TrioStat: Online Workload Estimation in Distributed Spatial Data Streaming Systems

Anas Daghistani
Purdue University
West Lafayette, Indiana
Umm Al-Qura University
Makkah, Saudi Arabia
ahdaghistani@uqu.edu.sa

Walid G. Aref
Purdue University
West Lafayette, Indiana
aref@purdue.edu

Arif Ghafoor
Purdue University
West Lafayette, Indiana
ghafoor@purdue.edu

ABSTRACT

The wide spread of GPS-enabled devices and the Internet of Things (IoT) has increased the amount of spatial data being generated every second. The current scale of spatial data cannot be handled using centralized systems. This has led to the development of distributed spatial data streaming systems that scale to process in real-time large amounts of streamed spatial data. The performance of distributed streaming systems relies on how even the workload is distributed among their machines. However, it is challenging to estimate the workload of each machine because spatial data and query streams are skewed and rapidly change with time and users' interests. Moreover, a distributed spatial streaming system often does not maintain a global system workload state because it requires high network and processing overheads to be collected from the machines in the system.

This paper introduces *TrioStat*; an online workload estimation technique that relies on a probabilistic model for estimating the workload of partitions and machines in a distributed spatial data streaming system. It is infeasible to collect and exchange statistics with a centralized unit because it requires high network overhead. Instead, TrioStat uses a decentralised technique to collect and maintain the required statistics in real-time locally in each machine. TrioStat enables distributed spatial data streaming systems to compare the workloads of machines as well as the workloads of data partitions. TrioStat requires minimal network and storage overhead. Moreover, the required storage is distributed across the system's machines.

CCS CONCEPTS

• **Information systems → Spatial-temporal systems**; **Parallel and distributed DBMSs**; **Geographic information systems**; • **Computing methodologies → Distributed algorithms**.

SIGSPATIAL '20, November 3–6, 2020, Seattle, WA, USA

ACM ISBN 978-1-4503-8019-5/20/11…$15.00
https://doi.org/10.1145/3397536.3422220

KEYWORDS

Workload estimation, distributed streaming systems, spatial stream processing, collecting statistics, load balancing

ACM Reference Format:
Anas Daghistani, Walid G. Aref, and Arif Ghafoor. 2020. TrioStat: Online Workload Estimation in Distributed Spatial Data Streaming Systems. In *28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '20), November 3–6, 2020, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3397536.3422220

1 INTRODUCTION

The ubiquity of GPS-enabled smart devices, the Internet-of-Things (IoT), and social networks has led to the development of location-based services that produce large volumes of data. For example, 500 million tweets are created every day, and they can be geo-tagged [3]. The current scale of the spatial data being generated cannot be handled using centralized environments. This has led to the development of distributed spatial streaming systems.

Distributed spatial streaming systems distribute the workload across machines by making each machine responsible for some data partitions. The partitions are generated by dividing the underlying space into spatial rectangles. User queries and data points are directed based on their locations to the machines that handle the overlapping partitions. A key challenge to improve the performance and scalability is to ensure even workload across all machines. One obstacle is to estimate the workload of spatial distributed streaming systems because spatial data and query workloads change rapidly. Moreover, spatial distributions of data and queries are skewed. This skewness changes rapidly with time and users' interests, e.g., different timezones can lead to significant changes in the distribution of the spatial data being streamed. Distributed spatial streaming systems often do not maintain a global system workload state because the data and queries are distributed across their machines. Collecting and maintaining statistics about the workload of every machine in a centralized location introduces high network, storage, and processing overheads.

This paper introduces *TrioStat*; an online workload estimation technique that relies on a probabilistic model to estimate the workload of partitions and machines in a distributed spatial data streaming system. TrioStat introduces a new statistics structure that requires minimal storage overhead. TrioStat uses a decentralised technique to collect and maintain the required statistics in real-time locally in each machine. Thus, TrioStat introduces negligible network overhead. Moreover, TrioStat has an efficient algorithm to collect the statistics with very localized overhead to process every

newly received data point or query. TrioStat enables distributed spatial data streaming systems to compare the workloads of both the machines and the data partitions.

The rest of this paper proceeds as follows. Section 3 presents a cost model for estimating the workloads of machines and data partitions. Section 2 discusses related work. Section 4 presents TrioStat's statistics, and how they are maintained. Section 5 presents how TrioStat uses the statistics in a cost model to estimate the workload. Section 6 studies TrioStat's performance. Section 7 concludes the paper.

2 RELATED WORK

Distributed streaming systems can be categorized based on their processing model into batch-at-a-time and tuple-at-a-time systems. Spark Streaming [25], M3 [5] are examples of batch-at-a-time systems that accumulate batches of data before distributing them for processing. In contrast, tuple-at-a-time systems process data tuple once it arrives to produce results with low latency. Example tuple-at-a-time systems are: Apache Storm [24] and Twitter Heron [15]. The performance of these systems relies on how evenly they distribute their workload among their machines. They lack a global state that provides workload estimates. TrioStat estimates the workload for distributed spatial streaming systems with minimum overhead. TrioStat applies to tuple-at-a-time systems that are suitable for real-time processing. It is challenging to track the workload as it is distributed across multiple machines.

The workload of distributed systems is skewed and is rapidly changing. Several techniques exist to track data changes and adapt accordingly. Belussi, et al. [9] propose an approach for SpatialHadoop [12] to detect the skewness degree in spatial data distribution using box-counting functions [8]. They choose the best partitioning strategy using a heuristic sketch and the detected skewness degree. However, this technique cannot track online changes in skewness in a distributed streaming setup. Also, they estimate the workloads based on the data distribution without considering the query workloads. Fang, et al. [13] introduce a key-based workload partitioning strategy to rebalance the workload with minimum migration overhead. SIMOIS [26] balances the workload by identifying the set of workload-heavy keys and optimizes join queries accordingly. Identifying hotspot keys is performed using an exponential counting scheme. PKG2 and PKG5 [18, 19] evenly distribute the received workload for each key among a limited number of system machines. They use only the key frequencies as an estimate for the workload and do not consider queries. Thus, they are not suitable for tracking changes in spatial data distribution and estimating the workload of their partitions.

Techniques have been proposed for efficient spatial data aggregation and summarization. Ho, et al. [14] introduce a technique to answer range-sum queries of the number of points in a window by maintaining prefix sums in a grid. Riedewald, et al. [20] generalize the idea of prefix sums to count the number of rectangles (queries) and support OLAP queries. Maintaining aggregates and summarizations for spatial regions is challenging because the counting could result in duplicate counting of some queries. Euler histograms [7] count rectangles that intersect a given region without duplicates. Euler histograms help estimate the selectivity of spatial joins [6, 23]. AQWA adaptively changes the partitioning of Hadoop [1] by maintaining statistics using the prefix sum technique and a variant of the Euler histogram. AQWA introduces a cost model to estimate both the data and query workloads. However, it is centralized and hence is not viable for distributed streaming systems because data and statistics are distributed on different machines.

Some distributed streaming systems use adaptive load-balancing that redistribute the workload based on AQWA's cost model for estimating the workload, e.g., STAR [10], Tornado [16, 17], Amoeba [21, 22], and PS^{2}Stream [11]. However, these techniques are relatively slow when updating the statistics and updating the workload cost model when the statistics change. The reason is that they consider only the history of data and queries without considering how persistent these estimates could be in the future. Most distributed streaming process the data in real-time and do not store the data for a long duration. Systems with adaptive load-balancing need a technique that can accurately predict the workload fast with minimum network and processing overheads.

3 THE COST MODEL

Distributed spatial streaming systems divide the whole space that the application serves into partitions. The partitions are distributed across the participating machines in the system. The system rebalances the workload across its machines by repartitioning and/or redistributing the partitions. TrioStat estimates the workload of a partition by computing its potential processing cost relative to all other partitions. Moreover, the workload of a machine is estimated according to the partitions served by the machine. TrioStat introduces a probabilistic cost model that relies on three terms (and hence the name TrioStat). The main factor of the cost model is the amount of data points that are received by each partition. Also, the cost model gives higher weight to partitions having a larger number of queries. The reason is that the number of queries indicates the required number of query checks against every new data point. Moreover, the cost model predicts the future workload of each partition based on its workload history. This prediction serves as a scale factor for the overall cost and workload of each partition. Assume that we have a distributed spatial streaming system, say S, that has a set of executor machines M. Each machine $m \in M$ holds some partitions P_m, where $|P_m| = n_m$, n_m is the number of partitions in Machine m. Each partition $p \in P_m$, locally maintains some statistics. The cost estimate $C(p)$ of a partition p is computed as follows:

$$C(p) = N(p) \times Q(p) \times Prob(p) \quad (1)$$

$N(p)$ is the number of points received by Partition p, $Q(p)$ is the number of queries that overlap p, and $Prob(p)$ is the probability that new data and queries land in p. $Prob(p)$ depends on the amount of data and queries that arrive during the last round of repartitioning. Note that the workload history is captured via N and Q while $Prob$ is a weighting factor to the cost of this history. The effect of old data can fade with time as in Section 4.2. $Prob(p)$ is estimated as follows:

$$Prob(p) = \frac{R(p)}{R(S)} \quad (2)$$

where $R(p)$ and $R(S)$ are the number of data points and queries received by p and all of S, respectively, during the last round of

repartitioning. $R(S)$ is computed as follows:

$$R(m) = \sum_{i=1}^{n_m} R(p_i),\ R(S) = \sum_{i=1}^{|M|} R(m_i) \tag{3}$$

By substituting Eqn. 2 into Eqn. 1, then:

$$C(p) = \frac{N(p)Q(p)R(p)}{R(S)} = \frac{Num(C(p))}{R(S)} \tag{4}$$

where $Num(C(p))$ is the numerator of Partition p's cost formula. Machine m's workload is computed based on Partitions P_m that m holds:

$$C(m) = \sum_{i=1}^{n_m} C(p_i) \tag{5}$$

Using Eqn. 4,

$$C(m) = \frac{N(p_1)Q(p_1)R(p_1)}{R(S)} + \cdots + \frac{N(p_{n_m})Q(p_{n_m})R(p_{n_m})}{R(S)}$$

$$C(m) = \frac{\sum_{i=1}^{n_m}\{N(p_i)Q(p_i)R(p_i)\}}{R(S)} = \frac{Num(C(m))}{R(S)} \tag{6}$$

where $Num(C(m))$ is the numerator of Machine m's cost formula. $Num(C(m))$ can be computed locally. In contrast, computing $R(S)$ requires data from all machines in S. $R(S)$ is the same for all machines, and hence is computed once using Eqn. 3 that requires only one number, $R(m)$, from each machine. Thus, comparing and ranking the machines based on their costs is the same as comparing and ranking them using only $Num(C(m))$. Also, computing $Num(C(p))$ for the partitions of a machines is enough to compare and rank by cost the partitions locally in their machine.

4 COLLECTING AND MAINTAINING STATISTICS

Collecting statistics in distributed streaming systems is challenging because the data arrives continuously in high volume. Also, most applications need real-time processing with minimum latency. Thus, any collected statistics should require minimum number of updates. Also, each partition should maintain its statistics locally without the need to communicate with other machines. TrioStat maintains minimum local statistics that help estimate the workloads of partitions and machines using the cost model in Section 3. TrioStat uses a hash table in every executor machine to link the ID of every partition in the machine with its statistic structure. TrioStat maintains the statistics of every partition in a simple multidimensional array in memory. The statistics of each row (or column) is located next to each other in memory. Therefore, TrioStat can provide workload estimations for a partition or a part of a partition fast by taking advantage of cache prefetching. Reading the first needed statistic to compute a workload estimation from a row (or column) result on having the remaining needed statistics in cache. The process of collecting and maintaining statistics is explained in greater detail below.

4.1 Required Statistics

TrioStat maintains minimal statistics that are needed by the cost model. The underlying space is divided into a grid of small cells that are aligned with partition boundaries. Figure 1 gives an example for dividing the space into a grid of 8X8 small cells. The arrangement of cells that cover a partition is passed to TrioStat with the partition ID of the executor machine that holds this partition. Increasing

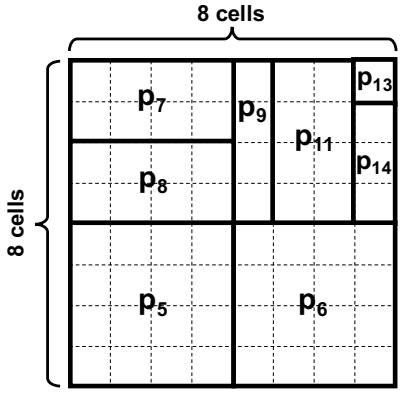

Figure 1: The space is divided into a grid of small cells.

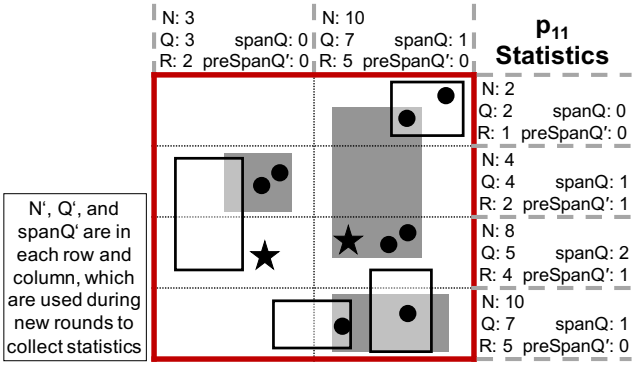

Figure 2: TrioStat statistics for Partition p_{11}

the number of cells that divide the space increases the storage and processing overhead of TrioStat and increases the resolution of workload estimation. We use Partition p_{11} in Figure 1 to illustrate how TrioStat maintains the statistics.

Systems periodically check if repartitioning could improve performance. TrioStart provides the needed workload estimates by the end of every repartitioning round. Figure 2 gives the statistics maintained by TrioStat to estimate p_{11}'s workload. The dots and rectangles represent the positions of the data points and the query ranges in p_{11}, respectively. The stars and the gray rectangles mark the data points and the queries received in the last round of repartitioning, respectively. p_{11} has a 4X2 cell matrix. TrioStat maintains in each row (column) 5 statistics; 3 of which are cumulative. Row i's (Column j's) cumulative statistics represent the total from the uppermost row (leftmost column) until Row i (Column j), respectively. The 5 maintained statistics in each row (column) are: (1) N: the cumulative number of data points, (2) Q: the cumulative number of queries, (3) R: the cumulative number of data points and queries received during the last round of repartitioning, (4) $spanQ$: the number of queries whose ranges span from the previous row/column, and (5) $preSpanQ'$: the number of queries received during the last repartitioning round whose ranges span from the previous row/column. Refer to Row 3 of p_{11} in Figure 2. All cumulative statistics reflect the objects in the first 3 rows. There are 8 data points (N) and 5 queries (Q). Two data points and two queries are received during the last round, hence $R = 4$. Two queries span from the second row ($spanQ = 2$). However, only one of them

is received during last round ($preSpanQ' = 1$). TrioStat uses these statistics to estimate the workload of each part of a partition. The overall statistics of a partition p ($N(p)$, $Q(p)$, and $R(p)$) are the ones in the last row/column. The statistics are only updated at the end of every repartitioning round to avoid the overhead of updating almost all the statistics whenever a new data point or query arrives. Three more statistics, termed *Statistics Collectors*, for each row and column are introduced, namely N', Q', and $spanQ'$. *Statistics Collectors* are used to update the statistics at the end of a round. They reduce the number of updates per received data point or query.

4.2 Maintaining the Statistics

TrioStat performs a few updates when receiving a data point or query. When a point arrives, TrioStat updates two of a partition's *Statistics Collectors*. However, when a query arrives, TrioStat updates the *Statistics Collectors* of the rows (columns) that overlap the query. Having more statistics to update will not affect performance because the arrival rate of data is much higher than that of queries. Three *Statistics Collectors*, N', Q', and $spanQ'$, are used in each row/column to count different types of received objects during the most recent round of repartitioning. N' and Q' count the new data points and queries, respectively. $spanQ'$ counts the number of queries that their ranges span from the previous row/column. When a data point arrives, TrioStat increments N' of the row (column) containing the data point. When a query arrives, TrioStat increments both Q' of the row (column) that overlap the top-left corner of the query, and $spanQ'$ of the remaining rows (columns) that overlap the query. To conclude a repartitioning round, TrioStat uses the *Statistics Collectors* to update all remaining statistics as follows. Let $i \geq 0$ be a row/column index. Then, the statistics are updated as follows:

$$N(i) = N(i) + \sum_{j=0}^{i} N'(j)$$
$$Q(i) = Q(i) + \sum_{j=0}^{i} Q'(j)$$
$$R(i) = \sum_{j=0}^{i} N'(j) + \sum_{j=0}^{i} Q'(j)$$
$$spanQ(i) = spanQ(i) + spanQ'(i)$$
$$preSpanQ'(i) = spanQ'(i)$$

The naive way to compute the cumulative statistics requires computing the summations from the beginning each time. Its time complexity is $O(k^2)$, where k is the number of rows and columns of the partition's statistics. However, TrioStat utilizes the fact that the summations in the equations can be carried out from one row/column to another. Hence, there is no need to compute the summations from scratch each time. With only one addition, we produce the statistics of the next row/column from these of the previous row/column. Algorithm 1 illustrates how to update the statistics of a partition by passing once through the partition's rows and columns. The time complexity of using Algorithm 1 to update the statistics of a partition is is $O(k)$. This algorithm runs as a separate background task. Note that all *Statistics Collectors* are reset to 0 to be ready for collecting the statistics of the next round of repartitioning.

Figure 3 illustrates the statistics of Partition p_{11} while receiving new data points and queries. Figure 3a illustrates the positions of the data points and the ranges of the queries in p_{11} at the beginning of a new repartitioning round. Also, it shows the current state of the maintained statistics, as in Section 4.1. The *Statistics Collectors* are all set to 0 at the beginning of the round. Figure 3b gives the *Statistics Collectors* at the end of the round after receiving 2 new data points and 3 new queries. During the repartitioning round, the two data points D_A and D_B are received first. Both data points are in the third row (Row_2), in Columns Col_0 and Col_1, respectively. Hence, $N'(Row_2)$ is incremented twice while $N'(Col_0)$ and $N'(Col_1)$ are each incremented once. Then, Queries Q_A, Q_B, and Q_C arrive into p_{11} in this order. The upper-left corner of Q_A is in the cell that overlaps Col_0 and Row_1. Also, the range of Q_A is contained within one cell. Thus, only $Q'(Row_1)$ and $Q'(Col_0)$ are incremented. Q_B starts in Row_0 and spans through Row_1 and Row_2. Thus, $spanQ'(Row_1)$ and $spanQ'(Row_2)$ are incremented in addition to the increment of $Q'(Row_0)$ and $Q'(Col_1)$. At the end of the round, *Statistics Collectors* are used to update the statistics using Algorithm 1. The results of the updated statistics are given in Figure 2.

Algorithm 1: updateStat(*PartitionID*, *rowOrColumn*)

```
1  stat[ ][ ] = partitionsHashMap.get(PartitionID)
       .statistics(rowOrColumn)    ▷ Multidimensional array
2  int sumN' = 0
3  int sumQ' = 0
4  for i = 0 to Num of rowOrColumn in PartitionID do
5      sumN' += stat[N'][i]
6      sumQ' += stat[Q'][i]
7      stat[N'][i] = 0              ▷ Reset current N'
8      stat[Q'][i] = 0              ▷ Reset current Q'
9      stat[N][i] += sumN'
10     stat[Q][i] += sumQ'
11     stat[preSpanQ'][i] =stat[spanQ'][i]
12     stat[spanQ][i] += stat[spanQ'][i]
13     stat[spanQ'][i] = 0          ▷ Reset current spanQ'
14     stat[R][i] = sumN' + sumQ'
15 end
```

Notice that the target of TrioStat is not to count the actual number of data points but rather to track the change in the spatial data workload. To diminish the effect of old data gradually, the number of data points N is divided by 2 before it is updated in each round of repartitioning. This is to reduce the effect of old data points on the current spatial distribution. In distributed streaming systems that support historical queries, TrioStat needs to be informed about data expiration to update N accordingly.

4.3 Correctness of the Statistics

In this section, we prove the correctness of the statistics that TrioStat collects and maintains about data points and queries. To show this, we need to prove that the maintained statistics always represent the true number of data points and queries without any over- or under-counting.

Correctness of the Statistics for Point Data. Assume that we have a partition that has k rows and only one column. This results in k cells in total as in Figure 4.

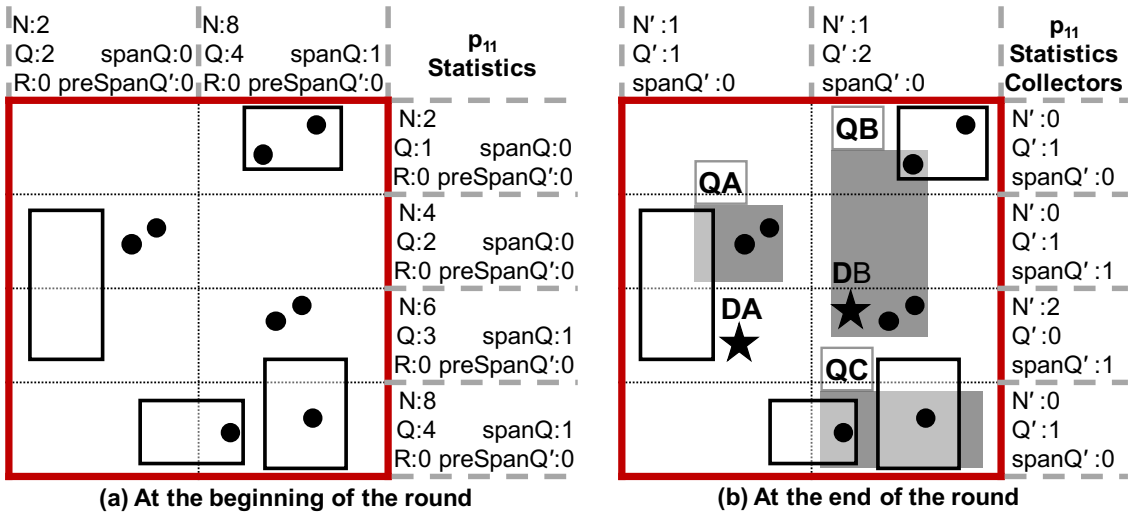

Figure 3: Updating Partition p_{11}'s *Statistics Collectors*

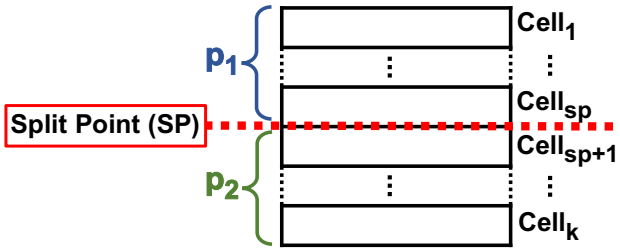

Figure 4: Partition with k cells (rows)

Let i be the row number of a cell, where $1 \leq i \leq k$, $n(i)$ be the true number of data points in $cell_i$, and $N(i)$ be the cumulative number of data points that TrioStat maintains in row_i. $n(i)$ can be obtained by simply counting the number of data points within $cell_i$. As mentioned before, the cumulative number $N(i)$ is computed from top to down for horizontal divisions. Therefore, $N(i) = \sum_{j=1}^{i} n(j)$.

In the initial case where $k = 1$, there is only one cell with $n(1)$ data points, hence $N(1) = n(1)$. For $k = 2$, $N(2) = n(1) + n(2)$. We can derive the number of data points in $cell_2$ as $n(2) = N(2) - N(1)$. In general (refer to Figure 4 for illustration), assume that a partition has a split point sp, where $1 <= sp <= k$, that divides the partition into two partitions, say p_1 and p_2. Let $n(p_i)$ be the true number of data points in Partition p_i. Thus, the number of data points in each partition can be computed as follows:

$$n(p_1) = n(1) + n(2) + ... + n(sp)$$
$$\therefore n(p_1) = \sum_{j=1}^{sp} n(j) = N(sp)$$
$$n(p_2) = n(sp+1) + n(sp+2) + ... + n(k) = \sum_{j=sp+1}^{k} n(j)$$
$$\therefore n(p_2) = \sum_{j=1}^{k} n(j) - \sum_{z=1}^{sp} n(z) = N(k) - N(sp)$$

This shows that the computed statistic N is equal to the true number of data points, i.e., n. An analogous proof can show that N is also correct when dividing cells vertically, and N is cumulatively computed from left to right.

Correctness of the Statistics for Queries. Refer to Figure 4. Given the input query boundaries, we can count all queries in each grid $cell_i$ by maintaining four variables, namely, q_s, q_e, q_{se}, and q_o (s, e, se, o are short for start, end, start and end, and overlap, resp. Let $q_s(i)$ be the number of queries whose upper boundary intersects $cell_i$ and whose lower boundary intersects another cell, $q_e(i)$ be the number of queries whose lower boundary intersects $cell_i$ and whose upper boundary intersects another cell, $q_{se}(i)$ be the number of queries whose upper and lower boundaries intersect $cell_i$, and $q_o(i)$ be the number of queries whose upper and lower boundaries do not intersect $cell_i$ but their ranges overlap $cell_i$. Thus, the true number, $q(i)$, of queries that intersect $cell_i$ is the sum of these four variables, i.e.,

$$q(i) = q_s(i) + q_e(i) + q_{se}(i) + q_o(i) \quad (7)$$

We extend this formula to compute the true number of queries, $q(u, l)$, that overlap a one-column sub-partition whose column of cells starts from Row u and ends in Row $l > u$. We need to avoid double counting of a query that overlaps multiple cells. $q(u, l)$ is the true number of queries in Row u and only queries that start in any row from Row $u + 1$ up to Row l, no matter where these queries end. For rows after u, only counting queries that start in any cell will exclude recounting any query that span over multiple cells. Therefore, $q(u, l)$ can be computed as follows:

$$q(u, l) = q(u) + \sum_{j=u+1}^{l} (q_s(j) + q_{se}(j)) \quad (8)$$

We demonstrate that the statistics gathered by TrioStat when counting the number of queries equals $q(u, l)$, the true number. Refer to Figure 4 for illustration. We have a partition with k cells from Cell 1 at the top to Cell k at the bottom. To maintain query statistics, for each Row i, $1 \leq i \leq k$, of a partition, TrioStat maintains only two statistics per row, namely, $Q(i)$ and $Q_{span}(i)$. $Q(i)$ is the cumulative number of queries from Row 1 of the partition to Row i. Thus, $Q(i)$ directly represents the number of queries that start at any row from the beginning of the partition until Row i. Recounting of queries can happen by considering queries that only end or overlap any of the cells as they are already counted where they start. Thus, they are excluded from $Q(i)$ as follows:

$$Q(i) = \sum_{j=1}^{i} (q_s(j) + q_{se}(j)) \quad (9)$$

Let $Q_{span}(i)$ be the number of queries that extend (span) from an upper row, say Row $(i-1)$, to Row i. Thus, $Q_{span}(i)$ is the number of queries that overlap or start without ending in Row $(i-1)$. $Q_{span}(1)$ is always 0 because there are no queries that extend from outside the partition to the first row. Thus, $Q_{span}(i)$ is formulated from the true values as follows:

$$Q_{span}(i) = \begin{cases} 0, \text{ if } i = 1 \\ q_s(i-1) + q_o(i-1), \text{ otherwise} \end{cases}$$

$Q_{span}(i)$ depends on the variables of the previous row $(i-1)$]. However, there is an equivalent way of computing $Q_{span}(i)$ with the variables from Row i that makes the proof sketch easier to follow. Any query that overlaps Row $(i-1)$ or that starts without ending in Row $(i-1)$ definitely extends to Row i and this query's range either ends at Row i or overlaps Row i and continues to the next row below Row i. This is reflected in the formula for calculating $Q_{span}(i)$ as follows:

$$\because q_s(i-1) + q_o(i-1) = q_e(i) + q_o(i)$$
$$\therefore Q_{span}(i) = q_e(i) + q_o(i) \quad (10)$$

This equation for calculating $Q_{span}(i)$ is correct also in the case when $i = 1$ because both $q_e(1)$ and $q_o(1)$ are always 0.

In the initial case, when $k = 1$, and there is only one cell in the partition with $q(1)$ queries, $Q(1) = q_s(1) + q_{se}(1) = q(1)$ and $Q_{span}(1) = 0$. This is correct because $q_s(1) = q_e(1) = q_o(1) = 0$ as $cell_1$ covers the whole partition and every query definitely starts and ends in this cell. For $k = 2$, by Eqns. 9 and 10, Q and Q_{span} for Rows 1, 2 are computed by:

$$Q(1) = q_s(1) + q_{se}(1)\ ,\ Q(2) = Q(1) + q_s(2) + q_{se}(2)$$
$$Q_{span}(1) = q_e(1) + q_o(1) = 0\ ,\ Q_{span}(2) = q_e(2) + q_o(2)$$

Notice that when $k = 2$, there are only the following three possible sub-partitions: a partition that has $cell_1$ only, $cell_2$ only, or $cell_1$ and $cell_2$. The computation of the true numbers can be computed using Eqn. 8 as follows:

$$\begin{aligned} q(1,1) &= q(1) = q_s(1) + q_e(1) + q_{se}(1) + q_o(1) \\ &\therefore = q_s(1) + 0 + q_{se}(1) + 0 = Q(1) \\ q(1,2) &= q(1) + q_s(2) + q_{se}(2) \\ &= q_s(1) + q_e(1) + q_{se}(1) + q_o(1) + q_s(2) + q_{se}(2) \\ &= q_s(1) + 0 + q_{se}(1) + 0 + q_s(2) + q_{se}(2) \\ &\therefore = Q(1) + q_s(2) + q_{se}(2) = Q(2) \\ q(2,2) &= q(2) = q_s(2) + q_e(2) + q_{se}(2) + q_o(2) \\ &\therefore = Q(2) - Q(1) + Q_{span}(2) \end{aligned}$$

Notice that the maintained statistics are enough to compute the true number of queries in all sub-partitions for $k = 2$. Refer to Figure 4 for illustration. For cases $k > 2$, assume that a partition has a split point sp, $1 <= sp <= k$, that splits the partition into two sub-partitions, say p_1 and p_2. TrioStat's query statistics are computed by Eqns. 9 and 10 as follows:

$$Q(sp) = \sum_{j=1}^{sp}(q_s(j) + q_{se}(j))$$
$$Q(k) = \sum_{j=1}^{k}(q_s(j) + q_{se}(j))$$
$$Q_{span}(sp+1) = q_o(sp+1) + q_e(sp+1)$$

The true number of queries in Partition p_1 is computed by Eqns. 8 and 7 as follows:

$$\begin{aligned} q(p_1) = q(1, sp) &= q(1) + \sum_{j=2}^{sp}(q_s(j) + q_{se}(j)) \\ &= q_o(1) + q_e(1) + q_{se}(1) + q_s(1) + \sum_{j=2}^{sp}(q_s(j) + q_{se}(j)) \\ &= q_o(1) + q_e(1) + \sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) \\ \therefore &= 0 + 0 + \sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) = Q(sp) \end{aligned}$$

Notice that the computed statistic $Q(sp)$ is equal to the true number of queries in $p1$, i.e., $q(p_1)$. The true number of queries in Partition p_2 is computed as follows:

$$\begin{aligned} q(p_2) = q(sp+1, k) &= q(sp+1) + \sum_{j=sp+2}^{k}(q_s(j) + q_{se}(j)) \\ &= q_o(sp+1) + q_e(sp+1) + q_{se}(sp+1) + q_s(sp+1) \\ &\quad + \sum_{j=sp+2}^{k}(q_s(j) + q_{se}(j)) \\ &= q_o(sp+1) + q_e(sp+1) + \sum_{j=sp+1}^{k}(q_s(j) + q_{se}(j)) \\ \therefore q(p_2) &= Q_{span}(sp+1) + \sum_{j=sp+1}^{k}(q_s(j) + q_{se}(j)) \\ &= Q_{span}(sp+1) + \sum_{j=1}^{k}(q_s(j) + q_{se}(j)) \\ &\quad - \sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) \\ \therefore q(p_2) &= Q_{span}(sp+1) + Q(k) - Q(sp) \end{aligned}$$

Therefore, TrioStat's statistics (Q and Q_{span}) are necessary and sufficient to compute the true number of queries. The same proof can be used to show that TrioStat's statistics are correct by using the computed cumulative number Q from left to right when dividing partitions vertically.

TrioStat's Statistic R represents the cumulative number of the newly received data points and queries. Thus, the proof of correctness for R is the same as those for the data points and the queries explained above. However, in the proofs, $Q_{preSpan}$ is to used instead of Q_{span}, where the former represents the span of only the new queries.

5 ESTIMATING THE WORKLOAD

Periodically, distributed spatial streaming systems need to evaluate the effectiveness of their partitioning and workload distribution. Hence, by the end of every repartitioning round, TrioStat collects the statistics as in Section 4. Using these statistics, TrioStat can estimate the workload in $O(1)$ for partitions, parts of a partition, and entire machines.

According to Eqns. 4 and 6, there is no need to divide by $R(S)$ to compare the workload of partitions and machines because $R(S)$ is common in all equations. Thus, TrioStat estimates the workload of a Partition p to be $W(p) = Num(C(p))$, and the workload estimation for a Machine m to be $W(m) = Num(C(m))$. Let $N(i)$, $Q(i)$, $R(i)$, $spanQ(i)$, and $preSpanQ'(i)$ be the statistics of p at Row (Column) Index i. Also, let L be the index of the last row (column) of p's statistics. TrioStat estimates the workload of Partition p by using p's statistics as follows:

$$W(p) = N(L) \times Q(L) \times R(L)$$

For example, the workload estimation of Partition p_{11} in Figure 5 is $W(p_{11}) = 10 \times 7 \times 5 = 350$.

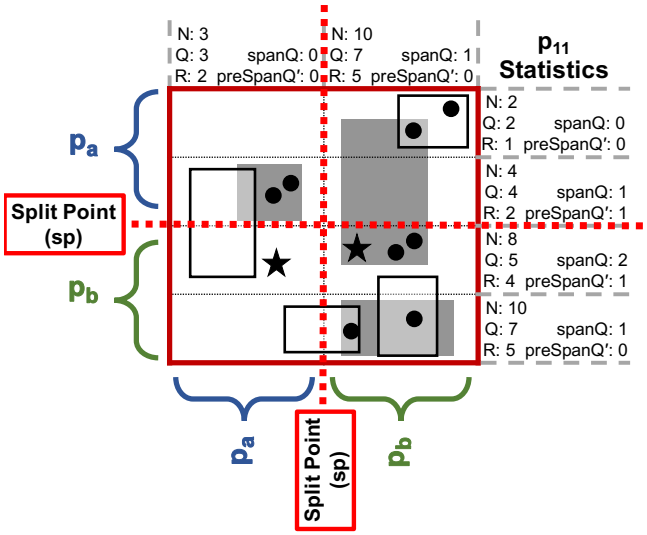

Figure 5: Estimating the workload when Partition p_{11} splits into the two Sub-partitions p_a and p_b

TrioStat estimates the workload of sub-partitions that could result from splitting p into 2 sub-partitions p_a and p_b. Figure 5 illustrates splitting p_{11} vertically or horizontally. The split point (sp) is the row (column) index, where the partition is split. TrioStat uses the maintained statistics directly to estimate the workloads of the sub-partitions as follows:

$$W(p_a) = N(sp) \times Q(sp) \times R(sp)$$
$$Q(p_b) = Q(L) - Q(sp) + spanQ(sp+1)$$
$$R(p_b) = R(L) - R(sp) + preSpanQ'(sp+1)$$
$$W(p_b) = [N(L) - N(sp)] \times Q(p_b) \times R(p_b)$$

For example, when P_{11} is split horizontally on the second row as in Figure 5: $W(p_a) = 32$, $Q(p_b) = 5$, $R(p_b) = 4$ and $W(p_b) = 120$. Notice that $W(p)$ is always greater than or equal to $W(p_a) + W(p_b)$ because the queries get distributed among the sub-partitions. The sum of the sub-partitions' workload estimations is usually smaller than the original partition's workload estimation. The reason is that the total required number of query checks against every newly received data point is decreased. Also, the probability of receiving new objects can be different for each sub-partition. The sum of the workload estimates of the sub-partitions can be equal to the workload estimate of the original partition only when the split point (sp) cuts all the queries of the original partition and the probability of receiving new objects is equal for both sub-partitions.

TrioStat estimates the workload of a machine $W(m)$ by summing the workload estimates of the partitions that m holds. To compare the machines according to their workloads, the machine that performs the comparison should probe all machines to share their workload estimates. Hence, TrioStat requires minimal network overhead. Moreover, $R(S)$ can be computed in a common machine by getting and summing $R(m)$ of all machines. $R(S)$ is a useful measurement to monitor the throughput of the system because it represents the number of objects (data points and quires) that have been served by the system during the last round of repartitioning.

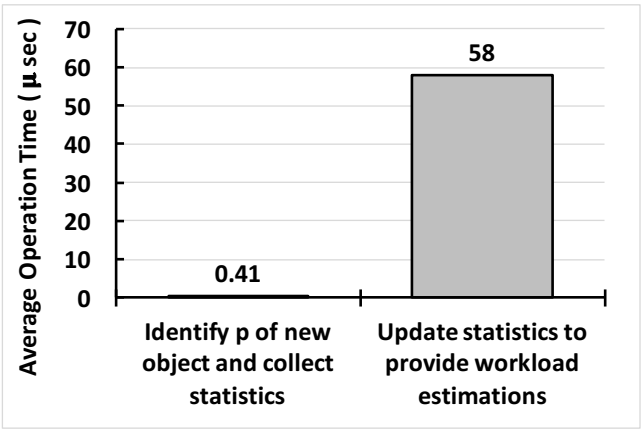

Figure 6: Overhead of TrioStat in executor machines

6 EXPERIMENTS

We realize TrioStat in Apache Storm [24]. However, TrioStat can be used with any other distributed spatial streaming system that processes streams in tuple-at-a-time manner. In these experiments, TrioStat provides workload estimates for an application that processes a real dataset from Twitter. The dataset is composed of 1 Billion geotagged tweets of size 140 GB in the US. The tweets are collected from January 2014 to March 2015. To simulate an infinite data stream, the 1 Billion tweets are replayed repeatedly each time they finish. The application's query workload is composed of continuous range queries. Locations of the real tweets are used as the query focal points. The continuous queries return tweets that overlap the queries' spatial ranges. TrioStat's grid that divides the whole space is of size 1000×1000. This size allows small cities in the US to be covered by multiple cells. The spatial side lengths of queries are 0.16% of the side length of the whole space (about the size of a university campus).

Experiments are performed using 6 Amazon EC2 instances. The network bandwidth is up to 10 Gbps. Apache Storm 1.0.0 runs in each instance over Ubuntu 18.04.2. One of the instances is of type m5.xlarge with 4 vCPU and 16 GB of memory. This instance has the Nimbus of Storm and a Zookeeper server [2] installed. The remaining 5 instances are of type m5.2xlarge, where each instance has 8 vCPU and 32 GB of memory. Each of the 5 instances is divided into 8 virtual machines each having one vCPU and 4 GB of memory. This results in a total of 40 virtual machines. 10 of the virtual machines act as Storm spouts to produce tweets and query streams. The application divides the remaining 30 virtual machines as 8 routing machines and 22 Executor machines. The routing machines distribute the workload among the executor machines based on the partitions held by each machine. A new repartitioning round is started every 15 seconds. By the end of every round, TrioStat updates the statistics, and requests from all executor machines to send their workload estimates ($W(m)$) and the number of newly received objects ($R(m)$) to one of the routing machines. All the experiments are performed from a cold start.

Figure 6 illustrates the overhead of TrioStat operations by showing the average time each operation takes in microseconds after running the system for an hour. Figure 6 illustrates that TrioStat adds 0.41 microseconds to the processing time of a new object to

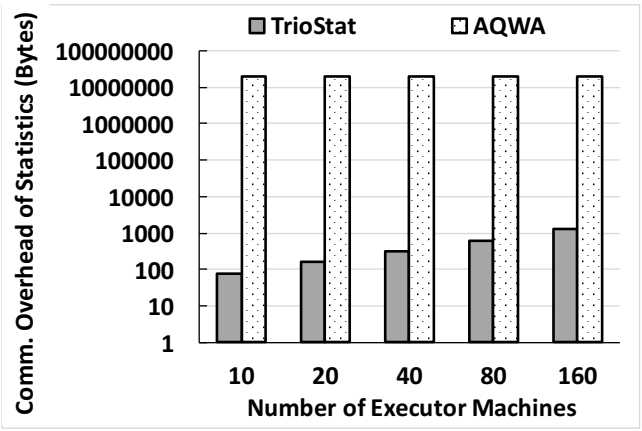

Figure 7: Network overhead of TrioStat statistics

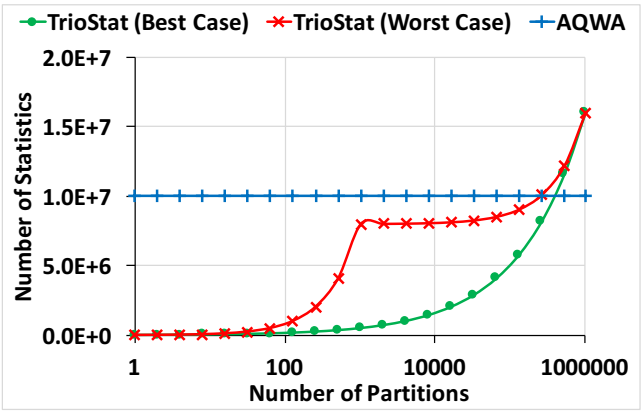

Figure 8: Total Storage for the statistics while varying the number of partitions

identify its partition and collect statistics about it. This demonstrates TrioStat's success in minimizing the added overhead to the processing of each new object. At the end of every repartitioning round, TrioStat needs 58 microseconds on average to update the statistics of all the partitions that an executor machine holds. After this update, TrioStat can estimate the workloads in $O(1)$, as in Section 5.

Figure 7 gives the network overhead of TrioStat's decentralized statistics compared to AQWA's centralized statistics approach [4]. Notice that the results are given in logarithmic scale. AQWA's statistics require maintaining one number per cell to count the data points, and four numbers per cell to count the queries. The four numbers in each cell are required to use the Euler Histogram [7] to count queries in a partition without re-counting queries that overlap multiple cells. Thus, AQWA collects the 5 statistics for every cell in the machine that holds the cell. By the end of every repartitioning round, all the collected statistics should be sent to one machine in order to be combined and used for workload estimation. Figure 7 compares the two approaches by measuring the number of bytes needed to be sent to one of the machines to monitor the performance of the system, compare the workload of all machines, and decide accordingly if repartitioning is needed. TrioStat's decentralized approach outperforms the centralized approach because TrioStat requires sending only two statistics **per executor machine**. The two numbers are the workload estimate of the machine ($W(m)$) and the number of the newly received objects by the machine ($R(m)$). In contrast, the centralized approach requires sending five statistics **per cell** in the system, i.e., five million statistics for the 1000×1000 grid that divides the space. TrioStat will always outperform the centralized approach because each machine can hold at least one cell sized partition, i.e., TrioStat's machines will send 2 statistics per machine while the centralized machines will send 5. However, having every machine holds only one partition with one cell is not practical. Hence, the grid size will be increased and that will increase the amount of statistics that the centralized approach have to send.

TrioStat requires having 8 numbers stored for every row and column of every partition. 5 of them are for the required statistics and 3 for the statistics collectors. Therefore, the storage that TrioStat requires is distributed across the machines according to the distribution of the partitions. On the other hand, AQWA requires storing a total of 10 numbers per cell. 5 of the numbers are used to collect the statistics and they are stored across the machines according to the distribution of the partitions. However, the remaining 5 numbers are all stored in a centralized machine to aggregate the collected statistics and can be used for workload estimation. This results in high storage overhead in one of the machines. Figure 8 gives the results of analyzing the total storage that TrioStat requires for statistics in comparison to AQWA while varying the number of partitions that divide the whole space. The grid that divides the space is 1000×1000 resulting in 1 million cells. Figure 8 gives the number of partitions in logarithmic scale between having 1 partition and 1 million partitions (all partitions are composed of a single cell). The required storage for TrioStat's statistics depends on the number of partitions and their shapes. Since TrioStat maintains statistics for every row and column of every partition, the total storage of TrioStat increases with the increase in the total circumferences of the partitions. Therefore, TrioStat requires the minimum storage when all partitions are squares (Best Case). On the other hand, The maximum storage (Worst Case) happens when the maximum number of partitions is of size 1 X grid side length. There is a fast increase in the worst case of TrioStat in Figure 8 before having 1000 partitions (equivalent to grid side length) because all partitions can be of size 1 X grid side length except one partition. In this worst case scenario, any increase in the number of partitions up to 1000 will result in increasing the total circumferences of the partitions by two times the grid side length. The increase in the worst case slows down after having more than 1000 partitions because any split after having 1000 partitions of size 1 X grid side length results in increasing the total circumferences of the partitions by exactly two. When the number of partitions become very large, each partition will be formed of very few cells. In this case, TrioStat requires higher storage than AQWA. However, it is not practical to have a large number of small partitions in the system. This is because having a large number of small partitions results on duplicating queries in a large number of partitions that needs to communicate to produce queries' final results. Usually, neighboring partitions in the same machines gets combined to reduce the overhead of query execution, which will result on having medium sized partitions. Hence, having

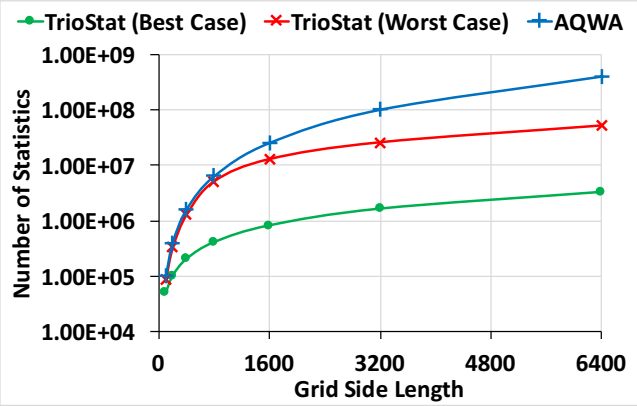

Figure 9: Total Storage for the statistics while varying the grid size

a large number of partitions that is close to the number of cells in the grid may never happen.

Figure 9 gives the results of analyzing the total storage that TrioStat requires for statistics in comparison to AQWA while varying the size of the grid that divides the whole space. The grid side length varies between 100 and 6400 cells. Notice that the number of statistics is represented in logarithmic scale. The number of partitions is fixed to be 1000 partitions. TrioStat outperforms AQWA in all cases. The gap between TrioStat in the best case and the worst case is small when the side length of the grid is less than 1000 (the number of partitions) for the same reason explained in the previous paragraph.

7 CONCLUSIONS

This paper introduces TrioStat, an online workload estimation technique that relies on a probabilistic model for estimating the workload of partitions and machines in a distributed spatial data streaming system. TrioStat introduces a new statistics structure that requires minimal storage overhead. TrioStat uses a decentralised technique to collect and maintain the required statistics in real-time locally in each machine. An efficient algorithm is presented for collecting statistics without adding much processing overhead with the arrival of every new data point or query. TrioStat is tested and compared against AQWA using an application that processes a real dataset from Twitter. TrioStat enables the application to compare the workload of its machines and data partitions with minimal network, storage, and processing overhead. TrioStat requires sharing only two numbers per machine to compare the machines based on their workloads and to monitor the performance of the system.

ACKNOWLEDGMENTS

Walid Aref acknowledges the support of the National Science Foundation under Grant Numbers III-1815796 and IIS-1910216.

REFERENCES

[1] 2020. Apatche Hadoop. http://hadoop.apache.org/.
[2] 2020. Apatche Zookeeper. https://zookeeper.apache.org.
[3] 2020. Internet live stats. https://internetlivestats.com/.
[4] 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. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2062–2073.
[5] Ahmed M Aly, Asmaa Sallam, Bala M Gnanasekaran, Long-Van Nguyen-Dinh, Walid G Aref, Mourad Ouzzani, and Arif Ghafoor. 2012. M3: Stream processing on main-memory mapreduce. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1253–1256.
[6] Ning An, Zhen-Yu Yang, and Anand Sivasubramaniam. 2001. Selectivity estimation for spatial joins. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 368–375.
[7] Richard Beigel and Egemen Tanin. 1998. The geometry of browsing. In *Latin American Symposium on Theoretical Informatics*. Springer, 331–340.
[8] Alberto Belussi and Christos Faloutsos. 1998. Self-spacial join selectivity estimation using fractal concepts. *ACM Transactions on Information Systems (TOIS)* 16, 2 (1998), 161–201.
[9] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. 2018. Detecting skewness of big spatial data in SpatialHadoop. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 432–435.
[10] 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*. 2761–2764.
[11] 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 *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 1095–1106.
[12] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 1352–1363.
[13] Junhua Fang, Rong Zhang, Tom ZJ Fu, Zhenjie Zhang, Aoying Zhou, and Junhua Zhu. 2017. Parallel stream processing against workload skewness and variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 15–26.
[14] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. 1997. Range queries in OLAP data cubes. *ACM SIGMOD Record* 26, 2 (1997), 73–88.
[15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.
[16] 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.
[17] 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 *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 219–228.
[18] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 137–148.
[19] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. 2016. When two choices are not enough: Balancing at scale in distributed stream processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 589–600.
[20] Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. 2001. Flexible data cubes for online aggregation. In *International Conference on Database Theory*. Springer, 159–173.
[21] Anil Shanbhag, Alekh Jindal, Yi Lu, and Samuel Madden. 2016. A moeba: a shape changing storage system for big data. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1569–1572.
[22] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J Elmore. 2017. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 229–241.
[23] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2002. Selectivity estimation for spatial joins with geometric selections. In *International Conference on Extending Database Technology*. Springer, 609–626.
[24] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.
[25] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. *HotCloud* 12 (2012), 10–10.
[26] F. Zhang, H. Chen, and H. Jin. 2019. Simois: A Scalable Distributed Stream Join System with Skewed Workloads. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 176–185.