DEBS Grand Challenge: Top-K Queries in RDF Graph-based Stream Processing with Actors

Syed Gillani, Abderrahmen Kammoun,
Julien Subercaze, Kamal Singh,
Gauthier Picard, and Frédérique
Laforest
Laboratoire Hubert Curien, UMR CNRS 5516,
and Institute Henri Fayol, EMSE
Saint Etienne, France
syed.gillani, a.kammoun,
julien.subercaze, kamal.singh,
frederique.laforest@univ-st-etienne.fr,
gauthier.picard@emse.fr

ABSTRACT

In this paper, we describe our novel system named as *RGraSPA* an RDF Graph-based Stream Processing with Actors, which adheres to the realm of RDF graph and knowledge reasoning, and uses an actor model for distribution of continuous queries. Furthermore, we present our approach to solve DEBS Grand Challenge by employing our system. *RGraSPA* uses RDF graph-based event model to encapsulate a set of triples and process them in continuous manner. We also present our *synchronised structure traversal* algorithm that uses *Range tree* to store results in a sorted view, where each node of the tree maintains a balanced Multimap Binary Search Tree (BST). The range of each node is adaptive and updated according to the incoming values and defined size of the Multimap BST for each node.

In order to solve the DEBS challenge, we provide a formal method to calculate cell IDs from the longitude and latitude in a streaming fashion and use two Range trees for 10 most frequent routes and profitable areas. Our experimental results show that the query execution time can be optimised by carefully adjusting the cardinality values of Range tree. Our solution processes 1 year worth of RDFised data (372 GB) (approx 3.4 billion triples) for Taxis in 1.8 hours.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees,Distributed data structures
; I.2.4 [**Knowledge Representation Formalisms and Methods**]: Semantic Networks

Keywords

Top-K Continuous Queries, RDF Graph Streaming, Range Trees

1. INTRODUCTION

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.

http://dx.doi.org/10.1145/2675743.2772587.

The real-time analytics of continuous top-k queries over sliding windows on high volume data streams has several interesting applications ranging from financial analysis to network traffic monitoring [14]. It involves in reporting a certain number (k) of top preferred objects from data streams within a time window. This challenging problem of designing an efficient top-k query system is the subject of this years 2015 DEBS grand challenge. The target problem is to analyse the data of New York Taxi service and efficiently identify the recent top-k frequent routes as well as profit areas. Additionally, the challenge also demands interesting techniques to find the cell IDs a taxi belongs to, by processing its drop-off and pick-up location. The full data consists of around 173 million events recorded in 2013 and corresponding to the whole year. The data consists of recorded information of taxi trips such as taxi identifier, pick-up and drop-off coordinates, timestamps and fare information.

DEBS Challenge Queries. This year's DEBS challenge consists of two queries. The objective of the Query 1 is to find the top 10 most frequent routes during the last 30 minutes. A route is defined as a pair of starting and ending cells. Each cell is of size 500mx500m and the area under consideration for the routes is divided into 300x300 cells. The two important issues to tackle in this query are as follows:

- Finding routes and cell IDs of each incoming event from longitude and latitude values in streaming fashion;
- Maintaining the top-10 routes and triggering the output if an event results in a change to top-10 routes' order or if it results in addition or replacement of a route in the top-10 list

Query 2, which is more complex than the first one involves in determining the most profitable cells for taxi drivers, where each cell is of size 250mx250m. The profitability of a cell is computed by dividing the median cell profit by the number of empty taxis in that cell. The median profit of a given cell is used for profitability computation, where profit of each trip (within 15 minutes) is the sum of the fair as well as the tip. The number of empty taxis is estimated to be the number of taxis that dropped off their clients in the given cell within the last 30 minutes and did not find any new client. The major challenges in this query are as follows:

- Finding cell ids for each event from longitude and latitude values in a streaming fashion;

- Finding a median value of the profits in a continuous manner;
- Determining the number of empty taxis in a cell from drop-off and pick-up cell IDs of each event;
- Simultaneously handling two different windows (profitability and empty taxis) on streams and maintaining top-10 profitable cells.

RDF Graph Stream. Before proceeding towards the detailed discussion of top-k queries, we would like to motivate readers towards a knowledge-based stream processing. Most of the existing stream processing systems [17, 8] are based on relational technologies and offer an efficient on-the-fly analysis of tuple based stream elements, usually called as *Events* – that enclose a certain set of attributes. However, they fall short of combining high-level knowledge representation with background knowledge. For instance, considering the requirements of DEBS challenge, taxi routes are determined by pick-up cell and drop-off cell IDs, where each event only contains the information about the geolocation of pick-up and drop-off places, hence an external knowledge base (KB) containing the boundaries of the cell IDs can easily be utilised to not only determine the routes and cell IDs but also the owner of the car, street names, weather information etc. Figure 1(a) presents the schema (ontology) of live RDF graphs and Figure 1(b,c) presents the static KB to enrich the live event. To fulfil, the knowledge part of the streams, RDF stream processing [7, 4, 12] was introduced. RDF can be realised as directed-labelled graph model that consists of a set of triples, where each triple consists of subject(s) (vertex), predicate(p) (edge) and object(o) (vertex) ($\langle s, p, o \rangle$). Existing solutions for RDF stream processing do provide interesting and additional benefits as compared to relational stream processing. However they fail to reach their target due to their triple-based model. Since each event within a stream consists of a set of attributes (e.g., taxi id, location, timestamps, fare etc.), therefore distributing each event into a set of triples (e.g, one triple for each attribute) and consuming as triple stream, not only shreds the general architecture of RDF graph but also fails to capture the boundaries on a set of attributes within an event. Therefore, we propose *RGraSPA*, an RDF Graph-based Stream Processing with Actors, which adheres to the realm of RDF graph and knowledge reasoning.

Top-K Continuous Queries. Existing techniques for top-k queries in conventional databases involves in pre-analysing the static data to prepare appropriate meta-information to subsequently answer incoming top-k queries [14]. However, in streaming environments, data evolves over time and single model is not efficient to cater such queries. Therefore, the key design for continuous top-k queries to is design a top-k maintenance mechanism that efficiently updates the top-k results even under high-input data rates and over huge query windows. Therefore, given a data stream S, window ($w(S)$) on stream and a preference function F, a continuous top-k query $Q(S, F, w, k)$ continuously returns (or updates) k objects(o) $\{o_1, o_2, \ldots, o_k\}$ from data stream within the window $w(S)$ that have the highest $F(o_i)$ score among all objects within the defined window. The query window can be either time or count-based. Existing solutions [9, 3, 16] for top-k continuous query processing aim at reducing computational costs by incrementally updating the top-k results upon each window slide. However, they all suffer from the performance bottleneck of periodically requiring a complete re-computation of the top-k results from scratch. Therefore, one of the motivation for DEBS challenge is to eliminate the performance bottlenecks of re-computing top-k results from scratch within a window. We propose a *synchronised structure traversal* algorithm that uses a *Linked-Hash queue* and an extended form of *segment* tree [5] to maintain a ranged sorted view, where each node maintains a balanced Multimap Binary Search Tree (BST) and involves in ((lognm)) operations – where n is the size of range tree and m is the size of Multimap BST. In this paper, we refer this structure as a *Range* tree. There are also some probabilistic based solutions [9] for top-k continuous queries but they fail in terms of accuracy (they are usually are application dependent) and involve in high-computation with the increase in volume of the data.

Distribution of Stream Processing. Existing solutions for graph processing rely either on a centralized index system with tremendous pre-computation overhead, or on a distributed graph processing system such as Pregel [13] that requires much synchronisation effort. Such solutions in streaming environments not only disregard on-the-fly requirements for streams, but also incur high indexing and storage costs. Furthermore, the performance of these systems degenerates with frequent updates due to the changes in streaming data. Therefore, we propose an actor based asynchronous solution to answer large numbers of top-k queries. Each query is registered as an actor (processing unit) and it receives data for a single or multiple actors (stream sources). Furthermore, incoming RDF graphs are also clustered and only relevant RDF graphs or parts of them are sent to the queries.

Contributions. Our general contributions towards the topic and specifically towards the DEBS challenge are as follows.

Top-K Queries. To tackle the re-computation bottlenecks, we design a *synchronised structure traversal* algorithm. It uses an integrated data structure composed of a Linked-Hash queues and Range tree, where each node is responsible for a range of values that are placed in a Multimap BST. Range tree is defined recursively and ranges are adaptive and updated according to the incoming values and defined size of the Multimap BST for each node. It is executed in a way that it arranges the top-k results in a sorted – ascending or descending order – and events belonging to those results in a queue structure. Due to the self-balancing tree structure, no re-computation is required and insertion and deletion is of $O(\log nm)$ complexity.

RDF Graph Streaming. We propose a light-weight on-the-fly RDF graph processing algorithm. Our algorithm makes use of the vertical partitioning techniques [2], where each distant predicate table stores all the subjects and objects bearing that predicates. These tables are cached into the main memory as multi-set hash tables and then used to perform hash-joins based on the RDF graph queries. Due to the vertical partitioning, our system also implements the *partition by* clause and only sends the predicate tables required by the queries. There is no pre-processing and indexing involved in our algorithm, thus it saves important storage and index computation costs, and also provides optimised performance for relatively small RDF dataset.

DEBS Challenge. We propose an efficient formal technique to estimate the cell IDs from longitude and latitude coordinates in streaming fashion. Our solution does not involves in high computation of converting longitude and latitude into other formats. We have also extended our synchronised structural traversal algorithm with a list-based Red-Black tree to store the Cell IDs in a list and their corresponding profits (fare+tip) in a tree structure to calculate the running median values. Furthermore, our knowledge-based stream processing provides an insight on how it can enrich taxi data stream to infer contextual information. This paper is organised as follows. Section 1 provides the introduction of the DEBS challenge and the target problems. Section 2 provides the background on RDF, SPARQL, our event model, and actor model, which are used in the following sections. Section 3 provides a brief presentation to our proposed system architecture. Section 4 provides a detailed dis-

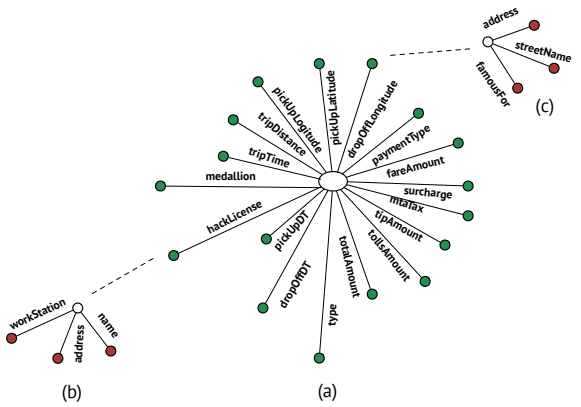

Figure 1: (a) Live RDF Event Schema, (b),(c) Static Background KB

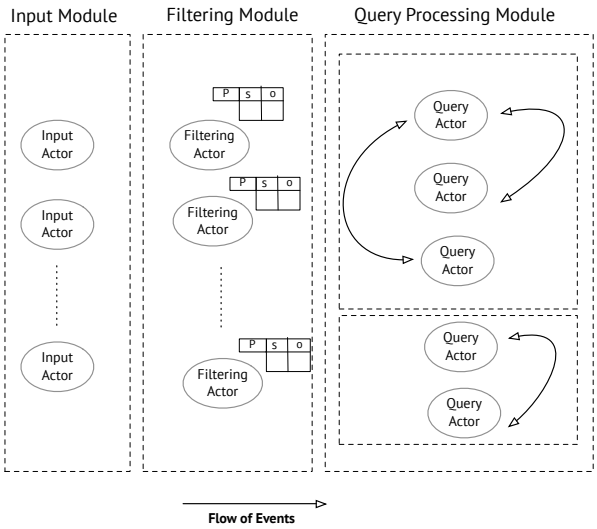

Figure 2: System Overview: RDF Graph Stream Processing

cussion on our algorithms to solve DEBS challenge queries. Section 4 discusses the results and Section 5 concludes the paper.

2. PRELIMINARIES

In this section, we present an introduction to the RDF graphs and underlying Akka actor model, which we use for distribution of stream processing.

RDF Graph. An RDF graph can be conceptualised as a directed edge labelled graph, where a finite set of triples ($G \subset (I \cup B) \times I \times (I \cup B \cup L)$) constitute an RDF graph [1]. Each triple is the union of three pair-wise disjoint sets: the set of all IRIs (I), the set of all literals (L) and the set of all blank nodes (B). Since RDF graphs are defined in terms of sets, it follows that the ordering of RDF triples in an RDF graph is entirely arbitrary and that RDF graphs do not allow for duplicate triples. The key construct for processing RDF data is triple pattern matching. Here, users describe the required structure of a set of triple patterns as queries and systems return as answers all occurrences of the pattern found in the data. A set of triple patterns, where each triple pattern $tp \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ and V is a set of variables allowed in the triple pattern – constitutes a query (considering only Basic Graph Pattern Queries [15]) in which at least one of the subject, predicate or object is a variable, denoted by a leading question mark. Therefore, the query attempts to match a set of triple patterns to sub-graphs in the RDF and the result of a triple patterns query is a list of all variables substituted from those graphs. The standard query language for expressing triple pattern matching queries on RDF data is SPARQL[1].

Event Data Model. In a traditional relational-based stream processing, a stream $\mathcal{S}$ is a countable infinite set of events $s \in \mathcal{S}$. Each stream event $s : \langle v, t \rangle$ consists of a relational tuple v conforming to a schema S and a system or application timestamp t. Each relational tuple v is usually a set of attributes describing various objects within an event. The Semantic Web community has mapped this stream event model into triple model [4] ($s : \langle triple, t \rangle$), where each event is mapped to an RDF triple. This provides the use of external Knowledge-base and RDF reasoning. However, this model fails to capture the set of attributes for each stream event. Therefore, we present a new event model, where each event of the stream $s : \langle G_n, t \rangle$ consists of a Named RDF graph – that provides the appropriate boundaries to a certain set of attributes – and a time-stamp associated with each event. Figure 1(a) shows the event model for live taxi data streams, which represent a star-shaped schema.

[1]http://www.w3.org/TR/rdf-sparql-query/

Actor Model. Actor model is a model of concurrent computation for developing parallel and distributed systems [6]. Each actor is an autonomous object that operates concurrently and asynchronously, receiving and sending messages to other actors, creating new actors, and updating its own local state. An actor system consists of a collection of actors, some of whom may send message to, or receive message from, actors outside or within the system. Our system uses Akka[2], which is an efficient actor model implementation for Java Virtual Machine (JVM) and provides a higher level of abstraction for writing concurrent and distributed systems. It reduces the need for context switching between threads and applies its own space scheduling as opposed to relying on OS scheduler. Therefore, this reduces the blocking function call and uses message queues to send or receive message – thus, releasing an event-based programming model. The Akka actor model can be represented as a Directed Acyclic Graph (DAG), where each node is an actor (or processing unit) and has a job queue as an edge to communicate with other actors.

3. RGraSPA

In this section, we review our generic system architecture, the motivation behind its design and optimisation strategies for parallel and distributed processing of streams. We first present our event model that is used for continuous RDF graph processing and later explain how we integrated it with the actor model to implement a high-performance Knowledge-enabled RDF graph stream processing.

3.1 System Overview

Data stream management system (DSMS) retrieves tuples from streams and process them using queries that are registered to it by complying to certain window constraints. However, as our system deals with RDF graphs instead of tuples or RDF triples, therefore, first we need to pre-process RDF graph data before triggering graph matching queries. We divide our system into three modules, where each module consists of a set of actors – that works in parallel (for standalone application by using threads) or in a distributed way (by distributing each actor on various nodes). Figure 2, presents an

[2]http://akka.io/

overview of our system design and the brief detail of each module is as follows.

Input Module. This module deals with the pre-processing of input data. To cache RDF data, we use a vertical partitioning approach. Traditionally, vertical partitioning involves in partitioning RDF data into multiple two-column tables $\langle s, o \rangle$. Each table of distinct predicates stores all the subjects (s) and objects (o) bearing that predicate (p). However, as triple pattern in RDF graph query can involve in s-s and o-s joins, we use two vertically partitioned multi-hash sets to cache all the RDF data. Similar to input RDF graphs, the external static KB graphs are also cached into vertical partitioned tables. Due to the large size of KB, our system only loads the tables that are required during the join of live and static RDF graphs.

Filtering Module. This module performs the required filtering on partitioned triple sets from input graphs. Each query is registered to a set of actors that send the required vertical partitioned tables to it. For instance if there are the following four triple patterns in an RDF query graph then the filtering module will send four vertical partitioned tables for each predicate:

(t_{p1}) $?s$ $hasName$ $?o$
(t_{p2}) $?s$ $liveIn$ $?o2$
(t_{p3}) $?s$ $friendOf$ $?o2$
(t_{p4}) $?s$ $worksFor$ $?o2$

Similarly, each query also receives all the predicate tables from external KB that are used in the triple patterns of the query. These tables are joined in the next module to report the match of query and other contextual information that is generated with joins of live and static RDF graphs.

Query Processing Module. This module joins various vertical partitioned tables according to the triple patterns described in the queries. We use Hash-based joining techniques, as it is more efficient for a relatively smaller set of data (as compared to Merge-join) [11]. As there could be a set of n queries $Q = \{Q_1, Q_2, \ldots, Q_n\}$ over the streaming RDF graphs, our system identifies whether there are some overlapping triple patterns among queries that can share the evaluated results. Each query is registered as an actor that consumes input from the filtering module. Our system partitions actors of queries into groups, where queries in the same group share common sub-patterns, thus those common sub-patterns are evaluated just once and results are distributed among the same group. The execution of these continuous queries is applied on windows of streams, and the result of the selected attributes are stored in top-k range trees. All data stream management systems provide some form of windowing functionality. Therefore, our system provides the two primary types of windows: *Tumbling window* and *Sliding window*. Both of these types are divided into two flavours: tuple-based – which in our case is RDF graph based – and time-based. These windows provide the required functionality to unblock otherwise blocking operators such as aggregations.

4. SOLVING DEBS CHALLENGE

This section presents the details of our algorithms to solve the two queries presented in DEBS challenge. As the data provided in the DEBS challenge is of raw nature, we constructed a small RDFS schema model to map the data into RDF graphs, where each RDF graph encapsulates a set of attributes for each taxi. These RDF graphs are then sent as streams and queries are executed continuously adhering to window constraints. Figure 1(a) presents an overview of our simple ontology. Please note that a detailed ontology can be used to infer and reason complex contexts within events. However, as the objective of the challenge was to only get the information regarding the routes frequency and profitability of cells, therefore, we inclined to use a simplistic RDFS schema for the triples.

4.1 Query 1

The main objective of Query 1 is to find the 10 most frequent routes within a time window of 30 minutes, where a route is represented by a starting and an ending grid cell. Each cell is a square of 500 meters x 500 meters and the cells shift towards east and south from a reference point. The two main challenges involved in this query are i) Efficient calculation of cell IDs from longitude and latitude of input streams events (graphs in our case) and ii) avoiding re-computation of top-10 results for each stream, while producing results only when there is a change in the top-10 result set. For the first challenge, we present a formal technique to compute cell IDs in a streaming fashion. Please note that an external KB containing the boundaries of cell IDs can also provide such functionality. However as the organiser specifically ask for stream-based computation, we use our formal technique for computing cell IDs.

4.1.1 Estimating Cell IDs

Here, we describe how we estimate the cell IDs from the longitude and latitude coordinates. Each event data has longitude and latitude coordinates that correspond to pick up location as well as drop off location for a given taxi trip.

Let x_0 be the longitude and y_0 be the latitude of the center of the first cell, whose cell ID is 1.1. Let $cell_x$ correspond to the cell ID sub-index increasing from west to east and $cell_y$ corresponds to the cell ID sub-index increasing from north to south. Further, let s denote the width as well as the length of a cell, as they are square cells, where $s = 500m$ for Query 1 and $s = 250m$ for Query 2.

Given a location with longitude coordinate x and latitude coordinate y, we have to find the corresponding values of $cell_x$ and $cell_y$. The given location is $(x_0 - x)/\delta x$ towards the east and $(y - y_0)/\delta y$ towards south from the center of the first cell as, in the region considered, the coordinates decrease while moving east and increase while moving south. Here, δx is the change in the longitude coordinates per distance in meters and δy is the change in the latitude coordinates per meter. They are assumed to be $0.005986/500$ degrees per meter and $0.004491556/500$ degrees per meter respectively as provided by DEBS organisers.

The starting points of a cell with indices $cell_x$ and $cell_y$ are $(cell_x - 1)s - s/2$ in the east and $(cell_y - 1)s - s/2$ in the south. The ending points of the cell are $(cell_x - 1)s + s/2$ in the east and $(cell_y - 1)s + s/2$. We add or subtract by $s/2$ as the relative distance in measured from the center of the first cell. Thus, if the given point lies in the cell given by $cell_x$ and $cell_y$, whose values we have to find, then following is true:

$$(cell_x - 1)s - \frac{s}{2} \leq \frac{x_0 - x}{\delta x} < (cell_x - 1)s + \frac{s}{2} \tag{1}$$

Using the left hand side of the above eq. (1), we can say that:

$$(cell_x - 1) \leq \frac{1}{2} + \frac{x_0 - x}{s\delta x} \tag{2}$$

and using the right hand side of eq. (1) we can say that:

$$-\frac{1}{2} + \frac{x_0 - x}{s\delta x} < (cell_x - 1)$$

or

$$\frac{1}{2} + \frac{x_0 - x}{s\delta x} < cell_x \tag{3}$$

Thus, from eq. (2) and eq. (3), we see that $\frac{1}{2} + \frac{x_0 - x}{s\delta x}$ lies between two consecutive integers $cell_x - 1$ and $cell_x$. Thus, the value of

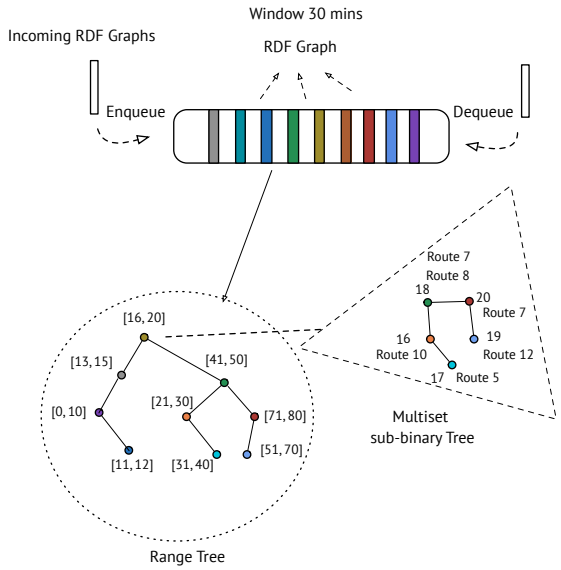

Figure 3: Stream Queue, Frequency Range Tree for Query 1

$cell_x$ can be found using a floor function, denoted by $\lfloor \rfloor$, which by definition maps a real number to the largest previous integer:

$$cell_x - 1 = \left\lfloor \frac{1}{2} + \frac{x_0 - x}{s\delta x} \right\rfloor$$

or

$$cell_x = 1 + \left\lfloor \frac{1}{2} + \frac{x_0 - x}{s\delta x} \right\rfloor \quad (4)$$

Using similar derivations as above, the value of $cell_y$ can be found as:

$$cell_y = 1 + \left\lfloor \frac{1}{2} + \frac{y - y_0}{s\delta y} \right\rfloor \quad (5)$$

Note that the difference here is $y - y_0$ as in the given region the coordinates increase while moving south and y_0 corresponds to the center of the north-most and the west-most cell.

The use of the above equations prevent the heavy computation during latitude and longitude conversion into radian or any other format, such as Universal Transverse Mercator (UTM) [10]. Therefore a function encompassing simple arithmetic operations accepts latitude and longitude values to provide drop-off and pick-up cell IDs of each event. We compute the cell IDs of each event in the filtering module (see Figure 2) and then add it as a triple with each event, which is later used in window queue and range trees, as described in next sections.

4.1.2 Sliding Windows and Top-K Results

The second part of the challenge is efficiently handled by our *synchronised structure traversal* algorithm. Each event from the stream is added to the Linked-hash queue and the corresponding route's frequency is used to search for its suitable position within a range tree. If a node is responsible for a range $[l, r]$, then its left child is responsible for the range $[r_2, l - i]$, where $i > 0$ and $r_2 << l$, while its right child is responsible for the range $[r + i, r_3]$, where $i > 0$ and $r_3 >> r$. Each node of the tree contains a range of routes frequency and uses a Multimap BST to store the set of routes and their frequencies. The size of the Multimap BST tree can be tweaked to get the optimal distribution of ranges. The range of each node is adaptive and if a new route is added to the tree and size of the Multimap BST is greater than the v – where $v >> k$ and k (10) is number of preferred objects – then the last k values of the Multimap BST are used to create a new node in the range tree and stored in a new Multimap BST connected to the newly created node. Figure 3 presents the route queue and the range tree to store the routes according to their frequency. The use of Range tree reduces the search space and provides an efficient look-up for 10 most frequent routes.

The window for Query 1 is a time-based one, but as there is no sliding value described in the challenge, therefore we use the original timestamps of each event– instead of using system timestamps – to determine the expiration condition of an event within a window. Algorithm 1 describes the execution of sliding window and top-k result computation. Each incoming event is first compared with the head (first event added to the queue) of the queue based window, if the difference between the timestamps of newly arrived event and the event placed at the head of the queue is greater than 30 minutes (line 17-21), then the older event is deleted from the queue and its corresponding route frequency is updated in the Multimap BST within a Range tree. This procedure continues until all the older events are deleted from the window. After adhering to the window constraint the newly arrived event is added to the tail of the queue and if the Multimap BST tree already contains corresponding route, its frequency value is updated (deletion and addition of tree node) (line 12-13). Multimap BST also maintains the freshness of results by sorting the data belonging to the same key in insertion order. Finally, the Range tree is post-orderly (right-most leave) traversed to get the top-10 frequent routes. Another, important requirement of DEBS challenge was to only display results if there is either a change in the top-10 values or change in the order of top-10 values. Therefore, we maintain an additional view of the last computed top-10 values and if there is a difference between last updated view and newly computed top-10 values (during post-order Range tree traversal) (line 24-26), the system produces the required results.

4.2 Query 2

Query 2 is more complex as compared to Query 1. Here the challenge is to track the profits associated with each cell, track the number of empty taxis in that cell, efficiently compute the continuous median of the profit from all the trips that started in a given cell and ended somewhere within the last 15 minutes, and design an efficient way of joining two different sliding windows – i.e, sliding window of 30 minutes for empty taxis and 15 minutes to determine the profits. Furthermore, the cell size is also reduced to 250mx250m, as compared to Query 1.

4.2.1 Estimation of Cell IDs

To estimate cell IDs for Query 2, we use the same method as described for Query 1. We use eq.(4) and eq.(5). However, as the cell size is different in Query 2 – cell of 250mx250m instead of 500mx500m – therefore we update the value of s= 250m in eq.(4) and eq.(5). Due to the reduction in the cell sizes, query 2 deals with 600x600 number of cells.

4.2.2 Sliding Windows and Top-K Results

Similar to the Query 1, we use the original timestamps entailed by each event to determine if an event has to drain from the window or not. However, as there are two different windows associated with this query, we use two different queues: one to store the event that should determine the number of empty taxis and another that determines the trip profitability (see Figure 4). One important extension

Algorithm 1: Query 1 Algorithm

```
1  Input: Event Stream (S), Event Queue (Q), Event (e)
2  Output: Top-10 most frequent routes
3  /* Let V be the view to maintain the changes in
   top-10 values */
4  /* Let R be the range route tree */
5  while e_i ∈ S do
6      ComputeCellIDs(e_i)
7      if not Q → φ then
8          CheckWindow(Q, e_i, R)
9      else
10         Q ← Q + e_i /* add the event to the Queue */
11         R ← R + r_i /* add route to the range route
           tree */
12         GetTop10(R, V);
13 function CheckWindow(Q, e_i, R)
14 while Q.size() > 0 and Q.peek().t_s < e_i.t_s − 30 do
15     e_old ← Q.RemoveTop()
16     R ← updateTree(e_old)
17 function GetTop10(R, V)
18 top10 ← R /* get top 10 entries (post-order
   traversal) */
19 if not (compare(top10, V)) then
20     Output(top10);
21     V ← top10 /* update view */
```

to our synchronised structural traversal algorithm for Query 2 is the addition of a hash-map and Multimap BST to store the sorted view of profits (fare + tip) for each cell ID. This enables us in calculating the continuous median value by first checking the size of the sorted values corresponding to each cell ID. If the size is of odd value, we simply get the median by selecting the middle element of the list and in case of even size list, the median is calculated by adding the two middle elements and dividing by two. Figure 4 shows the construction of profit window and other data structures. The execution of Query 2 is described in Algorithm 2. It starts by first checking the size of the profit window queue (empty taxis window works in parallel to this one, but due to space restriction, its not shown in the Algorithm 2). If there are already some events in the profit window queue (line 12), then the timestamps of newly arrived event is used to check the window constraints on the events in the profit window queue. This process continues until algorithm finds an event at the head of the window which satisfies window constraints (line 21-26). The next step is to add the newly arrived event in the window and profit (fare+tip) from the event to profit multi-set tree ans its cell ID to hash-map (line 13-14). In order to get the profitability of the cell, our algorithm gets the number of empty taxis in the pick-up cell from the other window of 30 minutes (line 15). If the number of taxis is greater than zero, it first gets the median value of the profit (fare+tip) for pick-up cell and then divides it by number of taxis. This value is updated in profitability tree (line 16-17)– i.e., deletion and insertion operation in the tree. As previously, we just calculated the profitability of pick-up cells, but there could be some cases when within a portability window a cell has some number of trips that exist in profit (fare+tip) tree but has zero profitability, as the number of empty taxis were zero. And if a new event arrives with drop-off location of that cell ID, then it would have more than zero taxis and we need to calculate its profitability as well. Figure 5, presents such case, where Taxi-1, Taxi-2, Taxi-3 have pick up

Algorithm 2: Query 2 Algorithm

```
1  Input: Event Stream (S), Profit Event Queue (Q)
2  Output: Top-10 most profitable areas
3  /* Let V be the view to maintain the changes in
   top-10 values */
4  /* Let P be the profitability range tree */
5  /* Let F be the hash-map with multimap binary
   tree fare+tip */
6  while e_i ∈ S do
7      ComputeCellIDs(e_i)
8      if not (Q → φ) then
9          CheckWindow(Q, e_i, F)
10     else
11         Q ← Q + e_i /* add event to the Queue */
12         F ← F + e_i.f /* add fare+tip to the fare
           tree with cell ID */
13         ET_p ← getEmptyTaxis(e_i.pickUpCell()) /* get
           empty taxi of pick-up cell from Empty
           Taxi Window Queue that works in parallel
           to this one */
14         if ET_p > 0 then
15             M ← getMedianValue(F, e_i.pickUpCell())
16             P → updateTree(M ÷ ET_p, e_i.pickUpCell()) /*
               if cell ID already exists, then update
               Tree */
17         ET_d ← getEmptyTaxis(e_i.dropOffCell()) /* get
           empty taxi of drop-off cell from Empty
           Taxi Window Queue that works in parallel
           to this one */
18         if ET_d > 0 then
19             M ← getMedianValue(F, e_i.dropOffCell())
20             P → updateTree(M ÷ ET_d, e_i.dropOffCell())
               /* if cell ID already exists, then
               update Tree */
21         GetTop10(P, V);
22 function CheckWindow(Q, e_i, F)
23 while Q.size() > 0 and Q.peek().t_s < e_i.t_s − 30 do
24     e_old ← Q.RemoveTop()
25     F ← F/e_old.f
       P ← updateTree(e_old.dropOffCell(), e_old.pickUpCell())
26 function GetTop10(P, V)
27 top10 ← P /* get top 10 entries (post-order
   traversal)*/
28 if not (compare(top10, V)) then
29     Output(top10);
30     V ← top10 /* update view */
```

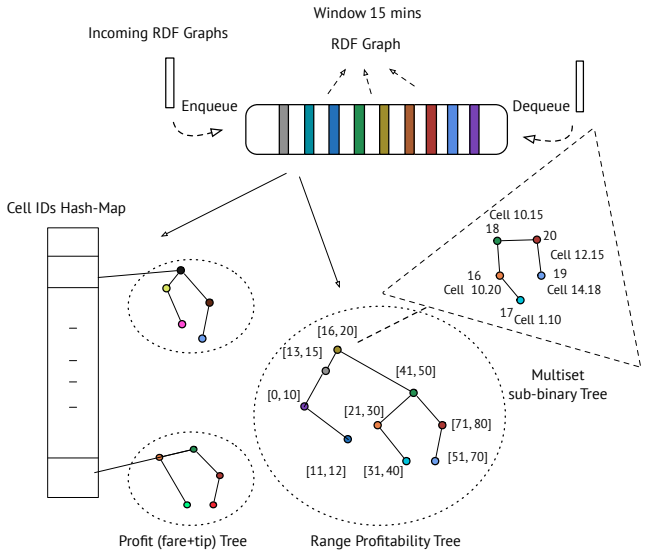

Figure 4: Stream Queue, Profit list and Mulimap tree and Profitability Range Tree for Query 2

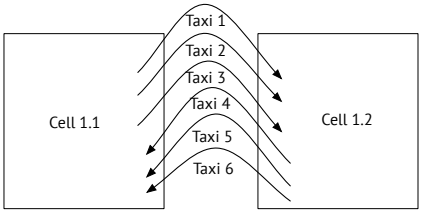

Figure 5: Profit and Empty Taxi Formation

location from Cell 1.1 and drip-off location at Cell 1.2. After the Taxi-3 event, there are no empty taxis in Cell 1.1 (therefore zero profitability), but it has three values of profit (fare+tips). However, at this point Cell 1.2 has three empty taxis but as there is no pick-up from this cell yet, therefore it also has zero profitability. Now when an event of Taxi 4 arrives, it reports that Cell 1.2 has a pick-up and thus will have profitability value. Furthermore, as now Cell 1.1 will also has an empty taxi, we need to calculate its profitability as well(line 19-22). Similar to Query 1, Query 2 also maintains a view of the top-10 profitable cells and if there is an update to these results as compared to the last maintained view it produces updated results (line 33-35).

5. RESULTS

In order to evaluate the results for our system, we use two sets of data, the first set contains 1 day of taxis data 500k graphs/events (0.85GB) and the second set contains data worth of 1 month around 15 million graphs/events (30.5GB). Each RDF graph consists of 18 triples. Our testbed machine has the configuration of Intel Xeon E3 1246v3 processor with 8MB of L3 cache. The system is equipped with 16GB of main memory and a 256Go PCI Express SSD. The system runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 7u67.

We next report an experimental evaluation of our system. we use the two data sets to check the performance of the system by varying the size (s) of Multimap BST. We used four sets of s values, two sets for Query 1 and two for Query 2. The choice is based on the fact that Query 1 involves in less range of values – as on average

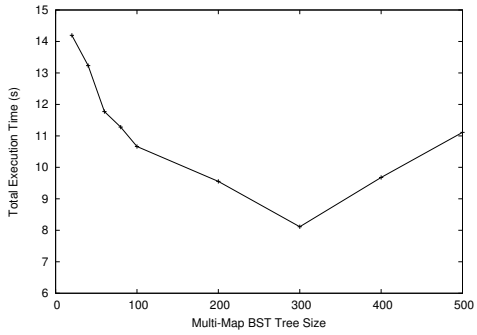

Figure 6: Execution Time (s) and the Varying Tree Size (1 Day) (Query 1)

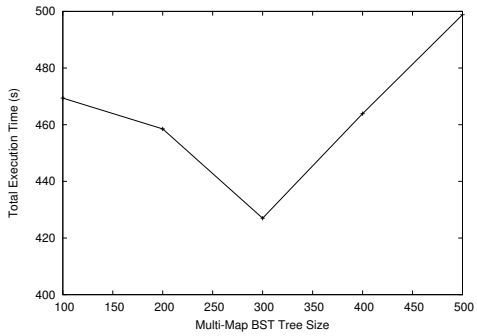

Figure 7: Execution Time (s) and Varying Tree Size (1 Month) (Query 1)

frequency of the taxis ranges from 0 till 30 in one day– while Query 2 covers a much broader range of values.

Figure 6 and 7 show the average execution time (after 10 runs) for the two sets of data (one day and one month) for Query 1, and Figure 8,9 for Query 2. For Query 1 the execution time starts at a higher value, as due to the smaller size of Multimap BST, there are more nodes in the Range tree, but it started to decrease with the increase in the size of Multimap BST, as the size of the Range tree starts to decrease. Not surprisingly, after reaching a certain point when the sizes of both trees are optimal the execution time started to increase with the increase in the size of Multimap BST and decrease in the size of Range tree. The same behaviour is observed in Query 2. We use a much larger values of Multimap BST tree size for one month data set and it shows the same behaviour for Query 1 and Query 2. With the use of Range tree our system has the faster time for top-k result lookup compared with randomly built BST trees or using list sorting. Our experimental results show that at large tree sizes the difference in top-k query performance between the two types of trees (Range tree and simple BST) is significant large. This means that Range trees are faster than BSTs or list sorting for this setting – despite still having a higher memory footprint. Based on these results, a streaming system can be tweaked – i.e., size of Multimap BST depending on different application.

6. CONCLUSION

This paper presents our RDF graph based streaming system *RGraSPA* that uses RDF graph as a data model for events and an actor model for the distribution of queries. We have also described our synchronised structure traversal algorithm that uses Range tree to efficiently process continuous top-k queries. We also show that a Knowledge-

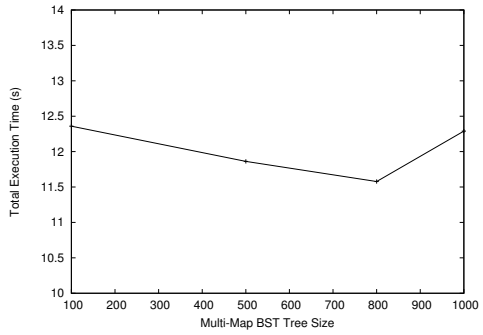

Figure 8: Execution Time (s) and the Varying Tree Size (1 Day) (Query 2)

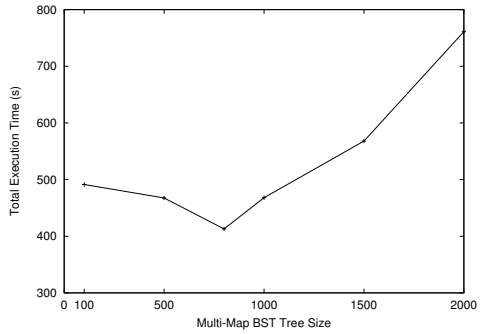

Figure 9: Execution Time(s) and Varying Tree Size (1 Month) (Query 2)

enabled stream processing system provides useful contextual information by joining live and static RDF graphs.

Our technique for calculating cell IDs from longitude and latitude uses simple arithmetic comparisons as compared to computational expensive techniques. Our system enabled us to implement all the required features in an efficient way (joins, windows) and our synchronised structural traversal algorithm avoids the bottlenecks of re-computation of top-k results and median values. Our experimental evaluation provides a clear picture of the operators involved in heavy computation and shows that the use of Range tree for large volume of data can dramatically decreases the execution time for top-k queries.

7. REFERENCES

[1] RDF 1.1 primer. Technical report, World Wide Web Consortium, 2014.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[3] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 28–39, New York, NY, USA, 2003. ACM.

[4] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 1061–1062, New York, NY, USA, 2009. ACM.

[5] J. Bentley. Algorithms for kleeâĂŹs rectangle problems. *Technical Report, Carnegie-Mellon University, Pittsburgh, Penn., Department of Computer Science*, 1977.

[6] C. Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[7] J. Huang, K. Venkatraman, and D. J. Abadi. Query optimization of distributed pattern matching. In *ICDE 2014*, 2014.

[8] N. Jain, S. Mishra, and Srinivasan. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390, Aug. 2008.

[9] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.

[10] C. Karney. Transverse mercator with an accuracy of a few nanometers. *Journal of Geodesy*, 85(8):475–485, 2011.

[11] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, Aug. 2009.

[12] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 370–388, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] G. Malewicz, M. H. Austern, and Bik. Pregel: A system for large-scale graph processing. SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[14] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. ICDT'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[15] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.

[16] W. Rao, L. Chen, S. Chen, and S. Tarkoma. Evaluating continuous top-k queries over document streams. *World Wide Web*, 17(1):59–83, Jan. 2014.

[17] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.