SSTD: A Distributed System on Streaming Spatio-Textual Data

Yue Chen[1], Zhida Chen[1], Gao Cong[1], Ahmed R. Mahmood[2], Walid G. Aref[2]
[1]Nanyang Technological University, Singapore
[2]Purdue University, USA
{yue004, chen0936}@e.ntu.edu.sg, gaocong@ntu.edu.sg, {amahmoo, aref}@cs.purdue.edu

ABSTRACT

Streaming *spatio-textual data* that contains geolocations and textual contents, e.g., geo-tagged tweets, is becoming increasingly available. Users can register *continuous queries* to receive up-to-date results continuously, or pose *snapshot queries* to receive results instantly. The large scale of spatio-textual data streams and huge amounts of queries pose great challenges to the current location-based services, and call for more efficient data management systems. In this paper, we present SSTD (**S**treaming **S**patio-**T**extual **D**ata), a distributed in-memory system supporting both continuous and snapshot queries with spatial, textual, and temporal constraints over data streams. Compared with existing distributed data stream management systems, SSTD has at least three novelty: (1) It supports more types of queries over streamed *spatio-textual data*; (2)SSTD adopts a novel workload partitioning method termed QT (**Q**uad-**T**ext) tree, that utilizes the joint distribution of queries and spatio-textual data to reduce query latency and enhance system throughput. (3) To achieve load balance and robustness, we develop three new workload adjustment methods for SSTD to fit the changes in the distributions of data or queries. Extensive experiments on real-life datasets demonstrate the superior performance of SSTD.

PVLDB Reference Format:
Yue Chen, Zhida Chen, Gao Cong, Ahmed R. Mahmood and Walid G. Aref. SSTD: A Distributed System on Streaming Spatio-Textual Data. *PVLDB*, 13(11): 2284-2296, 2020.
DOI: https://doi.org/10.14778/3407790.3407825

1. INTRODUCTION

With the prevalence of GPS-enabled mobile devices, there emerges massive amounts of data containing both geographical locations and textual contents, also termed spatio-textual data. Location-based services (LBS) are important sources of spatio-textual data, e.g., geo-tagged tweets in Twitter, and geo-tagged posts in Facebook. This data usually arrives at high rates, and can be modelled as a data stream. It conveys valuable information, e.g., trending topics and people's reactions towards them. Users may issue different types of queries to retrieve information of interest. For example, a filmmaker can pose a *snapshot query* to collect opinions of audiences within a city regarding her latest movie. Meanwhile, to get notified of every new comment on her movie from her home city, she can register a *continuous query* over the stream of social media data, e.g., tweets, to receive the up-to-date comments about her movie. These two types of queries are essentially different. A snapshot query is disposable and retrieves historical information already in the system. In contrast, a continuous query resides in the system after being registered, and will not be deleted until it expires or the user deregisters it. During its lifetime, new results satisfying the query will be reported. Both query types have many applications, e.g., social marketing, event detection and sentiment analysis.

It is challenging to develop a distributed stream processing system that supports both snapshot and continuous queries over a large scale of spatio-textual data with low latency and high throughput. First, the fast arrival speed of streaming spatio-textual data imposes high demands on the system performance. For example, Twitter receives up to 500 million tweets per day [27] and Foursquare receives over 9 million check-ins per day [23]. This calls for a distributed solution with efficient and effective workload partitioning and workload adjustment methods. Second, there exist notable differences in the mechanisms for processing snapshot vs. continuous queries. Efficient processing of snapshot queries requires effective index structures of the streaming objects. In contrast, efficient processing of continuous queries entails indexing the queries as well to continuously produce the query results over streaming objects efficiently. This calls for a careful design of the system architecture to seamlessly process snapshot queries and continuous queries.

There exist distributed systems for spatio-textual data streams [6, 16, 17] as well as distributed systems for spatial data [1,8,13,18,21,24,26], which are both related to our work. They either do not provide supports for both snapshot and continuous queries, or fail to satisfy the system performance requirements on low latency and high throughput for spatio-textual data streams.

In this paper, we present SSTD, a new distributed system for processing streaming spatio-textual data. SSTD supports more types of snapshot and continuous queries than previous spatio-textual streaming systems. We develop algorithms to handle various types of queries in a distributed

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407825

setting and support querying over large amounts of streaming spatio-textual data with low latency and high throughput. We propose a novel global index structure termed the QT-tree that is employed by multiple routers (dispatchers) to achieve effective workload partitioning for both streamed objects and queries while minimizing the total workload.

To support workload partitioning and workload balancing while minimizing the total workload in the QT-tree, we develop an effective cost model to capture the characteristics of the workload comprising multiple types of queries. Moreover, we develop three workload adjustment methods to handle the workload imbalance problem.

Our contributions can be summarized as follows:

- We develop SSTD, a distributed system that supports both snapshot and continuous queries over spatio-textual data streams.
- We design a new global index structure termed the QT-tree for workload partitioning while minimizing the total workload that utilizes both the spatial and textual properties of the data. For QT-tree, we design an effective cost model to capture the workload that is composed of various types of queries.
- To cope with the load imbalance incurred by workload changes, we design three workload adjustment methods to recover from load imbalance. Our workload adjustment mechanisms outperform existing work in three aspects: (a) more effective load balancing, (b) more efficient workload adjustment, and (c) guaranteed correct query results during workload adjustment.
- We implement SSTD, and deploy it on Amazon EC2. We conduct extensive experiments over real-life datasets to evaluate the performance of SSTD. Results demonstrate that SSTD outperforms existing systems with respect to latency, throughput, and scalability.

2. RELATED WORK

Querying spatio-textual data streams. There exists a lot of work on querying spatio-textul data streams. These proposals usually focus on a specific type of query. Some systems [3, 6, 12, 15, 22] focus on processing continuous queries with both spatial and keyword constraints. They aim to develop efficient indexing structures to organize the continuous queries to efficiently match the new spatio-textual objects with the registered continuous queries. There also exist proposals focusing on the top-k frequent term query [5, 20] that returns the top-k terms with the highest frequency in a region. These proposals focus on developing specialized index structures to efficiently compute the term frequencies.

These index structures have been developed to handle a specific type of spatial keyword queries. It would be difficult to generalize them to handle other types of queries. Worse still, they cannot be extended to act as a global index that needs to meet the following requirements: (1) effective workload distribution while minimizing the total workload, which is not considered by these indexes; (2) being efficiently updated over the new streaming data; however, these indexes are complex and expensive to maintain. Details will be discussed in Section 4. In contrast, SSTD may utilize these index structures as local indexes for processing queries, which in this sense, are complementary to our work.

Distributed spatial systems. Parallel Secondo [13], MD-HBase [18], Hadoop-GIS [1] and SpatialHadoop [8] are disk-based spatial analysis systems. Parallel Secondo [13] combines Hadoop with Secondo that is an extensible database system that supports non-standard data types e.g., spatial data. MD-HBase [18] extends HBase with multi-dimensional index structures (e.g., kd tree). Hadoop-GIS [1] extends Hive to support spatial indexes. SpatialHadoop [8] extends Hadoop to provide native support for spatial data. It partitions the spatial data using an R-tree or a grid index.

Simba [24], GeoSpark [26] and LocationSpark [21] are distributed in-memory spatial data analysis systems. Simba [24] extends Spark with SQL-like query language to express spatial queries and provides native support for spatial data. It uses an R-tree-based partitioning strategy to partition the spatial data. GeoSpark [26] is also based on Spark. It applies two spatial index structures (quadtree and R-tree) to support spatial join, range query and kNN query. LocationSpark [21] extends Spark with a query scheduler and local query executors.

These disk-based and in-memory systems focus on querying static spatial data but do not handle continuously arriving and large scale data streams. We discuss them in two categories as follows. (1) R-tree-based global index for workload distribution [1, 8, 24]. These are not easily adaptable to spatio-textual data streams for two reasons: (a) They are not suitable for high update rates. To distribute new queries and objects, the global index needs to invoke frequent updates of the MBRs of R-tree nodes due to the insertions of the arriving spatio-textual objects, which leads to poor performance of the global index. (b) Multiple routers undertake the role of global index layer, each maintaining a copy of the index. Inserting new streaming objects into R-tree will result in inconsistency of the global index on different routers. To solve this, we either do costly communication and synchronization in response to updates on a global index, or send each query to all routers. Either solution is infeasible and will result in very poor performance for handling queries. (2) Other spatial index based partitioning strategies, e.g., grid [8, 26], kd-tree or quadtree. They do not suffer from the two aforementioned issues of R-tree when employed as global index for spatio-textual data streams. However, both categories of global indexes do not have a cost model for various types of spatio-textual queries as does SSTD. Since the expected workload of SSTD includes processing many queries with textual constraints, these partitioning strategies are not effective.

Additionally, Cruncher [19] is a demonstration system that supports continuous and snapshot spatial queries over spatial data. Cruncher uses kd-tree based index that is based on AQWA [2]. Cruncher does not consider the textual property of data, making it inefficient for processing queries with textual constraints.

Distributed systems for spatio-textual data streams. Closest to our system SSTD are Tornado [16,17] and PS^2 [6], which are distributed systems supporting continuous range keyword queries only over spatio-textual data streams. We proceed to review the two key components of the two systems: (1) Workload distribution. Tornado uses an augmented grid structure (called A-Grid) that overlays spatial partitions from a kd-tree on top of the cells of A-Grid as the global index for partitioning the workload comprising continuous range keyword queries. Each spatial partition in

the A-Grid contains a summary of all keywords contained in continuous queries falling in the partition, which will be used together with spatial partitions to decide how to route a new query. PS^2 [6] adopts the global kd^t index structure for partitioning the workload. kd^t extends kd-tree by allowing a node to be split based on the spatial property or the textual property, both of which are used for routing a new query. However, both A-Grid and kd^t-tree are developed for processing one type of continuous queries, and their cost models do not consider a dynamic workload that involves object insertions and various types of queries including snapshot queries as does SSTD. Their partitioning strategies cannot reflect well the characteristics of the targeted workload of SSTD.

(2) Workload adjustment. To adapt to changes in the workload, Tornado [16, 17] uses split and merge operations. It splits an overloaded partition into two partitions and then merges two lightly-loaded partitions into one partition. In addition, Tornado adopts workload shift operations that transfer queries among neighbor partitions when no merging is possible. PS^2 [6] solves the load imbalance problem by transferring continuous queries from the most loaded worker to the least loaded worker. The workload adjustment strategies of Tornado and PS^2 do not consider the workload caused by snapshot queries, which makes them inapplicable to our adjustment problem. In contrast, we propose for SSTD three workload adjustment methods that work together to rebalance the workload while guaranteeing correct query results during workload adjustment.

3. SYSTEM OVERVIEW

In this section, we present the definitions of supported queries and describe the system architecture of SSTD.

3.1 Supported Queries

A spatio-textual object is in the form of $o =< \phi, \rho, t >$, where ϕ is a set of keywords, ρ is the object's geographical location (i.e., latitude and longitude), and t is the creation timestamp of object o. We next define the query types supported in SSTD.

DEFINITION 1. ***Range keyword query:*** *A range keyword query is in the form of $q =< \phi, r, \tau >$, where ϕ is a set of keywords, r is a rectangular region, and τ is a time duration.*

- *For the snapshot version, q returns objects that contain $q.\phi$, are located inside $q.r$, and that arrive within the last $q.\tau$ time units.*
- *For the continuous version, q keeps running in the system for a duration of $q.\tau$. During the lifetime of q, a newly arrived object is reported as the result of q if it contains $q.\phi$ and is located inside $q.r$.*

DEFINITION 2. ***kNN query:*** *A kNN query is of the form $q =< \phi, \rho, k, \tau >$, where ϕ is a set of keywords, ρ is the geographical location, k is an integer, and τ is a time duration.*

- *For the snapshot version, q returns the k-nearest objects to $q.\rho$ that contain $q.\phi$, and that arrive within the last $q.\tau$ time units.*
- *For the continuous version, q keeps running in the system for a duration of $q.\tau$. During the lifetime of q, it maintains the k-nearest objects to $q.\rho$ and that contain $q.\phi$.*

DEFINITION 3. ***Top-k frequent-term query:*** *A top-k frequent-term query is of the form $q =< r, k, \tau >$, where r is a rectangular region, k is an integer, and τ is a time duration.*

- *For the snapshot version, q returns the top-k most frequent terms appearing in the objects that are located inside $q.r$, and that arrive within the last $q.\tau$ time units.*
- *For the continuous version, different from the last two types of queries, q does not make instant response to the user. Instead, every δ time units, it performs aggregation operations on objects received within a period T, and returns the aggregation information (top-k frequent terms) to the user. Therefore, q has two additional parameters T and δ, where T denotes the sliding window size and δ denotes the step size. Query q keeps running in the system for a duration of $q.\tau$. During the lifetime of q, every δ time units, q reports the top-k most frequent terms appearing in the objects that are located inside $q.r$ and that arrive within the time window $[t_{now} - T, t_{now}]$, where t_{now} denotes the current timestamp.*

The parameter r in range keyword query and top-k frequent term query can also be a circular region. For brevity, in the paper hereafter, we use the terms object and spatio-textual object interchangeably if there is no ambiguity. We use SRQ, SKQ, and STQ to represent the snapshot versions of the range keyword, the kNN, and the top-k frequent-term queries, respectively, and use CRQ, CKQ, and CTQ to refer to the continuous versions of the range keyword, the kNN, and the top-k frequent-term query, respectively.

3.2 System Architecture

SSTD builds on Apache Storm to process large scale streaming spatio-textual data. Storm is a popular distributed platform that provides rich APIs to conduct real-time operations over streaming data. Storm is not optimized for the execution of spatial-keyword queries because it does not have built-in support for spatial or textual primitives.

To support the various types of queries over a large scale of spatio-textual data with low latency and high throughput, SSTD focuses on addressing the following challenges: (1) Efficiently and effectively distributing streaming data and queries across workers. This calls for a global index employed by routers to co-locate relevant data and queries in the same worker to reduce workload. (2) Maintaining load balancing as the workload changes by redistributing workload while guaranteeing the correctness of query execution. (3) The mechanisms for processing snapshot vs. continuous queries are significantly different as discussed in Introduction. Figure 1 illustrates the overall architecture of SSTD.

Routers To partition the workload and facilitate the processing of both snapshot and continuous queries, we design a novel global index structure, termed the QT-tree. QT-tree aims to achieve load balance while minimizing the total workload. To reduce the total workload, ideally spatially close or textually similar objects are assigned to the same worker and co-locate the relevant queries as well. To support this, we develop an effective cost model to capture the characteristics of the workload comprising multiple types of queries. To support high arrival rates of streamed data, the global index is replicated in multiple routers. Once an object or a query is received, a router is selected in a round-robin fashion to handle the object or query, and sends the object

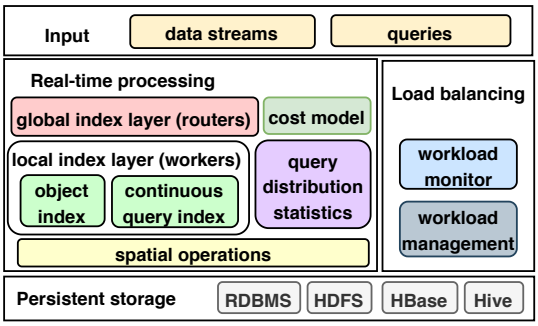


Figure 1: Architecture of SSTD

to a relevant worker or the query to relevant worker(s). We will explain the indexing structure in detail in Section 4.

Workers Each worker (1) maintains a *local* index for objects and indexes for various types of continuous queries, (2) checks the incoming data objects against the continuous queries and updates query results, and (3) checks the incoming snapshot queries over the object index to produce results. SSTD supports 6 kinds of queries. We will discuss their execution strategies in Section 6.

Load balancing Achieving load balancing is essential for excellent performance in a distributed system. A decent distributed stream processing system calls for an effective load-balancing mechanism that can adaptively adjust the workload of different workers to maintain load balance. The load balancing module periodically collects the statistics of objects and queries to estimate the workloads of the workers, and decides whether load imbalance occurs. If that happens, SSTD applies three different methods to change the workload assignment, and adjusts the workload of workers accordingly as to be discussed in Section 5.

Persistent storage Due to the nature that data streams are unbounded, it is impractical for SSTD to always preserve the whole data in memory, which is also unnecessary as users are usually interested in relatively "fresh" data. Therefore, every a period of time, SSTD writes the obsolete data into persistent storage to free memory space. SSTD also supports connections with other peripheral systems, e.g., RDMBS and HDFS for future processing.

Fault tolerance SSTD extends the fault tolerance mechanism of Storm. It has two major daemon processes: Nimbus and Supervisor. Nimbus assumes the responsibilities of scheduling and monitoring the Supervisors. In contrast, the Supervisor takes charge of launching and killing threads that undertake the roles of routers and workers. When a router/worker fails, the Supervisor daemon will restart it. If it continuously fails on startup, Nimbus will create a new router/worker. After recovery, (1) the "new" router will load the QT-tree from disk and continue receiving streamed objects and queries; (2) the "new" worker will load the lost objects from the persistent storage (if they have been persisted already) to support subsequent queries. For the fault tolerance of Nimbus, we maintain multiple Nimbus processes managed by Zookeeper. If the current Nimbus dies, another Nimbus will be elected as the new leader.

4. GLOBAL INDEXING

In SSTD, a global index is deployed on the routers to partition and distribute the workload to the workers, aiming to achieve load balance. When a router receives an object or a query, it looks up the global index, and dispatches to the

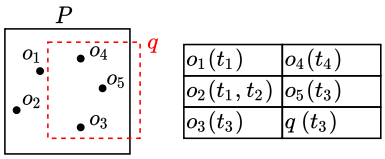


Figure 2: Before partitioning

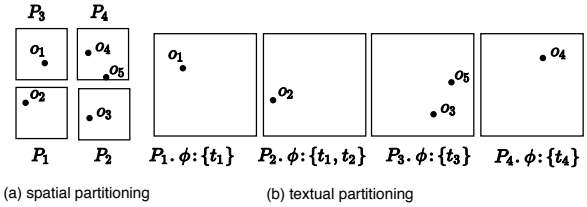


Figure 3: After partitioning

corresponding workers. We present the index construction in Section 4.1, and the index deployment in Section 4.2.

4.1 QT-tree Construction

Requirements on global index As discussed in Related Work, existing global indexes cannot reflect well the characteristics of the target workload of SSTD. We proceed to present the requirements on global index: (1) Optimize the cost. Ideally, each object is stored only once in the system. It reduces communication cost, saves the memory of storing objects, and avoids deduplication overhead in query processing. (2) Achieve load balance among the workers. The global index should assign balanced workload to each worker, as well as adapt and fit the change of workload of workers over time. (3) Minimize the total workload. Intuitively, it will reduce the workload of answering queries if spatially close or textually similar objects are assigned to the same worker and co-locate the relevant queries as well. Furthermore, different ways of workload partitioning will result in different workloads. To illustrate this, Figure 2 gives 5 objects and 1 SRQ together with their textual contents in Region P. If P is handled by a worker, the workload will be visiting 5 objects on P to answer q. If we partition the workload of P utilizing the spatial properties of the objects into 4 workers as in Figure 3(a), then we only need to visit objects in Regions P_2 and P_4 to check if they are results, the total workload is smaller. Alternatively, if we partition P utilizing textual properties of objects as in Figure 3(b), we only need to visit 2 objects in Region P_3 to answer q, which has even smaller total workload. (4) Efficient maintenance. The global index should be maintained efficiently by each router, under the premise that the objects with high arrival rate can be digested and queries can be correctly processed.

QT-tree To address these challenges, we propose the QT-tree, a variant of quadtree that allows a node to be split based on the spatial or textual properties of the objects. Figure 4 gives an example of QT-tree, where node N_2 is partitioned based on the spatial property, and N_4 is partitioned based on the textual property. It aims to minimize the total workload while preserving load balance among all workers. To achieve this goal, in Section 4.1.1, we introduce a cost model to estimate the workload of a region, which is a key component in QT-tree construction. We present the algorithm for constructing the QT-tree in Section 4.1.2.

4.1.1 Cost Model

Existing cost models for querying spatio-textual data focus on SRQ and cannot capture the characteristics of the workload comprising various type of queries. For example, Tornado [17] adopts a cost model called AQWA [2] that

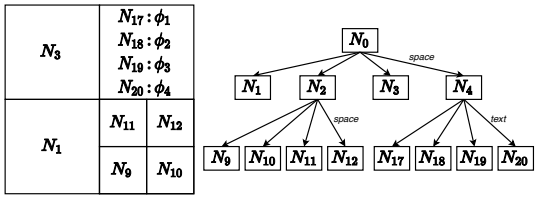


Figure 4: Example of QT-tree

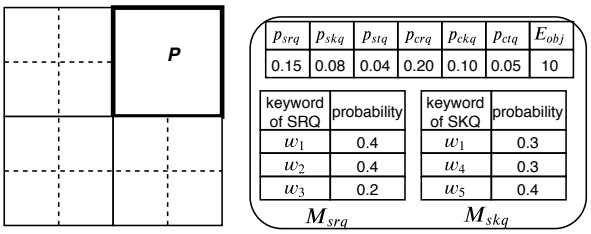


Figure 5: Query Distribution Statistics

defines the workload of a partition $\mathcal{P}$ as $W(\mathcal{P}) = |\mathcal{Q}| \cdot |\mathcal{O}|$, where $|\mathcal{Q}|$ and $|\mathcal{O}|$ denotes the number of queries and objects overlapping with $\mathcal{P}$, respectively. It does not consider the effect of textual dimension. Furthermore, the model cannot characterize the various types of snapshot and continuous queries supported in SSTD.

To this end, we propose a new cost model to better characterise the workload in our system. Before presenting the model, we first introduce the *query distribution statistics* which is required for computation in the cost model.

Query distribution statistics Query distribution statistics indicate the probabilities of each type of query arriving at a particular region and the distribution of query keywords in this region. For convenience, we use a quadtree denoted by J to store these statistics. Figure 5 gives an example of the query distribution statistics. The left part is Quadtree J and the right part is the query distribution statistics on Cell P. Initial statistics can be derived from historical data. Given historical queries and objects, for each Cell P of J, we estimate the probability of each type of query arriving at Cell P by counting the number of each type of query overlapping P and normalizing (e.g., p_{srq} in Figure 5). We also count the average number of objects arriving at P per time unit, denoted by E_{obj}. Furthermore, based on the term frequencies collected from queries that overlap P, we estimate the distribution of query keywords in P for SRQ and SKQ (i.e., M_{srq} and M_{skq}). Additionally, we compute the distribution w.r.t. the number of query keywords and the distribution w.r.t. the length of the query time-window. SSTD updates these statistics periodically based on the received objects and queries.

Cost model for SRQ Given partition $\mathcal{P}$ and objects $\mathcal{O}$ located in $\mathcal{P}$, let $\mathcal{P}.r, \mathcal{P}.\phi$ denote the MBR of $\mathcal{P}$ and the set of keywords extracted from $\mathcal{O}$, respectively. When an SRQ q arrives, $\mathcal{P}$ incurs workload if: (1) $\mathcal{P}.r$ overlaps $q.r$; and (2) $\mathcal{P}.\phi$ covers $q.\phi$. Thus we estimate the workload of $\mathcal{P}$ under an SRQ q as

$$W_{srq}(\mathcal{P}) = Pr(q.r \cap \mathcal{P}.r) \cdot Pr(q.\phi \subseteq \mathcal{P}.\phi \,|\, p.r \cap \mathcal{P}.r) \cdot |\mathcal{O}|.$$

To compute W_{srq}, we find the same cell as $\mathcal{P}$ from quadtree J, and use p_{srq} (see Figure 5) to estimate the probability that they overlap, i.e., $Pr(q.r \cap \mathcal{P}.r)$. In addition, we assume for any SRQ q arriving at $\mathcal{P}$, each query keyword of it is independently drawn from M_{srq} (Figure 5). Under condition that q overlaps $\mathcal{P}$, each keyword of q is contained by $\mathcal{P}.\phi$ with probability $H_{srq} = \sum_{w \in M_{srq} \cap \mathcal{P}.\phi} p_w$, where p_w is the probability of w in distribution M_{srq}. Then we have conditional probability

$$Pr(q.\phi \subseteq \mathcal{P}.\phi \,|\, p.r \cap \mathcal{P}.r) = \sum_l Pr(|q.\phi| = l) \cdot (H_{srq})^l, \quad (1)$$

where l is the number of keywords in q. The probability wrt the number of query keywords $Pr(|q.\phi| = l)$ can be obtained from query distribution statistics.

Cost model for SKQ When an SKQ q arrives, $\mathcal{P}$ incurs workload only if: (1) the minimum distance between $\mathcal{P}$ and q (denoted by $d(q, \mathcal{P})$) is smaller than the distance between q and its kth result (denoted by $d(q, q.R_k)$); (2) $\mathcal{P}.\phi$ covers $q.\phi$. Thus we estimate the workload of $\mathcal{P}$ under an SKQ as

$$\begin{aligned} W_{skq}(\mathcal{P}) =& Pr(d(q,\mathcal{P}) < d(q, q.R_k)) \cdot \\ & Pr(q.\phi \subseteq \mathcal{P}.\phi \,|\, d(q,\mathcal{P}) < d(q, q.R_k)) \cdot |\mathcal{O}|. \end{aligned}$$

To compute W_{skq}, we find the same cell as $\mathcal{P}$ from J. As p_{skq} (see Figure 5) denotes the probability an SKQ accessing $\mathcal{P}$, we use it to estimate the probability $\mathcal{P}$ overlapping the search region of an SKQ, i.e., $Pr(d(q,\mathcal{P}) < d(q, q.R_k))$. Based on the same assumption as SRQ, we can compute H_{skq} and $Pr(q.\phi \subseteq \mathcal{P}.\phi \,|\, d(q,\mathcal{P}) < d(q, q.R_k))$ in the same way (i.e., replace M_{srq} by M_{skq}).

Cost model for STQ When an STQ q arrives, $\mathcal{P}$ incurs workload if $\mathcal{P}.r$ overlaps $q.r$. The workload of $\mathcal{P}$ under an STQ q is estimated by

$$W_{stq}(\mathcal{P}) = Pr(q.r \cap \mathcal{P}.r) \cdot |\mathcal{O}|.$$

Similarly we use p_{stq} in Figure 5 to estimate the probability that they overlap, i.e., $Pr(q.r \cap \mathcal{P}.r)$.

Cost model for Continuous Query For continuous queries, we define their workload since they check the newly arrived objects during their lifetime in the system.

When a CRQ q arrives, if $q.r$ overlaps $\mathcal{P}.r$, then for each incoming stream object o in $\mathcal{P}$ during the lifetime of q, we need to check if o updates the results of q. Thus we estimate the workload of $\mathcal{P}$ under a CRQ as

$$W_{crq}(\mathcal{P}) = Pr(q.r \cap \mathcal{P}.r) \cdot \mathbb{E}_{\mathcal{P},\mathcal{O}}.$$

where $Pr(q.r \cap \mathcal{P}.r)$ is estimated by p_{crq} in Figure 5, and $\mathbb{E}_{\mathcal{P},\mathcal{O}}$ is the expected number of newly arrived objects in $\mathcal{P}$ during the lifetime of q, and is estimated by $\mathbb{E}_{\mathcal{P},\mathcal{O}} = \sum_T Pr(q.\tau = T) \cdot T \cdot \mathbb{E}_{\mathcal{P}}$, where T is the time window length of q, and $\mathbb{E}_{\mathcal{P}}$ is the average number of objects arriving at $\mathcal{P}$ per time unit. Both $Pr(q.\tau = T)$ and $\mathbb{E}_{\mathcal{P}}$ (E_{obj} in Figure 5) can be retrieved from the query distribution statistics.

For a CKQ q, if $\mathcal{P}.r$ overlaps the search region of q, then for each incoming object o in $\mathcal{P}$, we need to check if o updates the results of q. Thus we estimate the workload of $\mathcal{P}$ under a CKQ as $W_{ckq}(\mathcal{P}) = Pr(d(q,\mathcal{P}) < d(q, q.R_k)) \cdot \mathbb{E}_{\mathcal{P},\mathcal{O}}$, where the first term is estimated by p_{ckq} in Figure 5.

For a CTQ q, if $q.r$ overlaps $\mathcal{P}.r$, then for each incoming object o in $\mathcal{P}$ during q's lifetime, we need to check if o updates the term-frequency tables of q. Thus we estimate $\mathcal{P}$'s workload under a CTQ as $W_{ctq}(\mathcal{P}) = Pr(q.r \cap \mathcal{P}.r) \cdot \mathbb{E}_{\mathcal{P},\mathcal{O}}$.

By summing up the workload of all types of queries, we compute the workload of a partition $\mathcal{P}$ under a random query using Equation 2.

$$\begin{aligned} W(\mathcal{P}) =& Pr(srq) \cdot W_{srq}(\mathcal{P}) + Pr(skq) \cdot W_{skq}(\mathcal{P}) + \\ & Pr(stq) \cdot W_{stq}(\mathcal{P}) + Pr(crq) \cdot W_{crq}(\mathcal{P}) + \\ & Pr(ckq) \cdot W_{ckq}(\mathcal{P}) + Pr(ctq) \cdot W_{ctq}(\mathcal{P}), \end{aligned} \quad (2)$$

where $Pr(srq)$, $Pr(skq)$, $Pr(stq)$, $Pr(crq)$, $Pr(ckq)$ and $Pr(ctq)$ represent the probability that the system receives a SRQ, SKQ, STQ, CRQ, CKQ and CTQ, respectively. They can be obtained from query distribution statistics.

4.1.2 QT-tree Construction

Taking a set of historical queries $\mathcal{Q}$ and objects $\mathcal{O}$ as input, we build the QT-tree offline. The construction involves three steps to be presented below.

Generating query distribution statistics Given $\mathcal{Q}$ and $\mathcal{O}$, SSTD computes query distribution statistics as discussed previously, which is denoted by J, and will be used for computing the cost model.

Constructing QT-tree SSTD takes as input $\mathcal{O}$ and utilizes the cost model to build the QT-tree. Initially, we construct a root node that covers the whole spatial range, and contains all objects $\mathcal{O}$. Then, we recursively split the leaf nodes until the number of leaf nodes reaches a threshold θ. We design two partitioning methods for splitting a node: *Spatial* and *Textual* Partitioning. (1) **Spatial Partitioning**. It splits a node into 4 child nodes of equal size, and divides the objects on the parent node among the child nodes based on the locations of the objects, as illustrated in Figure 3(a). (2) **Textual Partitioning**. This method aims to store the textually similar objects in the same child node as in Figure 3(b). It splits a node into 4 child nodes, each with the same MBR as the parent node's MBR, and the objects on the parent node are assigned to the child nodes based on the textual contents of the objects. Specifically, for each object o on the parent node, we compute the *workload increment* using Equation 2 for each child node if this child takes o, and assign o to the child node that has the smallest workload increment. To compute the workload increment efficiently when attempting to add an object to a child node, for each child node, we maintain a *keyword list* that stores the keywords in its objects, and the associated H_{srq}, H_{skq} as defined by Equation 1. To compute the increment due to o, for a child node, we check if o brings new keywords into the node's keyword list and use their values in the query distribution statistics M_{srq}, M_{skq} to incrementally update H_{srq}, H_{skq}, and thus compute the increments for W_{srq} and W_{skq}. After assigning o to a child node, SSTD will update the node's keyword list and the associated H_{srq}, H_{skq} as needed. Since the number of keywords in an object is usually small, assigning o from the parent node to a child node takes constant time. When splitting a node, we consider both partitioning methods, and choose the partitioning method that incurs smaller amount of total workload.

Algorithm 1 lists the pseudo code. It takes as input a set of objects $\mathcal{O}$, a threshold θ for the number of leaf nodes and query distribution statistics J. First, we initialize a root node n_r of the QT-tree using $\mathcal{O}$ that covers the whole space (Line 1). Then, we initialize a max-heap H, which stores the nodes to be split in descending order of their workload, and push n_r into H (Line 2–3). If the number of leaf nodes is smaller than θ, we pop up a node from H, compute the workload of both partitioning methods, and adopt the one with lower workload. We keep partitioning nodes until the number of leaf nodes reaches θ (Line 4–13). In the end, we return, as output, the root node of the QT-tree (Line 14).

Remark. The time complexity of Algorithm 1 is $O(|\mathcal{O}|\log(\theta))$. As spatial partitioning and textual partitioning both take linear time, in each layer of the QT-tree, we visit $O(|\mathcal{O}|)$ objects, and the height of QT-tree is given by $\log(\theta)$, and thus the total complexity is $O(|\mathcal{O}|\log(\theta))$. In our experiments, we study the effect of θ.

Algorithm 1: QTtreeConstruction

```
   Input  : Objects O, Threshold θ, Query
            distribution statistics J
   Output: The root node of QT-tree
1  n_r ← Initialize a root node of the QT-tree using O;
2  Initialize a max-heap H;
3  H.push(n_r);
4  while the number of leaf nodes is smaller than θ do
5      n_p ← H.pop();
6      children_s ← spatialPartitioning(n_p);
7      children_t ← textualPartitioning(n_p);
8      if ∑W(children_s) ≤ ∑W(children_t) then
9          Set children_s to be child nodes of n_p;
10         Push each node in children_s into H;
11     else
12         Set children_t to be child nodes of n_p;
13         Push each node in children_t into H;
14 return n_r
```

Allocating the leaf nodes of the QT-tree to workers The objective of the allocation is to achieve load balance among workers. This problem can be reduced from the multi-way partitioning problem: Given a finite set of n real numbers, find the optimal solution to divide the set into k subsets such that the difference between the maximum subset sum and the minimum subset sum is minimized. Since the multi-way partitioning problem is NP-complete [10, 11], we apply a heuristic algorithm, namely the Karmarkar-Karp differencing algorithm [10], to solve SSTD's leaf-nodes allocation problem.

At the end of construction, SSTD writes the QT-tree to disk. Note that SSTD does not store the objects at each node of the QT-tree. SSTD stores the *keyword list* for nodes whose parent node is textually partitioned, and the keyword list will be used for routing objects. The QT-tree is lightweight and takes only dozens of KBs.

4.2 QT-tree Deployment

Each router in SSTD loads a copy of the QT-tree from disk for routing objects and queries, and receives streamed objects and queries in a round-robin fashion.

Routing Objects When a router receives an object, say o, the router traverses its QT-tree to assign o to a leaf node's corresponding worker as follows: (1) When a non-leaf node is partitioned by spatial properties, the router visits the child node that covers $o.\rho$; (2) When a node is partitioned by textual properties, the router assigns o to the child node that has the smallest workload increment after inclusion of o based on the cost model (i.e., in the same way as assigning objects using textual partitioning when building the QT-tree. It takes constant time, and updates the *keyword list* of that child node by the keywords of o. The router sends o to the worker corresponding to o's leaf node. Note that we do not maintain objects in the QT-tree.

Routing Queries When a router receives a query q with a region (SRQ, STQ, CRQ, CTQ), it traverses the QT-tree and finds out all leaf nodes that overlap q spatially, and sends q to the corresponding workers. As SKQ and

CKQ queries have no regions, SSTD adopts different routing strategies for them (See Section 6.2 for SKQ's routing strategy). For CKQ q, since each leaf node may contribute to the result of q, SSTD routes q to all workers. In routing queries, (1) we do not update the QT-tree; (2) we do not check query keywords. The reason is that SSTD updates the *keyword lists* of some nodes when routing objects, and thus each router may maintain slightly different *keyword lists* within a node. For correctness, we only check the spatial constraints when routing queries; only the workers will do pruning using the textual constraints.

QT-tree Maintenance Each router maintains its copy of the QT-tree by updating its keyword lists when routing objects. Although copies of the QT-tree are slightly different across routers, it does not affect the correct execution of queries in SSTD, and does not synchronize the QT-tree copies. However, when rebalancing load, we may need to change the QT-tree (see Section 5.3), and SSTD will synchronize the QT-tree among all routers in this case.

Remark. Since routing an object from a non-leaf node to a child node takes constant time, the complexity of routing an object depends on the height of the QT-tree, i.e., $O(\log(\theta))$, where θ is the number of leaf nodes in QT-tree. SSTD takes $O(a\log(\theta))$ time to route a query q, where a is the average number leaf nodes that overlap q.

5. LOAD BALANCING

As the data and query distribution evolve over time, the workload assigned to the workers will change gradually, which may result in load imbalance. To avoid system performance degradation, SSTD needs to conduct workload adjustments to rebalance the load. In this section, we introduce the definition of load imbalance, and then present two types of workload adjustment strategies: workload reallocation and QT-tree adjustment.

5.1 Overview

SSTD periodically collects the statistics of each leaf node from the workers and computes their workloads to detect load imbalance. We define load imbalance as follows.

Definition 4. ***Load Imbalance:*** *The system has load imbalance if $\frac{L_{max}}{L_{min}} > \lambda$, where L_{max} is the maximum load of workers, L_{min} is the minimum load of workers, and λ is a predefined threshold. Both L_{max} and L_{min} are computed using Eqn 2.*

When load imbalance occurs, SSTD conducts workload adjustments to recover load balance. Ideally, the workload adjustment procedure can meet the following requirements: (1) SSTD can recover load balance after workload adjustment, (2) workload adjustment should be conducted efficiently, and (3) query results should be correct during workload adjustment.

To meet these requirements, we propose two strategies for workload adjustment in SSTD. (1) Workload reallocation: Aims to change the leaf node allocation scheme that shuffles the leaf nodes and reallocates them to the workers. (2) QT-tree adjustment: Aims to update the QT-tree's structure, and then allocate the updated leaf nodes to the workers. Workload reallocation performs workload adjustment efficiently, and incurs small network cost. In many cases, it can recover load balance effectively. When it cannot balance the load, it is probably because the current QT-tree does not well characterize the workload. Thus, SSTD switches to the QT-tree adjustment method that updates the structure of the QT-tree, and reallocates the leaf nodes. The adjustment is conducted by one router, and will propagate the updated QT-tree to the other routers after completion. Next, we present these two workload adjustment strategies.

5.2 Workload Reallocation

When load imbalance is detected, workload reallocation will shuffle the leaf nodes, and reallocate them to the workers, aiming to regain the load balance. We illustrate the idea with an example. Assume that there are 3 workers (W_1, W_2, W_3) and 7 leaf nodes $(\mathcal{P}_1, \cdots, \mathcal{P}_7)$, current allocations are: $W_1(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_4)$, $W_2(\mathcal{P}_3, \mathcal{P}_5)$, $W_3(\mathcal{P}_6, \mathcal{P}_7)$. To conduct workload reallocation, we first apply the Karmarkar-Karp algorithm to split $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_7\}$ into 3 subsets: $X_1(\mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_6)$, $X_2(\mathcal{P}_2, \mathcal{P}_5)$, $X_3(\mathcal{P}_4, \mathcal{P}_7)$.

Next, we want to find the optimal assignment with minimum transfer cost to assign X_1, X_2, and X_3 to W_1, W_2, and W_3. During workload reallocation, we only transfer continuous queries between workers rather than objects. This is beneficial for reducing the network cost because the number of objects is much larger than the number of continuous queries. This design choice will result in a new challenge for handling snapshot queries to be discussed later. We proceed to define the transfer cost. Suppose we assign X_2 to W_1, X_1 to W_2 and X_3 to W_3, then we need to transfer $\mathcal{P}_5$ to W_1, $\mathcal{P}_1$ and $\mathcal{P}_6$ to W_2, and $\mathcal{P}_4$ to W_3. Therefore, the total cost invoked by workload reallocation is $\sum_{i\in\{1,4,5,6\}} |\mathcal{P}_i.\mathcal{Q}|$, where $\mathcal{P}_i.\mathcal{Q}$ is the set of continuous queries in $\mathcal{P}_i$. The problem of finding the optimal assignment is a combinatorial optimization problem termed the *assignment problem*. We apply the state-of-the-art *Hungarian algorithm* [9] to solve this problem in $O(n^3)$ time. We denote this adjustment method that involves all workers by **FM**.

It is not always necessary to shuffle all the leaf nodes. To reduce the network cost, we propose an alternative method which shuffles and reallocates the leaf nodes of only two workers: the most loaded worker and the least loaded worker. We term this method **PM**.

A challenge caused by our design of transferring continuous queries only is that this scheme will cause incomplete results for some snapshot queries that arrive after the workload reallocation. To illustrate this, let t be the time when workload reallocation is completed, and $\mathcal{P}$ be a leaf node that is transferred from worker W to W'. When the system receives a snapshot query q requesting to access $\mathcal{P}$ at time t' $(t' > t)$, based on the up-to-date mapping table (i.e., a hash table storing the mapping from leaf node id to worker id), the router will send q to W'. If q asks for historical objects that arrive at the system before t, then sending q to W' will lead to incomplete results as W' only stores the objects arriving after t. To address this issue, the router maintains different versions of the leaf node mapping table and each version is associated with a validation timestamp. When receiving a snapshot query q, the router checks these mapping tables and finds the workers that may contain the result objects. Since the workload reallocation is conducted infrequently, the router only maintains a small number of mapping tables and in most cases only the up-to-date mapping table is used for disseminating the query.

5.3 QT-tree Adjustment

In the case when workload reallocation fails to rebalance the load, this implies that the current QT-tree does not well distribute the workload, e.g., some leaf nodes are heavily loaded while others are lightly loaded. To rebalance the load, we propose to update the structure of current QT-tree through invoking two operations iteratively: splitting the heavily loaded leaf nodes, and merging the lightly loaded leaf nodes. After the QT-tree is updated, we invoke **FM** on the updated QT-tree to perform workload reallocation. We denote this strategy as **FG**. We proceed to present the two operations and the algorithm for updating QT-tree.

Split Operation: If we apply the spatial or textual partitioning method as described in Section 4.1.2, we need to retrieve the detailed information about objects from the workers, which will result in significant network overhead. Instead, we propose a heuristic algorithm to determine which partitioning method to use.

As mentioned in Section 4.1.1, we use a quadtree J to store the query distribution statistics. Suppose we want to split a leaf node $\mathcal{P}$, we first find the smallest node N_0 in J that covers the MBR of $\mathcal{P}$, and the child nodes of N_0 in J, denoted by N_1, N_2, N_3 and N_4. If the similarity of query distribution between N_0 and its child nodes is small, it indicates that queries are distinguishable across the spatial partitions and spatial partitioning would result in smaller workload than textual partitioning. We compute the similarity of query distribution by the Manhattan distance computed by $\hat{h} = \frac{\sum_{i=1}^{4}\sum_{j=1}^{6} N_i.p_j}{4 \cdot \sum_{j=1}^{6} N_0.p_j}$, where $N_i.p_1, \cdots, N_i.p_6$ are the probabilities that the SRQ, SKQ, STQ, CRQ, CKQ and CTQ queries overlap N_i, respectively. They can be derived from the statistics maintained in N_i. The smaller the $\hat{h}$, the larger the spatial discrepancy between N_0 and its child nodes. If $\hat{h}$ is small, most queries in $\mathcal{P}$ overlap a small number of child nodes, and it is better to use spatial partitioning.

We partition a leaf node $\mathcal{P}$ based on the following rule:

$$split(\mathcal{P}) = \begin{cases} \text{spatial partitioning}, & \hat{h} \leq \epsilon; \\ \text{textual partitioning}, & \hat{h} > \epsilon, \end{cases} \quad (3)$$

where ϵ is a threshold. After splitting $\mathcal{P}$, we move the continuous queries of $\mathcal{P}$ to its child nodes, accordingly.

Merge Operation: Merge is only applicable to a non-leaf node $\mathcal{P}$ whose four child nodes are all leaf nodes. It moves the continuous queries from all four child nodes to $\mathcal{P}$, and deletes the child nodes (making $\mathcal{P}$ a new leaf node).

Now, we show how we update a QT-tree G using the split and merge operations. We use L_{max} to record the maximum workload of leaf nodes in G, and use a min-heap H to store the nodes whose child nodes can be merged. For each candidate node, if its workload is smaller than L_{max}, we insert it into H. Then, we recursively apply the split and merge operations on G until the load balance constraint is satisfied. In each iteration, we split the most loaded leaf node $\mathcal{P}$ using Equation 3, and update the value of L_{max} accordingly. Next, we pop one node P' from H and if its workload is larger than L_{max}, then we cannot further decrease the difference in workload among the leaf nodes by more merge operations, and thus we terminate. Otherwise, we perform the merge operation on $\mathcal{P}'$. If the merge operation results in a new candidate node (i.e., the parent node of $\mathcal{P}'$) for the merge operation, we push the new candidate node into H. Then, we check whether the load balance constraint is met by invoking the Karmarkar-Karp algorithm, and terminate if this is the case. In each iteration, the split operation reduces the value of L_{max} and the workload variance in leaf nodes decreases further, and thus, eventually the algorithm will terminate.

As in the workload reallocation strategy, we only transfer continuous queries among workers in the QT-tree adjustment strategy. This design will also result in incomplete results for new snapshot queries after the QT-tree is adjusted. In remedy the router of SSTD maintains different versions of the QT-tree, each of which is associated with the mapping from leaf node id to worker id, and a validation timestamp. For simplicity, we consider two versions of the QT-tree G and G', where G' is the up-to-date version. For a snapshot query q that overlaps a leaf node $\mathcal{P}$ of G', where $\mathcal{P}$ is assigned to worker W_1, besides sending q to W_1, we conduct the following operations:

- If $\mathcal{P}$ is a leaf node in G and is assigned to another worker W_2, then we also send q to W_2.
- If $\mathcal{P}$ is not a leaf node in G but its parent (or ancestor) node is, and is assigned to another worker W_3, then we send q to W_3.
- If $\mathcal{P}$ is not a leaf node in G but its child (or descendant) nodes are, and are assigned to several workers, then we send q to each of them.

To summarize, SSTD maintains query distribution statistics and periodically checks whether load imbalance occurs. Once load imbalance is detected, to rebalance the load, SSTD first applies **PM**, and if it does not recover the balance, SSTD will next apply **FM**, followed by **FG**. After adjusting the workload, the updated mapping table and the QT-tree will be synchronized among all routers, and are written into disk for fault tolerance.

6. LOCAL INDEXES AND OPERATIONS

We proceed to present how to build local indexes and process the snapshot and continuous queries in the workers.

6.1 Local Indexes

After workload allocation, each worker is assigned with multiple leaf nodes of the QT-tree. For each leaf node assigned to a worker, the worker builds and maintains separate index structures for the objects and continuous queries in that leaf node. When a worker receives an object or a continuous query from a router, it assumes the responsibility of inserting the object or query into a corresponding local index. The reason for the design of building separate indexes for each leaf node rather than a single index for all leaf nodes assigned to a worker is that the worker can utilize the filtering power of the global index and only access the data/queries under a leaf node for query processing and index maintenance. We proceed to present object index and query index for each leaf node.

Object index. SSTD categorizes the data into different time intervals, e.g., $[t_0, t_0 + \Delta), [t_0 + \Delta, t_0 + 2\Delta), \cdots$, and builds index structures for each time interval. When receiving a new object o, the worker finds the time interval that covers the timestamp of o, and inserts o into the corresponding index structure. The benefits of this design are twofold: (1) When processing snapshot queries, only the indexes satisfying the temporal constraint need to be visited;

(2) SSTD periodically writes the obsolete data into the persistent storage to free memory space. SSTD supports all existing spatio-textual index structure(e.g., [4, 7]) for objects. Additionally, to improve the efficiency in answering STQ, we maintain a hash table to store term frequencies in each node of the object index.

Query index. SSTD also supports and indexes continuous queries that may be active for a long time duration and whose number may be large. Note that SSTD does not need to index snapshot queries. For CRQs, SSTD supports existing indexes, e.g., AP-tree [22], IQ-tree [3] and R^t-tree [12]. For CKQs, SSTD also supports existing indexes, e.g. the grid structure [25] or IQ-tree [3]. SSTD adopts FAST [15] for CRQ and a grid structure [25] for CKQ. For CTQs, SSTD reports the top-k most frequent terms every δ time units. To the best of our knowledge, no dedicated index structures are designed for CTQ queries. We extend the *hash table* [14] and *subscription group* [5] techniques to index CTQs. The main idea is to categorize the CTQs into groups based on their query regions. Figure 6 gives an example of the index of a group of CTQs. For each group, we maintain a common hash table that stores the term frequencies in the intersection region of all queries (e.g., shaded in Figure 6), and an individual hash table that stores the term frequency in the region excluding the intersection region for each query.

6.2 Snapshot Query Processing

After a worker receives an SRQ query q, it accesses the object indexes that satisfy the temporal constraint of q, finds the result objects using the algorithms proposed in previous work on spatial keyword queries and sends them to the **merger**, i.e., a particular worker responsible for collecting the partial results from the other workers and reporting the complete results to the users. SSTD processes an STQ q in a similar way as to that of an SRQ. Specifically, when a worker receives an STQ, it searches for the objects in the index that satisfy the temporal constraint, aggregates the term frequency table, and then sends this table to the merger.

SSTD processes an SKQ q in two phases. In the first phase, the router finds the leaf node closest to $q.\rho$ and sends q to the corresponding worker. Then, the worker retrieves k initial results of q based on the object indexes and outputs the radius r of q that is the distance between q and the kth farthest object in the initial results. The radius r is an overestimation of the distance between q and the kth object in the actual results. If the initial results contain fewer than k objects, we set r to be a large value. The worker then sends (q, r) back to the router. In the second phase, after receiving (q, r) from the worker, the router uses r to construct a range constraint for q, and processes q in the same way as processing an SRQ. At the end of the second phase, the workers report their k result objects to the merger.

6.3 Continuous Query Processing

Each worker processes the three types of continuous queries by inserting them into the corresponding index structures. With the query indexes, the update of query results is driven by the arrivals of new objects. If a new object o updates the results of a query, then we term this query as *affected*. When a worker receives a new object, it checks the continuous query indexes to find out the affected queries using the existing algorithms [15, 25]. Next, we present how we handle each affected continuous query.

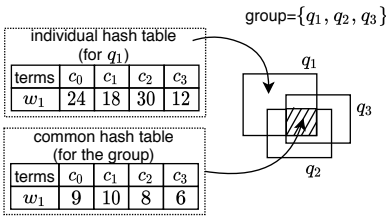


Figure 6: Index of CTQ

(1) For each affected CRQ q, the worker sends (o, q) to the merger. (2) For each affected CKQ q, the worker sends (o, q) to the merger. As the merger collects partial results from the workers and keeps the up-to-date results for q, if the merger finds $d(o, q) > r$, (r is the distance between q and its latest kth result object), then the merger will omit o and send r back to the source worker. (3) For CTQs, the worker updates the hash tables of each affected group of CTQs as described in Section 6.1. Every δ time units, the worker sends the hash tables storing the term frequency information to the merger, and the merger reports the top-k frequent terms of CTQs.

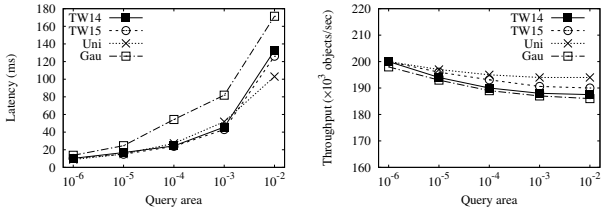
7. EXPERIMENTS

7.1 Experimental Setup

SSTD is deployed on Amazon EC2 platform that runs on a cluster of 10 c5.2xlarge instances. Each instance has 8 vCPUs running Intel Xeon E5-2680@3.4GHz and 16GB RAM. All instances are connected by a 10Gbps network, and run Ubuntu 16.04 LTS with Storm 1.1.0 and Kafka 1.1.0. We use one instance to serve as a Kafka server that presents the input data stream to SSTD. We use one instance to run Storm Nimbus daemon and 8 instances to run Storm supervisor daemon.

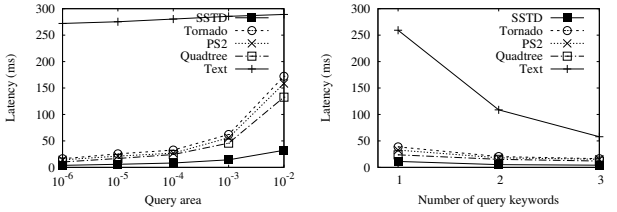
Datasets and Queries. We use two real datasets from Twitter. The first dataset contains 63 million tweets in August 2014 and its size is 5.9GB, and the second dataset contains 52 million tweets in August 2015 with a size of 4.6GB. Each tweet contains 10 keywords on average. The vocabulary size is about 800k. We denote them as TW14 and TW15, respectively. Since we do not have real queries, we synthesize them in the following way: we sample a small set of tweets and for each tweet, we utilize its textual content and geo-location to generate one query in SRQ, SKQ, STQ, CRQ, CKQ and CTQ such that (i) The number of query keywords ranges from 1 to 3 (default is 1), (ii) The area of the query region ranges from 0.0001% to 1% of the area of the global space (default is 0.01%), (iii) The value of k ranges from 10 to 40 (default is 10), and (iv) The time duration ranges from 24 hours to 168 hours (default is 72). We generate 400k queries for each type of query.

Compared methods. We use two state-of-the-art spatial data stream processing systems, namely Tornado [16] and PS^2 [6], as baselines to evaluate the performance of global index. (1) Tornado. It uses A-Grid as global index. A-Grid partitions the global space into equal-sized grid cells. We evaluate different granularities of grid cells varying from 100×100 to 1000×1000, and choose 600×600 as it performs the best in our experiments. (2) PS^2. It applies kd^t as global index, which extends a kd-tree by allowing some leaf nodes to be further partitioned using textual partitioning. PS^2 is developed for processing continuous queries and we extend it to support other queries as follows: Let N_1, N_2 be two leaf



(a) Size of query area (b) Size of query area

Figure 7: Data distribution



(a) Size of query area (b) Number of query keywords

Figure 8: SRQ query latency

nodes derived from textual partitioning on node N_0. When an incoming object o meets N_0, no matter whether o updates the results of continuous queries on N_1 and N_2, o will be sent to at least one worker for storage. If o updates the queries on both N_1 and N_2, o will be sent to both workers, but only one worker selected at random will store o in local index. When an incoming query q meets N_0, q will be sent to both N_1 and N_2 for range and top-k query, or one of them at random for kNN query.

We also adopt two simplified versions of QT-tree as baselines. (3) Quadtree. We always apply spatial partitioning in the construction of QT-tree. (4) Text. We always apply textual partitioning in the construction of QT-tree.

As we aim to evaluate the performance of different global indexing techniques, we use the same local index layer in each evaluated method. Specifically, in the local index layer, we use an R-tree augmented with inverted lists for storing objects, FAST [15] for indexing CRQs, Grid [25] for indexing CKQs, and hash table [5, 14] for indexing CTQs.

We evaluate the performance of SSTD in three aspects:(1) We evaluate the query latency of snapshot queries. The query latency is measured by the temporal difference between the time a query arrives at the system and the time the complete query results are produced. We send 50M objects to the system, then submit a batch of 200 queries every other second, and use the average query latency value to serve as the query latency. (2) We evaluate the system throughput when processing continuous queries. We initially register 200k queries into the system, and produce an input stream of objects with an arrival rate of 200,000 objects per second. We measure the throughput by the average number of objects being processed every second. (3) We evaluate the effectiveness of our load balancing methods.

7.2 Performance with Different Data

In this set of experiments, we study the performance of SSTD on different datasets. In addition to datasets TW14 and TW15, we synthesize another two datasets with different data distributions by changing the geographical locations of the objects in TW14 as follows: (i) **Uni**, where we assign uniformly distributed locations to the objects; and (ii) **Gau**, where we assign locations satisfying 2-dimensional Gaussian distribution to the objects, where σ_x and σ_y are set as 1, μ_x and μ_y are set as the mid points of the x and y axes, resp., and the correlation coefficient ρ_{xy} is set as 0.

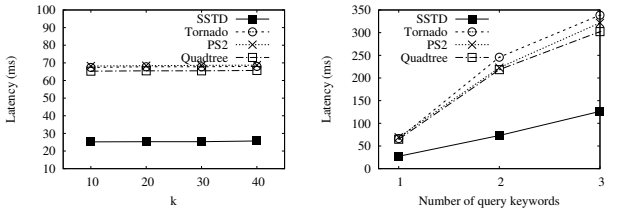
Figure 7(a) gives the query latencies of SRQs for the four datasets when query region is varied. The result shows that SSTD displays similar trend for datasets of different data distributions, even for the skewed dataset Gau. The latency for Gau is the largest as its skewed data distribution makes it difficult to achieve load balance for workers, and this affects the performance of SSTD's workload partitioning strategy. In contrast, the latency for Uni is the smallest in most cases due to its uniform data distribution. The results of TW14 and TW15 are similar and are between the latencies of Gau and Uni. Figure 7(b) gives the throughputs of CRQs for the four datasets. As expected, the throughput for Gau is the smallest due to its skewed data distribution while that for Uni is the largest. The throughput for TW15 is slightly larger ($< 2\%$) than the throughput for TW14.

The results demonstrate that SSTD displays similar trend on different datasets. Since the results on TW14 and TW15 are similar, due to space, we report only the results on TW14 in the remaining experiments.

7.3 Query Latency

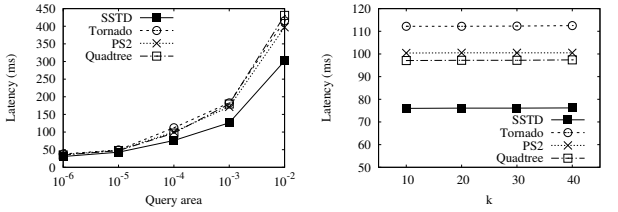
Figure 8 gives the query latencies of SRQ for different systems. Figure 8(a) shows the effect of the size of the query region. SSTD consistently has the smallest query latency, and Quadtree performs the second best. Text has the worst performance. When the query region is 0.0001% of the entire space, the query latencies of SSTD, Tornado, PS^2 and Quadtree are 4ms, 16ms, 14ms and 10ms, respectively; SSTD is 2–4 times faster than baselines although they overlap in the figure. The disparity becomes larger as the size of the query region increases. When the query region is 1% of the entire space, the query latency of SSTD is 32ms, whereas Tornado, PS^2, and Quadtree are 172ms, 158ms, and 132ms, respectively, which means that SSTD is 4–5 times faster than baselines. When the size of the query region increases, SRQ runs slower as it accesses more partitions. QT-tree can prune many irrelevant partitions, and thus its query latency increases slowly. Tornado and PS^2 have similar performance because they both use kd-tree-like structures and adopt similar cost models. The performance of Text is hardly affected by the query region due to its textual partitioning scheme. Figure 8(b) shows the effect of the number of query keywords. The query latency becomes smaller when the number of query keywords increases. SSTD has the best performance, which always has the query latency being smaller than 10ms. Though the query latency of Text reduces greatly when increasing the number of query keywords, it is still at least 4x slower than the others. Due to its poor performance, we do not show it subsequently.

Figure 9(a) shows the effect of the value of k of SKQ on TW14. SSTD performs significantly better than the other systems. The value of k does not have an obvious effect on the query latency. The reason is that the query latency is mainly dominated by the number of partitions being accessed, and as k is usually much smaller than the number of objects in a partition, k does not affect the number of partitions being accessed in most cases. Recall that SKQ is processed in two phases. SSTD outperforms the baselines since QT-tree can effectively find the partition producing a more accurate initial query results, which greatly reduces the searching cost in the second phase. Figure 9(b) shows that the query latency becomes larger when the number of query keywords increases. As the number increases, it is more



(a) k (b) Number of query keywords

Figure 9: SKQ query latency



(a) Size of query area (b) k

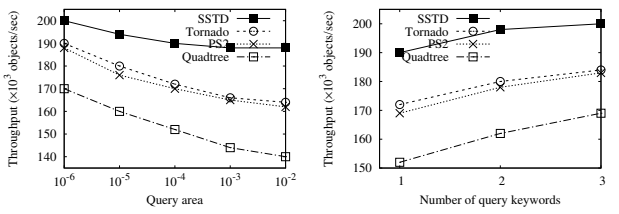
Figure 10: STQ query latency

likely that the initial results from the first phase contain fewer than k objects, and this will cause higher searching cost in the second phase.

Figure 10(a) gives the effect of the size of the query region of STQ on TW14. SSTD has the best performance. The query latency of an STQ is mainly decided by the number of term-frequency tables to be aggregated and the term distribution in each table. Textual partitioning in the QT-tree allows SSTD to store objects with different term distributions into different partitions, which is helpful in reducing the aggregate cost. For small query regions, a small number of term-frequency tables need aggregation. Thus the difference in latency among various systems is small. For large query regions, a larger number of term-frequency tables needs aggregation, and the QT-tree textual partitioning lowers the aggregate cost, thus reducing query latency. Figure 10(b) gives the effect of k. It shows that k has no obvious effect on query latency. The reason is that k does not affect the number of term-frequency tables to be aggregated, and it only affects the termination condition in aggregation, which has minor effect on the total aggregation cost. SSTD outperforms the other systems due to its textual partitioning method that puts textually similar objects together.

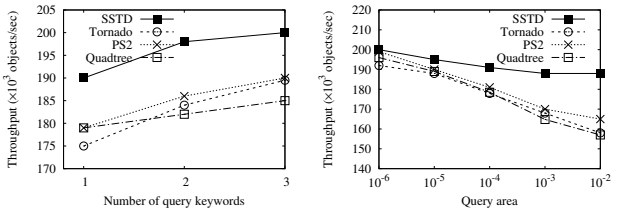
7.4 Throughput

Figure 11 compares the throughput of the systems when processing CRQs on TW14. Figure 11(a) gives the effect of the query region's size and Figure 11(b) gives the effect of the number of query keywords. SSTD has the largest throughput consistently, which outperforms Tornado by around 15–20%. The reason is that QT-tree utilizes both the spatial and textual properties to disseminate the objects, and does better in co-locating objects and relevant CRQs.



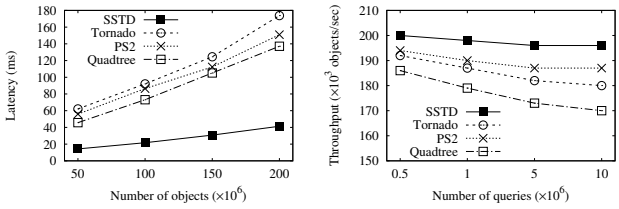
(a) Size of query area (b) Number of query keywords

Figure 11: CRQ throughput



(a) CKQ throughput (b) CTQ throughput

Figure 12: CKQ and CTQ throughput



(a) Number of objects (b) Number of queries

Figure 13: Scalability

Figure 12(a) gives the effect of the number of query keywords of CKQs. SSTD achieves the best throughput consistently. The result demonstrates that SSTD achieves better workload partitioning. As the number of query keywords increases, the throughput of the systems becomes larger. This is because the local index for the CKQs becomes more efficient when the number of query keywords is larger. Figure 12(b) gives the effect of the size of the query region of CTQs. The result shows that SSTD outperforms other systems. As the size of query region increases, the difference of throughput between SSTD and other systems becomes larger. The reason is similar as we discuss for STQ.

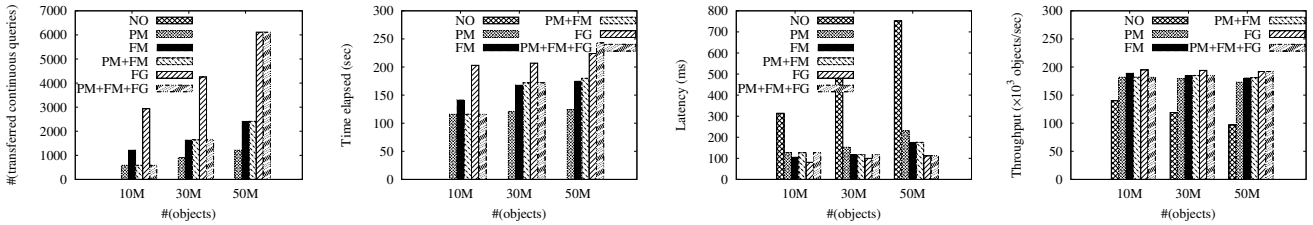
7.5 Scalability

We evaluate the scalability of SSTD in two scenarios: varying the number of objects and the number of continuous queries. To evaluate the effect of the number of objects, we duplicate Dataset TW14 to generate four datasets that contain 50M, 100M, 150M and 200M tweets, respectively. We report the average query latency for a batch of $10,000$ SRQs on the four datasets in Figure 13(a). We observe that SSTD scales well with the object size and is 3–4 times faster than baselines consistently. The disparity between SSTD and baselines becomes larger as the number of objects grows. This further demonstrate the effectiveness of the QT-tree in distributing spatially close or textually similar objects into the same partition, and its efficiency in processing snapshot queries.

To evaluate the scalability on the number of queries, we preload 0.5M, 1M, 5M and 10M CRQs, respectively, into the system, and present the streamed data at an arrival rate of $200,000$ objects per second. Figure 13(b) gives the system throughput. We observe that SSTD achieves the largest throughput consistently and its throughput drops slightly with the increase of number of CRQs.

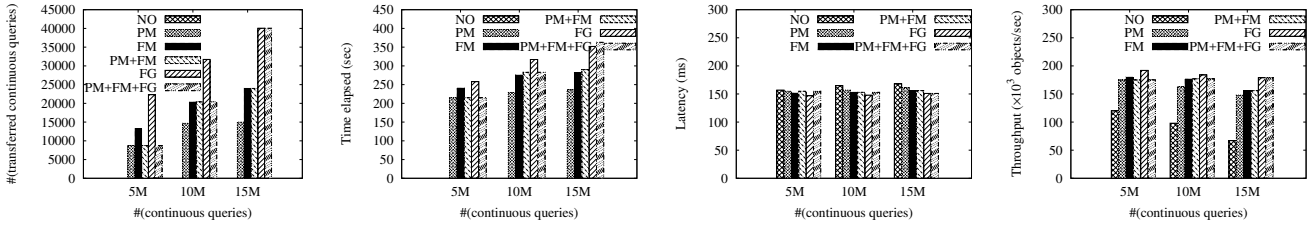
7.6 Robustness to Load Imbalance

To evaluate the effectiveness and efficiency of SSTD's proposed workload adjustment methods, we compare the following 6 plans after a load imbalance: (i) taking no adjustment actions, (ii) using PM, (iii) using FM, (iv) using PM+FM (i.e., using PM followed by FM), (v) using FG, and (vi) using PM+FM+FG. To simulate a load imbalance occurrence, we randomly select a region and change



(a) #(Transferred queries) (b) Time elapsed (c) Query latency (d) Throughput

Figure 14: Workload adjustment under object insertions



(a) #(Transferred queries) (b) Time elapsed (c) Query latency (d) Throughput

Figure 15: Workload adjustment under continuous query insertions

its load in either of the following ways: 1) insert a significantly large number of objects into the region; 2) register a significantly large number of continuous queries into the region. We preload 40k continuous queries and 30M objects into the system before triggering load imbalance. For each case of load imbalance, we perform workload adjustments and measure the network cost (i.e., the number of continuous queries being transferred during the adjustment), the required time for load adjustment, the average latency of 40k new snapshot queries after the adjustment, and the system throughput after the adjustment.

Figure 14 gives the performance after inserting different numbers of objects that results in different extents of imbalance. The result shows that PM, FM, and FG greatly improve load balance, and consequently, the query latency and throughput. For example, at 50M objects, PM, FM, and FG reduce the query latency by 69.3%, 76.6% and 85.1%, respectively, and enhance the throughputs by 78.4%, 86.6% and 97.9%, respectively, compared with taking no action. Comparing PM, FM, and FG in Figures 14(c) and 14(d), observe that FG achieves the best query latency and throughput, followed by FM, and then PM. However, Figures 14(a) and 14(b) indicate that FG incurs the largest network cost, and takes the longest time to conduct load adjustment. Comparing PM+FM with PM only, and FM only, observe that PM+FM has similar performance to that of PM in case of minor imbalance (i.e., #objects=10M), and is similar to FM's performance in moderate (i.e., #objects=30M) or severe (i.e., #objects=50M) imbalance. Observe that PM+FM+FG has similar performance with PM, FM and FG when #objects is 10M, 30M and 50M, respectively, which indicates that PM+FM+FG employs different strategies under different levels of imbalance. Figure 15 gives the performance after registering different numbers of continuous queries. Observe that PM, FM and FG often improve throughput by more than one time. Also, observe that FG achieves better balance than PM and FM, but it has larger network cost and needs more time to recover balance. In both cases, PM+FM+FG achieves similar effectiveness as FG, but often costs less in terms of network cost and adjustment time. Thus, PM+FM+FG is recommended.

In the experiments, we set parameters λ and ϵ at 6, 0.5, respectively, where λ denotes the threshold of load imbalance factor and ϵ is the threshold for choosing spatial or textual partitioning during QT-tree adjustment. We vary λ from 2 to 10 and ϵ from 0.1 to 0.9, and conduct experiments to evaluate their effects. However we observe similar trends.

7.7 Parameter tuning

This experiment is to study the impact of parameters on the performance of SSTD: (1) Parameter θ, the threshold of the minimum number of leaf nodes in the QT-tree, and (2) Parameter Δ, the length of the temporal interval of a local index storing the objects. We vary θ from 32 to 256, Δ from 1 day to 7 days, and set $\theta = 64$ and $\Delta = 3$ in our experiments. Due to the space limitation, the detailed results are reported in an online version of this paper[1].

8. CONCLUSION

In this paper, we present SSTD, a distributed in-memory system supporting continuous and snapshot queries over streaming spatio-textual data. We introduce a cost-based model to depict the workload, and propose a new global index for workload partitioning. Furthermore, we propose three workload adjustment methods to handle load imbalance. The experimental study demonstrates the superior performance of SSTD over the other systems.

9. ACKNOWLEDGEMENTS

Gao Cong acknowledges the support by Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU), which is a collaboration between Singapore Telecommunications Limited (Singtel) and Nanyang Technological University (NTU) that is funded by the Singapore Government through the Industry Alignment Fund - Industry Collaboration Projects Grant, and a Tier-1 project RG114/19. Walid G. Aref acknowledges the support of the U.S. National Science Foundation under Grant Numbers: III-1815796 and IIS-1910216.

[1]https://www.ntu.edu.sg/home/gaocong/sstdfull.pdf

10. REFERENCES

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-gis: a high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[2] A. M. Aly, A. R. Rahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. Aqwa: adaptive query-workload-aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.

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

[4] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[5] L. Chen, S. Shang, Z. Zhang, X. Cao, C. S. Jensen, and P. Kalnis. Location-aware top-k term publish/subscribe. In *ICDE*, pages 749–760, 2018.

[6] Z. Chen, G. Cong, Z. Zhang, T. Fu, and L. Chen. Distributed publish/subscribe query processing on the spatio-textual data stream. In *ICDE*, pages 1095–1106, 2017.

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

[8] A. Eldawy and M. Mokbel. Spatialhadoop: a mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[9] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[10] N. Karmarkar and R. M. Karp. The differencing method of set partitioning. Technical Report UCS/CSD 82/113, University of California, Berkeley, Conputer Science Division, University of California, Berkeley, 1982.

[11] R. E. Korf. Multi-way number partitioning. In *IJCAI*, pages 538–543, 2009.

[12] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *KDD*, pages 802–810, 2013.

[13] J. Lu and R. H. Guting. Parallel second: boosting database engines with hadoop. In *ICPADS*, pages 738–743, 2012.

[14] A. Magdy, A. M. Aly, M. F. Mokbel, S. Elnikety, Y. He, S. Nath, and W. G. Aref. Geotrend: spatial trending queries on real-time microblogs. In *SIGSPATIAL/GIS*, pages 7:1–7:10, 2016.

[15] A. Mahmood, A. Aly, and W. Aref. Fast: frequency-aware spatio-textual indexing for in-memory continuous filter query processing. In *ICDE*, pages 305–316, 2018.

[16] A. Mahmood, A. Aly, T. Qadah, E. Rezig, A. Daghistani, A. Madkour, A. Abdelhamid, M. Hassan, W. Aref, and S. Basalamah. Tornado: A distributed spatio-textual stream processing system. *PVLDB*, 8(12):2020–2023, 2015.

[17] A. Mahmood, A. Daghistani, A. Aly, M. Tang, S. Basalamah, S. Prabhakar, and W. Aref. Adaptive processing of spatial-keyword data over a distributed streaming cluster. In *SIGSPATIAL*, pages 219–228, 2018.

[18] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. Md-hbase: a scalable multi-dimensional data infrastructure for location aware services. In *MDM*, pages 7–16, 2011.

[19] A. S. Abdelhamid, M. Tang, A. M. Aly, A. R. Mahmood, T. Qadah, W. G. Aref, and S. Basalamah. Cruncher: distributed in-memory processing for location-based services. In *ICDE*, pages 1406–1409, 2016.

[20] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen. Scalable top-k spatio-temporal term querying. In *ICDE*, pages 148–159, 2014.

[21] M. Tang, Y. Yu, Q. Malluhi, M. Ouzzani, and W. Aref. Locationspark: A distributed in-memory data management system for big spatial data. *PVLDB*, 9(13):1565–1568, 2016.

[22] X. Wang, Z. Y., Z. W., X. Lin, and W. Wang. Ap-tree: efficiently support continuous spatial-keyword queries over stream. In *ICDE*, pages 1107–1118, 2015.

[23] Website. Foursquare, 2017.

[24] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.

[25] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: scalable processing of continuous k-nearest neighbor queries in saptio-temporal databases. In *ICDE*, pages 643–654, 2005.

[26] J. Yu, J. Wu, and M. Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*, 2015.

[27] K. Zhao, L. Chen, and G. Cong. Topic exploration in spatio-temporal document collections. In *SIGMOD*, pages 985–998, 2016.