disks. When receiving a query, it requires to first retrieve the Bloom Filters from the disks, which hinders its query efficiency. Moreover, if there are massive insertions, it first loads and updates the Bloom Filters, and then stores the modified Bloom Filters and data records into the key-value store, which deteriorates the insertion throughput.

KV-Hybird Method, the solution we proposed, is the first method that combines in-memory index with on-disk index based on key-value stores for spatio-temporal keyword queries. Since we store the data in key-value stores while maintaining the filters in memory with a replacement policy, our solution can achieve excellent scalability and high efficiency. In addition, compared with the work [1], we adopt the shard technique, which avoids the possible hotspot issue and enhances the throughput. More types of spatio-temporal keyword queries can be found in the article [38].

3 Preliminaries

In this section, we give some definitions and background knowledge.

3.1 Definitions

Definition 1 (Spatio-Temporal Keyword Object, STK Object)**.** *A spatio-temporal keyword object can be denoted as* $o = \langle p, t, \mathbb{W} \rangle$*, where* $p = (lat, lon)$ *represents the geo-location,* t *represents the time and* $\mathbb{W} = (w_1, w_2, ..., w_k)$ *is a set of keywords describing the textual information. In what follows, both STK data and STK object refer to a Spatio-Temporal Keyword Object.*

Definition 2 (Spatio-Temporal Keyword Query, STK Query)**.** *A spatio-temporal keyword query* q *is denoted as* $\langle r, t_s, t_e, \mathbb{W}' \rangle$*, where* $r = ([lat_{min}, lon_{min}], [lat_{max}, lon_{max}])$ *is a rectangle denoting the geographic scope,* t_s *and* t_e *denote the start time and end time respectively, and* $\mathbb{W}'$ *is a set of keywords. Given a set of STK objects* $\mathbb{D}$*, the STK query returns a set* $\mathbb{O} \subseteq \mathbb{D}$ *of objects, where each object* $o \in \mathbb{O}$ *satisfies the following three constraints simultaneously:*

(1) Spatial Constraint: the geo-location of o *lies within the query spatial range, i.e.,*

$$\begin{aligned} q.r.lat_{min} \leq o.p.lat \leq q.r.lat_{max} \\ q.r.lon_{min} \leq o.p.lon \leq q.r.lon_{max} \end{aligned} \tag{1}$$

(2) Temporal Constraint: the time of o *lies within the query time range, i.e.,*

$$q.t_s \leq o.t \leq q.t_e \tag{2}$$

(3) Keyword Constraint: the keyword sets $o.\mathbb{W}$ *and* $q.\mathbb{W}'$ *satisfy a specified boolean test, i.e.,*

$$test(o.\mathbb{W}, q.\mathbb{W}') = true \tag{3}$$

where $test(*, *)$ *could be any semantic. In this paper, we mainly focus on two semantics, i.e., 1) "OR Semantic", where* $test(o.\mathbb{W}, q.\mathbb{W}')$ *is denoted by* $q.\mathbb{W}' \cap o.\mathbb{W} \neq \emptyset$ *(i.e.,* o *contains at least one keyword in* q*); 2) and "AND Semantic", where* $test(o.\mathbb{W}, q.\mathbb{W}')$ *is represented by* $q.\mathbb{W}' \subseteq o.\mathbb{W}$ *(i.e.,* o *contains all keywords in* q*). It is worth noting that our method can be easily extended to arbitrary semantics (e.g., containing one keyword but not containing another).*

3.2 Background Knowledge

Key-Value Store. A key-value store (KVS) offers functions for storing and retrieving values associated with unique keys. One of its fundamental operations is to retrieve the value by a given key. Besides, many key-value stores also support value retrieval based on a range of keys. These operations are proved to be particularly beneficial for handling ordered datasets. Due to the high performance and strong scalability in the context of big data, key-value stores play a critical role in data management to provide high throughput and low latency data access. Representative key-value stores include Redis [39] and HBase [40].

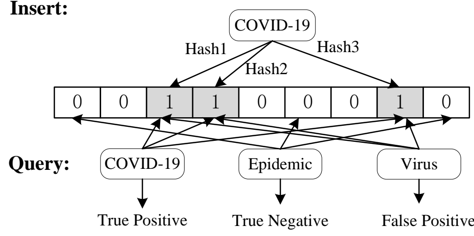


Fig. 2: An Example of Bloom Filter.

However, most key-value stores only support one-dimensional indexing capabilities, making them unsuitable for performing complex conditional queries. Therefore, they do not support spatio-temporal keyword queries directly.

Bloom Filter. A Bloom Filter [17] is used to determine whether an element y is in a set $\mathbb{S}$. It is composed of a fixed-size bitmap and k independent hash functions. Initially, all bits of the bitmap are set to 0. As shown in Fig. 2, when an element is inserted into the set, k hash functions are used to map the element to k bits in the bitmap, setting them to 1. To determine whether an element y exists, we first map y to k bits using the same k hash functions, and then check whether all the corresponding bits are 1. If so (e.g., 'COVID-19' in Fig. 2), it is of great possibility that y exists in $\mathbb{S}$; otherwise, y definitely does not exist in $\mathbb{S}$ (e.g., 'Epidemic' in Fig. 2). Since the hash functions have a certain probability of collision, y may not exist in $\mathbb{S}$ when all the corresponding bits are 1 (e.g., 'Virus' in Fig. 2), which is called false positive.

The false positive rate (FPR) of a Bloom Filter depends highly on the size of the bitmap. In our research scenario, the data are constantly inserted, and the data volume varies greatly among different spatio-temporal regions due to the severe unbalance of spatio-temporal data [41, 42]. However, traditional Bloom Filters need to allocate a fixed-size bitmap in advance. Therefore, it is difficult to determine an appropriate size for the bitmap before all data are ready. InfiniFilter [43] is a link of filters that stores a fingerprint along with a unary encoded age counter for each entry within a compact hash table. Its advantage lies in the ability to dynamically adjust the hash table size when data are continuously inserted to ensure a low false positive rate.

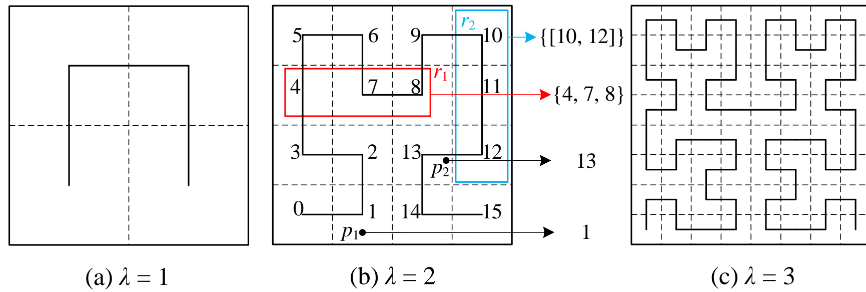


Fig. 3: An Example of Hilbert Curve.

Hilbert Curve. Hilbert Curve is a continuous fractal space-filling curve. It can cover all points in a multi-dimensional space by recursively dividing the space into smaller subspaces and connecting these subspaces to form a curve. The subspaces are numbered sequentially by the traversal order of the curve. Fig. 3 shows three two-dimensional Hilbert Curves with the resolutions $\lambda = 1, 2, 3$ respectively, where the resolution λ refers to the recursion depth of division. This transformation preserves spatial locality, ensuring that closely located points remain close in the curve. Other space-filling curves, such as Z-curve [44–48], also have similar features. However, Hilbert Curve exhibits better continuity[1, 49]. Therefore, we use Hilbert Curve in our framework. Hilbert Curve has three useful functions: 1) mapping a spatial point in the original space to a code, 2) mapping a spatial range to a set of codes and 3) mapping a spatial range to a set of ranges.

- $pToCode(p, \lambda)$: Given a spatial point $p = (lat, lon)$, $pToCode(p, \lambda)$ returns a code between $[0, 2^{2\lambda} - 1]$ that represents the grid number of point p under the Hilbert Curve with the resolution of λ.
- $rToCodes(r, \lambda)$: Given a spatial range $r = ([lat_{min}, lon_{min}], [lat_{max}, lon_{max}])$, $rToCodes(r, \lambda)$ returns a set of codes representing the grids intersecting with r, under the Hilbert Curve with the resolution of λ.
- $rToRanges(r, \lambda)$: Given a spatial range $r = ([lat_{min}, lon_{min}], [lat_{max}, lon_{max}])$, $rToRanges(r, \lambda)$ returns multiple ranges, each of which can be viewed as the concatenation of consecutive codes in $rToCodes(r, \lambda)$.

For example, as shown in Fig. 3(b) with $\lambda = 2$, we have $pToCode(p_1, 2) = 1$, $pToCode(p_2, 2) = 13$, $rToCodes(r_1, 2) = \{4, 7, 8\}$, and $rToRanges(r_2, 2) = \{[10, 12]\}$.

4 Overview

Fig. 4 gives an overview of our framework, which consists of two core modules: *Data Insertion* and *Query Processing*.

Data Insertion. The data insertion module, as illustrated in the left part of Fig. 4, is mainly responsible for receiving STK objects and performing the following two operations: 1) *Updating Key-Value Stores*. Through Key Generator and Value Convertor, each STK object is converted into a key-value pair and then inserted into the key-value store (i.e., KVS-Data). 2) *Updating Filter Manager*. The relationship between the spatio-temporal information and the keywords is sent to Filter Manager, which is used for speeding up STK queries later. Filter Manager also determines which filters should be moved out to the key-value store (i.e., KVS-Filter) to avoid running out of memory.

Query Processing. As shown in the right part of Fig. 4, given an STK query, Key Generator converts the spatio-temporal query range into several key ranges. Then, the filters in Filter Manager further narrow the key ranges according to the keyword constraints (e.g., contains all keywords in the query) to obtain the shrunk key ranges. After that, the shrunk key ranges are sent to KVS-Data to get the results in parallel.

5 Data Insertion

Fig. 5 gives the overview of Data Insertion, which consists of three steps. 1) *Insertion into KVS-Data*. When an STK object o_1 comes, we extract the necessary information and then insert it into the key-value stores (i.e., KVS-Data). 2) *Insertion into Filters*. To accelerate the query efficiency, we also extract some other information and insert this information into the two-layer filters. 3) *Managing Filters*. Since the memory size is limited, Filter Manager will evict the “cold” filters out of the memory and load necessary filters from the key-value stores (i.e., KVS-Filter).

5.1 Insertion into KVS-Data

Key-value stores can only index one-dimensional data, while STK objects contain multi-dimensional information. Therefore, we need to transform STK objects into one-dimensional data before we insert them into KVS-Data. There are two components for the transformation: Key Generator and Value Convertor.

Key Generator. In Key Generator, each STK object o_i is converted into the key of KVS-Data. To efficiently support spatio-temporal keyword queries, one intuitive method is combining the Hilbert Curve code with the keywords of o_i, followed by an object identifier to eliminate the possible key collision, where the Hilbert Curve code is the numbering result of a three-dimensional space consisting of latitude, longitude and time of o_i. However, this method has several drawbacks. First, as pointed out by the work [44], treating the spatial information and temporal information equally ignores the different scales of them, which hinders the spatial filtering ability in most cases. Second, it is hard to support STK queries with any semantics of keywords. Another method [1] combines the time information with the two-dimensional Hilbert Curve code of latitude and longitude as the key, but it may encounter the hotspot issue.

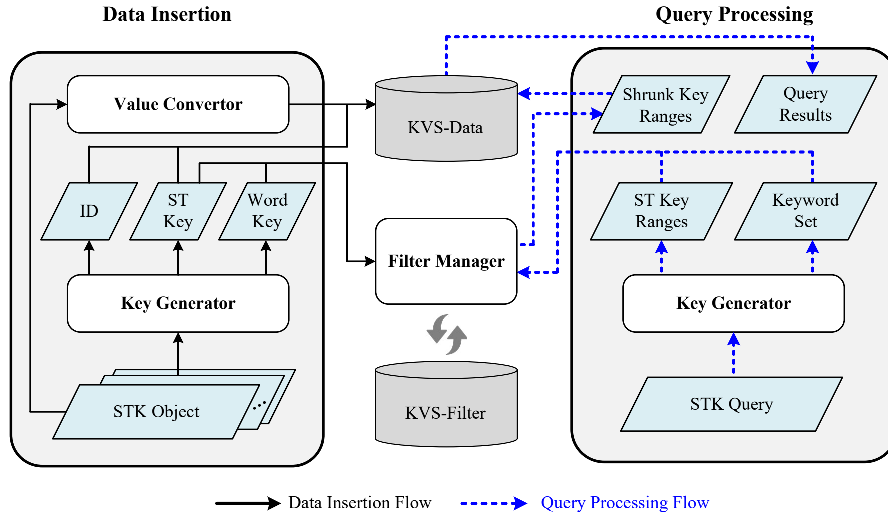


Fig. 4: Overview of KV-STK.

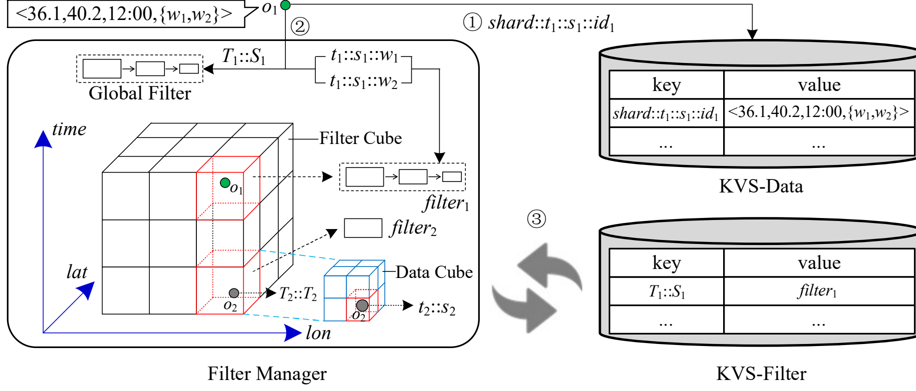


Fig. 5: Overview for Data Insertion.

To address the issues mentioned above, this paper designs the key combination of an STK object o_i in KVS-Data as follows:

$$key_{Data} = shard :: t_i :: s_i :: id_i \tag{4}$$

where "::" is the byte concatenation operation, *shard* is a random number that distribute s the objects across different cluster nodes to avoid hotspot issue, t_i is the temporal code that we will discuss it later, s_i is the Hilbert Curve code with the resolution of λ_{Data}, i.e., $s_i = pToCode(o_i.p, \lambda_{Data})$, and id_i is the unique identifier of o_i to avoid the possible key collision since there may exist multiple STK objects in the same spatio-temporal region. In our implementation, to guarantee the load balance among all cluster nodes, we let *shard* be a random number between $[0, n_c - 1]$, where n_c is the number of nodes in the cluster. λ_{Data} is set 14, which means that the grid size is about $1.4km \times 1.4km$, since we usually query the data in a region in about $1km^2$.

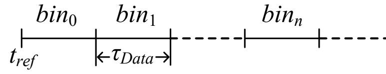


Fig. 6: Time Key Generator.

We adopt the time bin technique [45] to calculate the temporal code t_i of o_i. In specific, as shown in Fig. 6, we divide temporal dimension into disjoint bins with equal size, where the size of each bin is τ_{Data}. Then the temporal code t_i of o_i is calculated by the following equation:

$$t_i = \lfloor (o_i.t - t_{ref}) / \tau_{Data} \rfloor \tag{5}$$

where t_{ref} is a system parameter named reference time (e.g., 1970-01-01T00:00:00Z). In our implementation, we set τ_{Data} as 1 hour, since we usually query the data in a time range of one hour.

Value Convertor. In Value Convertor, we simply serialize each STK object into a byte array, forming the value portion of a key-value pair.

5.2 Insertion into Filters

The key combination in KVS-Data introduced in Equation (4) does not contain any keyword information, because there could be multiple keywords in an STK object. If we simply append the keywords in the tail of key_{Data}, the keyword filtering ability will become invalid, since it is hard to find the matching of the query keywords in key_{Data}. In this paper, we maintain the relationship between the keyword information and spatio-temporal information (e.g., whether a keyword appears in a given spatio-temporal region) in Bloom Filter-based structures.

Naïve Method. A naïve method is to store all relationship between keywords and spatio-temporal information in a single Bloom Filter [17]. However, STK data are generated constantly. As the data volume increases, the false positive rate of the traditional Bloom Filter will deteriorate. Although some dynamic filters [43, 50, 51] can adjust their memory consumption in accordance with the volume of data, they need to continuously increase their size to keep a low false positive rate, ending in running out of memory eventually.

Our Method. We observe that the numbers of STK objects vary significantly in different time periods and spatial regions. To this end, we partition the spatio-temporal space into multiple equal-sized disjoint cubes, and maintain at most one individual dynamic filter (called **local filter**) for each cube. We adopt InfiniFilter [43] as the local filter since it can expand its hash table infinitely while keeping a low false positive rate. Initially, we do not assign any filter for a cube. If there is at least one STK object falling in this cube, we build a filter with a small size (e.g., 1MB) for this cube, and insert the STK object into the filter. With more STK objects inserted into this filter, its size increases smoothly while its false positive rate keeps stable. For example, as shown in Fig. 5, we maintain a filter $filter_1$ for the cube where o_1 locates, and maintain another filter $filter_2$ for the cube where o_2 falls.

There are two advantages to maintain an individual filter for each cube. First, we can assign less memory to the cubes that contain fewer STK objects. In extreme circumstances, if there is no STK object in a cube, we will not assign any filter for this cube, which can save much memory. Second, maintaining multiple filters allows us to evict some less-visited filters out of the memory when the memory is full or load necessary filters from the disks if they are evoked (please see Subsection 5.3). Heretofore, there are still two questions left: 1) How to partition the cubes? 2) What are the hashing keys of the filters?

(1) Cube Partition. We disjointly partition the whole spatio-temporal space into cubes with equal-size, but the size of cubes should be carefully selected. In specific, the spatial space and temporal space are partitioned separately. For the temporal space partition, we divide the whole temporal space into time bins using the same technique

shown in Fig. 6 but with a different bin size of τ_{Filter}. For the spatial space partition, we divide the whole spatial space into grids and number each grid using Hilbert Curve with a resolution of λ_{Filter}. Therefore, if an STK object o_i is located in a cube, then the number "T_i::S_i" of the cube is calculated by:

$$\begin{cases} T_i = \lfloor (o_i.t - t_{ref})/\tau_{Filter} \rfloor \\ S_i = pToCode(o_i.p, \lambda_{Filter}) \end{cases} \tag{6}$$

Here, τ_{Filter} and λ_{Filter} are two user-defined parameters. We will investigate their effects on the performance of STK queries in Section 7.

(2) Filter Hashing Key. We leverage filters to determine whether a keyword exists in a specified spatio-temporal region. Therefore, for each STK object o_i, we combine each keyword with the spatio-temporal information, and regard each combination (a.k.a. filter hashing key) as the input of hashing functions. Specifically, if the STK object o_i has m keywords $\{w_1, w_2, ..., w_m\}$, it will produce m filter hashing keys, and each filter hashing key is composed of:

$$key_{Hash} = t_i :: s_i :: w_j \tag{7}$$

where "::" is the byte concatenation operation, $t_i = \lfloor (o_i.t - t_{ref})/\tau_{Data} \rfloor$ (i.e., Equation (5)), $s_i = pToCode(o_i.p, \lambda_{Data})$, and $w_j \in \{w_1, w_2, ..., w_m\}$. We choose the same bin size τ_{Data} and Hilbert Curve resolution λ_{Data} with key_{Data}, enabling us to quickly eliminate unsatisfied keys in KVS-Data when answering STK queries.

5.3 Managing Filters

As STK objects are constantly generated, with the time goes by, there are increasing number of cubes formed, leading to the risk of running out of memory. On one hand, STK objects are essentially inserted in a chronological order. On the other hand, users tend to query the data generated at the most recent time. Therefore, the filters associated with different cubes are visited unevenly. Hence, to avoid memory overflow, we can move the less-visited filters out of the memory, and load necessary filters from the disks if they are accessed. We call this module Filter Manager, as shown in Fig. 5.

To efficiently load the necessary filters from the disks, we store the less-visited filters in a key-value store (i.e., KVS-Filter). As shown in Fig. 5, each row of KVS-Filter stores one filter $filter_i$, where the key is the combination of the associated cube number (i.e., T_i::S_i), and the value is the serialized filter $filter_i$. By doing this, we can load the corresponding filter efficiently by a given cube.

LRU Policy. We adopt the Least Recently Used (LRU) replacement policy to determine which filter should be moved out of the memory and stored into KVS-Filter. In specific, we maintain a hash table and a linked-list for the filters in memory, as shown in Fig. 7, wherein the hash table can quickly find the associated filter in the linked-list by a given cube number. The most recently visited filter is placed in the head of the linked-list, while the least-used filter is in the tail. For both query and insertion operations, if the visited filter is already in the memory, we move it to the head of the linked-list directly. If the visited filter is not in the memory, we try to retrieve it from

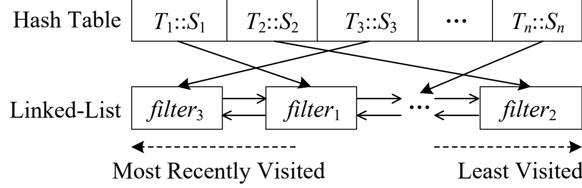


Fig. 7: Implementation of LRU Policy.

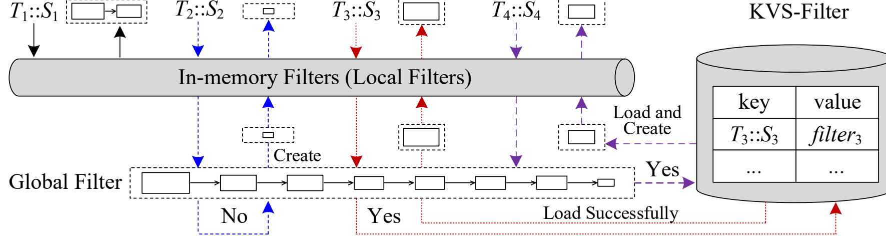


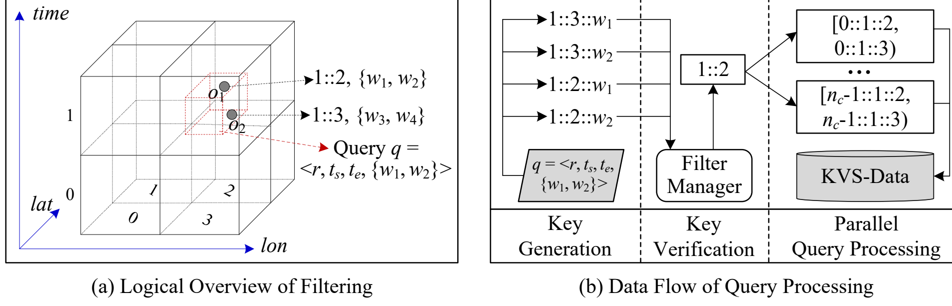
Fig. 8: Examples of Filter Management for Data Insertion.

KVS-Filter. If loaded successfully, the filter is prepended into the head of linked-list; otherwise, for insertion operations, we create a new filter, while for query operations, we return an empty result (detailed in Section 6). Once the memory is full, we sequentially evict some least-used filters out of the memory, and insert them into KVS-Filter if they have been updated, until the memory usage is below the given threshold.

Global Filter. We observe that if the required filter is not in the memory, we should always try to access KVS-Filter to check whether the corresponding filter is stored in KVS-Filter, which may affect the efficiency of insertion and query processing. To this end, we maintain a global filter (another InfiniFilter) to indicate whether there exists any STK object in a cube. Specifically, when an STK object o_i is generated, we obtain its cube number "T_i:: S_i" according to Equation (6), and then insert it as the hashing key into the global filter. If a required filter is not in the memory, instead of accessing KVS-Filter directly, we first check its existence with the help of the global filter. If the global filter tells us that there is no STK object in the cube (equivalent to the required filter does not exist), we would not access KVS-Filter any more. **In this paper, if not specified, the filter refers to the local filter.**

Until now, there are two types of logical cubes: one is for the calculation of "t_i::s_i" in Equation (4), and another is for the calculation of "T_i::S_i" in Equation (6). We call the former *Data Cube*, while the latter *Filter Cube*. As shown in Fig. 5, in general, the filter cube should be much larger than the data cube, because with a relatively larger filter cube (i.e., smaller λ_{Filter} and bigger τ_{Filter}), we can avoid frequently accessing KVS-Filter when maintaining filters. Therefore, the number of filter cubes should be limited, which means that the global filter would not be too large even for a very long time and could be entirely accommodated in memory.

Examples. Fig. 8 gives four cases of filter management for data insertion. **Case 1**: We want to access the filter associated with the cube "T_1::S_1" and the filter



(a) Logical Overview of Filtering (b) Data Flow of Query Processing

Fig. 9: Example for Query Processing

is in the memory, so we return the filter directly. **Case 2**: We access the filter associated with the cube "T_2::S_2" but it does not exist in the memory, so we resort to the global filter. The global filter reply "No", so we create a new filter, add it into memory and return the filter finally. **Case 3**: We access the filter associated with the cube "T_3::S_3" but it does not exist in the memory. After we get an answer of "Yes" from the global filter, we load the filter from KVS-Filter successfully, add the loaded filter to the memory cache, and finally return the filter. **Case 4**: We want to access the filter associated with the cube "T_4::S_4", but it is not in the memory. Although the global filter indicates that the filter is stored in KVS-Filter, we do not find it, so we create a new filter, then add it in memory, and finally return the filter. For Case 2, Case 3 and Case 4, if the memory size exceeds a given threshold after we add the accessed filter, we move the least-used filters out of memory to ensure the memory usage is below the given threshold.

6 Query Processing

STK data are stored in the key-value store, i.e., KVS-Data. Hence, we can efficiently retrieve the STK data using the corresponding keys. During query processing, we similarly transform the STK query into one or multiple keys. However, since the designed keys in KVS-Data do not contain textual information, relying solely on the pruning capability of key-value stores would significantly reduce the query efficiency. Therefore, we leverage filters to filter out the impossible keys to reduce the overall scanned data. The query processing of KV-STK consists of three steps: 1) *Key Generation*, 2) *Key Verification*, and 3) *Parallel Query Processing*.

6.1 Key Generation

In this step, we transform the query into a set of keys. Given a query $q = \langle r, t_s, t_e, \mathbb{W}' \rangle$, its spatio-temporal information can also form a cube, as shown in Fig. 9(a). We first calculate a set of time bins $\mathbb{T} = \{t_a, t_a + \tau_{Data}, t_a + 2\tau_{Data}, ...t_b\}$ that intersect with the query time range $[q.t_s, q.t_e]$ by the following equation:

$$\begin{cases} t_a = \lfloor (q.t_s - t_{ref})/\tau_{Data} \rfloor \\ t_b = \lfloor (q.t_e - t_{ref})/\tau_{Data} \rfloor \end{cases} \quad (8)$$

where t_{ref} and τ_{Data} have the same meanings with these in Equation (5). Then, we obtain a set of spatial codes, i.e., $\mathbb{S} = rToCodes(q.r, \lambda_{Data})$, which represent the grids that intersect with the query spatial range $q.r$. At last, for each $t \in \mathbb{T}$, $s \in \mathbb{S}$ and $w \in \mathbb{W}'$, we can form a key as "t::s::w". It is easy to infer that the number of keys we can generate is $|\mathbb{T}| \times |\mathbb{S}| \times |\mathbb{W}'|$.

For example, given the query $q = \langle r, t_s, t_e, \{w_1, w_2\}\rangle$ shown in Fig. 9(a), we have $\mathbb{T} = \{1\}$ and $\mathbb{S} = \{2, 3\}$. Since $q.\mathbb{W}' = \{w_1, w_2\}$, we can generate $1 \times 2 \times 2 = 4$ keys shown in Fig. 9(b).

6.2 Key Verification

In this step, we verify each key generated in the previous step with the help of built filters. We feed each key into Filter Manager, and Filter Manager will reply to us whether the key exists using the policy described in Section 5. In specific, this step consists of two main sub-steps: 1) *Finding the Related Filters*, and 2) *Checking the Existence of Keys*.

(1) **Finding the Related Filters.** Since the query may intersect with multiple Filter Cubes, we first calculate the cube number of each intersected Filter Cube, and then obtain the related filter with the help of Filter Manager. Given a query $q = \langle r, t_s, t_e, \mathbb{W}'\rangle$, we first compute a set of time bins $\mathbb{T}' = \{t'_a, t'_a + \tau_{Filter}, t'_a + 2\tau_{Filter}, ..., t'_b\}$ that intersect with the query time range $[q.t_s, q.t_e]$ by:

$$\begin{cases} t'_a = \lfloor (q.t_s - t_{ref})/\tau_{Filter} \rfloor \\ t'_b = \lfloor (q.t_e - t_{ref})/\tau_{Filter} \rfloor \end{cases} \tag{9}$$

where t_{ref} and τ_{Filter} are the reference time and size of time bins, respectively, which have the same meanings with these in Equation (6). Next, we calculate a set of spatial codes, *i.e.*, $\mathbb{S}' = rToCodes(q.r, \lambda_{Filter})$ that represent the grids intersecting the query spatial range $q.r$. For each $T \in \mathbb{T}'$ and $S \in \mathbb{S}'$, we form a filter cube number "T::S", with which we can load the related filter from Filter Manager. Note that there is minor difference with the logics of Data Insertion shown in Fig. 8. During query processing, if the global filter indicates the accessed filter does not exist (Case 2), or the global filter indicates the accessed filter exists but it is actually not stored in KVS-filter (Case 4), we do not create a new filter here.

(2) **Checking the Existence of Keys.** For each key "t::s::w" generated in Section 6.1, we first check if it exists in the related filter, and then conclude that if the spatio-temporal key combination "t::s" in KVS-Data should be accessed. In particular, if the related filter does not exist, we are sure that the key "t::s::w" is unqualified. If the related filter exists but it returns "No", then the key "t::s::w" can be definitely filtered out as well. Finally, we can figure out the validity of the spatio-temporal key "t::s" according to the semantic of the STK query: 1) For "OR Semantic", if there is at least one $w_i \in q.\mathbb{W}'$ letting "t::s::w_i" get "Yes" from the filter, then "t::s" is valid. 2) For "AND Semantic", if for all $w_i \in q.\mathbb{W}'$ such that "t::s::w_i" get "Yes" from the filter, then "t::s" is valid.

For example, as shown in Fig. 9, given the query $q = \langle r, t_s, t_e, \{w_1, w_2\}\rangle$, if the filters return "No" for "1::3::w_1" and "1::3::w_2", but return "Yes" for "1::2::w_1" and "1::2::w_2", then we regard "1::2" to be valid, and will send it to the next step.

6.3 Parallel Query Processing

After key verification, we obtain a set of spatio-temporal keys. For each spatio-temporal key, we can generate multiple key ranges, with which we can retrieve the required data from KVS-Data in parallel. Specifically, in this step, we perform the following operations: 1) *Combining Adjacent Keys*, 2) *Prepending Shard Numbers*, 3) *Retrieving Data Parallelly* and 4) *Refining Results*.

Combining Adjacent Keys. To reduce the number of accesses to KVS-Data, we combine the adjacent keys into key ranges. Here, "t_i::s_i" and "t_i::($s_i + 1$)" are considered to be adjacent. For example, given the spatio-temporal keys "2::2", "2::3", "2::4" and "1::2", we can combine them into two key ranges "[2::2, 2::4]" and "[1::2, 1::2]", thus reducing the number of accesses to KVS-Data from 4 to 2. Note here that a single key can be deemed as a range where its start key and end key are equal.

After that, to ensure all required data can be retrieved, for each time range, we change its closed end key to an open one, since we have appended the object identifier at the tail of the key when the object is inserted. For example, the key ranges "[2::2, 2::4]" and "[1::2, 1::2]" are transformed into "[2::2, 2::5)" and "[1::2, 1::3)", respectively.

Prepending Shard Numbers. To avoid hotspot issue, we prepend *shard* to the key of each inserted object, where *shard* is a random value between $[0, n_c - 1]$, and n_c is the number of cluster nodes. During query processing, we should also prepend all possible shard values to each key range. In specific, for each key range $[start, end)$, we produce n_c key ranges: [0::*start*, 0::*end*), [1::*start*, 1::*end*), ..., [($n_c - 1$)::*start*, ($n_c - 1$)::*end*). For example, suppose $n_c = 3$, the key range "[1::2, 1::3)" are transformed into 3 key ranges: [0::1::2, 0::1::3), [1::1::2, 1::1::3) and [2::1::2, 2::1::3).

Retrieving Data Parallelly. After prepending shard values, we obtain numerous key ranges, each of which triggers a scanning operation on the key-value store. We utilize a thread pool to execute the scanning operation parallelly. Thanks for the powerful parallel processing capability of the key-value store, we can efficiently perform scanning based on the key ranges in parallel on KVS-Data.

Refining Results. Due to the possible false positive rate of filters, there could be some STK objects that do not contain any query keyword. Besides, there could also be some objects that do not meet the spatio-temporal constraints. After retrieving the candidate objects, we check each retrieved STK object and delete those that do not satisfy the query constraints.

7 Experiment

7.1 Experimental Settings

Datasets. We validate the performance of KV-STK using two real-world datasets and one synthetic dataset. Table 2 gives the detailed information of the two real datasets. In specific, **Yelp** [52] comprises reviews provided by users on various businesses, serving as

Table 2: Information of Real-World Datasets.

Database	Latitude Range	Longitude Range	# Objects	Average # Keywords
Tweet	[-20.27,71.33]	[-178.71,178.89]	19,966,712	5.58
Yelp	[27.56,53.68]	[-120.10,-73.20]	6,990,033	70.04

Table 3: Parameter Settings.

Parameters	Settings
Query Temporal Range (h)	1, 2, **3**, 4, 5
Query Spatial Range (km^2)	1×1, 2×2, $\mathbf{3} \times \mathbf{3}$, 4×4, 5×5
# Query Keywords	1, 2, **3**, 4, 5
Semantic	**OR**, AND
λ_{Filter}-τ_{Filter}	14-1, 13-2, **12-4**, 11-8, 10-16
# Cluster Nodes	1, 3, **5**, 7
Key-Value Store	**HBase**, Redis

a valuable resource for analyzing consumer feedback and sentiments towards different establishments. **Tweet** [53] consists of tweets collected from the Twitter platform, offering insights into social media topics and public opinions. To test the scalability of the framework, we obtain a synthetic dataset by copying tweets and shifting time and coordinates to a certain extent, which includes up to 100 million pieces of data.

Baselines. We compare KV-STK with **three** competitors (i.e., JUST [44], BDIA [1] and STILT [37]) and **two** variants (i.e., KV-STK-G and KV-STK-F).

JUST [44]. JUST is a spatial-temporal engine that incorporates a distributed NoSQL data store to efficiently manage big spatio-temporal data. Since JUST does not natively support STK queries, we first retrieve the data with the spatio-temporal constraints, and then select the data meeting the keyword constraint from the results.

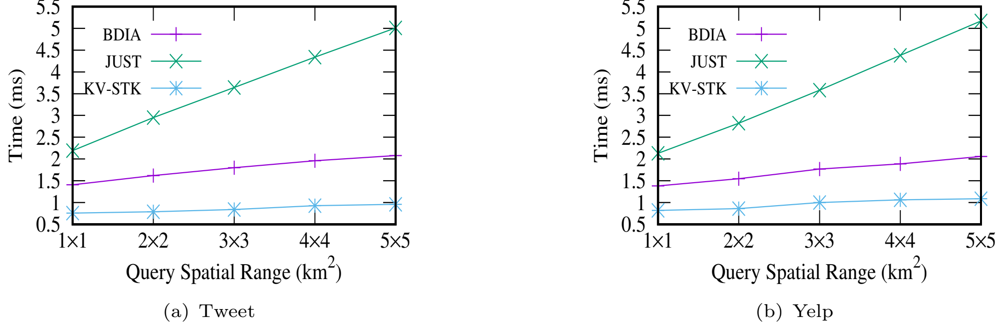
BDIA [1]. BDIA is the only existing method that supports spatio-temporal keyword queries based on key-value store.

STILT [37]. To verify the stability of KV-STK in different memory limitations, we also compare it with STILT. STILT designs a tree-based index and unifies the spatial, temporal and textual components within a single structure in a space-efficient manner.

KV-STK-G. It is a variant of KV-STK, but without the global filter.

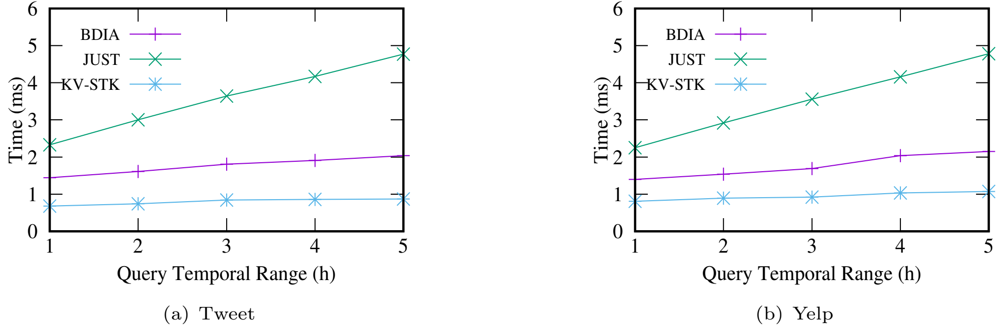
KV-STK-F. It is a variant of KV-STK, but without any filter in memory.

Metric. We mainly focus on the query response time. For each parameter setting, we perform a set of one hundred thousand queries (i.e., a query set), and the average query response time is reported. The temporal and spatial ranges of each query are generated based on the location and time of objects. To ensure that 50% of the queries in a query set return non-empty results, we generate the query keywords from the spatio-temporal related objects for half of queries. The rest of queries contain keywords randomly selected from the global keyword set in the corresponding dataset.



(a) Tweet (b) Yelp

Fig. 10: Performance with Different Query Spatial Ranges.



(a) Tweet (b) Yelp

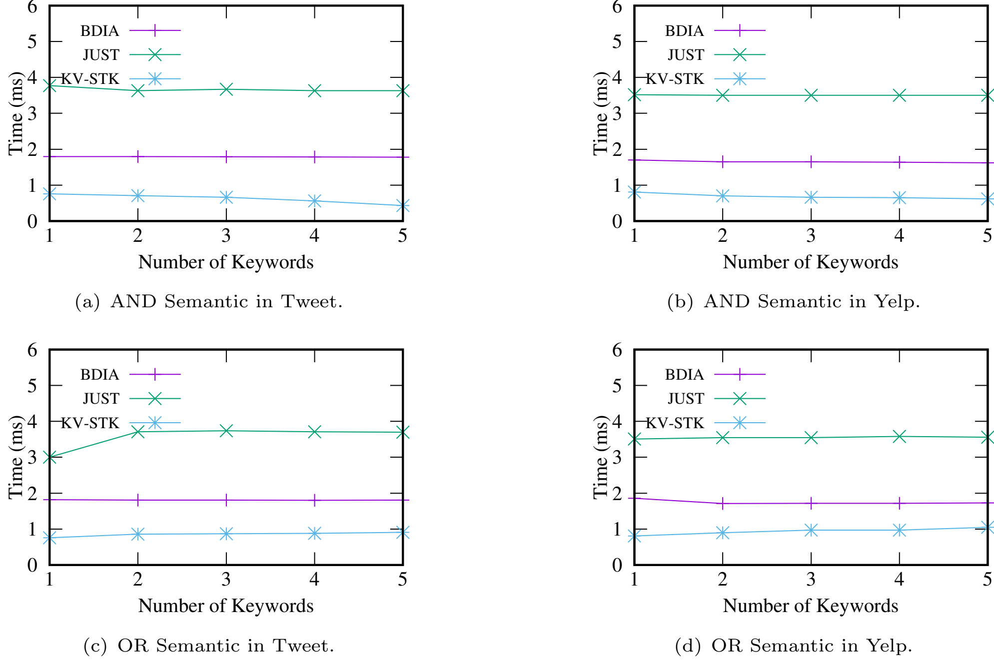
Fig. 11: Performance with Different Query Temporal Ranges.

Settings. Table 3 summarizes the parameters, where the default values are in bold if not specified. By default, the experiments are conducted on a cluster of five nodes, and each node is equipped with an 8-core CPU, 128GB RAM, and 4T disk. We assign a maximum 1GB of memory to the filter cache. All methods are implemented in Java language. The JDK (Java Development Kit) version is 1.8, Hadoop version is 2.8.5, and HBase version is 2.2.0. All source codes are published [3].

7.2 Different Query Parameters

Different Query Spatial/Temporal Ranges. To evaluate the performance variation when the query spatial/temporal range changes, we perform STK queries with different spatial/temporal ranges on both Tweet and Yelp. We do not compare STILT since it runs out of memory while building the index. From Fig. 10 and Fig. 11, we can observe that, 1) For both Tweet and Yelp, with the expansion of the query spatial/temporal range, all methods take linearly more time. However, compared with

[3] https://github.com/Spatio-Temporal-Lab/st-keyword-query



(a) AND Semantic in Tweet.

(b) AND Semantic in Yelp.

(c) OR Semantic in Tweet.

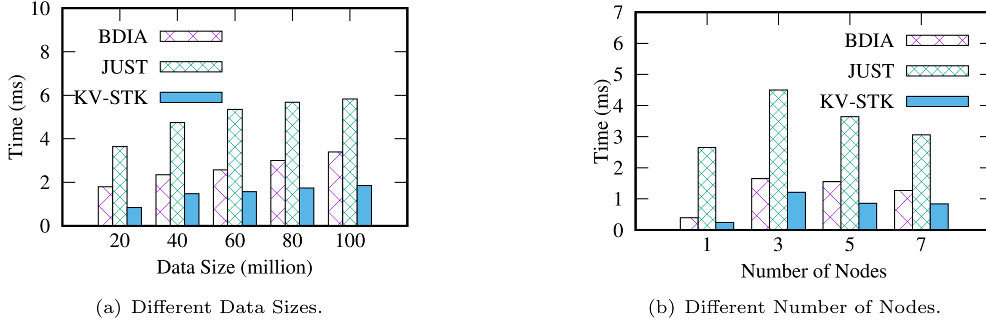
(d) OR Semantic in Yelp.

Fig. 12: Performance with Different Number of Keywords.

BDIA and JUST, our method has a flatter growth curve. 2) KV-STK is the most efficient among the three methods in all query spatial ranges for both datasets. JUST and BDIA are about 4.72 times and 1.24 times slower than KV-STK, respectively.

One reason could be that, for all methods, the number of keys to be scanned is proportional to the growth of query spatial/temporal range. Besides, the qualified results are supposed to linearly increase with a larger query spatial/temporal range, which leads to more processing time and transmission latency. For BDIA, the number of filters to be accessed by BDIA is also proportional to the growth of the spatial/temporal range, but it does not necessarily scan the specific data, so its performance is better than JUST. For KV-STK, although the number of filters accessed increases as well with the expansion of the query spatial/temporal range, most operations are performed in memory without disk I/Os, so it is more stable in different cases.

Different Numbers of Keywords. We vary the number of query keywords within the range of 1 to 5, and then compare the efficiency of different methods under AND Semantic and OR Semantic, respectively. The results are shown in Fig. 12, from which we have the following observations: 1) Under the same spatio-temporal constraint, with an increasing number of keywords, the query time of KV-STK slightly increases in OR Semantic but decreases in AND Semantic. This is because for OR Semantic, more keywords mean a looser constraint, which leads to more qualified results. However, for AND Semantic, more keywords mean a stricter constraint, leading to fewer satisfied results. 2) The two comparing methods have a smaller relative variation. Because for



(a) Different Data Sizes.

(b) Different Number of Nodes.

Fig. 13: Scalability Verification.

BDIA, the range of keys scanned is actually fixed; for JUST, it has no keyword filtering capability, so it is the most stable. 3) For both AND Semantic and OR Semantic, KV-STK is always faster than the competitors, because the filters in memory have filtered out many unqualified keys before actually accessing the key-value store. For BDIA, it needs to access the disks for loading the filters, which hinders its efficiency.

7.3 Scalability Verification

Different Data Sizes. To verify the scalability of our framework, we generate five large datasets of different sizes by copying and modifying the Tweet dataset and conduct experiments on them. For each piece of tweet data, while keeping the keywords unchanged, we shift the time forward or backward by 10 to 60 minutes, and shift the spatial coordinates by 100 to 500 meters to one of the four directions of north, south, east and west to generate a new piece of data.

The experimental results are shown in Fig. 13(a). As the amount of data increases, the query response time of each method continues to increase. However, KV-STK significantly outshines other two methods compared. In specific, the time required by BDIA and JUST is twice and four times that of ours, respectively. The outcomes not only validate the effectiveness of KV-STK in managing large-scale data but also underscore its potential in applications demanding high efficiency and scalability.

Different Numbers of Nodes. In order to demonstrate the query efficiency of KV-STK on different numbers of cluster nodes, we compared the query efficiency of our framework with the baselines when the number of nodes varies from 1 to 7. As shown in Fig. 13(b), we can observe that: 1) No matter how the number of nodes changes, KV-STK always has the highest query efficiency, JUST has the lowest efficiency, and BDIA is between the two; 2) In a distributed environment (i.e., the number of nodes is greater than 1), the fastest query time required by our solution is only 24% of JUST and 55% of BDIA; 3) When the number of nodes increases from 3 to 7, the query efficiency decreases in sequence. It is because that the parallel capability of the system is increased and the load pressure on a single node is reduced when increasing the number of nodes. When the number of nodes is 1, the query efficiency is the fastest for all methods because there is no network overhead. Note that this phenomenon does

not denote that the distributed system is inferior to the standalone one. For super large-scale data management, a distributed system is necessary since the resource of one machine is limited. Besides, the distributed system also provides fault tolerance.

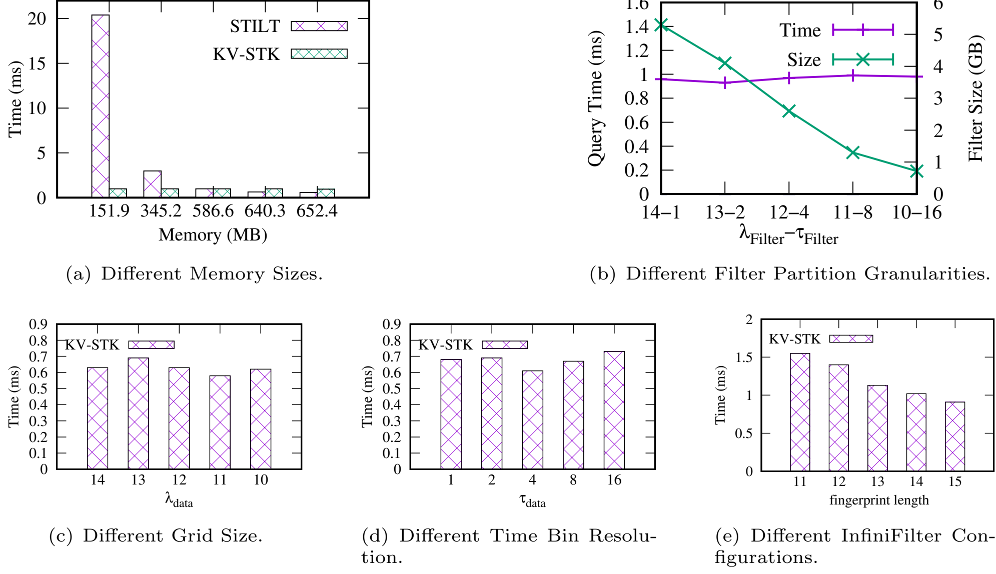
7.4 Different System Settings

Different Memory Sizes. In order to verify the stability of our framework under different memory sizes, we compared the query efficiency of KV-STK and STILT under different memory limitations. The reason why we do not compare the other two key-value based solutions (i.e., JUST and BDIA) is that they do not require the usage of memory resource. In this set of experiments, we utilize only 1 million of STK objects in Tweet. Because even with a memory limitation of 30GB, STILT cannot finish indexing the full data of Tweet, and it finally ends in running out of memory. We limit the maximum height of the STILT index to control the memory usage, and assign the same size of memory to the filter cache of KV-STK.

As shown in Fig. 14(a), when the memory is large, the query efficiency of STILT is slightly better than KV-STK , but when the memory is small, its query time reaches 20ms, which is about 20 times slower than our framework. Compared with the drastic change of STILT, KV-STK has stable query time under different memory limitations, which demonstrates the stability of our framework. For STILT, limiting the memory size of the tree is equivalent to limiting the number of partitions of the spatio-temporal region, which leads to a significant decline in its spatio-temporal pruning ability. However, our framework will only increase the number of I/Os of the filters when the memory is limited. The I/Os of the filters are much smaller than these of the data, so the impact is negligible. In the experiment, even with the lowest memory (i.e., 151.9MB in the figure), most of the filters to be accessed can be accommodated in memory, so KV-STK runs stably.

Different Filter Partition Granularities. We also study the performance of KV-STK in terms of query response time and total filter size (including the filters in both memory and KVS-Filter) under different filter partition granularities. We set the combination of λ_{Filter}-τ_{Filter} as 14-1, 13-2, 12-4, 11-8 and 10-16, respectively. Note that 14-1 is finer-grained than 13-2, and so on. We use the full dataset of Tweet.

The experimental results are shown in Fig. 14(b). On one hand, as the partition granularity becomes coarser, the query response time increases slightly, but its fluctuation is within only 0.95 ms, i.e., it is relatively stable. The granularity mainly affects the number of STK objects inserted into a specific filter. The adopted dynamic filter, i.e., InfiniFilter, would automatically change its size according to the number of inserted STK objects to keep a stable false positive rate. A coarser granularity results in more STK objects inserted into one filter, further leading to a bigger filter. It takes a bit more time to load a bigger filter from KVS-Filter if it is not in memory. Note that the filter is much smaller than the accessed data, so the increased time is marginal. On the other hand, with a coarser partition granularity, the total filter size drops,



(a) Different Memory Sizes.

(b) Different Filter Partition Granularities.

(c) Different Grid Size.

(d) Different Time Bin Resolution.

(e) Different InfiniFilter Configurations.

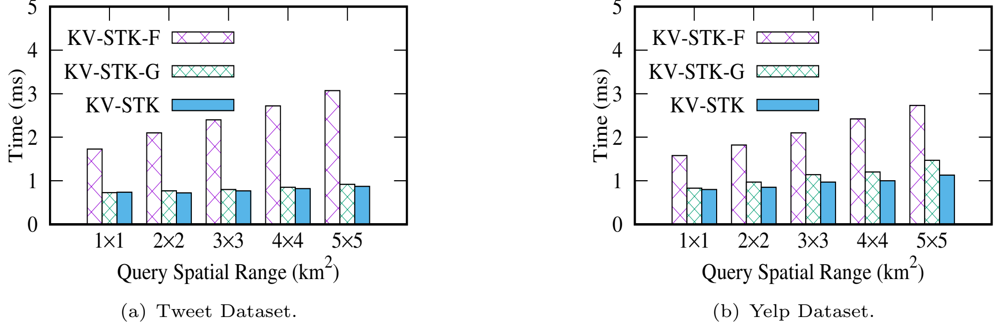
Fig. 14: Performance Over Different Parameters (Tweet).

since the number of filters becomes smaller. To this end, users can make a reasonable trade-off between the query efficiency and total filer size according to their own goals.

Different Data Partition Granularities. In order to verify the performance of KV-STK under different data partition granularities, we conduct a set of experiments by changing parameters λ_{data} and τ_{data} respectively and compare the query efficiency. We set the query spatial and temporal ranges to 1km*1km and 1 hour, respectively, to observe the trend in query time as the data partition granularities increase (i.e., smaller λ_{data} and bigger τ_{data}).

The experimental results are shown in Fig. 14(c), (d), where we can observe that as λ_{data} decreases (resulting in larger spatial grids), the query time fluctuates. This may be due to the fact that while the decrease in λ_{data} leads to more data being returned for each query range, the number of query ranges simultaneously decreases. We also observe a similar result as τ_{data} increases (resulting in larger time bins), but the overall trend is upward. This may be because time precedes space in the composition of key_{Data}, making the increase in τ_{data} have a greater impact on the number of results returned for each query range. From the granularity corresponding to the smallest query time, we can conclude that, in practical applications, setting both λ_{data} and τ_{data} slightly larger than the frequently queried range can achieve the lowest query time.

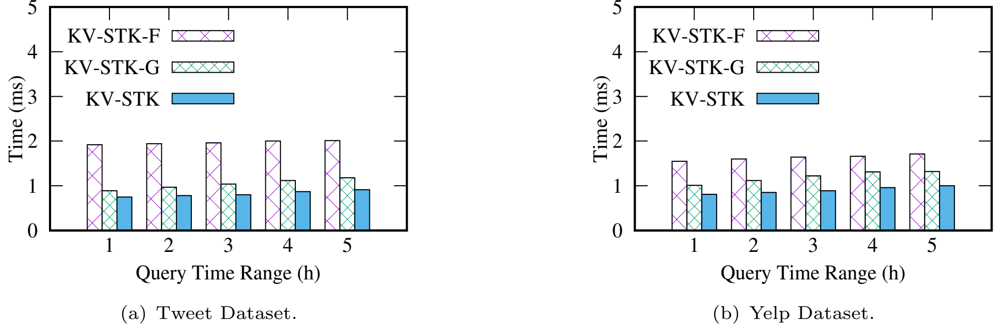
Different Filter Configurations. We study the performance of KV-STK in different InfiniFilter configurations. We adjust its fingerprint length . A smaller fingerprint length means lower space usage, but bigger false positive rate.



(a) Tweet Dataset.

(b) Yelp Dataset.

Fig. 15: Ablation Experiments in Different Query Spatial Ranges.



(a) Tweet Dataset.

(b) Yelp Dataset.

Fig. 16: Ablation Experiments in Different Query Temporal Ranges.

The experimental results are shown in Fig. 14(e). We can observe that as the fingerprint length of InfiniFilter increases, the time consumed by KV-STK decreases continuously. This is because the primary factor affecting the overall system performance is the time required to query KVS-Data. A lower false positive rate results in shorter query times for KVS-Data. Therefore, in practice, we should set a long fingerprint length for InfiniFilter to achieve a low false positive rate.

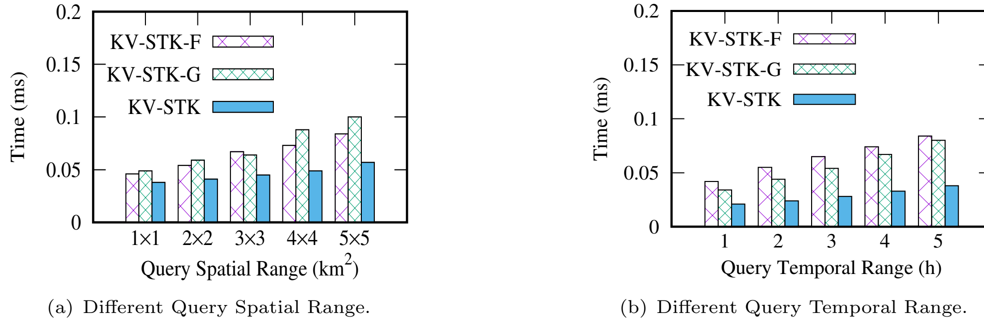
7.5 Ablation Experiments

In order to verify the effectiveness of local filters and the global filter in KV-STK, we conduct a set of ablation experiments with different query spatial ranges and query temporal ranges. We denote KV-STK without the global filter as KV-STK-G, and KV-STK without both the global filter and local filters as KV-STK-F.

From Fig. 15 and Fig. 16, we can observe that for both datasets: 1) Removing the global filter and local filters at the same time (KV-STK-F) seriously affects the query efficiency. This is because local filters are used to remove the spatio-temporal regions that do not meet the constraints, while the global filter is used to reduce the number of invalid accessed local filters. 2) As the query spatial range or temporal

Table 4: Performance on Insertion (Tweet).

Method	BDIA	JUST	KV-STK
Insertion Time (s)	54.962	24.854	31.841



(a) Different Query Spatial Range.

(b) Different Query Temporal Range.

Fig. 17: Performance on Redis (Tweet).

range expands, this impact becomes increasingly significant, because the growth rate of the spatio-temporal regions satisfying keyword constraints is smaller than that of the query range, which proves the powerful pruning capability of the filters. 3) Local filters produce a much greater positive effect than the global filter, since local filters can avoid massive data retrievals, while the global filter can only avoid unnecessary filters accessing. However, we can always see the improvement of the global filter for all query spatial ranges and temporal ranges.

7.6 Performance on Insertion

We also conduct a set of experiments on data insertion. One million pieces of data in Tweet are inserted into the three frameworks, respectively, with every 1,000 pieces of data being inserted in a batch. We measure the total time required to insert all the data into these systems, including the time for key generation, filter insertion, and key-value store insertion. In our experiments, key generation time accounts for only about 1% of the total time, while the main overhead being the time taken to insert data into the key-value store. The experimental results are shown in Table 4. JUST is the fastest among them, because it does not need to update any filter. However, its query efficiency is much slower than KV-STK. The insertion time of BDIA is about 1.73 times that of our solution. Because BDIA may encounter the problem of hotspot. Besides, BDIA should update the filters in disks, which triggers many disk I/Os.

7.7 Extended Experiment: Deployed on Redis

There are a wide range of key-value stores. To verify the wide applications of KV-STK for various key-value stores, we rigorously assess the performance of KV-STK on Redis. We implement KV-STK and its variants, i.e., KV-STK-G and KV-STK-F,

and evaluate their query time efficiency across various temporal and spatial ranges. We do not compare JUST and BDIA since they are tailored for HBase. As illustrated in Fig. 17, all methods exhibit increasing query time as the query spatial range or temporal range expands, but KV-STK always performs the best, which demonstrates the effectiveness of filters. On average, KV-STK reduces query time by approximately 45% compared to KV-STK-F and by about 40% compared to KV-STK-G. Sometimes KV-STK-F (without any filter) performs better than KV-STK-G (without only the global filter) in Fig. 17(a), since the time of accessing the local filters in Redis differs little with that of loading data from Redis. However, our findings highlight the pivotal role of the global filter, because it can reduce the probability of accessing the local filters in Redis. When both local filters and data are stored in Redis, the proportion of time for loading local filters from Redis becomes critical. The global filter can avoid many invalid Redis accesses for the local filters. This set of experiments not only confirms the effectiveness of our framework on Redis but also demonstrates the essential contribution of the global filter to maintaining high efficiency across diverse query conditions.

8 Conclusion and Future Works

This paper proposes an efficient and scalable key-value based solution named KV-STK for spatio-temporal keyword queries. In particular, we provide the first attempt by combining in-memory index with on-disk index to efficiently support STK queries. Extensive experiments using large datasets verify the powerful performance of KV-STK. Specifically, in the fastest case, the query time of KV-STK is only 12% of JUST and 25% of BDIA, respectively. Moreover, KV-STK can support an infinite amount of data with limited memory usage theoretically. We implement KV-STK on HBase and Redis, both of which have excellent query efficiency. Considering that kNN queries also play a significant role in many scenarios, extending our work to kNN spatio-temporal keyword queries will be left as a future research.

Acknowledgments. This paper is supported by the National Natural Science Foundation of China (62202070, 62322601, 62172066) and China Postdoctoral Science Foundation (2022M720567).

References

[1] Chen, X., Zhang, C., Shi, Z., Xiao, W.: Spatio-temporal keywords queries in hbase. Big Data & Information Analytics **1**(1), 81–91 (2015)

[2] Shepherd, T.S.: 23 Essential Twitter (X) Statistics You Need to Know in 2024. https://thesocialshepherd.com/blog/twitter-statistics (2024)

[3] BLOG, T.S.B.: Yelp Demographics: How Many People Use Yelp In 2024? https://thesmallbusinessblog.net/how-many-people-use-yelp/ (2024)

[4] Zhao, J., Gao, Y., Chen, G., Chen, R.: Towards efficient framework for time-aware spatial keyword queries on road networks. ACM Transactions on Information

Systems (TOIS) **36**(3), 1–48 (2017)

[5] Liu, X., Wan, C., Xiong, N.N., Liu, D., Liao, G., Deng, S.: What happened then and there: Top-k spatio-temporal keyword query. Information Sciences **453**, 281–301 (2018)

[6] Chen, L., Shang, S.: Region-based message exploration over spatio-temporal data streams. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 873–880 (2019)

[7] Huang, X., Gao, Y., Gao, X., Chen, G.: Netr-tree: An efficient framework for social-based time-aware spatial keyword query. In: 2021 IEEE International Conference on Web Services (ICWS), pp. 198–207 (2021). IEEE

[8] Hoang-Vu, T.-A., Vo, H.T., Freire, J.: A unified index for spatio-temporal keyword queries. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 135–144 (2016)

[9] Huang, Q., Du, J., Yan, G., Yang, Y., Wei, Q.: Privacy-preserving spatio-temporal keyword search for outsourced location-based services. IEEE Transactions on Services Computing **15**(6), 3443–3456 (2021)

[10] Guan, Y., Lu, R., Zheng, Y., Zhang, S., Shao, J., Wei, G.: Toward privacy-preserving cybertwin-based spatiotemporal keyword query for its in 6g era. IEEE Internet of Things Journal **8**(22), 16243–16255 (2021)

[11] Luo, C., Wang, P., Li, Y., Zheng, B., Li, G.: Efficient time-interval augmented spatial keyword queries on road networks. Information Sciences **593**, 505–526 (2022)

[12] Andrade, D.C., Rocha-Junior, J.B., Costa, D.G.: Efficient processing of spatio-temporal-textual queries. In: Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web, pp. 165–172 (2017)

[13] Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57 (1984)

[14] Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta informatica **4**, 1–9 (1974)

[15] Almaslukh, A., Magdy, A.: Temporal geo-social personalized search over streaming data. In: Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 189–198 (2019)

[16] Almaslukh, A., Kang, Y., Magdy, A.: Temporal geo-social personalized keyword search over streaming data. ACM Transactions on Spatial Algorithms and

Systems (TSAS) **7**(4), 1–28 (2021)

[17] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)

[18] Nishio, S., Amagata, D., Hara, T.: Lamps: Location-aware moving top-k pub/sub. IEEE Transactions on Knowledge and Data Engineering **34**(1), 352–364 (2020)

[19] Kalamatianos, G., Fakas, G.J., Mamoulis, N.: Proportionality in spatial keyword search. In: Proceedings of the 2021 International Conference on Management of Data, pp. 885–897 (2021)

[20] Dong, Y., Xiao, C., Chen, H., Yu, J.X., Takeoka, K., Oyamada, M., Kitagawa, H.: Continuous top-k spatial–keyword search on dynamic objects. The VLDB Journal **30**(2), 141–161 (2021)

[21] Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. Proceedings of the VLDB Endowment **2**(1), 337–348 (2009)

[22] Bao, J., Li, R., Yi, X., Zheng, Y.: Managing massive trajectories on the cloud. In: Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 1–10 (2016)

[23] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM computing surveys (CSUR) **38**(2), 6 (2006)

[24] Chen, L., Cui, Y., Cong, G., Cao, X.: Sops: A system for efficient processing of spatial-keyword publish/subscribe. Proceedings of the VLDB Endowment **7**(13), 1601–1604 (2014)

[25] Ding, X., Zheng, Y., Wang, Z., Choo, K.-K.R., Jin, H.: A learned spatial textual index for efficient keyword queries. Journal of Intelligent Information Systems **60**(3), 803–827 (2023)

[26] Sheng, Y., Cao, X., Fang, Y., Zhao, K., Qi, J., Cong, G., Zhang, W.: Wisk: A workload-aware learned index for spatial keyword queries. Proceedings of the ACM on Management of Data **1**(2), 1–27 (2023)

[27] Yin, Z., Feng, S., Liu, S., Cong, G., Ong, Y.S., Cui, B.: List: Learning to index spatio-textual data for embedding based spatial keyword queries. arXiv preprint arXiv:2403.07331 (2024)

[28] Ding, J., Nathan, V., Alizadeh, M., Kraska, T.: Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. Proceedings of the VLDB Endowment **14**(2), 74–86 (2020)

[29] Nathan, V., Ding, J., Alizadeh, M., Kraska, T.: Learning multi-dimensional indexes. In: Proceedings of the 2020 ACM SIGMOD International Conference on

Management of Data, pp. 985–1000 (2020)

[30] Anand, A., Bedathur, S., Berberich, K., Schenkel, R.: Efficient temporal keyword search over versioned text. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, pp. 699–708 (2010)

[31] Xia, F., Yu, C., Qian, W., Zhou, A.: Top-k temporal keyword query over social media data. In: Web Technologies and Applications: 18th Asia-Pacific Web Conference, APWeb 2016, Suzhou, China, September 23-25, 2016. Proceedings, Part I, pp. 183–195 (2016). Springer

[32] Chen, L., Cong, G., Cao, X., Tan, K.-L.: Temporal spatial-keyword top-k publish/subscribe. In: 2015 IEEE 31St International Conference on Data Engineering, pp. 255–266 (2015). IEEE

[33] Chen, G., Zhao, J., Gao, Y., Chen, L., Chen, R.: Time-aware boolean spatial keyword queries. IEEE Transactions on Knowledge and Data Engineering **29**(11), 2601–2614 (2017)

[34] Chen, Z., Zhao, T., Liu, W.: Time-aware collective spatial keyword query. Computer Science and Information Systems **18**(3), 1077–1100 (2021)

[35] Ray, S., Nickerson, B.: Temporally relevant parallel top-k spatial keyword search. Journal of Spatial Information Science (24), 115–156 (2022)

[36] Luo, C., Liu, Q., Gao, Y., Chen, L., Wei, Z., Ge, C.: Task: An efficient framework for instant error-tolerant spatial keyword queries on road networks. Proceedings of the VLDB Endowment **16**(10), 2418–2430 (2023)

[37] Arseneau, Y., Gautam, S., Nickerson, B., Ray, S.: Stilt: Unifying spatial, temporal and textual search using a generalized multi-dimensional index. In: Proceedings of the 32nd International Conference on Scientific and Statistical Database Management, pp. 1–12 (2020)

[38] Chen, Z., Chen, L., Cong, G., Jensen, C.S.: Location-and keyword-based querying of geo-textual data: a survey. The VLDB Journal **30**(4), 603–640 (2021)

[39] Ltd., R.: Redis. https://redis.io/ (2024)

[40] Foundation, T.A.S.: Apache HBase. https://hbase.apache.org/ (2024)

[41] Li, R., Wang, R., Liu, J., Yu, Z., He, H., He, T., Ruan, S., Bao, J., Chen, C., Gu, F., *et al.*: Distributed spatio-temporal k nearest neighbors join. In: Proceedings of the 29th International Conference on Advances in Geographic Information Systems, pp. 435–445 (2021)

[42] Li, R., Bao, J., He, H., Ruan, S., He, T., Hong, L., Jiang, Z., Zheng, Y.: Discovering real-time reachable area using trajectory connections. In: Database Systems

for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part II 25, pp. 36–53 (2020). Springer

[43] Dayan, N., Bercea, I., Reviriego, P., Pagh, R.: Infinifilter: Expanding filters to infinity and beyond. Proc. ACM Manag. Data **1**(2) (2023) https://doi.org/10.1145/3589285

[44] Li, R., He, H., Wang, R., Huang, Y., Liu, J., Ruan, S., He, T., Bao, J., Zheng, Y.: Just: Jd urban spatio-temporal data engine. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1558–1569 (2020). IEEE

[45] Li, R., He, H., Wang, R., Ruan, S., He, T., Bao, J., Zhang, J., Hong, L., Zheng, Y.: Trajmesa: A distributed nosql-based trajectory data management system. IEEE Transactions on Knowledge and Data Engineering **35**(1), 1013–1027 (2021)

[46] Li, R., He, H., Wang, R., Ruan, S., Sui, Y., Bao, J., Zheng, Y.: Trajmesa: A distributed nosql storage engine for big trajectory data. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 2002–2005 (2020). IEEE

[47] He, H., Li, R., Ruan, S., He, T., Bao, J., Li, T., Zheng, Y.: Trass: Efficient trajectory similarity search based on key-value data stores. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 2306–2318 (2022). IEEE

[48] He, H., Xu, Z., Li, R., Bao, J., Li, T., Zheng, Y.: Tman: A high-performance trajectory data management system based on key-value stores. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE) (2024). IEEE

[49] Hilbert, D.: Über die stetige abbildung einer linie auf ein flächenstück. Mathematische Annalen **38**, 459–460 (1891)

[50] Wang, H., Dai, H., Li, M., Yu, J., Gu, R., Zheng, J., Chen, G.: Bamboo filters: Make resizing smooth. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 979–991 (2022). IEEE

[51] Zhang, F., Chen, H., Jin, H., Reviriego, P.: The logarithmic dynamic cuckoo filter. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 948–959 (2021). IEEE

[52] Inc., Y.: Yelp Open Dataset. https://www.yelp.com/dataset (2024)

[53] Chen, Z., Cong, G., Zhang, Z., Fuz, T.Z., Chen, L.: Distributed publish/subscribe query processing on the spatio-textual data stream. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 1095–1106 (2017). IEEE