Adaptive Processing of Spatial-Keyword Data Over a Distributed Streaming Cluster*

Ahmed R. Mahmood[1], Anas Daghistani[1], Ahmed M. Aly[2], Mingjie Tang[1],
Saleh Basalamah[3], Sunil Prabhakar[1], Walid G. Aref[1]
[1]Purdue University [2]Google Inc. [3]Umm Al-Qura University
[1]{amahmoo, tang49, sunil, aref}@cs.purdue.edu, anas@purdue.edu
[2]aaly@google.com [3]smbasalamah@uqu.edu.sa

ABSTRACT

The widespread use of GPS-enabled smartphones along with the popularity of micro-blogging and social networking applications, e.g., Twitter and Facebook, has resulted in the generation of huge streams of geo-tagged textual data. Many applications require real-time processing of these streams. For example, location-based ad-targeting systems enable advertisers to register millions of ads to millions of users based on the users' location and textual profile. Existing streaming systems are either centralized or are not spatial-keyword aware, and hence these systems cannot efficiently support the processing of rapidly arriving spatial-keyword data streams. In this paper, we introduce a *two-layered indexing* scheme for the distributed processing of spatial-keyword data streams. We realize this indexing scheme in Tornado, a distributed spatial-keyword streaming system. The first layer, termed the routing layer, is used to fairly distribute the workload, and furthermore, co-locate the data objects and the corresponding queries at the same processing units. The routing layer uses the Augmented-Grid, a novel structure that is equipped with an efficient search algorithm for distributing the data objects and queries. The second layer, termed the evaluation layer, resides within each processing unit to reduce the processing overhead. The two-layered index adapts to changes in the workload by applying a cost formula that continuously represents the processing overhead at each processing unit. Extensive experimental evaluation using real Twitter data indicates that Tornado achieves high scalability and more than 2x improvement over the baseline approach in terms of the overall system throughput.

CCS CONCEPTS

• **Information systems → Parallel and distributed DBMSs**; **Main memory engines**; **Stream management**; **Spatial-temporal systems**; **Data streaming**;

KEYWORDS

Distributed streaming, Spatial-keyword processing

ACM Reference Format:
Ahmed R. Mahmood, Anas Daghistani, Ahmed M. Aly, Mingjie Tang, Saleh Basalamah, Sunil Prabhakar, Walid G. Aref. 2018. Adaptive Processing of Spatial-Keyword Data Over a Distributed Streaming Cluster. In *26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18), November 6–9, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3274895.3274932

1 INTRODUCTION

Recently, there has been an unprecedented widespread of GPS-enabled smartphones and an increased popularity of micro-blogging and social networking applications, e.g., Twitter, Flickr, and Facebook. In addition, the increased amount of time individuals spend online motivates advertising agencies to store traces of the transactions performed by internet users for further processing and analysis. These online traces often include both spatial and textual attributes. For example, the online trace of a web search includes both the geo-location and the keywords of each search query. This resulted in the generation of massive amounts of rapidly-arriving geo-tagged textual streams, i.e., spatial-keyword data streams. For example, about 600 million tweets and 5 billion Google searches are generated every day [2]. These rapid spatial-keyword streams call for efficient and distributed data processing platforms. Several applications require continuous processing of spatial-keyword data streams (in real-time). One example is location-aware ad-targeting, i.e., publish-subscribe systems [5]. In these systems, millions of users can subscribe for specific promotions, i.e., continuous queries. For example, a user may subscribe for promotions regarding *nearby restaurants and cafes*. Every subscription has a specific spatial range and an associated set of keywords. Each promotion has a spatial location and a textual profile that describes it. An e-coupon is qualified for a user when it is located inside the spatial range of the user's subscription and when the keywords of the user's textual profile overlap the keywords of the user's subscription. In this application, the number of users and e-coupons can be very high.

Despite being in the era of big data, existing systems fall short when processing rapid spatial-keyword streams. These systems belong to one of three categories: (1) *centralized* spatial-keyword systems, e.g., [5], that cannot scale to high arrival rates of data, (2) *distributed* batch-based spatial/spatial-keyword systems, e.g., [6, 21], that have high query-latency (where in some cases, it may require several minutes or even hours to execute a single query), and (3) non-spatial-keyword streaming systems, e.g., [16, 20], that do not have direct support for spatial-keyword queries. This calls for

*This research was supported in part by the National Science Foundation under Grant Number III-1815796.

SIGSPATIAL '18, November 6–9, 2018, Seattle, WA, USA

ACM ISBN 978-1-4503-5889-7/18/11…$15.00
https://doi.org/10.1145/3274895.3274932

distributed spatial-keyword streaming systems that are equipped with efficient spatial-keyword query evaluation algorithms and structures.

In this paper, we describe Tornado [13] a distributed and real-time system for the processing of spatial-keyword data streams. Tornado extends Storm [16]. Storm is a distributed, fault-tolerant, and general-purpose streaming system. Tornado addresses the following challenges:

(1) **Scalability with respect to data and query workload**: Tornado scales to process a large number of data objects per second against a large number of spatial-keyword queries with minimal latency.

(2) **Skew and variability in workload distribution across time**: It is highly unlikely to have a uniform or a fixed distribution of the data or the query workload. Tornado achieves load balancing, and adapts according to changes in the workload (with minimal overhead).

(3) **No downtime**: As Tornado adapts to changes in the workload, it is essential to ensure that Tornado remains functional during the transitioning phase, and that the query results are correct, i.e., no missing or duplicate results.

(4) **Limited network bandwidth**: The underlying network of the computing cluster can easily become a bottleneck under high arrival rates of the data and queries. Tornado minimizes network usage to improve the overall system performance.

To address these challenges, Tornado introduces two main processing layers, namely: 1) the *evaluation layer*, and 2) the *routing layer*.

The Evaluation Layer is composed of multiple evaluators, where each evaluator is assigned a spatial region, i.e., a *Partition* of the space. The entire space is collectively covered by all the partitions with each partition covering a *non-overlapping rectangle*. The evaluation layer uses FAST [5], an efficient spatial-keyword index that has been designed to improve the performance of Tornado.

The Routing Layer distributes data and queries across the processing units, i.e., evaluators. The distribution is location-based, where each evaluator is assigned a spatial region, i.e., a *partition* of the space. One can argue that the distribution of the data and queries can alternatively be text-based. However, text-based distribution is inefficient when compared to location-based distribution. The reason is that a data object, e.g., tweet, has multiple keywords, but only one point location. Text-based distribution may forward a data object to multiple processing units (one per keyword), while space-based distribution forwards a data object to one and only one evaluator. The routing layer employs the *Augmented-Grid* (A-Grid, for short), a novel spatial grid structure. The A-Grid stores the non-overlapping rectangular partitions that are assigned to evaluators. The A-Grid adopts a new algorithm that uses shortcuts to assign data and queries to evaluators. We analytically show that using the A-Grid, the routing time of a query, say q, is $O(N_p)$, where N_p is the number of processing units that are relevant to q. To reduce the network communication overhead, the A-Grid maintains a textual summary of all the query keywords for every evaluator. Before transmitting a data object, say O, to an evaluator, say A, the textual summary of A is checked. If the keywords of A do not overlap the keywords of O, i.e., O does not contribute to the answer of any query, then O is not transmitted. This textual summary is useful when the keywords of queries are very selective, i.e., not popular.

Adaptivity. In Tornado, overloaded evaluators can delay the processing and reduce the overall system throughput. Underutilized evaluators waste processing resources. Hence, Tornado maintains a balanced distribution of the workload across all the evaluators. It is expected that the system workload will not be the same at all times, and hence having a static routing layer can result in poor system performance. Existing systems, e.g.,[6], address the problem of adaptive workload-aware processing of big data by providing mechanisms for updating the partitioning the data. These systems keep centralized workload statistics, and halt the processing of the data and queries during the re-partitioning phase. However, in distributed real-time applications, workload statistics are distributed across evaluators and it is unacceptable to pause the query processing. This calls for a real-time load-balancing technique that does not interrupt the query processing. It is challenging to implement such a distributed and real-time load-balancing mechanism in Tornado for the following reasons:

- **No Global System View:** In Tornado, the workload statistics are distributed across evaluators. Sending detailed workload statistics from one process to another requires high network overhead. The load-balancing protocol should minimize the overhead needed to collect, transfer, and process the workload statistics.
- **Correctness of Evaluation:** during the re-partitioning phase, Tornado redefines the boundaries of the evaluators. This requires moving queries from one evaluator to another. Meanwhile, the data objects continuously update their locations, and the answer to each query needs to be continuously updated as well. Hence, unless the incoming data objects are carefully directed, missing (or duplicate) results can occur.
- **Overhead of Re-partitioning:** Moving the queries between the evaluators incurs network overhead. The re-balancing algorithm should be aware of the re-balancing overhead, and avoid unnecessary re-balancing.

Tornado employs a load-balancing mechanism, where the choice of the new spatial boundaries of the evaluators is mostly delegated to the evaluators themselves. This reduces communication overhead needed to transfer detailed workload statistics and distributes the computational overhead across the evaluators. The load-balancing mechanism is incremental, i.e., rather than redefining all the partitions, only a few partitions are updated using simple shift, split, and merge operations. Furthermore, Tornado ensures the correctness of evaluation during the transient phase using a two-stage re-partitioning protocol. In summary, the contributions of this paper are as follows:

- We present the structure of Tornado, a scalable spatial-keyword data streaming system that uses an efficient two-layered indexing scheme.
- We develop an *Augmented-Grid* structure and an optimal *neighbor-based routing* algorithm that minimizes the overhead of routing the data and queries.
- We present an *incremental* and *adaptive*load-balancing mechanism that ensures fairness in the workload distribution across the evaluators.

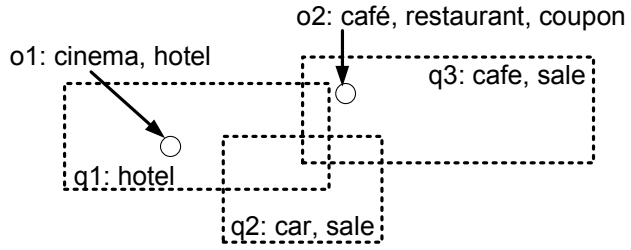


Figure 1: E-coupon example .

- Using real datasets from Twitter, we show that Tornado achieves 2x performance gain in comparison to a baseline approach.

The rest of this paper proceeds as follows. Section 2 presents the notations used throughout the paper. Section 3 describes the structure of Tornado. Section 4 describes the load balancing mechanism in Tornado. Detailed experimental evaluation is given in Section 5. The related work is presented in Section 6. Section 7 contains concluding remarks.

2 PRELIMINARIES

In this section, we present the notations that are used throughout the paper. A spatial-keyword data stream is an unbounded sequence of spatial-keyword objects. A spatial-keyword object, say O, has the following format: $O = [oid, loc, text, ts]$, where *oid* is the object identifier, *loc* is the geo-location of the object at Timestamp ts, and $text$ is the set of keywords associated with the object. We use Tornado to answer the spatial-keyword filter query defined as follows.

A continuous spatial-keyword filter query, say q, is defined as $q = [qid, MBR, text, t]$, where *qid* is the query identifier, *MBR* is minimum bounding rectangle representing the spatial range of the query, and $text$ is the set of keywords of the query. The continuous query q is registered, i.e., keeps running for a specific duration, say t. During t, the query continuously reports the data objects that satisfy the query's spatial and textual predicates. To ***match*** a query, a data object needs to be located inside the spatial range of the query, and needs to contain all query keywords.

Figure 1 gives an example of multiple spatial-keyword filter queries from an e-coupon application. In Figure 1, three subscriptions, i.e., queries, q_1, q_2, and q_3 are registered in the system. E-coupon o_1 qualifies for subscription q_1 because it is located inside the spatial range of q_1 and the textual content of o_1, i.e., "cinema, hotel" contains the keywords of q_1, i.e., "hotel".

3 TORNADO SYSTEM ARCHITECTURE

In this section, we present the architecture of Tornado and its main components alongside with query processing algorithms. Tornado [13] extends Storm [16]. Storm is a cluster-based, distributed, fault-tolerant, and general-purpose streaming system that achieves real-time processing with high throughput and low latency. Storm provides three abstractions, namely: *spout*, *bolt*, and *topology*. A spout is a source of input data streams. A bolt is a data processing unit. A topology is a directed graph of bolts and spouts that resembles a pipeline of streamed data evaluation. Storm is not optimized for the execution of spatial-keyword queries, simply because it does not have built-in support for spatial or textual primitives, e.g., points, rectangles, or containment of keyword lists.

In order to efficiently support the evaluation of spatial-keyword queries, we need to guarantee that relevant data and queries are collocated in the same processing unit, i.e., a Storm bolt. This is challenging because the system needs to distribute data and queries across processing units in a way that achieves the following properties: (1) **Optimize the memory usage** across the machines by not storing queries in multiple processing units, (2) **Optimize the CPU usage** by checking each data object against as few queries as possible, and (3) **Maintain good load balancing as the workload changes**, and distribute the data and queries across the processing units while guaranteeing the correctness of evaluation, i.e., without missing output tuples and without producing duplicate results.

Tornado extends the bolt abstraction from Storm into *routing units* and *evaluators*. The routing units are light-weight components that are responsible for co-locating the queries and data objects together. The evaluators are processing units that check the incoming data objects against the continuous queries and produce query results.

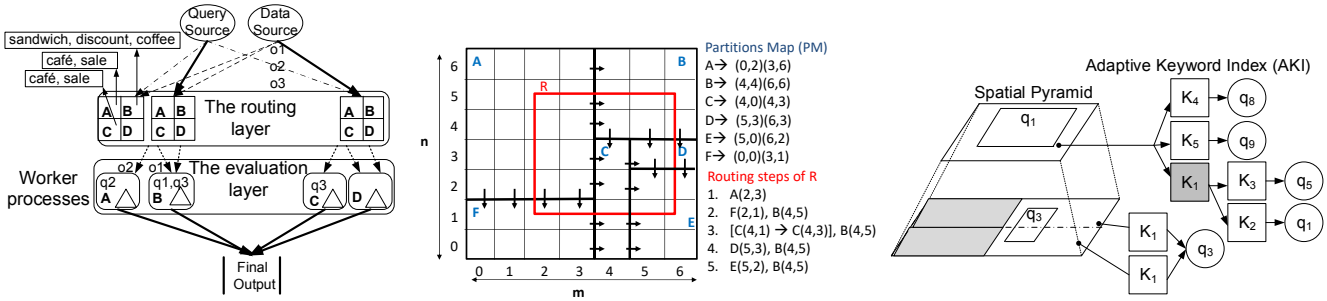
Tornado makes use of the fact that a data object has a single point location, but multiple keywords. This is typical in many location services, e.g., as in tweets, where a tweet is associated with a single location and multiple keywords. Accordingly, the routing layer in Tornado partitions the space into non-overlapping MBRs. Every evaluator is responsible for a single MBR. The benefit of having non-overlapping MBRs is to optimize the network utilization by forwarding each data object to a single evaluator.

To support high arrival rates of streamed data, the routing layer applies replication, i.e., multiple identical routing units are employed. The routing layer maintains a textual summary for every evaluator. The textual summary of an evaluator, say E, contains all keywords of queries stored in E. In the routing units, the textual summary for an evaluator is stored as a **hash set** of keywords.

Before forwarding a data object, say O, to an evaluator, say E, the textual summary of E is consulted to check if there are some queries in E that have keywords that overlap the keywords of O. Figure 2(a) illustrates how Tornado processes spatial-keyword queries. Once a query is received, a routing unit is selected at random, and the query is forwarded to that routing unit, where the latter sends the query to the spatially relevant evaluator(s). Based on the textual summary of the evaluators, stored on the routing layer, some data objects are not forwarded to any evaluator, e.g., o_3 in Figure 2(a).

3.1 The Routing Units: The Augmented-Grid (A-Grid)

The routing layer is composed of multiple identical routing units. Routing units are used to *store and index the non-overlapping spatial partitions of evaluators*. Recall that every evaluator, i.e., worker process, is responsible for a specific spatial range. In real applications, the size of the distributed cluster and the number of worker processes is large, e.g., Yahoo has about 40000 machines running



(a) The routing units and evaluators. (b) Neighbor-based query routing in Tornado. (c) FAST local index in Tornado.

Figure 2: The architecture and system components of Tornado.

Hadoop with a total of 100000 CPU cores [4]. This calls for an efficient index to store the partitions of evaluators.

An incoming data object or query goes to a random instance of the routing units to be dispatched to the corresponding evaluator(s). The smaller the routing time, the higher the throughput of the entire system. Moreover, having light-weight routing units can save more resources that can be used for query evaluation rather than for routing. In Tornado, the location of a data object is represented as a single point in space. Because the partitions are non-overlapping, a data object is routed to a single evaluator. This routing is achieved in $O(1)$ using uniform grid partitioning. However, a query has a spatial range, that may overlap with multiple partitions, and hence a query needs to be routed to multiple evaluators.

Definition 1. ***The Routing Problem*** *Given a rectangular query-range, say r, and a set, say S, of N_e non-overlapping rectangular partitions that cover the entire space, find the partitions that overlap r.*

We propose the *A-Grid* and the *Neighbor-Based Routing*, a routing technique that requires $O(N_p)$ operations to route a spatial range, where N_p is the number of evaluators that overlap the spatial range. This is lower than the time needed in both the traditional grid, i.e., $O(m \times n)$ and hierarchical structures, i.e., $O(\log(N_e) + N_p)$.

The A-Grid stores the non-overlapping rectangular partitions that are assigned to evaluators. The entire space is partitioned into N_e non-overlapping spatial partitions. Initially, the A-Grid partitions the entire space into a virtual fine grid *FG*. Then, the N_e partitions are overlaid on top of *FG*. Each partition, say *p*, corresponds to one evaluator, and is defined as follows: $[pid, xcellmin, ycellmin, xcellmax, ycellmax]$, where *pid* is the identifier of the partition, *xcellmin* and *ycellmin* define bottom left grid cell of *p*, *xcellmax* and *ycellmax* define the top right grid cell of *p*.

The main idea of neighbor-based search algorithm is to follow shortcuts to jump directly from *dominant cells* belonging *neighboring* partition.

Definition 2. ***Dominant cell*** *A dominant cell of a partition, say A, with respect to a spatial range, say R, is the top left cell of A that is inside R. Dominant cells are the only cells that need to be visited while searching the A-Grid.*

For example in Figure 2(b), the dominant cell of the Partition *A* with respect the spatial range *R* is (2,5). Observe that each grid cell is spatially contained inside the spatial range of a single evaluator. Boundaries of partitions are maintained an a hash table termed the *Partitions Map*, PM for short as illustrated in Figure 2(b).

Each grid cell, say *c*, maintains the identifier of the partition that contains *c*. To find the right dominant cell with respect to a range *R* we follow the following steps:(1) find the right cell *RC* belonging to a different partition, and (2) find the dominant cell of *RC* that is the top-left of *RC* belonging to the same partition and inside the spatial range *R*.

Refer to Figure 2(b) for illustration. Assume that we need to identify the right dominant cell, say *RC* of Cell $(2, 5)$ within Partition *A*. From the PM we know that the partition *A* spans cells $[(0, 2), (3, 6)]$. The right cell *RC* of the cell $(2, 5)$ is of the form (xp, yp), where xp is the index on the horizontal coordinate that is to the right of cell $(2, 5)$. The yp is 5 because the Cell *RC* is to the right of Partition *A* and has the same position on the vertical axis. From the PM, the partition *A* ranges from 0 to 3 on the horizontal coordinate, where 0 and 3 are $xmin$ and $xmax$ of the Partition *A* respectively, the value xp is equal to 4 that is $1 + xmax$. This means that the Cell *RC* is $(4, 5)$ that is covered by Partition *B*. The dominant cell of $(4, 5)$ is also $(4, 5)$ as this is the top-left cell within *R*. The same logic applies when finding the bottom dominant cell.

To route a spatial range, say *R*, we start from the upper-left corner of *R*. We find the partition that is covered by that corner (this is trivial because the partition identifier is stored in the cell corresponding to that corner). Then, we follow the right and bottom dominant shortcuts of that corner.

We recursively apply this procedure until we reach a cell from which the pointers lead to a cell that is outside *R* or to a previously visited partition. We use a Boolean array to mark the visited partitions and avoid visiting the same partition more than once.

Refer to Figure 2(b) for illustration. To route the red rectangle, we start from Cell $(2, 5)$ covered by Partitions *A*. Then, we follow the pointers to Cells $(4, 5)$ and $(2, 1)$, covering Partitions *F* and *B*, respectively. Then, we follow the bottom pointer of Cell $(2, 1)$ to reach Cell $(4, 1)$ inside Partition *C*. From the PM, we identify that Cell $(4, 3)$ is the dominant cell of the Partition *C* with respect to *R*. We follow dominant cell shortcuts visiting the following cells: Cell $(5, 3)$ within Partition *D*, Cell $(5, 2)$ within Partition *E*, and Cell

(4, 5) within Partition B. The Pseudocode of the algorithm is given in Algorithm 1.

Algorithm 1: *neighborSearch*(MBR r)

```
1  Stack S
2  Cell c(x,y)← TopLeft corner of r
3  S.push(c)
4  while S not empty do
5      c←S.pop
6      if c overlaps r and c.partition is not visited then
7          add c.partition to result
8          mark c.partition as visited
9          rightCell = getDom(getRightCell(c.y))
10         bottomCell = getDom(getBottomCell(c.x))
11         S.push(bottomCell),S.push(rightCell)
12     end
13 end
```

Lemma 1. ***The neighbor-based routing requires*** $O(N_p)$ ***and does not depend on the granularity of the grid***
The interconnection between dominant cells in the A-Grid can be abstracted as a hypothetical Directed Acyclic Graph (DAG). For the traversal performed by the neighbor-based search algorithm, the number of nodes V in the hypothetical DAG is N_P. The number of the edges E visited is $2N_P$ because for every node, we follow at most two pointers. The total traversal time is $O(V + E) = O(N_p)$. The run time of the algorithm cannot be less than $O(N_p)$ as this is the size of the output.

The routing units maintains a summary of query keywords within each evaluator. As described in Section 2, to match a data object say o with a query say q, the keywords of O need to contain all the keywords of q. Hence, it is sufficient to store *only a single keyword* from q in the textual summary of the evaluator corresponding to q. A data object is spatially assigned to only one evaluator. If the keywords of the object contain any of the evaluator's textual summary keywords, the object is forwarded to that evaluator.

Notice that Tornado maintains multiple identical routing units. One way to keep track of the query keywords within each evaluator is to *broadcast* each query to all the routing units. To avoid unnecessary communication, an incoming query, say q, goes to an arbitrary instance of the routing units, say U. If q adds new keywords to any evaluator, say E, then U *forwards* the added keywords to the other replicas of the routing units with negligible latency. As queries expire, the textual summary of the evaluators may contain redundant keywords. Evaluators periodically *inform routing units with expired keywords* to allow routing units to remove redundant keywords.

3.2 Evaluators

To improve the overall system performance, each evaluator maintains a local spatial-keyword index. Evaluators in Tornado use FAST [5], an efficient spatial-keyword index that requires minimal memory overhead that has been designed to improve the scalability of Tornado. FAST integrates the spatial pyramid [7] with a new textual index termed the adaptive-keyword-index (AKI). The spatial pyramid is a multi-resolution spatial index. AKI is a hybrid textual index that integrates the trie structure [12] with inverted lists [22] and uses the frequencies of keywords to improve the indexing and searching performance. FAST uses multiple optimizations to reduce its overall memory overhead, e.g., avoid query replication by sharing lists of queries among neighboring pyramid cells. Figure 2(c) illustrates the structure of FAST. The main responsibilities of an evaluator are as follows:
(1) Store and index continuous queries and drop expired queries.
(2) Process incoming data objects against stored queries.
(3) Keep track of usage and workload statistics.

Continuous queries are persisted in an evaluator by indexing them in the local instance of FAST that is maintained in the evaluator. To process an incoming data object, say O, we search FAST for matching continuous queries. The keywords of matching queries need to be fully contained in the keywords of O. Also, the location of O needs to be inside the spatial range of a matching query. Incoming data objects are evicted as soon as they are matched against the indexed continuous queries.

4 REAL-TIME LOAD BALANCING

In Tornado, each evaluator is responsible for a certain spatial range that covers a partition in the fine grid FG. To achieve high throughput, Tornado keeps a balanced distribution of the workload across the evaluators. To compute the workload corresponding to an evaluator, Tornado keeps workload statistics at the same granularity of FG. For each data object, say O_l, that is received by $FG[i][j]$, where i and j are the horizontal and vertical coordinates of the Cell $FG[i][j]$, respectively, let q_l be the number of queries that contain any of the keywords of O_l.

For each grid cell $FG[i][j]$, we define the workload overhead, i.e., the computational cost, as the sum of q_l over all the data objects O_l received by that cell:

$$cost(FG[i][j]) = \sum_{l} q_l \tag{1}$$

Given a partition, say P_w, that is bounded by $[(xmin,\ ymin),\ (xmax,\ ymax)]$, the overall computational cost is the sum of the costs of all the grid cells in P, i.e.,

$$cost(P_w) = \sum cost(FG[i][j]) \tag{2}$$

where $xmin \leq i \leq xmax$ and $ymin \leq j \leq ymax$.

Below, we describe the load-balancing protocol in Tornado.

4.1 Initialization

Tornado partitions the entire space into N_e partitions, where N_e is the number of evaluators. To choose the initial boundaries of the partitions, Tornado uses a sample of the data and query workload, and calculates the computational cost of each fine grid cell. Let α be the maximum computational cost of the partition P_w, i.e.,

$$\alpha = \max_{P_w}(cost(P_w)) \tag{3}$$

In the initialization phase, the objective is to minimize α across all the N_e partitions. The best-case distribution is to have all evaluators process equal portions of the workload. The problem of finding the optimal rectangular partitioning that minimizes α is

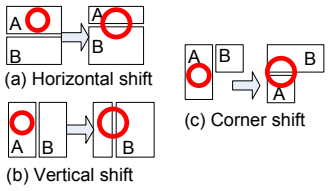
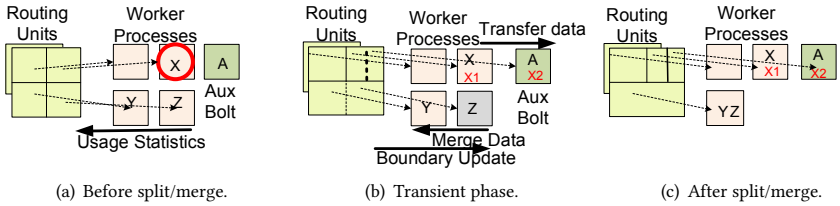


Figure 3: Shift variations.



(a) Before split/merge. (b) Transient phase. (c) After split/merge.

Figure 4: The split/merge operation.

NP-Hard (see [10]). Tornado employs a hierarchical recursive space decomposition similar to that of a k-d tree decomposition [6, 14]. In particular, Tornado maintains a priority queue of the partitions to be split, where the partitions are sorted according to their cost. First, the entire space represents a single partition that is inserted into the priority queue. Then, the top partition from the queue, i.e, the one with the highest cost, is retrieved, and then is split into two partitions. The split is chosen in a way that minimizes the maximum cost of the resulting two sub-partitions. Then, the resulting sub-partitions are inserted into the priority queue. This process is repeated until a single grid cell is reached (that cannot be split), or the maximum allowed number of evaluators in the system is reached. The maximum number of evaluators is a system parameter that depends on the number of CPU cores in the cluster.

4.2 Adaptivity in Tornado

Due to limited cluster resources, it is important to preserve fairness in workload distribution while keeping the number of evaluators fixed. Tornado uses two *incremental* load-balancing operations, namely: (1) *shift* and (2) *split/merge*.

A ***shift*** operation involves a transfer of the workload, i.e., fine grid cells, from an overloaded evaluator to an underutilized spatially adjacent evaluator. Shifting to a neighbor evaluator is meant to prevent excessively fragmenting the indexed queries into multiple evaluators, hence reducing the overall memory requirements. The objective is to store queries in the fewest evaluators possible while maintaining load balancing. Tornado uses three variants of the shift operation, namely: *horizontal*, *vertical* and *corner* shifts. Refer to Figure 3 for illustration. The red circle in the figure represents an area with a high workload. A horizontal shift is applicable to two evaluators that share a horizontal boundary, e.g., see Figure 3(a). Similarly, a vertical shift is applicable to two evaluators that share a vertical boundary, e.g., see Figure 3(b). A corner shift is applicable when two neighboring evaluators form a corner shape, e.g., see Figure 3(c). The corner shift allows a transfer of workload between two non-mergeable evaluators, i.e., ones that do not share an entire horizontal or vertical boundary. The details for finding the best point to shift are described in Section 4.3.

A ***split/merge*** operation involves a split of an overloaded evaluator into two evaluators, followed by a merger of two neighboring underutilized evaluators into a single evaluator. The split is either horizontal or vertical. The split position is chosen to minimize the difference in cost between the resulting two partitions. The details of finding the best point to split an evaluator are given in Section 4.3. During a split, Tornado transfers some grid cells from an overloaded evaluator to an auxiliary evaluator. Refer to Figure 4 for illustration. Figure 4(a) illustrates an overloaded evaluator X before a split/merge operation. An instance of the routing units makes a decision to split/merge and initiates a split of Evaluator X into X_1 and X_2, and a merge of Evaluators Y and Z, as in Figure 4(b). Observe that, according to the new boundaries, some of the fine grid cells are being transmitted from evaluator X to an auxiliary evaluator A. All the fine grid cells that are stored in Evaluator Z are transferred to Evaluator Y. Figure 4(c) gives the state at the end of the split/merge operation.

The decision of whether to initiate a rebalancing operation or not depends on two factors, namely, the *cost reduction* C_r resulting from the re-balance operation, and the *cell transfer overhead* C_t involved in the re-balance operation. **The cost reduction** C_r of a re-balance operation is the difference between the maximum partition cost before and after the re-balance operation. Consider the split/merge operation in Figure 4, and assume that Evaluator X has the highest cost. The cost before split/merge=$cost(X)$. The cost after split/merge is $max(cost(X_1), cost(X_2), (cost(Y)+cost(Z)))$. The cost reduction of the split/merge operation is:

$$C_r(split/merge, X, X_1, X_2, Y, Z) = \\ cost(X) - max(cost(X_1), cost(X_2), (cost(Y) + cost(Z))) \quad (4)$$

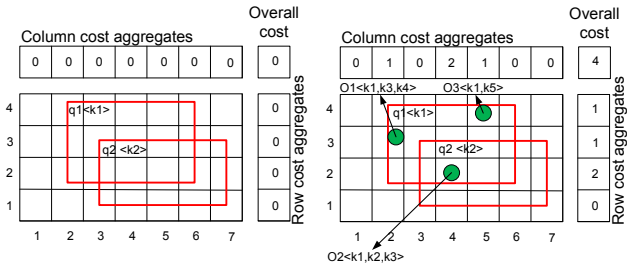
The above idea applies to the shift operation, where the cost reduction is computed as the difference between the maximum cost before and after the shift operation. **The cell transfer overhead** C_t is an estimate of the overhead of transferring cells during the re-balance operation. $C_t(p) = \beta \times queryCount(p)$, where $queryCount(p)$ is the number of queries in Partition p, and β is the average time needed to transfer a query. $queryCount(p)$ is incremented whenever a query is registered at p, and is decremented whenever a query in p expires. For example, for the split/merge operation in Figure 4, the cell transfer overhead of the split/merge operation is calculated as follows:

$$C_t(split/merge, X, X_1, X_2, Y, Z) = \\ \beta \times (queryCount(X_2) + queryCount(Z)) \quad (5)$$

Tornado chooses the operation that maximize that value of C_r while having $C_r > C_t$.

4.3 Distributed Load-Balancing

Existing load-balancing approaches are centralized [6], i.e., require having a single unit that receives all the workload statistics. In contrast, in Tornado, the computation of the costs of the fine grid cells is distributed across the evaluators as follows. (1) *The evaluators keep detailed workload statistics* and choose the split coordinates



(a) Initial evaluator statistics. (b) After data objects arrive.

Figure 5: Cost aggregation within an evaluator.

that are needed to perform the shift and split/merge operations and (2) *The routing layer periodically receives a summary of the workload statistics from the evaluators*, and then makes a decision as to whether to change the partitioning or not.

For the routing layer to make a decision whether to re-balance or not, it does not need the detailed costs within every evaluator. The decision to rebalance can be made using the overall evaluator costs from Equation 2.

Tornado keeps three aggregates at each evaluator, namely, row, column, and overall aggregates. Refer to Figure 5 for illustration. Figure 5(a) gives the initial values of these aggregates. Figure 5(b) gives the values after processing three data objects O_1, O_2, and O_3. O_1 satisfies one query at Cell $(2, 3)$, and hence the aggregates of Row 3 and Column 2 are incremented. O_2 satisfies two queries at Cell $(4, 2)$, and hence the aggregates of Row 2 and Column 4 increase by 2. O_3, satisfies one query at Cell $(4, 4)$, and hence the aggregates of Row 4 and Column 4 are incremented. The overall cost of the evaluator gets the value of 4. Maintaining these aggregates requires $O(1)$ processing time per data object. Tornado maintains similar row, column and overall aggregates for the number of queries within grid cells.

For the split/merge operation, to maximize the cost reduction resulting from splitting a partition, say X, into X_1 and X_2, Tornado tries to minimize the value of $|cost(X_1) - cost(X_2)|$ by trying all possible vertical and horizontal splits. If Equation 2 is applied directly, it requires $O(m \times n)$ to find the best split. Instead, Tornado uses the row and column aggregates to find the best split in $O(m + n)$. In particular, Tornado scans the column aggregates and keeps a sum of the scanned aggregates, say S_a. Initially, $S_a = 0$, and keeps accumulating values from the column aggregates as long as S_a is less than half the overall cost of the evaluator, say (O_{half}). If S_a is equal to (O_{half}), no more aggregates are scanned. If S_a is greater than (O_{half}), then the split position is marked, and the same process is repeated, but with the row aggregates. The split position that minimizes the value of $|cost(X_1) - cost(X_2)|$ is chosen.

For example, in Figure 5(b), the best vertical split is between Columns 3 and 4, with a difference of 3 in cost. However, the best horizontal split is between Rows 2 and 3, with a difference of 0 in cost. Hence, the horizontal split is chosen.

For the shift operation, we need to distinguish between a corner shift and a horizontal/vertical shift. In the corner shift in Figure 3(c), there are no multiple choices for the shift coordinate in A. The corner shift coordinate depends on the position of B relative to A. This allows A to identify the cost of the cells involved in any shift operation as well as the cell transfer overhead. Notice that there are at most 8 possible corner shifts for any given evaluator. However, there is no fixed coordinate for the horizontal/vertical shift in A. The reason is that the optimal coordinate for a horizontal/vertical shift depends on the cost of B that is unknown to A. To address this issue, Tornado delays the choice of the best shift coordinate in A until the routing unit makes a decision to perform a horizontal/vertical shift.

At the time when the routing unit makes a decision as to whether to re-balance or not, it has accurate statistics for both the split/merge and the corner shift operations. The routing unit does not know the exact cost reduction and cell transfer overhead of horizontal/vertical shift operations. The routing unit estimates that an optimal horizontal/vertical shift from evaluator A to evaluator B results in an optimal division of workload between A and B. Thus, the estimated cost reduction is computed as $cost(A) - \frac{cost(A)+cost(B)}{2}$. Assuming uniform query distribution in A, the routing unit estimates the cell transfer overhead to be proportional to the amount of workload transferred, i.e., $\beta \times queryCount(A) \times \frac{cost(A) - \frac{cost(A)+cost(B)}{2}}{cost(A)}$. Then, the routing unit chooses the re-balancing operation if necessary. If the re-balancing operation is a horizontal/vertical shift, then the routing unit *informs the evaluators involved in this horizontal/vertical shift operation* with the costs necessary to make an optimal shift operation similar to finding the optimal split described previously.

4.4 Correctness during Load-balancing

A rebalancing operation affects both the routing and the evaluation layers. In the routing layer, the partitioning of the evaluators changes according to the rebalancing operation. In the evaluators, index cells and queries move from one evaluator, say E_1, to another evaluator, say E_2. It is challenging to guarantee the correctness during the re-balancing process because data objects and queries arrive during re-balancing and Tornado cannot afford to halt the processing until the entire re-balancing is done.

An important question to address is *which evaluator should receive the incoming data objects and queries during the transient phase? E_1*, or E_2, or both? Tornado splits the transient phase into two steps. In every step, we define a set of rules that guarantee correct processing in that phase. The steps of the transient phase are: (1) *Index cells transfer phase* during which queries from index cells are moved across evaluators, and (2) *Routing unit update phase* during which routing units update their partitioning according to the adaptivity operation.

Processing during the index cells transfer phase During the cell transfer phase, all incoming data and queries will be routed to E_1 because all routing units use the partitioning before re-balancing. Incoming queries to the area to be shifted are processed according to the following steps:

(1) All incoming queries are processed and indexed in E_1

(2) If a query arrives at a transmitted cell, forward the query to E_2

Incoming data objects are processed in Evaluator E_1.

Processing during the routing update phase, Due to network delays, it is not possible that all routing units update their partitioning instantaneously. This means that even after the *cell transfer phase*, some routing units may send data and queries to E_1 while

Table 1: The values of the parameters used in the experimental evaluation.

Parameter	Value
Number of routing units	1, 3, 5, 7, **10**, 12
Number of queries (million)	5, **10**, 20, 30, 40
Number of query keywords	1, 2, **3**, 5, 7
Spatial side length of a query	.01%,.05%,.1%,**.5**%,1%,1.5%

others send data and queries to E_2. To address this issue, we adopt the following approach during the routing update phase: *any data object or query that is routed to a* ***transmitted cell*** *in* E_1 *is neither processed nor indexed in* E_1 *and is instantaneously forwarded to* E_2.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of Tornado. Our experiments are conducted on a cluster of Dell r720xd servers that have a total of 48 TB of local storage, and a 40 Gigabit Ethernet interconnect. The cluster runs 5 virtual machines where each virtual machine has 16 cores and 32 GB of memory. Each virtual machine runs Storm 1.0.0 over Centos Linux 6.5. We evaluate the performance of Tornado using real datasets and a synthetic query workload. We use a real dataset from Twitter that is composed of 1 billion tweets with geo-locations inside the US and of size 140 GB. These tweets are collected from January 2014 to March 2015. The format of the tweet is "id, geo-location, text". We use these tweets to simulate a *continuous* and *infinite* stream of spatio-textual objects such that when all the tweets are streamed, we restart streaming the tweets from the beginning.

We use three query datasets, namely; (1) *normal tweets*, (2) *spatially-condensed*, and (3) *textually-selective*. The *normal tweets* dataset uses the locations and keywords of the tweets as the locations and the keywords of the query. The *spatially-condensed* dataset is used to study the effectiveness of load-balancing techniques by shrinking, i.e., scaling down, the spatial area covered by the dataset into a smaller range. Hence, creating load imbalance across evaluators that requires the adaptivity protocol to redistribute the workload. The *textually-selective* dataset chooses the keywords of queries based on how frequent the keywords are. To build this dataset, all the keywords of tweets are sorted based on their frequencies. Then the keywords of queries are randomly chosen from the k^{th} percentile frequent keywords, where k is the keyword frequency threshold. For example, setting k to 90%, does not include the 10% most frequent data objects keywords into query keywords.

Table 1 summarizes the values of the parameters we use. We set the default number of query keywords to 3, which resembles the average number of keywords in web searches [3]. The default spatial range of queries is .5% of the entire spatial range. For the A-Grid, we use a 1000×1000 granularity.

5.1 Performance of Tornado

In this experiment, we measure the performance of the following processing alternatives. (1) **Tornado (FAST)**, where the A-Grid is used as the routing structure and FAST is used as the local spatial-keyword inside the evaluators. (2) **GI^2**, where the A-Grid is used as the routing structure and GI^2 [17] is used as the local index inside the evaluators. The Grid Inverted Index, i.e., GI^2, is a spatial-keyword structure that is used in the PS2Stream [9] distributed publish-subscribe streaming system. GI^2 indexes spatial-keyword queries over a spatial grid where every cell of the spatial grid has an inverted list to queries. (3) **Text-Rout**, where the routing units use keywords of data objects and queries to hash and route data objects and queries to evaluators, i.e., every evaluator is assigned a set of keywords. In **Text-Rout**, FAST is used as the local spatial-keyword index inside evaluators. (4) **Uni-Space-Rout**, where the partitions assigned to evaluators span equal and non-overlapping spatial ranges. These spatial ranges are derived from a uniform spatial grid partitioning of the entire space regardless of the spatial and textual distribution of underlying workload. Also, in **Uni-Space-Rout**, FAST is used as the local index.

Figures 6 (a) and (b) show that using the **Tornado (FAST)** achieves the highest throughput and the least processing latency. This is due to the efficiency of both A-Grid and FAST. **Tornado (FAST)** achieves more than 2X improvement in the overall throughput than other processing alternatives. The reason is that the A-Grid ensures fair workload distribution to evaluators with minimal routing overhead. Also, FAST [5] ensures efficient indexing and searching performance with low memory overhead.

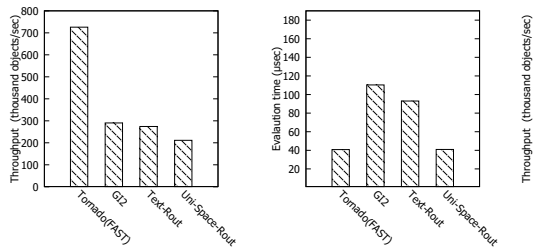
GI^2 results in low throughput and high execution latency because of the underlying local GI^2 index. GI^2 suffers from a high memory overhead due to the replication of queries over spatial grid cells. Also, the inverted lists inside the spatial grid cells of GI^2 do not provide high textual discrimination abilities [12]. This leads to poor searching performance, low overall system throughput, and high execution latency.

Text-Rout suffers from poor performance because the text-based routing replicates data objects and queries to multiple evaluators. For example, assume that Evaluator $E1$ is responsible for Keyword $k1$ and Evaluator $E2$ is responsible for Keyword $k2$. Any incoming data object containing Keywords $k1$ and $k2$ will be replicated to both Evaluators $E1$ and $E2$. This creates a bottleneck in the network bandwidth and reduces the overall throughput and results in having a single data object being processed in more than one evaluator.

The uniform spatial routing, i.e., **Uni-Space-Rout**, results in a throughput that is 2 times lower than that of **Tornado (FAST)**. The reason is that using uniform spatial partitioning does not account for the skewed spatial distribution of data objects and queries and results in an unfair workload distribution across evaluators. This significantly reduces the overall performance due to workload imbalance.

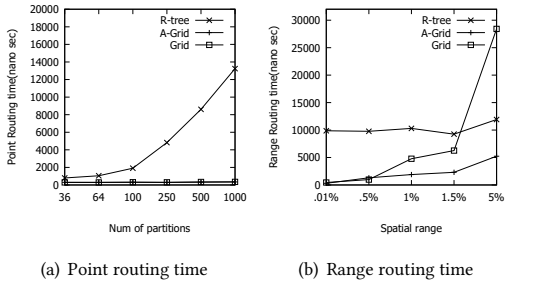
Also, we measured the performance of a native Storm implementation that replicates all queries to all evaluators and does not use any internal spatial keyword indexing. This native storm implementation resulted in an extremely low throughput, i.e., less than one thousand objects per second, that is not comparable with Tornado.

Figure 6(c) demonstrates the effectiveness of spatial-keyword routing against spatial-only routing using the *textually-selective* dataset. In this experiment, we vary the frequency of query keywords from 0%, i.e., least frequent keywords that do not match any of the keywords of data objects, to 100%, i.e., query keywords include all popular keywords and follow the same distribution as



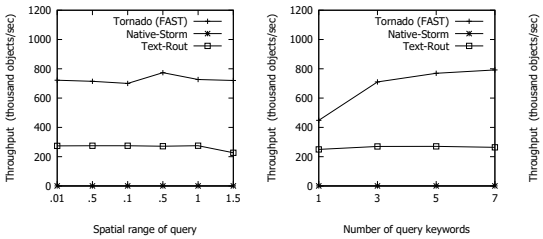
(a) Throughput. (b) Evaluation latency. (c) Effect of keyword frequency.

Figure 6: The performance of routing alternatives.



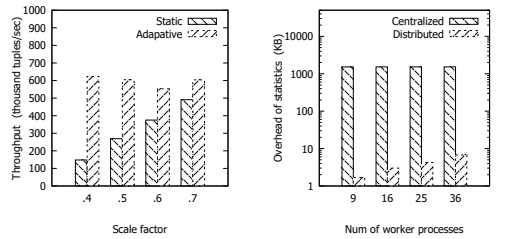
(a) Point routing time (b) Range routing time

Figure 7: Spatial routing time for points and ranges.



(a) Spatial range effect. (b) Number of keywords effect. (c) Number of queries effect.

Figure 8: Scalability.



(a) Hotspots Throughput (b) Statistics overhead.

Figure 9: Adaptivity.

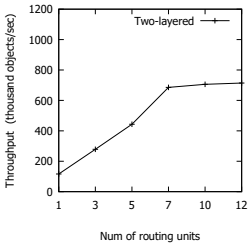


Figure 10: Number of routing units effect.

the keywords of data objects. Figure 6(c) illustrates that, as the keyword frequency percentile of queries decreases, the overall system throughput increases. The reason is that, as the frequency percentile of query keyword decreases, the number of data objects with keywords overlapping with the textual summaries in the A-Grid decreases. This results in having fewer data objects being forwarded to evaluators and hence a reduction of both the computational overhead in the evaluators and the communication overhead between the routing units and the evaluators.

5.2 Performance of Routing Layer

In Figure 7, we contrast the performance of the **A-Grid** against the performance of traditional uniform **Grid** and the R-tree. Figure 7(a) gives the routing times for data points while increasing the number of partitions. As the number of partitions increases, the routing time of the data points increases remains constant for both the Grid and the A-Grid and increases for the R-tree index. Although the Grid and the A-Grid have similar performance for point routing, Figure 7(b) shows that the A-Grid outperforms the Grid and R-tree for range routing as we increase the spatial range of queries from .1% to 5% of the entire spatial range.

In Figure 10, we study the effect of the number of routing units on the overall system throughput. Figure 10 gives the throughput when increasing the number of routing units. If there is only one routing instance, then the routing layer becomes a bottleneck. As we increase the number of routing instances, the system throughput increases. The increase in throughput saturates after 10 routing instances. After that, the bottleneck moves from the routing layer to the evaluation layer.

5.3 Scalability

In this experiment, we study the scalability of Tornado under various query workloads. In Figure 8(a), we vary the spatial range of the queries from .01% to 1.5% of the maximum spatial range. Figure 8(a) illustrates that Tornado is scalable and that the system throughput is stable and is not affected by the increase in the spatial extent of the query. In Figure 8(b), we increase the number of query keywords from 1 to 7. Figure 8(b) illustrates that Tornado is scalable and that the system throughput increases with the increase in the number of query keywords. The reason is that when queries contain more keywords fewer objects match with queries. This reduces the number of output tuples generated and improves the

overall throughput. This resembles the same performance trend found in FAST [5]. To demonstrate the scalability of Tornado, we increase the number of continuous queries from 5 million queries to 40 million queries. Figure 8(c), shows that Tornado scales well when increasing the number of queries. The overall throughput is slightly reduced due to the increased number of output tuples that resulted from having more queries in the system. The scalability of Tornado is due to the scalability of the local spatial-keyword index, i.e., FAST. Fast is able to index a large number of queries with an efficient searching performance and a low memory overhead [5]

5.4 Adaptivity

In this experiment, we demonstrate the adaptivity in Tornado using the spatially condensed dataset. The spatially condensed dataset is used to introduce hotspots and to direct all workload into a small subset of evaluators. We vary the scaling factor for shrinking the dataset's spatial range from .4 to .7 of the entire spatial range. A smaller scale factor results in a stronger hotspot that is focused in a small subset of evaluators. A scale Figure 9(a) illustrates that the adaptive version of Tornado is able to maintain a stable throughput in contrast to the static version of Tornado. The smaller the scale factor, the lower the throughput for static partitioning. The reason is that, in the static partitioning, fewer evaluators handle the entire workload. This results in a bottleneck in the evaluation layer.

In Figure 9(b), we compare the communication overhead between the distributed and the centralized load balancing approaches. In the centralized load-balancing approach detailed workload statistics are transmitted to the routing layer. However, in the distributed load-balancing approach only summaries of statistics are transmitted to the routing layer. Figure 9(b) illustrates that the communication overhead of the distributed load-balancing is much less than the overhead of the centralized load-balancing approach.

6 RELATED WORK

The work related to Tornado can be categorized into three main categories: 1) Centralized spatial and spatial-keyword query-processing, 2) Distributed query-processing, and 3) Adaptive query-processing.

Centralized spatial-keyword systems: Several centralized spatial-keyword indexes have been proposed to process spatial-keyword queries e.g., [5, 8, 18]. These access methods integrate a spatial index, e.g., the R-tree [11] or the Quad-tree [15] with a keyword index, e.g., Inverted lists [22]. These access methods are centralized and do not scale across multiple machines. FAST [5] is a centralized spatial-keyword index that has been designed as a local index for Tornado.

Distributed Query-Processing: Many systems have been developed to process large-scale datasets. Batch-based systems, e.g., Apache Hadoop [1], are designed to process large amounts of data in an offline manner (i.e., on disk). In these systems, a single job can take several minutes or even hours to complete. Apache Spark [19] has been introduced to improve the latency of Hadoop. Streaming systems, e.g., Storm[16], process data streams of high arrival rates in real-time. However, none of the aforementioned systems is optimized for processing spatial-keyword queries. PS2Stream [9] is a distributed location-aware publish/subscribe streaming system. PS2Stream uses the Grid Inverted Index, i.e., GI^2. PS2Stream does not use newly optimized spatial-keyword indexes, e.g., [5, 18] that improve the overall system performance.

Adaptive Query-Processing: AQWA [6] is an adaptive spatial-only processing system that is based on Hadoop. AQWA executes snapshot spatial queries over static data.

7 CONCLUSIONS

In this paper, we present Tornado, a distributed system for the processing spatial-keyword data streams. We use Tornado to realize a location-aware publish/subscribe application. Tornado uses several optimizations, e.g., global routing, neighbor-based spatial routing, to alleviate performance bottlenecks in the system. Tornado is adaptive to changes in data distribution and query workload and is able to preserve the system throughput under varying workloads. Tornado achieves 2x improvements over the performance of the baseline approaches.

REFERENCES

[1] 2018. Hadoop. http://hadoop.apache.org/.
[2] 2018. Internet live stats. https://internetlivestats.com/.
[3] 2018. Keyword search statistics. http://www.keyworddiscovery.com/keyword-stats.html.
[4] 2018. The size of Hadoop clusters in Yahoo! https://wiki.apache.org/hadoop/PoweredBy.
[5] Ahmed M. Aly Ahmed R. Mahmood and Walid G. Aref. 2018. FAST: Frequency-Aware Indexing for Spatio-Textual Data Streams. In *ICDE*.
[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. *PVLDB* 8, 13 (2015), 2062–2073.
[7] Walid G Aref and Hanan Samet. 1990. Efficient processing of window queries in the pyramid data structure. In *PODS*. ACM, 265–272.
[8] Lisi Chen, Gao Cong, Christian S Jensen, and Dingming Wu. 2013. Spatial keyword query processing: An experimental evaluation. In *VLDB*, Vol. 6. 217–228.
[9] 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.
[10] Michelangelo Grigni and Fredrik Manne. 1996. On the complexity of the generalized block distribution. In *Parallel Algorithms for Irregularly Structured Problems*. Springer, 319–326.
[11] Antonin Guttman. 1984. *R-trees: a dynamic index structure for spatial searching*. Vol. 14. ACM.
[12] Zeinab Hmedeh, Harris Kourdounakis, Vassilis Christophides, Cédric Du Mouza, Michel Scholl, and Nicolas Travers. 2012. Subscription indexes for web syndication systems. In *EDBT*. ACM, 312–323.
[13] 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.
[14] Beng Chin Ooi, Ken J McDonell, and Ron Sacks-Davis. 1987. Spatial kd-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC*, Vol. 87. 85.
[15] Hanan Samet. 1990. *The design and analysis of spatial data structures*. Vol. 85. Addison-Wesley Reading, MA.
[16] 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 *SIGMOD*. ACM, 147–156.
[17] Subodh Vaid, Christopher B Jones, Hideo Joho, and Mark Sanderson. 2005. Spatio-textual indexing for geographical search on the web. In *Advances in Spatial and Temporal Databases*. 218–235.
[18] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. 2015. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In *ICDE*. 1107–1118.
[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. , 10–10 pages.
[20] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM Symposium on Operating Systems Principles*. ACM, 423–438.
[21] Yu Zhang, Youzhong Ma, and Xiaofeng Meng. 2014. Efficient Spatio-textual Similarity Join Using MapReduce. In *IAT*, Vol. 1. 52–59.
[22] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM computing surveys* 38, 2 (2006), 6.