STAR: A Cache-based Distributed Warehouse System for Spatial Data Streams

Zhida Chen
SCALE@Nanyang Technological University
Singapore
chen0936@e.ntu.edu.sg

Gao Cong
SCALE@Nanyang Technological University
Singapore
gaocong@ntu.edu.sg

Walid G. Aref
Purdue University
USA
aref@purdue.edu

ABSTRACT

The proliferation of mobile phones and location-based services has given rise to an explosive growth in spatial data. In order to enable spatial data analytics, spatial data needs to be streamed into a data stream warehouse system that can provide real-time analytical results over the most recent and historical spatial data in the warehouse. Existing data stream warehouse systems are not tailored for spatial data. In this paper, we introduce the STAR (Spatial Data Stream Warehouse) system. STAR is a distributed in-memory data stream warehouse system that provides low-latency and up-to-date analytical results over a fast-arriving spatial data stream. STAR supports queries that are composed of aggregate functions and ad hoc query constraints over spatial, textual, and temporal data attributes. STAR implements a cache-based mechanism to facilitate the processing of queries that collectively utilizes the techniques of query-based caching (i.e., view materialization) and object-based caching. Extensive experiments over real data sets demonstrate the superior performance of STAR over existing systems.

CCS CONCEPTS

• **Information systems** → **Location based services**; **Data streaming**; **Parallel and distributed DBMSs**.

KEYWORDS

spatial data, data stream, warehouse system, distributed system

ACM Reference Format:
Zhida Chen, Gao Cong, and Walid G. Aref. 2021. STAR: A Cache-based Distributed Warehouse System for Spatial Data Streams. In *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21), November 2–5, 2021, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474717.3484265

1 INTRODUCTION

With the proliferation of GPS-equipped mobile devices and social media services, there has been an explosive growth in the spatial data sizes. Numerous users of social media upload posts on Twitter or Facebook using their smart phones, giving rise to a fast arriving spatial data stream. This spatially annotated data contains valuable information, and is beneficial for spatial data analytics. For instance, consider a marketing manager who wants to know the popularity of some product in various regions so that she can decide whether or not to adjust the advertising strategy. She can issue an ad hoc aggregate query that returns the frequencies grouped by region of the newly uploaded posts on social networks that mention the product. Techniques already exist for processing aggregate queries over data warehouses. However, most of these techniques are for batch-oriented systems that operate over static data sets, and are not suitable for handling highly dynamic data streams.

To reduce the gap between data production and data analysis, a data stream warehouse system (DSWS, for short) [17, 21] provides real-time analytics over data streams. DSWSs efficiently ingest data, and enable online analytical processing over streamed data. A DSWS allows users to issue continuous queries that monitor changes in the streamed data as well as snapshot queries that report the current or past status of warehoused data.

Although spatial data is explosive in size, research on distributed DSWSs that offer native spatial data stream analytics is still lacking. Most existing distributed systems, e.g., [4, 5, 14, 42] focus on developing spatial data management systems over static data sets, but are not designed for streamed data, and do not support ad hoc aggregate queries over spatial data streams. Existing distributed spatial data stream systems, e.g., [11, 30], do not support ad hoc aggregate queries. Furthermore, they only support continuous but not snapshot queries.

It is challenging to develop a DSWS that supports ad hoc queries over spatial data streams. First, the fast arrival speed of streamed spatial data imposes high demand on system performance, of which the accompanying workload will overwhelm a centralized system. It calls for a distributed and scalable solution with an effective workload partitioning scheme that is tailored for the workloads of processing objects and analytical queries. Second, it is difficult to pre-compute and maintain a set of materialized views for ad hoc aggregate queries over spatial data streams, which are essential for the performance of a warehouse system. Classic view materialization algorithms (e.g., [25, 34]) do not apply here as they do not support spatial data. It demands for novel view materialization algorithms that can optimize processing analytical spatial queries.

In this paper, we introduce STAR, an in-memory cache-based Spatial Data Stream Warehouse for spatial data analytics over spatial data streams. STAR supports ad hoc aggregate queries that can have constraints over spatial, textual, and temporal data attributes. STAR supports algebraic aggregate functions, e.g., *Count*, *Avg*, and *Sum* in addition to a holistic aggregate function *TopK*. STAR adopts

SIGSPATIAL '21, November 2–5, 2021, Beijing, China

ACM ISBN 978-1-4503-8664-7/21/11…$15.00
https://doi.org/10.1145/3474717.3484265

an effective workload partitioning method that collectively considers data locality and load balance to solve the distributed view materialization problem. Moreover, we develop a cache-based mechanism to facilitate the processing of analytical spatial queries that collectively utilizes query-based caching (i.e., view materialization) as well as object-based caching.

The contributions of our work are summarized as follows:

- We propose a distributed stream warehouse system STAR for spatial data analytics. STAR supports a rich set of aggregate queries over spatial data streams. STAR supports both snapshot and continuous queries that are composed of algebraic or *Topk* aggregate functions and ad hoc query constraints over spatial, textual, and temporal data attributes.
- We design an effective and efficient workload partitioning strategy that reduces the costs of processing objects, processing queries, and maintaining materialized views, as well as achieves load balance and data locality.
- We present a cache-based mechanism for the efficient processing of snapshot ad hoc aggregate queries. The cache-based mechanism collectively utilizes the techniques of query- and object-based caching. Query-based caching considers spatial and textual attributes to define views for aggregate functions, and selects and maintains a set of views in-memory to speedup query processing. Object-based caching is complementary to query-based caching when a query cannot be answered using the materialized views only. We develop an approximation algorithm for object-based caching that provides a competitive solution with theoretical bounds.
- We evaluate STAR on Amazon EC2 with real spatial data. STAR achieves excellent performance with respect to both object and query processing, and outperforms the best baseline systems by up to 5×.

2 RELATED WORK

Data Stream Warehouse Systems. Based on the architecture, existing distributed warehouse systems can be classified into three different types: (1) Systems that extend a database system with the abilities of fast data ingestion and real-time data evaluation [18] (2) Systems that extend a data stream processing system with the ability of exploring historical data [6], and (3) Systems that extend a distributed analytics framework, e.g., Hadoop, with the abilities of fast data ingestion and real-time data evaluation [26, 33]. However, existing distributed warehouse systems do not provide native support for spatial data streams, and are difficult to optimize for aggregate operations over streamed spatial data.

Some work exists for developing centralized stream warehouse systems over spatial data. Gorawski and Malczok [19] present an index structure to store spatial data in a stream warehouse. Lins *et al.* [28] and Giampi *et al.* [12] consider the problem of exploring streamed spaito-temporal data, and they propose a new data structure of views to achieve this. Feng et al. [15] propose solutions for exploring events from streamed geo-tagged tweets. These systems do not provide native support for aggregate queries with spatial, textual, or temporal constraints as STAR does. Moreover, these systems are centralized systems while STAR is distributed.

Systems for Spatial Data. A host of systems has been proposed for exploring a static spatial data set. Most are extensions to popular data analytics frameworks: SpatialHadoop [14] and Hadoop-GIS [4] extend the Hadoop framework; Simba [42], SpatialSpark [45], LocationSpark [38] and GeoSpark [46] extend the Spark framework. They extend Hadoop or Spark with operations to support spatial queries, e.g., range query and kNN query, over a large scale of spatial data. STAR differs from these systems in at least three aspects: 1) STAR operates over data streams, while these systems consider a static data set with few or no updates. 2) STAR is optimized for ad hoc aggregate queries while the other systems mainly consider object-finding queries. 3) Apart from operations on spatial attributes, STAR supports operations on textual attributes while the other systems focus only on operations over spatial attributes. LocationSpark [38] adopts a caching strategy that maintains frequently accessed data in-memory for object-finding queries. However, it does not support caching query results for aggregate queries, which is the main focus of STAR.

Systems for Streamed Spatial Data. The problem of querying spatial data streams has been studied extensively. Many centralized solutions have been proposed. Some efforts are made to find top-k frequent terms given a spatio-temporal range [36]. Another line of work considers answering spatio-keyword queries. A spatio-keyword query has a spatial and a textual arguments. An object is in the result of the query if the object qualifies both arguments [8] or if the object's similarity is larger than a threshold [24]. Another body of work studies the top-k spatio-keyword query [40, 41] that returns objects having the top-k highest similarities to the input query. Several distributed systems [9, 11, 29, 30, 37] have been proposed for querying streamed spatial data. However, these systems do not have native support for aggregate operations over spatial data. In contrast, STAR is optimized for processing ad hoc aggregate queries, which focuses on computing the aggregate results over all objects rather than finding individual objects, i.e., STAR treats aggregate queries over spatial objects as a first class operation. A preliminary version of this work has been demonstrated [10]. Xiong *et al.* [43] develop a system PLACE* that supports aggregate queries over moving spatial data. PLACE* focuses on query plan generation for reducing network cost. STAR is complementary to PLACE*.

View Materialization. Labio *et al.* [25] and Ross *et al.* [34] propose exhaustive algorithms to materialize views in a single machine that takes significantly long time to finish. Many other research focuses on designing greedy algorithms, e.g., [20, 22, 44], or randomized algorithms including genetic algorithms, e.g., [23] and simulated annealing algorithms, e.g., [13]. Ghanem *et al.* [16] consider the problem of supporting materialized views in a data stream management system. They propose a synchronized SQL query language to express continuous queries over data streams and create continuous query execution plans. However, they do not support aggregate or analytics queries over these views.

3 SYSTEM OVERVIEW

STAR is a web-based system built upon Apache Storm [2], an open source distributed real-time computation framework. First, we introduce the data types and queries supported by STAR. Then, we present STAR's architecture.

3.1 Data Types and Queries

Data Types: Each object has *primitive* and/or *extracted or derived* attributes. Primitive attributes store raw streamed data while the extracted or derived attributes store data that is extracted or derived from the primitive attributes. We assume that the raw data has at least the primitive attributes *loc* and *time*, where *loc* represents the geographical latitude and longitude, and *time* represents the timestamp. The raw data can also have other primitive attributes, e.g., *text* that contains a set of terms. STAR integrates a set of tools to extract data from these primitive attributes. For example, data in Attribute *topic* can be extracted from *text* by employing a pre-trained Latent Dirichlet Allocation (LDA) model [7].

Supported Queries: STAR is optimized to support aggregate queries with ad hoc constraints, e.g., over *loc*, *text*, and *time*. STAR supports *algebraic* aggregate functions and a *holistic* aggregate function *TopK*. Algebraic aggregate functions, e.g., *Count*, can be computed over the disjoint data partitions, and then the partial results are aggregated to obtain the final aggregate results. In contrast, *TopK* aggregate the entire data set to obtain the k most-frequent terms appearing in Attribute *text*.

STAR supports range and keyword constraints over Attributes *loc* and *text*, respectively. STAR focuses on time-window constraints that consider only recently streamed data. STAR expresses these constraints using SQL-like syntax, e.g.,
SELECT *aggr_func()* **FROM** *stream*
WHERE *condition(s)* **GROUP BY** *attribute(s)* [**SYNC** *freq*].
aggr_func() is an aggregate function, *condition(s)* are the constraints, and *attribute(s)* are the grouping attributes. STAR focuses on optimizations for processing snapshot queries, but supports continuous queries as well. STAR defines a continuous query via the **SYNC** operator. **SYNC** *freq* indicates that the query result is to be refreshed every *freq* time, which is inspired by [16].

Example 1: Snapshot Aggregate Query. Find the popularity trend of the iPhone in Region R grouping by date.
SELECT *Count()*, *date* **FROM** *stream*
WHERE *loc* **INSIDE** R **AND** *text* **CONTAINS** "iphone"
GROUP BY *date*.

Example 2: Snapshot Aggregate Query. Find the hot topics in the given range in the last 10 minutes.
SELECT *Count*(), *topic*, *minute* **FROM** *stream*
WHERE *loc* **INSIDE** R **AND** *time* **AFTER** "10 mins ago"
GROUP BY *topic*, *minute* **ORDER BY** *Count*() **DESC**.

Example 3: Continuous Aggregate Query. Find the most-frequent terms of each topic on the objects that are within a region R. Continuously produce the result every 1 minute.
SELECT *TopK()*, *topic* **FROM** *stream*
WHERE *loc* **INSIDE** R **GROUP BY** *topic* **SYNC** 1 minute.

3.2 System Architecture

STAR has four components: parser, router, worker and aggregator.

Parser. The parser takes as input the streamed spatial objects and the queries from users. It parses the primitive attributes of each object, and generates the extracted ones. Then, it transforms a user's SQL query into a predefined data structure in STAR. The parsed queries are sent to the router.

Router. The router is responsible for workload partitioning. It maintains a global index to facilitate partitioning the workload.

Worker. The worker processes objects and queries. It builds in-memory object and query indexes. The worker performs the following operations: (1) On receiving an object, say o, the worker inserts o into the object index. Then, it checks the continuous-query index to find the queries whose results are affected due to o's arrival. If any query qualifies, then the worker sends the updated results to the aggregator. (2) On receiving a snapshot query, say q_s, the worker leverages the cached data to answer q_s by checking whether the maintained query cache structures can be used. Otherwise, it checks the indexed objects to answer q_s. The results are sent to the aggregator. (3) On receiving a continuous query q_c, the worker registers q_c into the in-memory continuous-query index.

Aggregator. The aggregator collects the partial results from workers, and computes the final result. It maintains an index to store the partial results for each query. When receiving a notification that a new query has arrived, it stores the query id, and waits for the results from workers. For a snapshot query, after receiving all the partial results, the aggregator computes and outputs the final result immediately. For a continuous query, the aggregator outputs the result according to the result's refresh-rate specified by the query.

4 WORKLOAD PARTITIONING

STAR partitions workload with three considerations: **(1) Data Locality.** Records that are close to each other should be assigned to the same partition. **(2) Load Balance.** Partitions should be roughly of the same load. **(3) View Maintenance.** STAR materializes views to support queries. Maintenance of views needs to be considered.

SpatialHadoop [14] partitions spatial data using an R-tree or a grid index. In-memory distributed spatial systems, e.g., Simba [42], SpatialSpark [45], LocationSpark [39], and GeoSpark [46] typically use spatial partitioning methods, e.g., R-tree-based partitioning strategy, quadtree, and grid indexes. These systems do not consider the effect of view maintenance on workload partitioning in addition to the workload from querying the streamed data.

4.1 Workload and Partitioning

An important design decision of STAR is to build and maintain in-memory materialized views to support ad hoc aggregate queries. First, we introduce these views, and then present their effect on workload and partitioning.

Materialized Views. A view is a derived relation that is defined by a query. A view is said to be **materialized** [20, 22] if its derived relation that contains the result of the view's query is stored persistently into the system. STAR materializes views into memory. Materialized views can accelerate the processing of queries in STAR. To process a query, STAR reads the content of the corresponding materialized view that is most relevant to the query. Alternatively, STAR can rewrite the query to make use of the most relevant materialized views to answer the query.

STAR selects a set of views to be materialized in a cluster of workers. The target is to optimize system performance with these in-memory materialized views. Lot of work exists for addressing various aspects of view maintenance, e.g., [20, 22, 44]. STAR is the

first work to utilize materialized views to optimize the performance of spatial data analytics. However, maintaining these views comes at a cost. Thus, we also consider balancing the load of workers where the load includes spatial object processing, query processing and view maintenance. Next, we define the load of a worker.

Definition 1. ***Load of a Worker:*** *Given a time period, the load of a worker w_i during this period can be estimated as follows:*

$$L_i = c_1 \cdot |O| + c_2 \sum_{o \in O} n_1(o, Q_c) + c_3 \sum_{o \in O} n_2(o, V) + c_4 \sum_{q \in Q_s} (n_3(q, V) + n_4(q, O)), \quad (1)$$

where O is a set of spatial objects that has arrived to the worker in this time period, Q_c is a set of continuous queries handled by this worker, Q_s is a set of snapshot queries handled by this worker, V is a set of materialized views stored in this worker, $n_1(o, Q_c)$ is the number of continuous queries processed for o, $n_2(o, V)$ is the number of views updated for o, $n_3(q, V)$ is the sum of the sizes of the views accessed for q, and $n_4(q, O)$ is the number of the objects accessed for q. c_1 is the average cost of inserting an object, c_2 is the average cost of processing continuous queries for an object, c_3 is the average cost of updating views, and c_4 is the average cost of processing a snapshot query. ■

The load of one worker comprises processing spatial objects, processing queries, and maintaining the views managed by this worker. We define the cost of processing a snapshot query as the sum of the costs of accessing materialized views and objects.

Definition 2. ***Partitioning and View Materialization Problem:*** *Given a set of spatial objects O, a set of snapshot queries Q_s, and a set of continuous queries Q_c, the Partitioning and View Materialization problem is to split O, Q_s and Q_c into m subsets, where m is the number of workers, and to materialize a set of views S_i for each triplet (O_i, Q_i^s, Q_i^c) $(1 \leq i \leq m)$, where O_i is a subset of O, Q_i^s is a subset of Q_s, and Q_i^c is a subset of Q_c. The objective is to minimize $\sum_{i=1}^{m} L_i$, subject to the following constraints: (1) $\forall 1 \leq i \leq m$, $Mem(S_i) \leq C_i$, where $Mem(S_i)$ is the memory usage of S_i, and C_i is the memory capacity of worker w_i, and (2) $\forall i \neq j$, $L_i/L_j \leq \sigma$, where σ is a small constant value larger than 1.* ■

The partitioning and view materialization problem materializes a set of views for each worker, and aims at minimizing the total amount of load. The first constraint is that the memory usage of the materialized views does not exceed the memory capacity of each worker. The second constraint is that the workers should have balanced load. The Problem is NP-hard as it can be reduced from the minimum set cover problem [20].

4.2 Partitioning Algorithm

Due to the hardness of the partitioning and view materialization problem, we investigate heuristic algorithms to solve it. The fact that the results of data partitioning and view materialization are dependent on each other makes the task even harder. A heuristic solution is to partition the data first, and then to select the set of views to be materialized for each partition. Unfortunately, even this simple solution is challenging because either data partitioning or view materialization is a formidable task to be handled (both problems are NP-hard). We assume for now that we have an oracle to materialize views for a data partition, and focus on explaining the partitioning procedure. In the next section, we discuss the view materialization algorithm.

Algorithm Overview. The main idea is to construct a quad-tree [35] by recursively partitioning the most loaded node, and then assigning leaf nodes of the quad-tree to workers, aiming to achieve load balance and data locality. The algorithm can be divided into two phases. In Phase 1, we initialize a quad-tree with one root node, and recursively partition the node with the maximum estimated load until the number of nodes is larger than the required number of partitions. In each iteration, we call a function to estimate the load of each node, and partition the node having the maximum load. According to Definition 1, the load of a worker is related to the objects, queries, and materialized views. Because we cannot determine in advance which views will be materialized before assigning nodes to a partition (i.e., a worker), we estimate the load of a node by only considering the objects and queries, which is computed by $c_1|O| + c_2|O| \cdot |Q_c| + c_4|O| \cdot |Q_s|$.

In Phase 2, we assign leaf nodes to different partitions, and compute the set of views to be materialized for each partition. Then, we check if the load balance constraint can be satisfied. If this is the case, then we output the quad-tree and the partitions. Otherwise, we partition the leaf node having the maximum load, and repeat the above procedure. We have two objectives for the assignment of nodes to partitions: (1) We attempt to locate neighboring leaf nodes into the same partition. The reason is that some queries may overlap multiple adjacent nodes. Assigning them to different partitions will increase the total amount of load. (2) We attempt to balance the load of different workers.

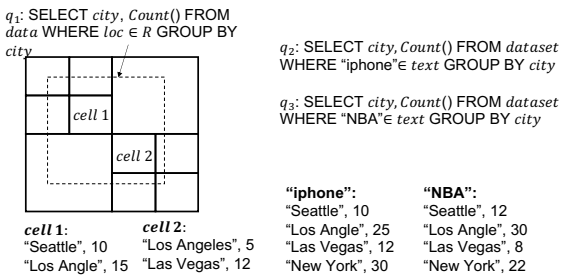
Assigning Nodes to Partitions. This function assigns leaf nodes of the quad-tree to partitions, aiming to achieve the two design objectives above. To achieve load balance, first, we estimate the average load each partition should have that we denote by L_{avg}. Then, we access the leaf nodes of the quad-tree in a depth-first manner, and assign the leaf nodes to different partitions so that the load of each partition is close to L_{avg}, and the adjacent nodes in the quad-tree order are assigned to the same partition.

5 CACHE-BASED OPTIMIZATIONS

In this section, we explain the cache-based mechanism that STAR adopts for processing snapshot queries, which is the main novelty of STAR. It comprises query- and object-based caching.

5.1 Query-based Caching

Query-based caching facilitates processing snapshot queries by materializing a set of views based on historical queries (and that is why we name it "query-based caching"). It maintains a selected set of views per worker. View selection is a classical problem in data warehousing, and has been extensively studied [20, 31]. However, in STAR, we investigate whether views defined for spatial and textual attributes can optimize processing aggregate queries over spatial data streams. STAR is the first to utilize materialized views to optimize spatial data analytics. However, materializing stream-based views may induce significant overhead, and deserves more consideration. STAR materializes the following views into



(a) Views for the range constraint. (b) Views for the keyword constraint.

Figure 1: Views for the queries.

memory: (1) Views for algebraic aggregate functions, and (2) Views for *TopK* aggregate functions. The former is similar to those for relational databases while the latter is not investigated in the view selection literature. For the first type of views, STAR has a new load-aware view materialization algorithm, and introduces the notion of *domination* among views that is defined based on the load, and that improves the effectiveness of the classic greedy algorithm by more than 50% according to our experiments. For the second type of views, STAR has an approximate solution that maintains a summary structure with a performance guarantee.

5.1.1 Views for Algebraic Aggregate Functions. The result of an algebraic aggregate function can be computed as follows: (1) Partition the input into disjoint subsets, (2) Compute the aggregate result for each subset, and (3) Aggregate the partial results. For simplicity, we illustrate using Aggregate Function *Count*. However, the proposed techniques can be extended easily to support other algebraic aggregate functions, e.g., *Avg* and *Sum*.

Example. Figure 1 gives an example of views. In Figure 1(a), q_1 returns the number of objects for each city, and has a range constraint that covers Cells 1 and 2. The views maintained in Cells 1 and 2 are two sets of key-value pairs. Only the cells covered by the query need to maintain this view. To answer q_1, we merge the views in Cells 1 and 2, and scan the objects in the other overlapped cells to compute the result. Figure 1(b) gives two views for the queries that have a keyword constraint. To answer q_2 and q_3, STAR produces the corresponding view as output.

Domination. Observe that when a set of views is materialized, selecting another view to be materialized may result in a bigger load. Consider a candidate view v_c, and a set of materialized views S, $L(S \cup \{v_c\}) - L(S) > 0$, where $L(.)$ is computed using Eqn 1. Although STAR can benefit from materializing a new view v_c by gaining efficiency in answering a set of queries, this benefit can be outweighed by the burden of maintaining the new view. We define *domination* between views to capture this.

Definition 3. ***Domination:*** *Given two views v_a and v_b, v_a is dominated by v_b iff (1) $Q(v_a)$ can be answered using v_b, where $Q(v_a)$ represents the set of queries that can be answered using v_a, and (2) $L(\{v_b\}) < L(\{v_a, v_b\})$.* ■

Based on this definition, when a view, say v_c, is selected for materialization, the views dominated by v_c are removed from the candidate views for materialization.

Generating Candidate Views. Another problem is how to generate the set of candidate views. We insert every query q into a quad-tree, and find the largest quad-tree node (denoted by n_q) that is covered by q's query range. Then, the view defined over the objects in n_q that can help answer q will be added to a list of candidate views. This strategy is based on the domination definition. The rationale for it is to reduce the number of candidate views.

Load-aware View Materialization. STAR selects recursively the view that has the largest benefit per unit space. The benefit of a view w.r.t. a set of materialized views S is computed by:

$$B(v, S) = \max(L(S) - L(S \cup \{v\}), 0), \tag{2}$$

where $L(S)$ is the load of the worker due to S (Eqn 1). The benefit of a view v per unit space is $B(v, S)/n_v$. The main operation in this algorithm is finding the view that has the maximum benefit, and thus has a time complexity of $O(n^2)$, where n is the number of candidate views. It runs at most C iterations, where C is the memory capacity for the materialized views and is a system-defined parameter. Thus, the time complexity of the algorithm is $O(Cn^2)$.

5.1.2 Views for the TopK Aggregate Function. The result of an algebraic aggregate function can be computed by aggregating the partial results for each subset of the data. However, this technique does not work for the *TopK* aggregate function as it requires computing over the complete dataset. Due to the fast arrival of streamed objects and large vocabulary size, it is not practical to maintain an accurate view for a *TopK* query. We propose to maintain a summary structure as a view that contains a small number of key-value pairs. To save memory space, we do not employ techniques that have dynamic summary size, e.g., [36]. Instead, we maintain a SpaceSaving summary [32] that estimates the frequency of any term t with additive error ϵn using $O(1/\epsilon)$ memory space, where n is the number of objects. For a parameter m that is specified based on the available memory size, the SpaceSaving summary maintains at most m counters. m is set automatically by the system or is provided by a system administrator. When a new term t arrives, SpaceSaving summary checks if t has been maintained in the summary, and increments t's counter. Otherwise, let t_m be the term having the least frequency in the summary. t replaces t_m and increases the counter by 1. To answer a *TopK* query, the summary can output the k terms having the largest counters. A term t is guaranteed to be among the top-k most-frequent terms if $C[t] - \epsilon_t > C_{k+1}$, where C_{k+1} is the $(k+1)$-th largest counter.

Theorem 1. *For a TopK query, by using $O(1/\epsilon)$ memory space, SpaceSaving summary guarantees that terms having frequency larger than $(1-\epsilon)F_k$ are included in the result, where F_k is the frequency of the k-th most-frequent term.*

Proof: Assume that the SpaceSaving summary maintains $\frac{n}{\epsilon F_k}$ counters that take $O(1/\epsilon)$ memory space. Then, the maximal possible overestimation error will be ϵF_k. Therefore, all the terms included in the result have frequencies that are larger than $(1-\epsilon)F_k$. ■

According to Theorem 1, SpaceSaving summary provides accuracy guarantees for a *TopK* aggregate function. In the case that a

query has ad hoc constraints, STAR maintains multiple SpaceSaving summaries, one for each subset of data that is partitioned according to the constraints. Agarwal *et al.* [3] have proven that Theorem 1 still holds when merging multiple SpaceSaving summaries.

5.1.3 Using Views for Processing Queries. STAR organizes views using a quad-tree. Each node in the quad-tree maintains a set of materialized views (the empty set is also possible for some nodes). To explain the procedure for processing a query, say q, we start with a simplified case when q's query range matches a quad-tree node, say n_s. If n_s is a leaf node, we select the most cost-efficient view(s) in n_s to answer q, or access the objects in n_s if these view(s) do not exist. When n_s is a non-leaf node, we compare the costs of using n_s and using the child nodes of n_s to answer q, and choose the one that has the smaller cost. STAR uses Eqn 1 to compute the cost. If one of the child nodes n_c is also a non-leaf node, we recursively compare the costs of using n_c and using n_c's child nodes. Specifically,

$$L(q, n_s) = \begin{cases} cost(n_s), & \text{if } n_s \text{ is a leaf} \\ min(cost(n_s), \sum_{n_c \in n_s.children} L(q, n_c)), & \text{otherwise} \end{cases}$$

This recursive computation is efficient because we maintain the sizes of the materialized views and the number of objects in each node. Then, we access either the materialized views or the objects, accordingly, to compute the result.

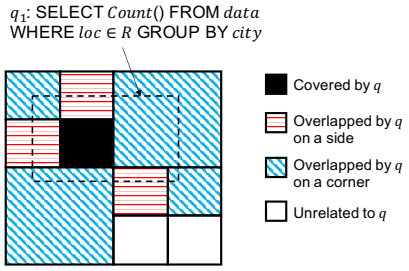


Figure 2: A query range overlaps multiple quad-tree nodes.

Next, we explain the case when the query range is not a quad-tree node. The overall procedure is identical to processing the simplified case except that we cannot utilize the views of a node that is not covered by the query range. For nodes that are partially overlapped, we need to access objects in them to compute the query result. Figure 2 gives an example query that overlaps multiple quad-tree nodes. Nodes may overlap the query by a side or by a corner, e.g., nodes with slash pattern and with horizontal line pattern, respectively. The overhead may be large when many partially overlapped nodes exist with many objects in them. To reduce query processing time, we propose to cache objects in the next subsection.

5.2 Object-based Caching

We present the idea of caching objects that works seamlessly with the materialized views. The main idea is to cache objects in the borders of a node that overlap the query range so that we only need to access the cached objects rather than the entire object set in the node to answer queries. We propose an approximation algorithm to decide a set of cached objects to optimize query processing with theoretical guarantees.

Definition 4. ***Caching Region:*** *For a quad-tree node n_p that partially overlaps queries, there are two types of caching regions for n_p. A side caching region of n_p is a rectangle inside n_p that has one side being set as one of the border lines of n_p. A corner caching region of n_p is a circular sector inside n_p whose center is a corner point of n_p and has the angle being equal to 90 degree.* ■

Figure 3 gives example caching regions. The smaller gray node has two candidate side caching regions that have a height equal to the shorter or longer bidirectional arrow, respectively. The larger gray node has two candidate corner caching regions that have a radius equal to the shorter or longer bidirectional arrow, respectively.

For a quad-tree node, we maintain at most 8 caching regions, i.e., four for the sides and the other four for the corners. For a query q that covers a side of a node n_c (but not the full node region), only the closest side caching region of n_c to q can be used (the overlapped area should be inside the caching region, otherwise, we do not use the cache). For a query q' that covers a corner of a node n_c (again, not the full node region), we select among the closest corner caching region and the two closest side caching regions. We use the one that has the smallest number of cached objects among the ones covering the overlapped area. Next, we define the *Object Caching Problem.*

Definition 5. ***Object Caching Problem****: Given a set of spatial objects O, a set of snapshot queries Q_s, and a quad-tree T organizing the materialized views, the Object Caching problem is to decide a caching region (can be empty) for each border or corner of the nodes in T. We aim to maximize $\sum_{q \in Q_s} \sum_{n_p \in N_p} (|O_{n_p}| - |O_c|)$ subject to the constraint that the total number of cached objects is smaller than B, where N_p is the set of partially overlapped nodes for q, O_{n_p} is the set of objects in n_p, O_c is the set of objects in the caching region that are used for answering q, and B is the memory capacity for caching.* ■

Theorem 2. *The Object Caching problem is NP-hard.*

Proof Sketch: This can be proved by reducing from the Knapsack problem. ■

Load-based Greedy Algorithm. We introduce a greedy algorithm to determine caching regions. Considering a set of partially overlapped queries Q and the overlapped area being A, a caching region r that covers A can accelerate the processing of each query in Q. The overall load improvement will be $\Delta L = c_2 \sum_{q \in Q} (n - n_r)$ based on Eqn 1, where n and n_r denote the number of objects in the node and in r, respectively. Figure 3 gives an example of caching regions. Both q_1 and q_2 partially overlap the smaller gray node. If we build a caching region with the height being equal to the shorter bidirectional arrow, we can reduce the running time of q_1. If the height of the caching region equals to the longer bidirectional arrow, both q_1 and q_2 can be accelerated. Based on this observation, we propose a greedy algorithm that decides the caching regions in descending order of $\frac{\Delta L(R,r)}{n_r}$, where R denotes the current set of caching regions, and $\Delta L(R, r)$ denotes the load improvement after adding r. When adding r into R, we delete candidate caching regions that are covered by r.

Theorem 3. *The load improvement of the caching regions R produced by the greedy algorithm is at least 63% of that of the optimal solution using the same amount of space as R.*

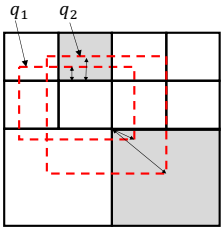


Figure 3: Caching regions of a node for queries.

Proof: The proof is based on the observation that

$$\Delta L(\{r_1, r_2\}, R) \leq \Delta L(r_1, R) + \Delta L(r_2, R)$$

that can be easily extended to $\Delta L(G, R) \leq \sum_{r \in G} \Delta L(r, R)$, where G denotes a set of caching regions.

Let R be the set of caching regions produced by the greedy algorithm, $\Delta L(R)$ be the load improvement of R, and θ_R be the memory used by R. Assume that the optimal solution using θ_R units of memory space produces R^*, and the load improvement of R^* is $\Delta L(R^*)$.

Consider that during the running of the greedy algorithm, Caching Region R_k has been selected, and R_k consumes k units of memory space. R_k has the load improvement $\sum_{i=1}^{k} b_i$, where b_i is the load improvement of adding the ith unit of memory space. Observe that the load improvement of the set $R^* \cup R_k$ is at least $\Delta L(R^*)$, i.e., the load improvement of R^* with respect to R_k is at least $\Delta L(R^*) - \sum_{1 \leq i \leq k} b_i$: $\Delta L(R^*, R_k) \geq \Delta L(R^*) - \sum_{i=1}^{k} b_i$.

According to this earlier observation, we deduce that

$$\Delta L(R^*, R_k) \leq \sum_{r \in R^*} \Delta L(r, R_k). \tag{3}$$

There exists a caching region $r_t \in R^*$ satisfying $\Delta L(r_t, R_k)/\theta_{r_t} \geq \Delta L(R^*, R_k)/\theta_R \geq (\Delta L(R^*) - \sum_{i=1}^{k} b_i)/\theta_R$, where θ_{r_t} is the memory space of r_t. Otherwise, inequality 3 will not hold.

The load improvement per unit space of the caching region r_g selected by the greedy algorithm with respect to R_k is at least $\Delta L(r_t, R_k)/\theta_{r_t}$ that is at least $(\Delta L(R^*) - \sum_{i=1}^{k} b_i)/\theta_R$. By distributing the benefit of r_g over each of its unit memory spaces, we get $b_{k+j} \geq (\Delta L(R^*) - \sum_{i=1}^{k} b_i)/\theta_R$, for $0 < j \leq \theta_{r_g}$, where θ_{r_g} is the memory used by r_g. The above equation applies to each caching region that is selected by the greedy algorithm. Thus,

$$\Delta L(R^*) \leq \theta_R b_j + \sum_{i=1}^{j-1} b_i, \text{ for } 0 < j \leq \theta_R. \tag{4}$$

Multiplying the jth equation by $(\frac{\theta_R - 1}{\theta_R})^{\theta_R - j}$ and adding all the equations, then $\sum_{i=1}^{\theta_R} b_i / \Delta L(R^*) \geq 1 - 1/e \approx 0.63$. ∎

Theorem 3 provides a theoretical bound on the performance of the greedy algorithm. Notice that the savings achieved from having caching regions are orthogonal to the query-based caching, i.e., maintaining materialized views. Thus, having caching regions can reduce query time even without using query-based caching.

Maintaining Caching Regions. Due to insertions of new objects, the memory constraints for the caching regions may get violated. To handle this, STAR adopts an eviction policy that removes the Least Recently Used (LRU) caching region. It keeps removing caching regions in the LRU order until the memory constraint is satisfied.

6 SUPPORT FOR CONTINUOUS QUERIES

Unlike existing distributed streamed systems [11, 30] for spatial data that are optimized for continuous range queries, STAR deals with aggregate continuous queries with ad hoc constraints, whose results are a set of key-value pairs. STAR adopts the following strategy: We maintain the result of each continuous query, and update the result using new spatial objects. For a new object, we check it against each continuous query and find the set of queries whose constraints are satisfied by the object. For these queries, we update their query results accordingly: (1) For a query with an algebraic aggregate function, we compute the effect of the new object on the query result, and update the result. (2) For a query with *TopK* aggregate function, we use a SpaceSaving summary to store its result, and update its result for each new object as we do for maintaining views for the *TopK* aggregate function.

STAR also exploits the following optimization. Consider that there are a set of queries that arrive at the system at almost the same time, and that have the same group-by attribute(s). They have query ranges that overlap with each other, and have the same ad hoc constraints except query ranges. STAR divides the result of such query into two parts: one part that is in the overlapped area and the other part that is not. For the first part, STAR maintains a single copy of result for the queries so that reducing the result updating cost due to objects falling in the overlapped area. For the other part, STAR still maintains a distinct result for each query.

We term these overlapping queries *template queries*, and we index them using a grid index. For a cell that is covered by template queries, we maintain one copy of the result. However, the additional cost is that we will need to aggregate the partial results of multiple cells when the sync period of a query is reached. To avoid large aggregation costs, we predefine a threshold θ of aggregation costs that can be set based on user tolerance to query response time. If a query, say q, satisfies $\sum_{g \in G_q} n_e(q, g) A(q)/A(g) > \theta$, where G_q is the set of cells overlapping q, $A(\cdot)$ is a function that computes the area of a query range or a cell, and $n_e(q, g)$ is an estimation of the cardinality of q's partial result in g, we maintain q's complete result in the same way we do for non-template queries.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup

We deploy STAR on the Amazon EC2 platform using a cluster of 8 c5d.2xlarge instances with 10G network bandwidth. Each c5d.2xlarge has 8 vCPUs running Intel Xeon Platinum 8000-series Processors with 3.5GHz and 16GB RAM. To simulate the streaming scenario, we deploy Apache Kafka on another storage optimized instance i3.4xlarge for emitting streamed data to STAR, which has 16 vCPUs running Intel Xeon E5 2686 v4 Processor at 2.3GHz and 122GB RAM. Apache Kafka [1] is a popular framework for building real-time data pipelines and streamed applications.

Datasets and Queries. We evaluate STAR using a real dataset *Tweets*. The Tweets dataset consists of 500 million tweets in America, each of which has the attributes of *loc*, *text* and *time*. We use tools to extract derived attributes from *loc*, *text* and *time*, respectively. Due to lacking of real-life ad hoc aggregation queries over Tweets, we synthesize both snapshot and continuous queries based

Parameter	Value
Aggregate function	$Count$, $TopK()$
Number of constraints	1, 2, 3, 4
Side length of the range	0.05%, 0.1%, 0.2%
Number of keywords	1, 2, 3
Interval length on $time$	10min, 20min, 30min
Equality value on $topic$	a random value among 50 topics
Sync time	1min, 5min, 10min

Table 1: Possible values for parameters.

on Tweets for evaluation. To synthesize a query with ad hoc constraints, we synthesize a constraint on Attribute loc, $text$, $time$ and $topic$ (a derived attribute extracted from $text$), respectively, each of which is of a different type: a range constraint on loc, a keyword constraint on $text$, an interval constraint on $time$, and an equality constraint on $topic$. we also define an *aggregation function* and *group-by attribute(s)*. For continuous queries, we define an additional *sync time*.

Table 1 shows the possible values of each query parameter. $TopK()$ uses a default parameter $k = 10$. Each query has a number of constraints that are selected randomly. *Range constraint* is created by defining a square whose upper left point is the coordinates of a random tweet in Tweets. *Keyword constraint* is created by selecting a set of keywords randomly from Tweets. *Interval constraint* wants the result on the objects that arrived within the past a period of time. *Equality constraint* requires that Attribute $topic$ equals to a value selected from 50 topics. The parameter value for each constraint is selected randomly from the values in Table 1.

For both snapshot and continuous queries, we synthesize two types of queries that have different data distributions of *group-by attribute(s)*. We first enumerate all possible combinations of derived attributes to create the set of *group-by attribute(s)*. For the first type of queries, we select the *group-by attribute(s)* from the set randomly. However, in real-life scenario, users are usually more interested in a small ratio of *group-by attribute(s)*, and users at different positions tend to have different interested *group-by attribute(s)*. Therefore, we synthesize another type of queries. We partition the spatial space into 10×10 uniform cells, and for each cell we randomly pick a *group-by attribute(s)* from the set of *group-by attribute(s)*, which we call it as *pivot*. Each query, based on the cell which its upper left point resides, has a probability of P using the corresponding *pivot* as the *group-by attribute(s)*, and $1 - P$ probability using a random *group-by attribute(s)*. In our experiments, we set P as 0.7. We classify our queries as follows:

Q_{S1}-Count, Q_{S1}-TopK: Both are snapshot queries. The *group-by attribute(s)* are randomly selected. Q_{S1}-Count uses $Count()$ as the aggregation function, and Q_{S1}-TopK uses $TopK()$ as the aggregation function.

Q_{S2}-Count, Q_{S2}-TopK: Both are snapshot queries. The *group-by attribute(s)* are selected using the *pivot* based method. Q_{S2}-Count uses $Count()$ as the aggregation function, and Q_{S2}-TopK uses $TopK()$ as the aggregation function.

Q_{C1}-Count, Q_{C1}-TopK: Both are continuous queries. The other settings are the same as Q_{S1}-Count and Q_{S1}-TopK.

Q_{C2}-Count, Q_{C2}-TopK: Both are continuous queries. The other settings are the same as Q_{S2}-Count and Q_{S2}-TopK.

Workload. The arrival speed of a spatio-textual object is approximately 10 times of the arrival speed of a snapshot or a continuous query. We evaluate our system after the system digests and processes objects and queries for 10 minutes.

7.2 Evaluation on Snapshot Queries

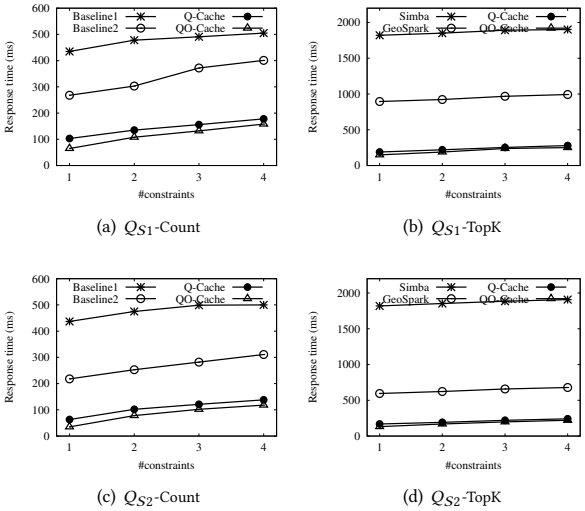
To evaluate the performance of STAR on processing snapshot queries, we compare STAR with the following two baselines that are variants of STAR:

Baseline-1. Baseline-1 does not use any cache-based technique. The other techniques used are the same as STAR.

Baseline-2. Baseline-2 differs from Baseline-1 only in that it uses a classic greedy algorithm to materialize views for queries without a spatial or keyword constraint.

Note that no existing system is designed to support snapshot aggregate queries over spatial data streams. We will extend representative existing systems for comparison in Section 7.3. We evaluate the performance by the query response time.

Query Response Time. Query response time is the average time required for answering a query. To avoid long queuing time in the buffer, we measure query response time by using a moderate input speed of the data stream. We evaluate the performance of our cache-based algorithms: Q-cache represents using the query-based caching algorithm and QO-cache represents using both query-based caching and object-based caching algorithms.

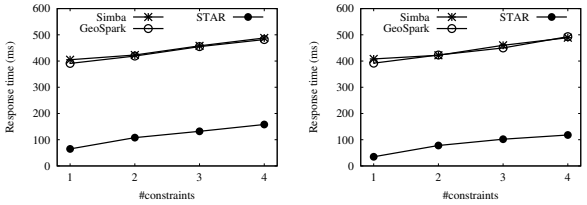


(a) Q_{S1}-Count (b) Q_{S1}-TopK

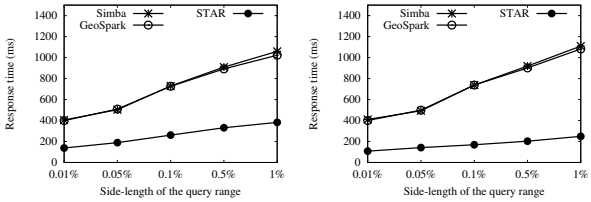
(c) Q_{S2}-Count (d) Q_{S2}-TopK

Figure 4: Query response time comparison for snapshot queries.

Figure 4 gives the experimental results. We observe that both Q-cache and QO-cache show a significant performance improvement over the baselines: they are about one magnitude faster than Baseline-1 when the number of constraints is 1 and 4–9 times faster when there are more than one constraint; they are 2–3 times faster than Baseline-2. QO-cache has the best performance, which improves the performance of Q-cache by 10% – 40%. This is because that QO-cache maintains materialized views and uses cached objects to help processing queries, which avoids checking a large amount of objects. Baseline-1 performs the worst as it always needs

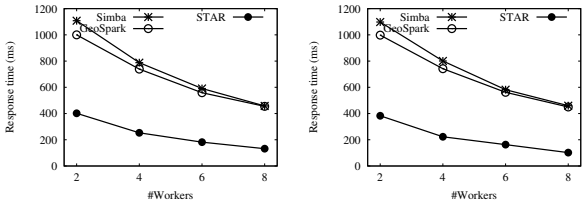


(a) Q_{S1}-Count (b) Q_{S2}-Count



(c) Q_{S1}-Count (#constraints=3) (d) Q_{S2}-Count (#constraints=3)

Figure 5: Comparison with existing systems



(a) Q_{S1}-Count (#constraints=3) (b) Q_{S2}-Count (#constraints=3)

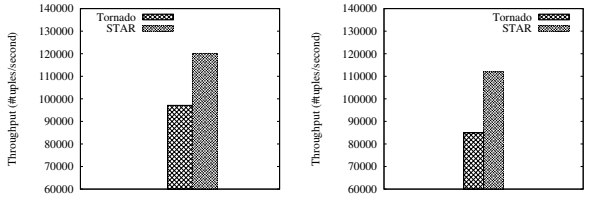
Figure 6: Scalability

to check the objects to answer queries. Baseline-2 is at least 1 time faster than Baseline-1, indicating that views without spatial or textual attributes are also helpful. We also observe that Baseline-2, Q-cache and QO-cache have smaller query response time for Q_{S2} queries than for Q_{S1} queries. The reason is that for the uneven distribution of *group-by attribute(s)* in Q_{S2} queries, the query-based caching algorithm is more likely to materialize the views having larger benefits, which helps reducing the query response time.

7.3 Comparison with Existing Systems

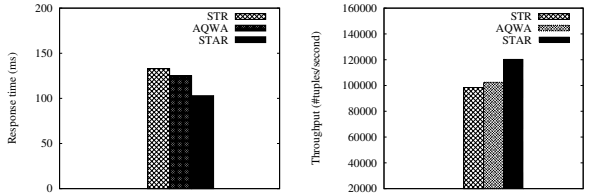
No existing system is able to support snapshot aggregate queries over spatial data streams, and thus no existing systems can be compared directly. We extend Simba [42] and GeoSpark [46], two representative distributed spatial data analytics systems, for comparison. However, both Simba and GeoSpark cannot work on streamed data. To make the comparison feasible, we introduce the following setting: (1) We create a static spatial data set by pre-loading STAR, Simba, and GeoSpark with a static set of tweets. (2) We input the queries with at least one range constraint.

Figures 5(a) and 5(b) show the query response time with respect to the number of constraints. STAR is about 3 times faster than Simba and GeoSpark. Though Simba and GeoSpark are designed for spatial data analytics, they are not optimized for aggregate queries with ad hoc constraints. The results demonstrate the effectiveness of the cache-based algorithms adopted by STAR. To further compare their performance, we vary the size of the query range to investigate the impact on the query response time. Figures 5(c)



(a) Q_{C2}-Count (b) Q_{C2}-TopK

Figure 7: Query throughput comparison with Tornado.



(a) Q_{S1}-Count (#constraints=3) (b) Q_{C1}-Count (#constraints=3)

Figure 8: Comparing different partitioning schemes.

and 5(d) show that STAR has a much smaller query response time, e.g., in Figure 5(d), the query response time of STAR is smaller than 40% of the query response times of other systems. The running time of all the systems increases with the increase of the query range. However, the running time of STAR is more stable, which is ascribed to the cached data maintained in STAR.

Figure 6 gives the result on scalability with the number of workers. STAR is 2–3 times faster than Simba and GeoSpark no matter how many workers are used. The results show that STAR scales well with the system size.

The results show that STAR outperforms Simba and GeoSpark in processing ad hoc aggregate snapshot queries over a static set of spatial data, although STAR is designed for streamed data. The main reason is that STAR exploits cache-based algorithms to optimize processing ad hoc aggregate queries over spatial data.

7.4 Evaluation on Continuous Queries

For continuous aggregate queries, we compare STAR with Tornado [30], a state-of-the-art system that supports continuous queries with ad hoc spatial and textual constraints over spatial data streams. We extend Tornado to support aggregate continuous queries for spatial data. Tornado only indexes continuous queries, but not spatial objects (and thus it cannot answer snapshot queries). Figure 7 gives the result of comparing throughputs. STAR has larger throughputs than Tornado for all groups of queries. For example, in Figures 7(a) and 7(b), the throughput of STAR is larger than Tornado by 23% and 32%, respectively. This is because that the optimization techniques adopted by STAR can reduce the cost of processing continuous queries. The results demonstrates that STAR outperforms Tornado in processing aggregate continuous queries.

7.5 Workload Partitioning

We evaluate our workload partitioning scheme by comparing it with two partitioning schemes: STR [27] and AQWA [5]. Figure 8 gives the experimental results. We observe that our partitioning scheme has the best performance: In Figure 8(a), STAR has about

20% smaller query response time than the others; In Figure 8(b), STAR has about 18% larger throughput than the others. The results demonstrate the effectiveness of our partitioning scheme.

8 CONCLUSIONS

In this paper, we present STAR; a distributed in-memory data stream warehouse system that provides low-latency and up-to-date analytical results over a fast arriving spatial data stream. STAR supports aggregate queries that have ad hoc constraints over spatial, textual and temporal data attributes. STAR adopts an effective workload partitioning strategy that partitions the workload composed of object processing, query processing, and view maintenance. Moreover, STAR implement a novel cache-based mechanism that significantly reduces the run-time of analytical queries over streamed spatial data. Extensive experiments over real data sets demonstrate the superior performance of STAR over existing systems.

ACKNOWLEDGMENTS

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF–ICP) Funding Initiative, as well as cash and in-kind contribution from Singapore Telecommunications Limited (Singtel), through Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU). Walid G. Aref acknowledges the support of the U.S. National Science Foundation under Grant Numbers: IIS-1910216 and III-1815796.

REFERENCES

[1] [n.d.]. Apache Kafka. https://kafka.apache.org. Accessed: 2021-01-05.
[2] [n.d.]. Apache Storm. http://storm.apache.org/. Accessed: 2021-01-05.
[3] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *TODS* (2013).
[4] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *PVLDB* (2013), 1009–1020.
[5] Ahmed M Aly, Ahmed R Mahmood, Mohamed S Hassan, Walid G Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: adaptive query workload aware partitioning of big spatial data. *PVLDB* 8 (2015), 2062–2073.
[6] Magdalena Balazinska, YongChul Kwon, Nathan Kuchta, and Dennis Lee. 2007. Moirae: History-Enhanced Monitoring.. In *CIDR*. 375–386.
[7] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.
[8] Lisi Chen, Gao Cong, and Xin Cao. 2013. An Efficient Query Indexing Mechanism for Filtering Geo-textual Data. In *SIGMOD*. 749–760.
[9] Yue Chen, Zhida Chen, Gao Cong, Ahmed R Mahmood, and Walid G Aref. 2020. SSTD: A Distributed System on Streaming Spatio-Textual Data. *PVLDB* 13 (2020).
[10] Zhida Chen, Gao Cong, and Walid G Aref. 2020. STAR: A Distributed Stream Warehouse System for Spatial Data. In *SIGMOD*. 2761–2764.
[11] Zhida Chen, Gao Cong, Zhenjie Zhang, Tom Z.J. Fu, and Lisi Chen. 2017. Distributed Publish/Subscribe Query Processing on the Spatio-Textual Data Stream. In *ICDE*. 1095–1106.
[12] Anna Ciampi, Annalisa Appice, Donato Malerba, and Angelo Muolo. 2011. Space-time roll-up and drill-down into geo-trend stream cubes. *Foundations of Intelligent Systems* (2011), 365–375.
[13] Roozbeh Derakhshan, Bela Stantic, Othmar Korn, and Frank Dehne. 2008. Parallel simulated annealing for materialized view selection in data warehousing environments. *Lecture Notes in Computer Science* 5022 (2008), 121–132.
[14] Ahmed Eldawy and Mohamed F Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*. 1352–1363.
[15] Wei Feng, Chao Zhang, Wei Zhang, Jiawei Han, Jianyong Wang, Charu Aggarwal, and Jianbin Huang. 2015. STREAMCUBE: Hierarchical spatio-temporal hashtag clustering for event exploration over the Twitter stream. In *ICDE*. 1561–1572.
[16] Thanaa M Ghanem, Ahmed K Elmagarmid, Per-Åke Larson, and Walid G Aref. 2010. Supporting views in data stream management systems. *TODS* 35 (2010).
[17] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk. 2009. Stream Warehousing with DataDepot. In *SIGMOD*. 847–854.
[18] Lukasz Golab, Theodore Johnson, Subhabrata Sen, and Jennifer Yates. 2012. A Sequence-Oriented Stream Warehouse Paradigm for Network Monitoring Applications. In *PAM*. Springer, 53–63.
[19] Marcin Gorawski and Rafal Malczok. 2010. Indexing Spatial Objects in Stream Data Warehouse. *Advances in Intelligent Information and Database Systems* 283 (2010), 53–65.
[20] Himanshu Gupta and Inderpal Singh Mumick. 1999. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *ICDT*. 453–470.
[21] Jiawei Han, Yixin Chen, Guozhu Dong, Jian Pei, Benjamin W. Wah, Jianyong Wang, and Y. Dora Cai. 2005. Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams. *Distributed and Parallel Databases* 18, 2 (2005), 173–197.
[22] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *SIGMOD*. 205–216.
[23] J.-T. Horng, Y.-J. Chang, and B.-J. Liu. 2003. Applying evolutionary algorithms to materialized view selection in a data warehouse. *Soft Computing* 7 (2003), 574–581.
[24] Huiqi Hu, Yiqun Liu, Guoliang Li, Jianhua Feng, and Kian-Lee Tan. 2015. A location-aware publish/subscribe framework for parameterized spatio-textual subscriptions. In *ICDE*. 711–722.
[25] Wilburt Juan Labio, Dallan Quass, and Brad Adelberg. 1997. Physical database design for data warehouses. In *ICDE*. 277–288.
[26] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. 2012. Muppet: MapReduce-style Processing of Fast Data. *PVLDB* 5 (2012), 1814–1825.
[27] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*. 497–506.
[28] Lauro Lins, James T Klosowski, and Carlos Scheidegger. 2013. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *TVCG* 19 (2013), 2456–2465.
[29] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, and Saleh Basalamah. 2015. Tornado: A Distributed Spatio-textual Stream Processing System. *PVLDB* 8 (2015), 2020–2023.
[30] Ahmed R Mahmood, Anas Daghistani, Ahmed M Aly, Mingjie Tang, Saleh Basalamah, Sunil Prabhakar, and Walid G Aref. 2018. Adaptive processing of spatial-keyword data over a distributed streaming cluster. In *SIGSPATIAL*. 219–228.
[31] Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. In *SIGMOD*. 20–29.
[32] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *ICDT*. 398–412.
[33] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. 2011. Nova: Continuous Pig/Hadoop Workflows. In *SIGMOD*. 1081–1090.
[34] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD*. 447–458.
[35] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Academic Press.
[36] Anders Skovsgaard, Darius Sidlauskas, and Christian S Jensen. 2014. Scalable top-k spatio-temporal term querying. In *ICDE*. 148–159.
[37] Mingjie Tang, Ahmed M Aly, Ahmed R Mahmood, Thamir Qadah, Walid G Aref, and Saleh Basalamah. 2016. Cruncher: Distributed in-memory processing for location-based services. In *ICDE*. 1406–1409.
[38] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. Locationspark: A distributed in-memory data management system for big spatial data. *PVLDB* 9 (2016), 1565–1568.
[39] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB* 9 (2016), 1565–1568.
[40] Bin Wang, Rui Zhu, Xiaochun Yang, and Guoren Wang. 2017. Top-k representative documents query over geo-textual data stream. *WWW* (2017), 1–19.
[41] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Zengfeng Huang. 2016. Skype: Top-k Spatial-keyword Publish/Subscribe over Sliding Window. *PVLDB* 9 (2016), 588–599.
[42] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*. 1071–1085.
[43] Xiaopeng Xiong, Hicham G Elmongui, Xiaoyong Chai, and Walid G Aref. 2007. Place: A distributed spatio-temporal data stream management system for moving objects. In *MDM*. 44–51.
[44] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. Algorithms for materialized view design in data warehousing environment. In *VLDB*, Vol. 97. 136–145.
[45] S. You, J. Zhang, and L. Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *ICDEW*. 34–41.
[46] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. GeoSpark: A Cluster Computing Framework for Processing Large-scale Spatial Data. In *SIGSPATIAL*.