STAR: A Distributed Stream Warehouse System for Spatial Data

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

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

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

ABSTRACT

The proliferation of mobile phones and location-based services gives rise to an explosive growth of spatial data. This spatial data contains valuable information, and calls for data stream warehouse systems that can provide real-time analytical results with the latest integrated spatial data. In this demonstration, we present the STAR (Spatial Data Stream Warehouse) system. STAR is a distributed in-memory spatial data stream warehouse system that provides low-latency and up-to-date analytical results over a fast spatial data stream. STAR supports a rich set of aggregate queries for spatial data analytics, e.g., contrasting the frequencies of spatial objects that appear in different spatial regions, or showing the most frequently mentioned topics being tweeted in different cities. STAR processes aggregate queries by maintaining distributed materialized views. Additionally, STAR supports dynamic load adjustment that makes STAR scalable and adaptive. We demonstrate STAR on top of Amazon EC2 clusters using real data sets.

CCS CONCEPTS

• **Information systems → Query optimization**; **Stream management**; *Online analytical processing engines.*

KEYWORDS

Spatial Data; Distributed System; Stream; Data Warehouse

ACM Reference Format:
Zhida Chen, Gao Cong, and Walid G. Aref. 2020. STAR: A Distributed Stream Warehouse System for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384699

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384699

1 INTRODUCTION

With the proliferation of GPS-equipped mobile devices and location based services, there has been an explosive growth in spatial data. Numerous social-media users upload posts on Twitter or Facebook using their smart phones, giving rise to a fast-arriving spatial-data stream. This spatially annotated data contains valuable information, and it is beneficial for performing spatial data analytics. Ad-hoc aggregate queries provide very valuable insights on the data, and facilitate the decision making process. 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 or not. She can issue an ad-hoc aggregate query that returns the frequencies grouped by region of the newly uploaded posts on social networks that talk about the product.

To minimize the gap between data production and data analysis, a data stream warehouse system (DSWS, for short) [1, 3, 4] provides real-time analytics over data streams. DSWSs are designed to efficiently ingest data and enable online analytical processing (OLAP) over the streamed data. A full-fledged DSWS allows users to issue queries that monitor changes in characteristics of streamed data (i.e., continuous queries) as well as queries that ask for statistics of already arrived data (i.e., snapshot queries).

Although spatial data is explosive in size, research on distributed DSWSs that provide native spatial data stream analytics is still lacking. Most existing distributed systems [6] focus on developing spatial data management systems over static data sets, and these systems 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., [2, 5], aim to support search queries, e.g., continuous spatial-keyword range queries, but are not optimized for or do not support ad-hoc aggregate queries. Furthermore, they only support continuous queries but not snapshot queries.

This demonstration presents STAR, a Spatial Data Stream Warehouse) system for spatial data analytics over spatial data streams. STAR supports both snapshot and continuous types of aggregate queries that can have arbitrary constraints over spatial, textual, and temporal data attributes. STAR supports algebraic aggregate functions, e.g., *Count*, *Avg*, and

Sum. Also, STAR supports holistic aggregate functions, e.g., *TopK*. STAR has a novel workload partitioning strategy that materializes distributed views in-memory to efficiently partition the workload. In particular, STAR has following features:

- STAR is the first system that supports a rich set of aggregate queries over spatial data streams. STAR supports both snapshot and continuous queries that are composed of algebraic or holistic aggregate functions and ad-hoc query constraints over spatial, textual and temporal data attributes.
- STAR supports SQL-like queries. It allows users to interact with the system in real-time. STAR achieves high throughput and low-latency by adopting a new workload partitioning strategy and maintaining distributed materialized views when processing and answering aggregate queries over spatial data streams.
- STAR is scalable and adaptive. It repartitions the workload dynamically to adapt to changes in the workload.

2 OVERVIEW OF STAR

Data Types: Each object in STAR 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 data in the primitive attributes. We assume that the raw data has at least 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 produce extracted data from primitive ones. For example, data in Attribute *topic* can be extracted from *text* by employing a pre-trained Latent Dirichlet Allocation (LDA) model. Figure 1 gives an example of primitive and extracted attributes.

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

STAR supports range and keyword constraints over Attributes *loc* and *text*, respectively. For Attribute *time*, STAR focuses on the time-window constraint that focuses only on the more 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() represents an aggregate function, *condition(s)* represents the constraints, and *attribute(s)* represent the ones used for grouping. STAR allows users to define a continuous query by providing **SYNC** operator: **SYNC** *freq* indicates that the query result is refreshed every $freq$ time.

For example, the following shows a snapshot query that returns the popularity trend of iPhone in a certain region R (The result is grouped by date):
SELECT *Count()*, *date* **FROM** *stream*
WHERE *loc* **INSIDE** R **AND** *text* **CONTAINS** "iphone"
GROUP BY *date*.

As another example, the following shows a continuous query that finds the most frequent terms of each topic on the objects that are within a region R every 1 minute:
SELECT *TopK()*, *topic* **FROM** *stream*
WHERE *loc* **INSIDE** R
GROUP BY *topic* **SYNC** 1 minute.

3 SYSTEM ARCHITECTURE

Figure 2 gives the system architecture for STAR. We implement STAR on top of Apache Storm[1], a popular distributed stream processing system. We select Storm as it provides great flexibility for extensibility. However, our work is not limited to Strom and can be applied to other streaming processing frameworks like Flink[2].

3.1 The Parser

The parser component takes as input streamed spatial objects and query requests from users. It parses primitive attributes of each object, and generates extracted attributes accordingly. A set of tools are embedded to support the procedure. The query parser module is responsible for parsing user queries, and transforms the SQL statements into a pre-defined data structure in STAR. The parsed queries are forwarded to the router component for further evaluation.

3.2 The Router

The router component is responsible for workload distribution and dynamic load adjustment.

The Global Index Manager. The index manager module maintains a global index structure to facilitate workload distribution that is built based on the workload partitioning strategy. The workload of STAR is composed of object processing, query processing, and view maintenance.

STAR uses a quadtree to partition the workload for its efficiency in both querying and updating. The workload partitioning algorithm performs two phases. In the first phase, it initializes a quadtree as one root node, and recursively partitions the nodes until the number of nodes exceeds the

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

Primitive Attributes			Extracted Attributes		
loc	*text*	*time*	*city*	*topic*	*date*
40.7, 74.0	Can't wait to see the NBA final.	2020-01-02 13:12:02	New York	sport	2020-01-02
42.3, 71.0	The iPhone X looks amazing.	2020-01-02 09:03:32	Boston	IT	2020-01-02
34.0, 118.2	I am watching the latest Star War movie.	2020-01-03 18:02:49	Los Angeles	movie	2020-01-03

Figure 1: Example of primitive and extracted attributes.

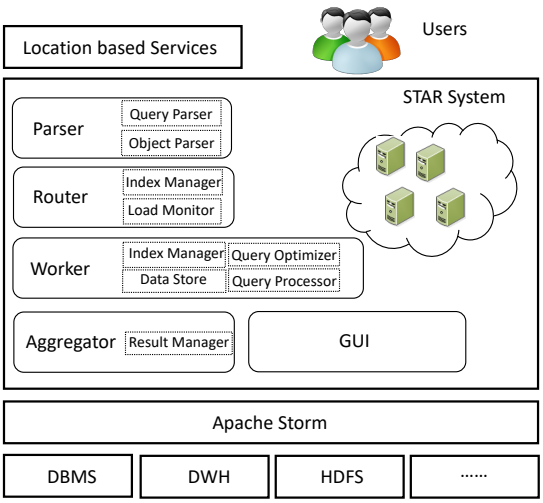

Figure 2: System Architecture of STAR.

number of workers. It decides which node to partition based on the load information. It estimates the load of a node as $|O| \times |Q|$, where O and Q are the objects and queries, respectively, that are inside or overlap the node. Let δ be a load imbalance threshold, and L_{max} (resp. L_{min}) be the maximum (resp. minimum) load of the candidate nodes. If $L_{max}/L_{min} > \delta$, then the workload partitioning algorithm chooses the node with maximum load to partition as, otherwise it would result in load imbalance. If $L_{max}/L_{min} \leq \delta$, then the workload partitioning algorithm chooses the node that invokes the minimum sum of loads after being partitioned. In the second phase, it assigns the leaf nodes of the quadtree to different partitions. It uses depth-first search to allocate adjacent cells into the same partition. The goal is to reduce the total load as some queries may overlap multiple adjacent cells, and assigning them to different partitions may increase the computation overhead. For each partition, we select a set of views to be materialized, and compute the load of each partition. We check if the load balance constraint can be satisfied. In this case, we produce as output, the quadtree index and the corresponding partitions. Else, we further partition the leaf nodes, and repeat the above procedure.

The quadtree acts as a global index for workload distribution. To determine the destination worker(s) where an object or a query should be sent to, the router needs to traverse from the root to the leaf node(s). This procedure takes $O(h)$ time, where h is the height of the quadtree. This imposes heavy burden on the router when the arrival speeds of objects and queries are fast.

To reduce the load on the router, we transform the quadtree into a grid index. The granularity of the grid index is decided by the leaf nodes of the quadtree. Figure 3 gives a quadtree and its corresponding grid index. The grid index partitions the global space into 4×4 cells as the height of the quadtree is 2. By using a grid index, the router takes $O(1)$ time to locate the local worker for an object. The time complexity of deciding the local worker(s) for a query is $O(R/c)$, where R is the area of the query range, and c is the cell size.

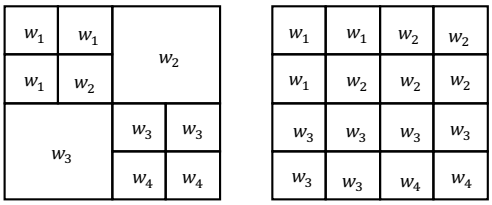

Figure 3: The quad-tree and corresponding grid index.

Load Monitor. The load monitor is responsible for dynamic load adjustment. It monitors the load of each worker. When it detects load imbalance among workers, 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 leaf nodes of the global index into other workers.

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 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.

3.3 The Worker

The worker component is responsible for processing objects and queries, and maintaining materialized views.

The Local Index Manager. This module maintains an in-memory local index structure to process objects and queries.

It also maintains a set of materialized views in-memory to accelerate answering the queries.

First, 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 (#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. Each node of the quad-tree may maintain a set of materialized views, which can be used for answering queries whose range constraints fully cover the spatial range of the node. The materialized views are stored as key-value pairs.

Query Processor. On receiving a query q, the query processor checks whether the materialized views can be used to answer q. Then, it selects the view(s) that incur the minimum cost to process q. Otherwise, it processes q by using the index structure that stores the objects and outputs an intermediate result. Then, the intermediate result is forwarded to the aggregator to compute the final result.

4 DEMO SCENARIO

We demonstrate the capabilities of STAR using two additional application scenarios. STAR is the first system that natively supports both snapshot and continuous types of such queries and we are not aware of other existing systems supporting such queries. Figure 4 shows the map assisted user interface of STAR.

Taxi Service. In this scenario, a manager of a taxi company wants to adjust the distribution of taxis in a region. To maximize the profit, the manager would like to distribute taxis according to the number of customers in different streets. The manager can issue the following query to track the number of customers on each street in a region every minute.
SELECT *Count*(), *streetName*, *minute* **FROM** *stream*
WHERE *loc* **INSIDE** *R* **AND** *time* **AFTER** "10 mins ago"
GROUP BY *streetName*, *minute* **SYNC** 1 minute.

News Recommendation. In this scenario, a system manager of a news recommendation system wants to provide localized news recommendation service. For the quality of service, the manager would like to recommend news on the latest "hot topic". The manager can issue the following query to find the hot topics for a region in the last 10 minutes.
SELECT *Count*(), *topic*, *minute* **FROM** *stream*
WHERE *loc* **INSIDE** *R* **AND** *time* **AFTER** "10 mins ago"
GROUP BY *topic*, *minute*
ORDER BY *Count*() **DESC**.

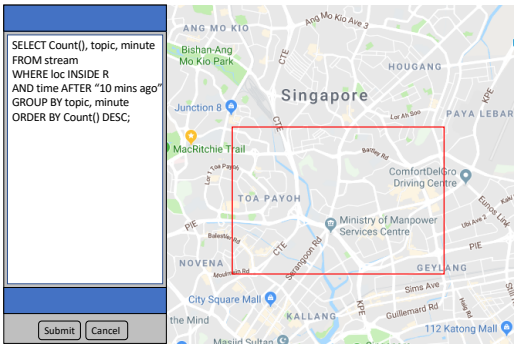

Figure 4: Map assisted user interface.

ACKNOWLEDGMENTS

We would like to thank Tom Z.J. Fu and Zhenjie Zhang for helpful discussion. Gao Cong is partially supported by Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU), which is a collaboration between Singapore Telecommunications Limited (Singtel) and Nanyang Technological University (NTU) that is funded by the Singapore Government through the Industry Alignment Fund - Industry Collaboration Projects Grant, a MOE Tier-2 grant MOE2016-T2-1-137, and a NTU ACE grant. Walid G. Aref acknowledges the support of the U.S. National Science Foundation under Grant Numbers: IIS-1910216 and III-1815796.

REFERENCES

[1] Michael Carey, Steven Jacobs, and Vassilis Tsotras. 2016. Breaking BAD: a data serving vision for big active data. In *DEBS*. 181–186.
[2] 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.
[3] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *NSDI*, Vol. 10.
[4] Lukasz Golab, Theodore Johnson, J Spencer Seidel, and Vladislav Shkapenyuk. 2009. Stream warehousing with DataDepot. In *SIGMOD*. ACM, 847–854.
[5] 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.
[6] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *PVLDB* 11, 11 (2018), 1661–1673.