STAR: A Cache-based Stream Warehouse System for Spatial Data

ZHIDA CHEN, SCALE@Nanyang Technological University, Singapore
GAO CONG, SCALE@Nanyang Technological University, Singapore
WALID G. AREF, Purdue University, USA

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 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 both snapshot and continuous 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 snapshot queries that collectively utilizes the techniques of query-based caching (i.e., view materialization) and object-based caching. Moreover, to speed-up processing continuous queries, STAR proposes a novel index structure that achieves high efficiency in both object checking and result updating. 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**.

Additional Key Words and Phrases: spatial data, data stream, warehouse system, distributed system

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 smartphones, giving rise to a fast arriving spatial data stream. This spatially annotated data contains valuable information, and it 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 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. As another example, consider a manager of a food delivery company who wants to improve the company's delivery service by adjusting the distribution of deliverymen in different regions at different time of the day. To do so, he can issue aggregate queries that return the number of orders and deliverymen grouped by region and time. 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 [35], 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) [25, 30, 48, 53] provides real-time analytics over data streams. A DSWS efficiently ingests data and enables online analytical processing over the streamed data. It allows users to issue continuous queries that monitor

Authors' addresses: Zhida Chen, chen0936@e.ntu.edu.sg, SCALE@Nanyang Technological University, Singapore; Gao Cong, gaocong@ntu.edu.sg, SCALE@Nanyang Technological University, Singapore; Walid G. Aref, aref@purdue.edu, Purdue University, USA.

2374-0353/2023/6-ART $15.00
https://doi.org/10.1145/3605944

changes in the streamed data, as well as snapshot queries that report the current or past status of the warehoused data.

Although spatial data is explosive, research on distributed DSWSs that offer native spatial data stream analytics is still lacking. Most existing distributed systems, e.g., [4–6, 21, 63] focus on developing spatial data management systems over static data sets, but they are not designed for streamed data, and do not support continuous or snapshot ad hoc aggregate queries over spatial data streams. Existing distributed spatial data stream systems, e.g., [16, 45], do not support ad hoc aggregate queries. Besides, they only support continuous but not snapshot queries. BBoxDB [49] is a distributed system to store multidimensional data. BBoxDB Streams [50] is developed on top of BBoxDB that supports both snapshot and continuous queries on multidimensional data. However, BBoxDB Streams does not consider aggregate queries that are essential to a DSWS. Moreover, though supporting multidimensional data, BBoxDB Streams does not offer functions to process the spatial, temporal, and textual data attributes, collectively, thus making it difficult to extend BBoxDB Streams to support aggregate queries with ad hoc constraints on spatial, temporal, and textual data attributes.

It is challenging to develop a DSWS that supports both snapshot and continuous 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 solution with an effective workload partitioning scheme that is tailored for the workloads of processing objects, and analytical queries. Second, there exist notable differences in the mechanisms for processing snapshot and continuous ad hoc aggregate queries. Processing snapshot queries requires an efficient caching strategy that uses the cached data to answer the queries without having to recheck numerous indexed objects. In contrast, processing continuous queries requires an efficient indexing mechanism that categorizes and groups the continuous queries and incurs small update overheads over the streamed data.

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 both snapshot and continuous 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*. Currently, STAR does not support the spatial join operation that we plan to consider in future work. We develop a cache-based mechanism to facilitate the processing of snapshot queries that collectively utilizes query-based caching (i.e., view materialization) as well as object-based caching. Moreover, to speed-up processing continuous queries, STAR employs new indexing techniques to facilitate the processing of various types of queries. Finally, STAR adopts a workload partitioning method that achieves data locality and load balance. This paper extends our published conference paper [14]. We improve it extensively by adding optimizations of processing continuous queries, which is realized by proposing a novel index to categorize continuous queries with various predicates.

In particular, STAR has the following novelties:

- 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.
- STAR implements 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 it selects and maintains a set of views in-memory to speed-up 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 the object-based caching that provides competitive solution with theoretical bounds.

- To efficiently process continuous ad hoc aggregate queries, STAR features a novel index that categorizes the queries with various predicates, and that reduces the amount of queries to be checked for streamed objects.
- STAR adopts an effective and efficient workload partitioning strategy that is tailored to workloads that process objects, snapshot and continuous queries, as well as achieves load balance and data locality.

We evaluate STAR on Amazon EC2 with real spatial data. STAR achieves excellent performance for both snapshot and continuous queries, and outperforms the best baseline systems by up to 5× and 1.5×, respectively.

2 RELATED WORK

Data Stream Warehouse Systems. Based on the architecture, existing distributed warehouse systems can be classified into three types: (1) Systems that extend a database system with the abilities of fast data ingestion and real-time data evaluation [24, 26, 53], (2) Systems that extend a data stream processing system with the ability of exploring historical data [8, 40, 48], and (3) Systems that extend a distributed analytics framework, with the abilities of fast data ingestion and real-time data evaluation [18, 37, 52]. However, existing distributed warehouse systems do not provide native support for spatial data stream, and they are difficult to optimize for aggregate operations over the spatial data stream.

Some work exists for developing centralized stream warehouse systems over spatial data. Gorawski and Malczok [27] present an index structure to store spatial data in a stream warehouse. Lins *et al.* [41] and Giampi *et al.* [17] consider the problem of exploring streamed spatio-temporal data, and they propose a new data structure of views to achieve this. Feng et al. [22] propose solutions for exploring events from streamed geotagged 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 spatial data. We categorize them as systems for static spatial data and systems for streamed spatial data.

(1) Systems for static spatial data [4, 5, 21, 42, 51, 58, 58, 63, 66, 67]. Most systems are extensions to popular data analytics frameworks, which allow users to write UDFs (user-defined functions) for deploying an analytical job: SpatialHadoop [21], Hadoop-GIS [4], Parallel-Secondo [42], MD-HBase [51], and ST-Hadoop [5] extend the Hadoop framework; Simba [63], SpatialSpark [66], LocationSpark [58] and GeoSpark [67] extend the Spark framework. They extend Hadoop or Spark with the operations to support spatial queries, e.g., range 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 they consider a static data set with few or no updates. 2) STAR is optimized for ad hoc aggregate queries, while they mainly consider object-finding queries. 3) Apart from operations on spatial attributes, STAR supports operations on textual attributes, while they focus only on operations over spatial attributes. LocationSpark [58] 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 .

(2) Systems for streamed spatial data [1, 3, 10, 11, 13, 16, 33, 39, 44, 45, 57, 59, 60, 64, 68]. 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 [3, 57]. Another line of work considers answering spatio-keyword queries. A spatio-keyword query has a spatial and a textual argument. An object is in the result of the query if the object qualifies both arguments [10, 39, 68] or if the object's similarity is larger than a threshold [33]. Another body of work studies the top-k spatio-keyword query [11, 59, 60] that returns objects having the top-k highest similarities to the input query. Several distributed systems [1, 13, 16, 44, 45] 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 [7].

(3) Systems for both static and streamed spatial data [49, 50]. BBoxDB [49] is a distributed store system for multidimensional data. The authors develop BBoxDB Streams [50] on top of BBoxDB that supports both snapshot and continuous queries over multidimensional data. Our work has some key differences from BBoxDB Streams. First, BBoxDB Streams mainly considers object-finding queries and the join query, and it does not support aggregate queries that are essential to a DSWS and are the focus of this paper. Second, BBoxDB Streams does not support the collective processing of spatial, temporal, and textual data attributes, thus making it difficult to extend BBoxDB Streams to support aggregate queries with ad hoc constraints on these data attributes. In contrast, STAR is designed to consider spatial, temporal, and textual data attributes, collectively, and applies effective optimizations on ad hoc aggregate queries. Lastly, BBoxDB Streams does not provide indexing mechanisms for continuous queries. In contrast, STAR implements two indexes, termed the CH-tree and the MT-index to organize the continuous queries.

View Materialization. Labio *et al.* [36] and Ross *et al.* [54] 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., [28, 29, 31, 65], or randomized algorithms including genetic algorithms, e.g., [32] and simulated annealing algorithms, e.g., [19, 20]. Ghanem *et al.* [23] 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 analytical queries over these views.

Indexing Continuous Queries. Many indexes exist for continuous queries over relational data streams, e.g., k-index [62], BE-tree [55] and OpIndex [69]. These indexes do not consider data with spatial and textual attributes, making them not efficient for our problem. Another line of research investigates indexes for continuous queries over spatial data streams [10, 12, 15, 43, 60, 61, 68], which only work for continuous queries that have some spatial constraint. STAR differs in that it accepts continuous queries that do not have a spatial constraint.

3 SYSTEM OVERVIEW

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 [9].

Supported Queries: STAR is optimized to support aggregate queries with arbitrary 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 result. In contrast, holistic aggregate functions, e.g., *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 the 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 allows users to define 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 [23].

Example 1: Snapshot Aggregate Query. To find the popularity trend for iPhones in Region *R* grouped by date, we find the number of tweets mentioning 'iPhone' and the average length of these tweets.
SELECT *Count(id), Avg(length), 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 $Count(id)$, *topic*, *minute* **FROM** *stream*
WHERE *loc* **INSIDE** *R* **AND** *time* **AFTER** "10 mins ago"
GROUP BY *topic*, *minute* **ORDER BY** $Count(id)$ **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(text), topic* **FROM** *stream*
WHERE *loc* **INSIDE** *R* **GROUP BY** *topic* **SYNC** 1 minute.

Example 4: Continuous Aggregate Query. Track the number of customers on each street in a region every minute.
SELECT $Count(id)$, $streetName$, *minute* **FROM** *stream*
WHERE *loc* **INSIDE** *R* **AND** *time* **AFTER** "10 mins ago"
GROUP BY $streetName$, *minute* **SYNC** 1 minute.

3.2 System Architecture

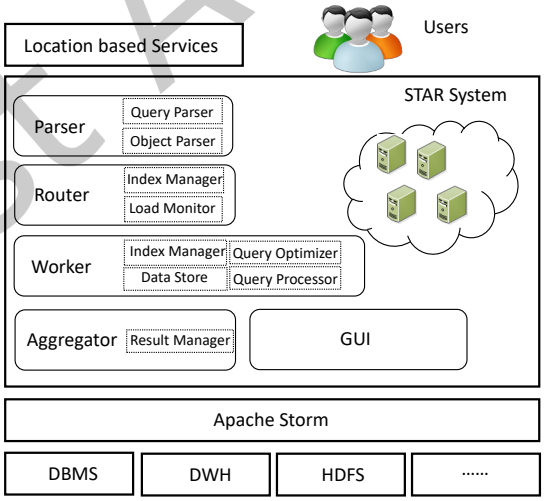


Fig. 1. System Architecture of STAR .

Figure 1 gives the system architecture of STAR .

STAR is based on Apache Storm[1], a distributed stream processing framework that provides great flexibility for extension. STAR can also be built on other stream processing frameworks, e.g., Flink[2]. STAR has four main component types: 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 an in-memory continuous-query index.

Aggregator. The aggregator collects the partial results from the 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 CACHE-BASED QUERY PROCESSING

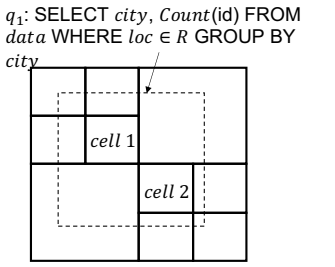
STAR offers a cache-based mechanism for answering snapshot aggregate queries. It combines query- and object-based caching.

4.1 Query-based Caching

Query-based caching facilitates processing snapshot queries by materializing a set of views based on historical queries. It maintains a selected set of views per worker. View selection is a classical problem in data warehousing, and it has been extensively studied [28, 29, 46]. 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. Materializing stream-based views may induce significant overhead, and it deserves more consideration. STAR materializes the following views into 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.

4.1.1 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*.

[1]http://storm.apache.org/
[2]https://flink.apache.org/



(a) Views for the range constraint.

q_2: SELECT $city, Count$(id) FROM $data$ WHERE "iphone"$\in text$ GROUP BY $city$

q_3: SELECT $city, Count$(id) FROM $dataset$ WHERE "NBA"$\in text$ GROUP BY $city$

"iphone":	**"NBA":**
"Seattle", 10	"Seattle", 12
"Los Angle", 25	"Los Angle", 30
"Las Vegas", 12	"Las Vegas", 8
"New York", 30	"New York", 22

(b) Views for the keyword constraint.

Fig. 2. Views for the queries.

The reason is that a view is essentially a set of key-value pairs, and the views for *Count* and other algebraic aggregate functions only differ in the way of computing *value* for each disjoint subset of objects.

Example. Figure 2 gives an example of views. In Figure 2(a), q_1 returns the number of objects for each city, and it 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 2(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.

Worker Overhead for Processing Snapshot Queries. We define the overhead of a worker for processing snapshot queries in Definition 4.1.

Definition 4.1. **Worker Overhead:** The overhead for processing snapshot queries for a worker, say w_i, during a certain time period can be estimated as follows:

$$L_i = c_1 \sum_{o \in O} n_1(o, V) + c_2 \sum_{q \in Q_s} (n_2(q, V) + n_3(q, O)), \tag{1}$$

where O is the set of spatial objects that has arrived to w_i in this time period, Q_s is the set of snapshot queries handled by w_i, V is the set of materialized views stored in w_i, $n_1(o, V)$ is the number of views updated for o, $n_2(q, V)$ is the sum of the sizes of the views checked for q, $n_3(q, O)$ is the number of indexed objects checked for q, c_1 is the average cost of updating a view, and c_2 is the average cost of processing a snapshot query. ■

Note that the load of a worker is composed of the processing overheads of both snapshot and continuous queries. Definition 4.1 gives the overhead of processing the snapshot queries. We will discuss the processing of the continuous queries in Section 5.

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 4.2. **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 [56], 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 the 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 worker overhead 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 per unit space, and thus it 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 it is a system-defined parameter. Thus, the time complexity of the algorithm is $O(Cn^2)$.

4.1.2 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 needs to compute over the complete dataset. Due to the fast arrival of the streamed objects and the 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 few key-value pairs. To save memory space, we do not employ techniques that have dynamic summary size, e.g., [57]. Instead, we maintain a SpaceSaving summary [47] 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 it 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 increase 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 4.3. *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 4.3, the SpaceSaving summary provides guarantees on the accuracy of the result 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.* [2] have proven that the guarantee in Theorem 4.3 still holds when merging multiple SpaceSaving summaries.

4.1.3 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 of 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.

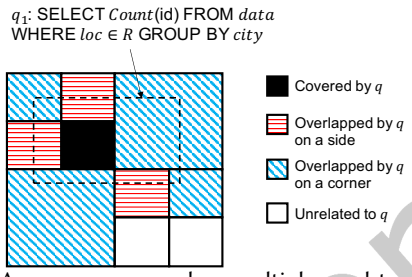


Fig. 3. A query range overlaps multiple quad-tree nodes.

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 3 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 time, we propose to cache objects in the next subsection.

4.2 Object-based Caching

We present the idea of caching the bordering objects that works seamlessly with the materialized views. The main idea is to cache the 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 a query. We propose an approximation algorithm to decide a set of cached objects to optimize query processing with theoretical guarantees.

Definition 4.4. **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 a border line of n_p. A corner caching region of n_p is a circular sector inside n_p whose centre is a corner point of n_p and has the angle being equal to 90 degree. ■

Figure 4 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 one 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 4.5. **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 4.6. *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 the caching regions. We need to decide caching regions for a node that overlaps queries. The optimal solution is based on the overlapped area with queries. 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 4 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 the caching regions that are covered by r.

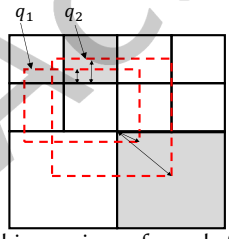


Fig. 4. Caching regions of a node for queries.

Theorem 4.7. *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.*

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 4.7 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 issue, STAR adopts an eviction policy that removes the Least Recently Used (LRU, for short) caching region. It keeps removing these caching regions in the LRU order until the memory constraint is satisfied.

5 INDEXING CONTINUOUS QUERIES

Unlike existing spatial data streaming systems [13, 16, 45] that are optimized for continuous range queries with keyword filtering, STAR deals with aggregate continuous queries with ad hoc constraints, whose results are a set of key-value pairs. In their setting, each query has a range and keyword constraints, and spatial indexes, e.g., a quad-tree, are adapted to index the continuous range queries [10, 13, 16, 45, 68]. However, these adapted indexing techniques are not applicable to STAR because the query predicates STAR supports are more complex. Another line of work [55, 62, 69] proposes indexes for continuous queries on relational data streams. They do not consider queries with range and keyword constraint as in STAR , making them not efficient for our problem.

The challenges are twofold: First, STAR needs to store and update the query results against the fast streamed objects. It imposes higher requirements on the efficiency of the index, as we not only check whether or not a new object satisfies a query, but also need to update the query results in real-time. Second, STAR needs to handle various types of predicates. Furthermore, for the queries with temporal constraints that focus on the most recent objects, e.g., a query that wants the aggregate information of the latest 10 minutes of data, STAR needs to exclude the effect of the outdated objects from the stored results. STAR implements new indexing mechanisms to address the above challenges. In Section 5.1, we introduce cost-based hybrid tree index (CH-tree) for STAR to handle the continuous queries without temporal constraints. In Section 5.2, we present the multi-layer time-window index (MT-index) for STAR to process the queries with temporal constraints that works together with the CH-tree.

5.1 Indexing Non-Temporal Queries

We introduce a new index, termed the cost-based hybrid tree index (CH-tree, for short) that stores continuous queries without a temporal constraint. The CH-tree is novel in that it applies to a variety of query predicates, e.g., spatial and keyword constraints, and it considers the selectivities of query predicates that enable it to achieve better efficiency over existing indexes proposed for continuous queries.

A query may have ad hoc constraints on several attributes. The constraints can be represented as a conjunction of Boolean predicates. Each predicate is a triple $P^{(attr,opt,val)}$, where $attr$ is an attribute, opt is a predicate operator, and val is a set of values. Given an object o, $P^{(attr,opt,val)}(o)$ returns true if all the predicates are satisfied, and returns false, otherwise. For ease of explanation, we use the following 4 types of predicates as representatives: (1) $P^{(x,=,v)}$, (2) $P^{(y,\in,[lb,ub])}$, (3) $P^{(loc,in,R)}$, and (4) $P^{(text,has,\{k_1,k_2,...\})}$. Note that the CH-Tree applies to other types of predicates.

The main idea of the CH-Tree is to categorize the queries based on their predicates, and use the common predicates to group them. However, it is difficult to find the optimal ordering of the common predicates that can minimize the number of predicates to be checked. Actually, the problem is NP-hard, as it can be reduced from the set cover problem. Therefore, we introduce a greedy method that uses a cost model to find a good ordering of the predicates.

Cost model. The cost model takes into consideration three factors: (1) the selectivity of the predicate, (2) the execution cost of the predicate, and (3) the number of queries that contain the predicate. Intuitively, for a new object, we want to filter the most irrelevant queries as soon as possible with small costs. We construct our cost model using sets of historical objects and queries. We estimate the cost due to a predicate $P^{(attr,opt,val)}$ by

$$Cost(P) = c_1^P \cdot |O| + \sum_{o \in O} (c_2^P \cdot |Q_o| + c_3^P \cdot |Q_v|), \tag{5}$$

where O is the set of objects, Q_o is the set of queries with which $P^{(attr,opt,val)}$ is satisfied without a need of verification for an object o, and Q_v is the set of queries that require verification; c_1^P is the average cost of checking Attribute $attr$ of objects for $P^{(attr,opt,val)}$, c_2^P is the average cost of processing the queries without verification, and c_3^P is the average cost of processing the queries that require verification. To explain the difference between Q_o and Q_v, consider predicates $P^{(x,=,v)}$ and $P^{(loc,in,R)}$. If we use a hash table to categorize $P^{(x,=,v)}$, e.g., v_0 is the key and the value is a list of predicates having $x = v_0$, Q_o is the set of queries that the predicates having $x = v_0$ corresponds to, and $Q_v = \Phi$. The reason is that the property of the hash table guarantees that the returned $P^{(x,=,v)}$ is true for the new object, so the corresponding queries require no verification. For $P^{(loc,in,R)}$, we use a spatial index, e.g., a quad-tree, to categorize the predicates. For an object o, let $Leaf$ be the leaf node that o falls into and P_{Leaf} be the set of predicates stored in $Leaf$, Q_v is the set of queries that P_{Leaf} corresponds to, and $Q_o = \Phi$. The reason is that P_{Leaf} may not be true and requires verification, i.e., checking whether o falls into the query range. It is worth noting that Q_o may be not empty if we store additional information for each leaf node of that the set of predicates whose R covers the leaf node range.

Categorizing the Predicates. Since the costs of predicates depend on the way that they are categorized, we first introduce the indexing mechanism for each type of predicate, respectively.

$P^{(x,=,v)}$: We use a hash table that uses each distinct value of v as the key and maps the key to the predicates that x equals to the key.
$P^{(y,\in,[lb,ub])}$: We use an interval tree that is built by inserting intervals specified by the predicates. Each tree node has a centre point, and the predicates whose intervals overlap it are stored in that node.
$P^{(loc,in,R)}$: We use a quad-tree that is built by inserting ranges specified by the predicates. The predicates are stored in the leaf nodes that overlap the query range.
$P^{(text,has,\{k_1,k_2,...\})}$: We use an inverted index that is built by selecting a representative keyword as the key for each predicate and inserting the predicates into the list of the corresponding key. For each predicate, we choose the most selective keyword (i.e., the keyword that has the least frequency in the objects) as the key.

Cost-based Hybrid Tree. We propose a cost-based hybrid tree index (CH-tree, for short) to store the continuous queries. We use the cost model to predefine the ordering of the predicates to be considered when building CH-tree, which is in ascending order of $Cost(P)$: the root node (level 1) consider the first predicate, the nodes at level 2

considers the second predicate, and so on. The queries are stored in the leaf nodes and for each non-leaf node N, we categorize the queries based on a selected $P^{(attr,opt,val)}$ in the following way:

$P^{(x,=,v)}$: If a query q has Predicate $P^{(x,=,v)}$, we pass q to the child node labelled by the value of v (if such node does not exist, we create one). Otherwise, we pass q to a special child node labelled "none".
$P^{(y,\in,[lb,ub])}$: If a query q has Predicate $P^{(y,\in,[lb,ub])}$, we insert the query interval and query id into an interval tree that is integrated in N. Each node of the interval tree has a pointer linking to a child node of N (if the child node does not exist, we create one). Then we pass q to that child node. If q does not have Predicate $P^{(y,\in,[lb,ub])}$, we pass q to a special child node labelled "none".
$P^{(loc,in,R)}$: If a query q has Predicate $P^{(loc,in,R)}$, we insert the query range and query id into a quad-tree that is integrated in N. Each leaf node of the quad-tree has a pointer linking to a child node of N (if the child node does not exist, we create one). Then we pass q to that child node. If q does not have predicate $P^{(loc,in,R)}$, we pass q to a special child node labelled "none".
$P^{(text,has,\{k_1,k_2,...\})}$: If a query q has Predicate $P^{(text,has,\{k_1,k_2,...\})}$, we find the most selectivity keyword of q and pass q to the child node labelled by that keyword (if such node does not exist, we create one). Otherwise, we pass q to a special child node labelled "none".

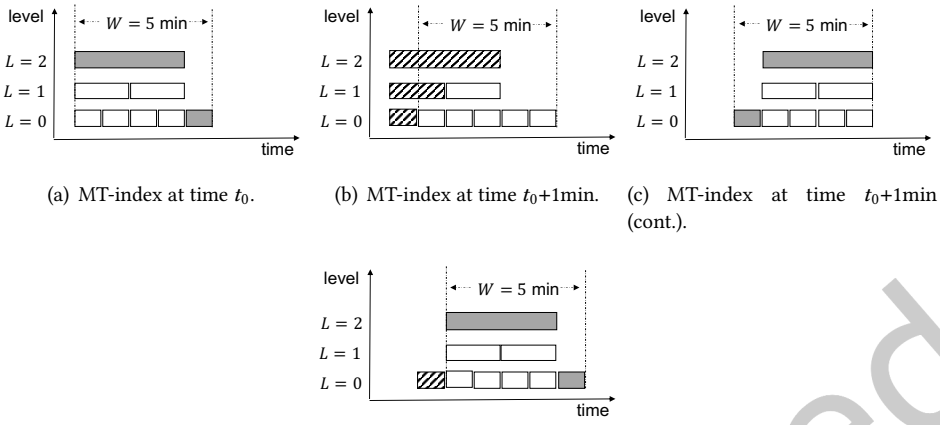
For predicates $P^{(y,\in,[lb,ub])}$ and $P^{(loc,in,R)}$, we employ an interval tree or quad-tree to categorize the predicates. The update overhead is large when the insertion of a new query incurs node splitting operations, which will trigger a sequence of update operations on the successor nodes. For the purpose of reducing the update costs, we decide in-prior the structure of the interval tree and quad-tree using a set of historical set of queries. After deploying them to the CH-tree, we do not update their structure when inserting new queries.

To avoid degrading the performance of a CH-tree greatly due to changes in the workload and data distribution, we use a threshold to limit the maximum number of continuous queries that a CH-tree can store. When a CH-tree is full, we build a new CH-tree to store the new queries, which is based on a *fresh* cost model derived from the most recent set of queries and objects. This strategy performs well when minor changes happen to the workload and data distribution before a CH-tree becomes full. However, the threshold cannot be too small, otherwise it will result in considerable index building cost. We set the threshold at an empirical value of 1,000. We leverage a *lazy* deletion strategy to delete queries from the CH-tree. When the user drops a query, we do not delete that query immediately, but set an inactive state to it. We set a threshold for the maximum number of inactive queries and delete all the inactive queries when the threshold is reached. A CH-tree is deleted if all of its queries are inactive.

Processing Objects. When receiving an object o, we traverse the CH-tree to find the queries whose result requires updating. At each non-leaf node, based on the selected predicate $P^{(attr,opt,val)}$, we check Attribute $attr$ of o to find the child node(s) that o should be passed to: (1) For $P^{(x,=,v)}$, we pass o to the child node that is labelled by the value of $Attr$. (2) For $P^{(y,\in,[lb,ub])}$ and $P^{(loc,in,R)}$, we check the interval tree or quad-tree to find the child node that o should be passed to. During the procedure, for the node that o falls in, we check o against its intervals or ranges, and mark the corresponding query id as "invalid" if o is not in an interval or range. (3) For $P^{(text,has,\{k_1,k_2,...\})}$, we scan $text$ of o and pass o to the child node(s) whose labels are contained in $text$. At each level, we also pass o to the special child node labelled "none". After reaching to a leaf node, we first filter the queries whose ids are marked as "invalid". For the remaining queries without predicate $P^{(text,has,\{k_1,k_2,...\})}$, we update their results directly. For the query with predicate $P^{(text,has,\{k_1,k_2,...\})}$, we update its result only if o contains all the keywords.

5.2 Indexing Temporal Queries

STAR supports aggregate queries with a time-window constraint, as illustrated in Section 3.1 (Example 4 with the temporal constraint "time AFTER 10 mins ago"). Processing this and similar queries is challenging as we not only need to maintain the query results but also need to exclude the effect of the outdated objects from the results.



(a) MT-index at time t_0.

(b) MT-index at time t_0+1min.

(c) MT-index at time t_0+1min (cont.).

(d) MT-index at time t_0+2min.

Fig. 5. The MT-index and a query with a 5-min time-window and 1-min refresh rate.

To address this, we present a multi-layer time-window index (MT-index) that works along with the CH-tree by integrating the MT-index into the leaf nodes of the CH-tree.

Each continuous query has a "sync" clause to define its result refresh rate, e.g., every 1 minute. In STAR , we assume that the length of the refresh interval is always smaller than the time-window size and the window size is a multiple of the refresh interval. The assumption is not necessarily true, as we can always split a refresh interval into smaller intervals that can be divided evenly by the window size and wait to report the result when the refresh interval is reached. For ease of explanation, we consider the simple case of one query and then generalize to multiple queries. The MT-index is composed of multi-layer time intervals. Each layer consists of adjacent equal-sized time intervals. At layer L_0, each interval is of size r, where r is the length of the refresh interval. At layer L_i, each interval is of size $2^i \times r$. The highest layer is L_m, where $m = \arg\max_i 2^i \leq \frac{W}{r}$ (W is the time-window size). The interval at layer L_{i+1} is built by combining two adjacent intervals at layer L_i. Each interval maintains a partial query result on the objects that arrives within it.

When a new object arrives, we update the partial result in the newest L_0 interval. When the timestamp reaches the end of the newest interval, it indicates that the result refresh interval has been reached. Before computing the result, we delete the intervals that are not covered by the time-window and consider whether to conduct merging operations: Staring from layer L_0, if the second-newest interval is not covered by any interval at layer L_1, we merge the two newest intervals (and the partial results in them) into a new interval at layer L_1. Then we move to the higher layer and conduct the same operation recursively. The iteration ends if the second-newest interval at the current layer has been covered or the highest layer is reached. After that, we compute the result by finding a minimum number of intervals that compose the query time-window and combining the partial results in them.

Figure 5 gives an example of MT-index and the procedure of processing a query with a time-window of 5 minutes and 1 minute refresh rate. In Figure 5(a), we use one L_0 interval and one L_2 interval, i.e., the grey boxes, to answer the query. In Figure 5(b)), after one minute, the refresh interval is reached, so we delete intervals that are not covered by the time-window, i.e., the slashed boxes. In Figure 5(c), we conduct merging operations and build two new intervals at L_1 and L_2, and compute the result using intervals at L_0 and L_2. In Figure 5(d), another refresh interval is reached. We delete the outdated interval, and as no merging is required, we compute the result directly.

Time Complexity. When the refresh interval is not reached, only the newest interval at layer L_0 requires updating, so the time complexity is $O(1)$. When the refresh interval is reached, for each layer, at most one deletion and merging are conducted. Therefore, the time complexity is $O(\lceil \log W/r \rceil)$.

Extending to General Cases. When there are more than one query, we set W as the largest time-window size, and set r as the largest common divisor of all the query refresh rates that can be divided evenly by W (in the most extreme case, r is set as 1-min if all the time-windows are in minutes). We can shift slightly the refresh period of a new query so that it can be put into an existing MT-index. The only difference is that we need to decide the set of queries whose results need updating when the timestamp reaches the end of the newest L_0 interval. To avoid a very large W and small r, we categorize the queries into groups based on the time-window size and maintain an MT-index for each group of queries.

6 WORKLOAD PARTITIONING

STAR partitions the streaming workload with two main 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.

The load of one worker comprises processing spatial objects, processing queries, and maintaining caches, i.e., materialized views and cached objects. The workload partitioning problem aims at minimizing the total amount of load. The first constraint is that the memory usage of the indexed data and the index structures does not exceed the memory capacity of the worker. The second constraint is that the workers should have balanced load. The Problem is NP-hard [29]. AQWA [6] is a disk-based system that is based on Hadoop, and hence is not well suited for streaming. However, AQWA's partitioning algorithm utilizes a kd-tree for spatial workload partitioning problem. AQWA's partitioning algorithm is not suitable for the problem at hand because AQWA only considers the query processing cost, while in our problem, the cache maintenance cost accounts for an important part of the worker load. Therefore, we propose a new workload partitioning algorithm.

Algorithm overview. We use a quad-tree to partition the workload. The main idea is to construct a quad-tree 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. We estimate the load of a node by $c_1|O| + c_2|O| \cdot |Q_c| + c_3|O| \cdot |Q_s|$, where O is the set of objects, Q_c is the set of continuous queries, Q_s is the set of snapshot queries, c_1 is the average cost of processing an object, c_2 is the average cost of processing continuous queries against the objects, c_3 is the average cost of processing snapshot queries against the objects.

In Phase 2, we assign leaf nodes to different partitions, and 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 neighbouring 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 workload 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.

Workload adjustment. We implement dynamic load adjustment mechanism in STAR to adapt to the changing workload. When the router detects that the load balance constraint is violated, it notifies the most loaded worker, say w_o, to transfer part of w_o's workload to other workers. We adjust the workload by migrating the cells of the global index to other workers. We expect that after adjustment, each worker still maintains an adjacent set of cells. The purpose is to reduce the total amount of load, as some queries and views may overlap multiple adjacent cells.

After receiving notification from the router, Worker w_o, having the maximum load, computes the amount of load that needs to be transferred. Let B be the cells bordering w_o. w_o sorts B in descending order of $load(g)/size(g)$, where $g \in B$. For each $g \in B$, w_o transfers g to the router, and the router sends g to another worker that contains cells being adjacent to g. If multiple candidates exist, the one having the minimum load is selected. This procedure repeats until the load balance constraint is satisfied, or until w_o has finished transferring B to other workers.

7 DATA ORGANIZATION

The objects are categorized over the timeline into a set of time slots, where different time slots have different granularities. The more recent data has a finer granularity, while the older data has coarser granularity. The granularity follows an exponential function $f(x) = 2^x$, where x represents the lifetime of the data in the system (e.g., #hours). The system periodically checks whether adjacent time slots can be merged. Two adjacent time slots can be merged when their granularities are the same. Since users usually focus more on the more recent data, they can tolerate minor accuracy loss in the old data. The system periodically checks the data size and deletes the oldest data when the data size exceeds a predefined threshold, which can be efficiently achieved by deleting the oldest time slot(s). This design is good for memory efficiency of STAR and allows efficient deletion of old data. In each time slot, a quad-tree is employed to index the objects. Objects having the $text$ attribute are further categorized using an inverted index.

8 EXPERIMENTAL EVALUATION

8.1 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 [34] is a popular framework for building real-time data pipelines and streaming 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 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

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.

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.

8.2 Evaluation on Snapshot Queries

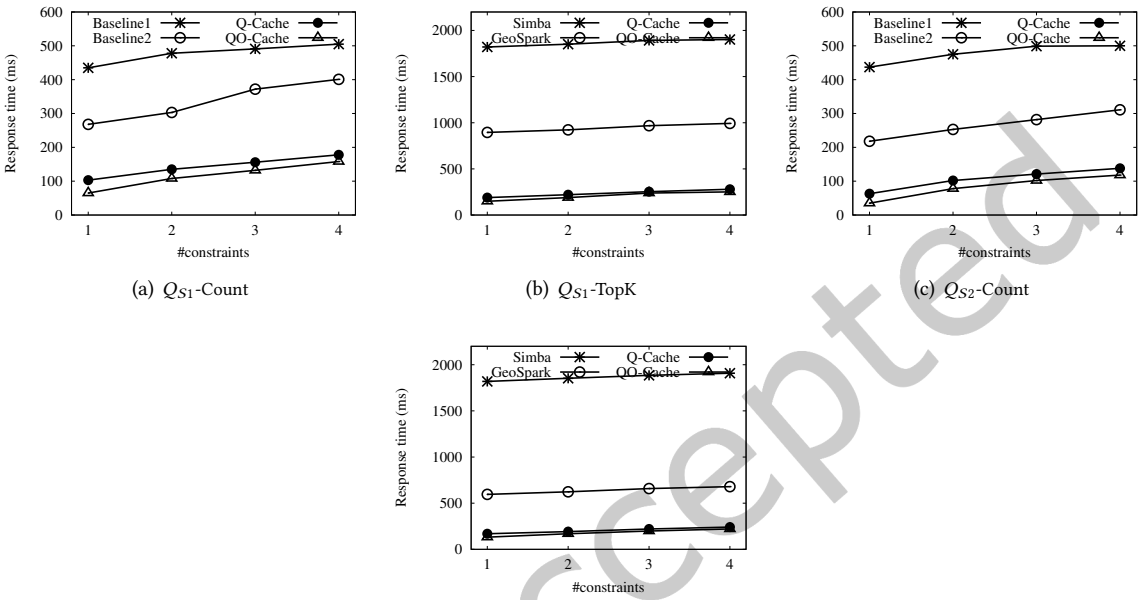
To evaluate the performance of STAR on processing snapshot queries, we compare STAR with the following two baselines that are variants of STAR :

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 spatial data stream system is designed to provide native support for ad hoc snapshot aggregate queries. We will extend representative existing systems for comparison in Section 8.4. We evaluate the performance by measuring 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 latency by using a moderate input speed of the data stream. We evaluate the performance of our cache-based algorithms: Q-cache represents using query-based caching and QO-cache represents using both query- and object-based caching.



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

(d) Q_{S2}-TopK

Fig. 6. Query response time comparison for snapshot queries.

Figure 6 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 from 10% to 40%. This is because that QO-cache maintains materialized views and uses cached objects to help process queries, which avoids checking a large amount of objects. Baseline-1 performs the worst as it always needs 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 to reduce the query response time.

8.3 Evaluation on Continuous Queries

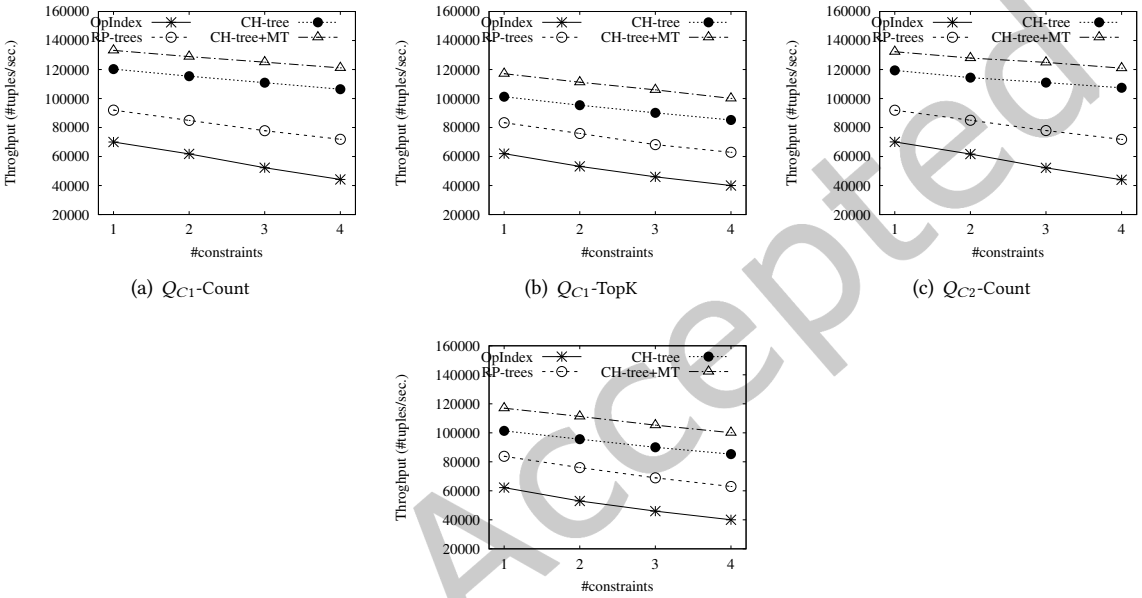
We evaluate the performance of our CH-tree and MT-index on processing continuous queries. We compare them with two state-of-the-art indexes that were proposed for publish/subscribe systems.

OpIndex [69]. OpIndex adopts a two-level partitioning scheme. In the first level, continuous queries are partitioned into query lists based on a selected pivot attribute. In the second level, the predicates in each query list

are further partitioned into predicate lists based on the predicate operator. OpIndex does not consider range or keyword constraint, and we transform each range constraint into two interval constraints and leave the keyword constraint to be checked at the last step.

RP-trees [70]. RP-trees index partitions the continuous queries into query lists based on a selected pivot attribute. For each query list, it maintains an R-tree, which stores the range constraints that are specified by the queries. As it does not consider keyword constraint, we check the keyword constraint at the last step.

We evaluate the performance by measuring throughput, as it is done in previous work on continuous queries.



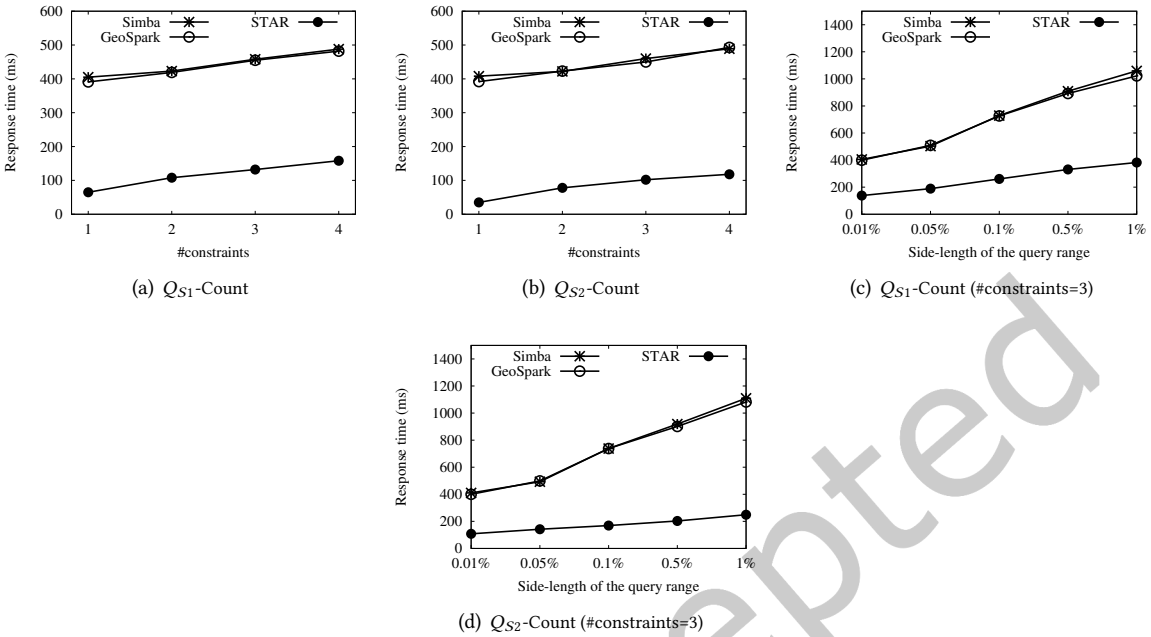
(a) Q_{C1}-Count (b) Q_{C1}-TopK (c) Q_{C2}-Count

(d) Q_{C2}-TopK

Fig. 7. Throughput comparison for continuous queries.

Throughput. The throughput is the number of tuples (including objects and queries) a system can handle when the processing capacity of the system is reached. We measure the throughput of each method by tuning the input speed of the data stream to be close to its processing capacity.

Figure 7 gives the throughputs of CH-tree, CH-tree+MT (using both CH-tree and MT-index), OpIndex and RP-trees for the four groups of queries. CH-tree+MT has the best performance, followed by CH-tree, which improves CH-tree by 10% – 20%. CH-tree+MT's throughput is 2–3 times of that of OpIndex and about 1.5 times of that of PR-trees. OpIndex has the worst performance, probably due to its poor performance in processing queries with a range constraint. RP-trees index performs better than OpIndex as it supports range constraints natively. We also observe that the throughputs of CH-tree and CH-tree+MT decrease slower than OpIndex and RP-trees' with respect to the number of constraints. This is because that the cost model in CH-tree can find a good ordering of predicates so that more queries can be filtered in an early stage, thus reducing the processing cost. The results demonstrate the superior performance of our indexes over state-of-the-art index structures.



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

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

Fig. 8. Comparison with existing systems (snapshot queries).

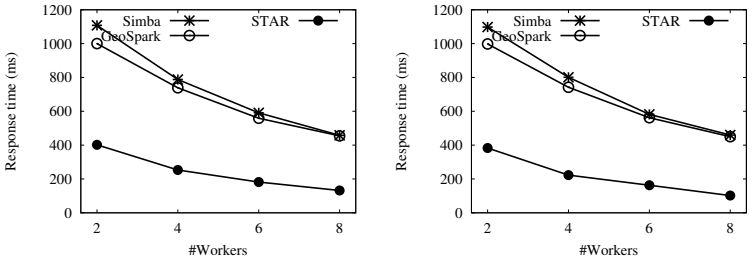
8.4 Comparison with Existing Systems

Since existing systems are not optimized to support both snapshot and continuous aggregate queries over spatial data streams, we compare STAR with them on processing snapshot and continuous queries, separately. We evaluate the performance of each system by running it exclusively on the cluster.

Comparison with systems over static data. We extend Simba [63] and GeoSpark [67], two representative distributed spatial data analytics systems, for comparison. Because 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 preloading STAR , Simba and GeoSpark with a static set of tweets. (2) We input the queries with at least one range constraint.

Figures 8(a) and 8(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 8(c) and 8(d) show that STAR has a much smaller query response time, e.g., in Figure 8(d), the query response time of STAR is smaller than 40% of the 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 9 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.



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

Fig. 9. Scalability (snapshot queries).

The experimental results demonstrate 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.

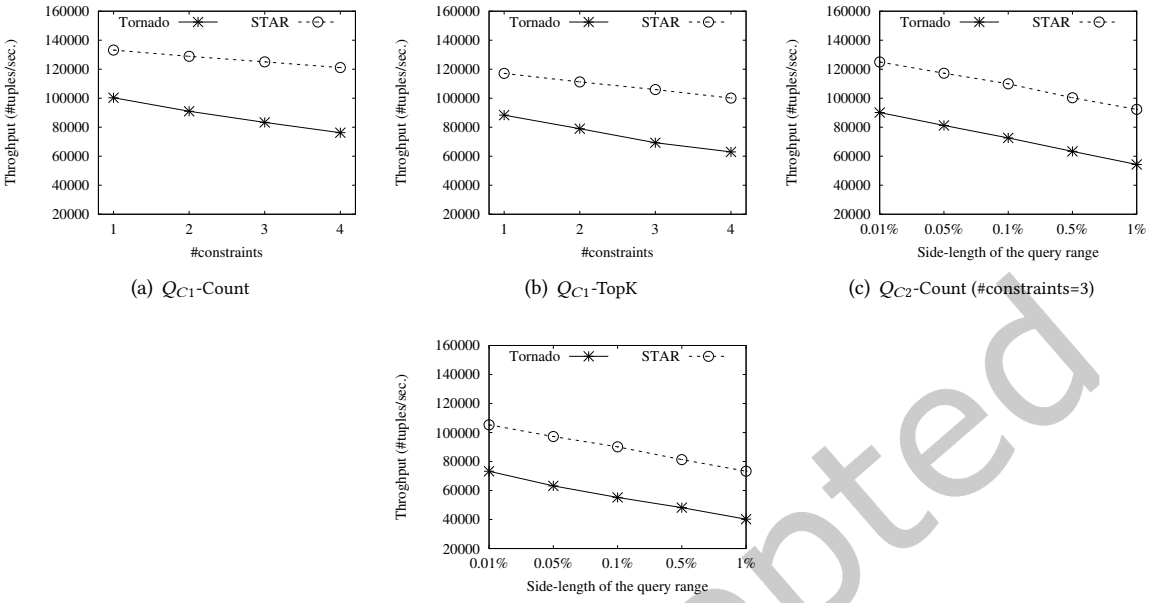
Comparison with Tornado. For continuous aggregate queries, we compare STAR with Tornado [45], a state-of-the-art system that supports continuous queries with ad hoc spatial and textual constraints over spatial data streams. Tornado utilizes FAST [43], an indexing framework to organize continuous queries. FAST has a spatial pyramid structure similar to a quad-tree, and is equipped with a frequency-aware indexing mechanism for continuous spatio-textual queries. 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 10 gives the experimental results. STAR has larger throughputs than Tornado for all types of queries. For example, in Figures 10(a) and 10(b), the throughputs of STAR are larger than Tornado by 35%–55%. The difference becomes more significant when increasing the number of query constraints, e.g., in Figure 10(a), STAR outperforms Tornado by about 55%. This is because that the CH-tree in STAR uses a cost model to determine the ordering of predicates, which can effectively reduce the cost of checking objects in the presence of multiple query constraints. Figures 10(c) and 10(d) show the throughputs of STAR and Tornado with respect to the size of the query range. STAR has a more stable performance than Tornado: the throughput of STAR decreases slower than Tornado when increasing the size of the query range. This is ascribed to the better filtering effect of the indexes used by STAR .

Figure 11 gives the result on scalability with the number of workers. STAR achieves larger throughputs than Tornado, e.g., in Figure 11(a), the throughput of STAR is consistently about 1.5 times of that of Tornado. The results demonstrate that STAR outperforms Tornado in processing aggregate continuous queries.

8.5 Workload Partitioning

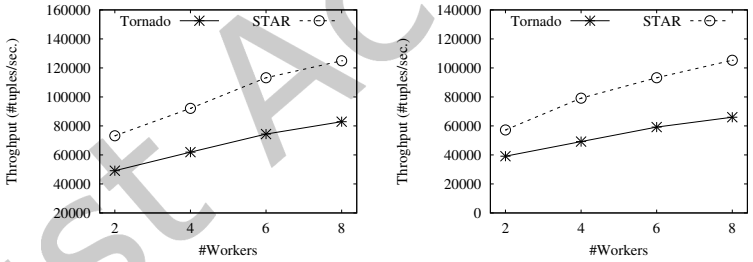
We evaluate our workload partitioning scheme by comparing it with two partitioning schemes: STR [38] and AQWA [6]. Figure 12 gives the experimental results. We observe that our partitioning scheme has the best performance: In Figure 12(a), STAR has about 20% smaller query response time than the others; In Figure 12(b), STAR has about 18% larger throughput than the others. The results demonstrate the effectiveness of our partitioning scheme.



(a) Q_{C1}-Count (b) Q_{C1}-TopK (c) Q_{C2}-Count (#constraints=3)

(d) Q_{C2}-TopK (#constraints=3)

Fig. 10. Comparison with Tornado (continuous queries).



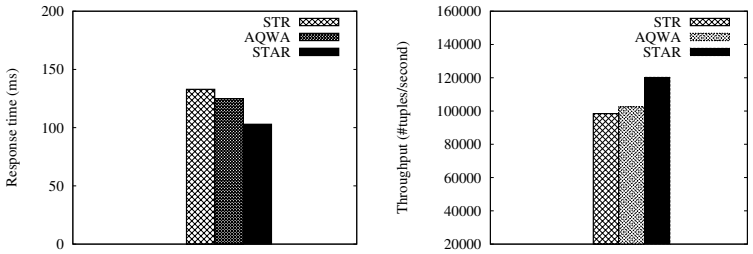
(a) Q_{S2}-Count (#constraints=3) (b) Q_{S2}-TopK (#constraints=3)

Fig. 11. Scalability (continuous queries).

8.6 Memory

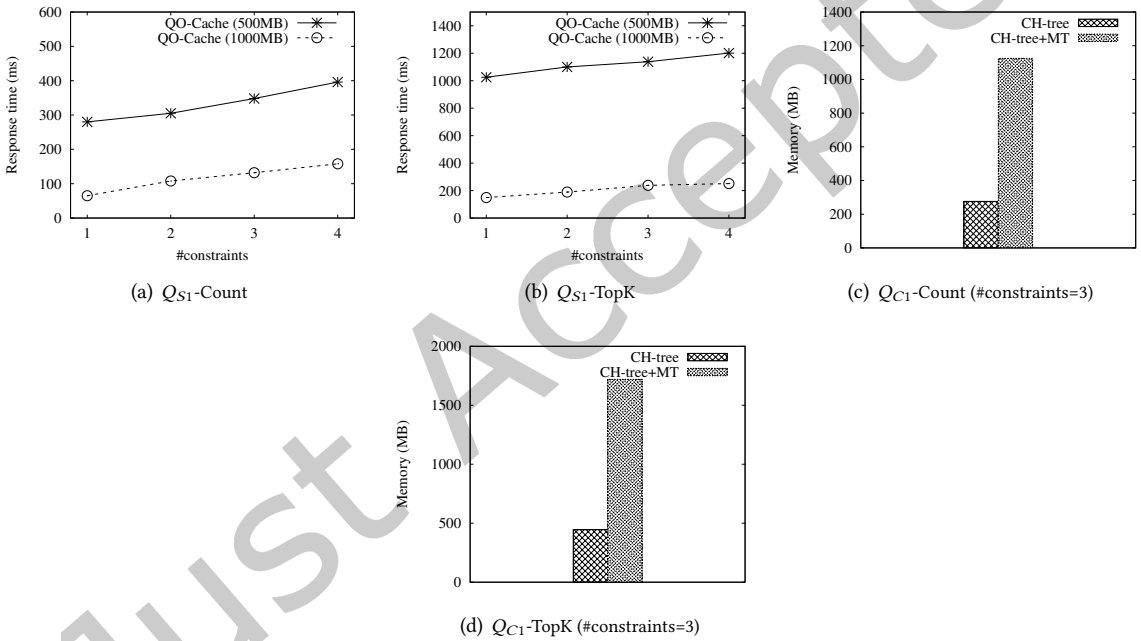
In this set of experiments, we evaluate the impact of the memory constraint on snapshot query processing and the memory usage of the indexes for continuous query processing.

Figures 13(a) and 13(b) give the results of snapshot query processing. When the memory constraint is 1,000MB, the query response time of QO-Cache is significantly smaller than that of QO-Cache when the memory constraint is 500MB. For Q_{S1}-Count, QO-Cache(1,000MB) takes less than 40% time of QO-Cache(500MB). For Q_{S1}-TopK, QO-Cache(1,000MB) takes less than 20% time than that of QO-Cache (500MB). This is expected because when there is more available memory, more queries can be answered using query- and object-based caching, which



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

Fig. 12. Comparing different partitioning schemes.



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

(d) Q_{C1}-TopK (#constraints=3)

Fig. 13. Evaluating the impact of memories on the snapshot query processing and the memories of continuous query indexes.

saves considerable time. The improvement is more prominent for Q_{S1}-TopK because computing top-k results from scratch is costly.

Figures 13(c) and 13(d) give the memory usage of both the CH-tree and CH-tree+MT for continuous query processing. The CH-tree+MT occupies about 4 times the memory for the CH-tree. This is due to the structure of CH-tree+MT that maintains multiple CH-tree instances.

9 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 both snapshot and continuous aggregate queries that have ad hoc constraints over spatial, textual and temporal data attributes. STAR adopts a cache-based mechanism to facilitate the processing of snapshot queries. STAR implements a novel index to categorize the continuous queries, which outperforms existing indexes in the procedures of checking objects against the indexed queries and maintaining the query results. Extensive experiments over real data sets demonstrate the superior performance of STAR over existing systems.

10 ACKNOWLEDGEMENTS

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] A. S. Abdelhamid, M. Tang, A. M. Aly, A. R. Mahmood, T. Qadah, W. G. Aref, and S. Basalamah. 2016. Cruncher: Distributed in-memory processing for location-based services. In *ICDE*. 1406–1409.
[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *TODS* (2013).
[3] Pritom Ahmed, Mahbub Hasan, Abhijith Kashyap, Vagelis Hristidis, and Vassilis J Tsotras. 2017. Efficient Computation of Top-k Frequent Terms over Spatio-temporal Ranges. In *SIGMOD*. ACM, 1227–1241.
[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] Louai Alarabi and Mohamed F. Mokbel. 2017. A Demonstration of ST-hadoop: A MapReduce Framework for Big Spatio-temporal Data. *PVLDB* (2017), 1961–1964.
[6] 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. *VLDB* 8, 13 (2015), 2062–2073.
[7] Anonymous. Anonymous. Reference omitted for anonymity purposes.
[8] Magdalena Balazinska, YongChul Kwon, Nathan Kuchta, and Dennis Lee. 2007. Moirae: History-Enhanced Monitoring.. In *CIDR*. 375–386.
[9] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.
[10] Lisi Chen, Gao Cong, and Xin Cao. 2013. An Efficient Query Indexing Mechanism for Filtering Geo-textual Data. In *SIGMOD*. ACM, 749–760.
[11] L. Chen, G. Cong, X. Cao, and K. L. Tan. 2015. Temporal Spatial-Keyword Top-k publish/subscribe. In *ICDE*. 255–266.
[12] Lisi Chen, Shuo Shang, Kai Zheng, and Panos Kalnis. 2019. Cluster-based subscription matching for geo-textual data streams. In *ICDE*. IEEE, 890–901.
[13] Yue Chen, Zhida Chen, Gao Cong, Ahmed R Mahmood, and Walid G Aref. 2020. SSTD: A Distributed System on Streaming Spatio-Textual Data. *VLDB* 13, 11 (2020).
[14] Zhida Chen, Gao Cong, and Walid G. Aref. 2021. STAR: A Cache-Based Distributed Warehouse System for Spatial Data Streams. In *SIGSPATIAL*. 606–615.
[15] Zhida Chen, Gao Cong, Zhenjie Zhang, Tom ZJ Fuz, and Lisi Chen. 2017. Distributed publish/subscribe query processing on the spatio-textual data stream. In *ICDE*. IEEE, 1095–1106.
[16] Z. Chen, G. Cong, Z. Zhang, T. Z. J. Fuz, and L. Chen. 2017. Distributed Publish/Subscribe Query Processing on the Spatio-Textual Data Stream. In *ICDE*. 1095–1106.
[17] 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.
[18] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *NSDI*, Vol. 10. 20.

[19] Roozbeh Derakhshan, Frank KHA Dehne, Othmar Korn, and Bela Stantic. 2006. Simulated Annealing for Materialized View Selection in Data Warehousing Environment.. In *Databases and Applications*. 89–94.
[20] 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.
[21] A. Eldawy and M. F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*. 1352–1363.
[22] W. Feng, C. Zhang, W. Zhang, J. Han, J. Wang, C. Aggarwal, and J. Huang. 2015. STREAMCUBE: Hierarchical spatio-temporal hashtag clustering for event exploration over the Twitter stream. In *ICDE*. 1561–1572.
[23] Thanaa M Ghanem, Ahmed K Elmagarmid, Per-Åke Larson, and Walid G Aref. 2010. Supporting views in data stream management systems. *TODS* 35, 1 (2010), 1.
[24] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads. *PVLDB* 8, 12 (2015), 1716–1727.
[25] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk. 2009. Stream Warehousing with DataDepot. In *SIGMOD*. ACM, 847–854.
[26] 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.
[27] Marcin Gorawski and Rafal Malczok. 2010. Indexing Spatial Objects in Stream Data Warehouse. *Advances in Intelligent Information and Database Systems* 283 (2010), 53–65.
[28] Himanshu Gupta. 1997. *Selection of views to materialize in a data warehouse*. Springer Berlin Heidelberg, Berlin, Heidelberg, 98–112.
[29] Himanshu Gupta and Inderpal Singh Mumick. 1999. *Selection of Views to Materialize Under a Maintenance Cost Constraint*. Springer Berlin Heidelberg, 453–470.
[30] 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.
[31] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *SIGMOD*. ACM, 205–216.
[32] 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, 8 (2003), 574–581.
[33] H. Hu, Y. Liu, G. Li, J. Feng, and K. L. Tan. 2015. A location-aware publish/subscribe framework for parameterized spatio-textual subscriptions. In *ICDE*. 711–722.
[34] Apache Kafka. 2020. https://kafka.apache.org/.
[35] Ralph Kimball and Margy Ross. 2013. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons.
[36] W. J. Labio, D. Quass, and B. Adelberg. 1997. Physical database design for data warehouses. In *ICDE*. 277–288.
[37] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. 2012. Muppet: MapReduce-style Processing of Fast Data. *PVLDB* 5, 12 (2012), 1814–1825.
[38] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*. IEEE, 497–506.
[39] Guoliang Li, Yang Wang, Ting Wang, and Jianhua Feng. 2013. Location-aware Publish/Subscribe. In *SIGKDD*. ACM, 802–810.
[40] G. Liang, L. Runheng, J. Yan, and J. Xin. 2010. Compressed StreamCube: Implementation of Compressed Data Cube in DSMS. In *CCIE*, Vol. 1. 345–349.
[41] L. Lins, J. T. Klosowski, and C. Scheidegger. 2013. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *TVCG* 19, 12 (Dec 2013), 2456–2465.
[42] J. Lu and R. H. Güting. 2014. Parallel SECONDO: A practical system for large-scale processing of moving objects. In *2014 IEEE 30th International Conference on Data Engineering*. 1190–1193. https://doi.org/10.1109/ICDE.2014.6816738
[43] Ahmed R Mahmood, Ahmed M Aly, and Walid G Aref. 2018. FAST: frequency-aware indexing for spatio-textual data streams. In *ICDE*. IEEE, 305–316.
[44] 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, 12 (2015), 2020–2023.
[45] 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*. ACM, 219–228.
[46] Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD* 41, 1 (2012), 20–29.
[47] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *ICDT*. Springer, 398–412.
[48] K. Nakabasami, T. Amagasa, S. A. Shaikh, F. Gass, and H. Kitagawa. 2015. An architecture for stream OLAP exploiting SPE and OLAP engine. In *IEEE Big Data*. 319–326.

[49] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2018. BBoxDB - A Scalable Data Store for Multi-Dimensional Big Data. In *CIKM* (Torino, Italy). Association for Computing Machinery, 1867–1870.
[50] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2022. BBoxDB streams: scalable processing of multi-dimensional data streams. *Distributed and Parallel Databases* (2022), 1573–7578.
[51] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1. 7–16. https://doi.org/10.1109/MDM.2011.41
[52] 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*. ACM, 1081–1090.
[53] Pedro Pedreira, Chris Croswhite, and Luis Bona. 2016. Cubrick: Indexing Millions of Records Per Second for Interactive Analytics. *PVLDB* 9, 13 (2016), 1305–1316.
[54] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD*. ACM, 447–458.
[55] Mohammad Sadoghi and Hans-Arno Jacobsen. 2011. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *SIGMOD*. 637–648.
[56] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Academic Press.
[57] Anders Skovsgaard, Darius Sidlauskas, and Christian S Jensen. 2014. Scalable top-k spatio-temporal term querying. In *ICDE*. IEEE, 148–159.
[58] 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. *VLDB* 9, 13 (2016), 1565–1568.
[59] Bin Wang, Rui Zhu, Xiaochun Yang, and Guoren Wang. 2017. Top-k representative documents query over geo-textual data stream. *WWW* (2017), 1–19.
[60] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Zengfeng Huang. 2016. Skype: Top-k Spatial-keyword Publish/Subscribe over Sliding Window. *PVLDB* 9, 7 (2016), 588–599.
[61] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. 2015. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In *ICDE*. IEEE, 1107–1118.
[62] Steven Euijong Whang, Hector Garcia-Molina, Chad Brower, Jayavel Shanmugasundaram, Sergei Vassilvitskii, Erik Vee, and Ramana Yerneni. 2009. Indexing Boolean Expressions. 2, 1 (Aug. 2009), 37–48.
[63] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*. 1071–1085.
[64] 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*. IEEE, 44–51.
[65] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. Algorithms for materialized view design in data warehousing environment. In *VLDB*, Vol. 97. 136–145.
[66] S. You, J. Zhang, and L. Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *ICDEW*. 34–41.
[67] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. GeoSpark: A Cluster Computing Framework for Processing Large-scale Spatial Data. In *SIGSPATIAL*. ACM.
[68] M. Yu, G. Li, T. Wang, J. Feng, and Z. Gong. 2015. Efficient Filtering Algorithms for Location-Aware Publish/Subscribe. *TKDE* 27, 4 (April 2015), 950–963.
[69] Dongxiang Zhang, Chee-Yong Chan, and Kian-Lee Tan. 2014. An Efficient Publish/Subscribe Index for e-Commerce Databases. 7, 8 (April 2014), 613–624.
[70] Pengpeng Zhao, Hanhan Jiang, Jiajie Xu, Victor S Sheng, Guanfeng Liu, An Liu, Jian Wu, and Zhiming Cui. 2017. Location-aware publish/subscribe index with complex boolean expressions. *WWW* 20, 6 (2017), 1363–1384.