DEBS Grand Challenge: Real Time Data Analysis of Taxi Rides using StreamMine3G

André Martin, Andrey Brito*, Christof Fetzer*
* Technische Universität Dresden - Dresden, Germany
* Universidade Federal de Campina Grande - Campina Grande, PB, Brazil
andre.martin@tu-dresden.de, andrey.brito@dsc.ufcg.edu.br, christof.fetzer@tu-dresden.de

ABSTRACT
In this paper, we present our approach for solving the DEBS Grand Challenge 2015 using StreamMine3G, a distributed, highly scalable, elastic and fault tolerant ESP system. We first provide an overview about the system architecture of StreamMine3G followed by a thorough description of our implementation for the two queries that provide continuously up-to-date information about (i) the top-$k$ most frequently driven routes and (ii) most profitable areas.

Novel aspects of our implementation include two self-balancing double linked list implementations to efficiently update and determine a top-$k$ as well as a median from a set of samples. Furthermore, we present a solution that supports data partitioning which allows the application to scale without bounds while still guaranteeing semantic transparency through the deterministic processing approach offered by the StreamMine3G runtime. In our evaluation, we provide measurements that show that our system can scale horizontally as well as vertically and can process 13 $k$Events/s on a single node which translates to a processing of 3.8 hours of real time data within a second and a latency under 1 ms.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed applications

General Terms
Algorithms, Design, Reliability

Keywords
Complex Event Processing, CEP, Event Stream Processing, ESP, Scalability, Migration, State Management, Fault Tolerance

1. INTRODUCTION
With the recent advent of fairly priced mobile devices and data plans, we witness an increasing amount of mobile applications that use geospatial data in order to carry out some Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.
Copyright 2015 ACM 978-1-4503-3286-6/15/06 ...$15.00.
http://dx.doi.org/10.1145/2775743.2772584.
form of service. Examples for such services range from simple dating apps where individuals can easily spot peers sharing common interests in close proximity, to services such as Uber, where customers can quickly find a nearby available taxi for a ride. Although all those applications follow a different purpose targeting a specific group of customers, they all have in common to process geospatial data in near real time in order to carry out their tasks.

The processing of such data is often done by back-end systems on server-side rather than on the mobile device itself where the application solely acts as a client constantly sending its location updates and receiving notifications in exchange such as the availability of a taxi in close proximity if desired. Since the incoming data stream must be processed with low latency in order to provide up-to-date information to customers, such data processing tasks are often carried out using Event Stream Processing (ESP) Systems rather than database or MapReduce [6] like systems.

The fifth edition of the DEBS Grand Challenge calls for applications centered around the processing of geospatial data originating from taxi rides. Applications in the challenge are provided with a continuous stream of records consisting of taxi rides with the objective to deriving two new streams answering the following two queries: (i) which are the top ten most frequent routes, and (ii) what are the top ten most profitable areas?

In this paper, we present the design and implementation of the two queries on top of StreamMine3G, an elastic and fault tolerant distributed ESP system. We highlight the challenges we solved to ensure correctness and to achieve scalability at the same time. We furthermore provide a performance evaluation of our solution in a single node and distributed setup.

The rest of the paper is structured as follows: In Section 2, we provide background information on ESP systems to contextualize our contribution and introduce the reader to the architecture and programming model of StreamMine3G, the ESP system we used to implement the DEBS challenge. In Section 3, we present our approach and provide implementation details for the two queries solving the challenge, followed by a performance evaluation of our approach in Section 4. Section 5 concludes and summarizes the contributions of the paper.

2. BACKGROUND
Although ESP systems exist already for more than a decade, they are currently facing a true renaissance. Inspired by the simplicity of MapReduce [6], a number of open source
ESP systems have evolved over the past years addressing the strong need for near-real-time computation using a simple interface as provided with the popular MapReduce implementation of Hadoop [4]. Examples for such systems range from Apache S4 [9, 1] (Yahoo!), Storm [3] (Twitter) to more recent developments such as Apache Samza [2] (LinkedIn). Although all of those systems originate primarily from industry, there exists also a number of projects that evolved from academia such as StreamCloud [7], SEEP [5] and StreamMine3G.

The previously mentioned ESP systems share an almost identical programming interface enabling developers to implement their own custom operators using user-defined-functions (UDF). The availability of those UDFs does not only increase the flexibility and productivity of the data analyst when tackling very specific problems such as given by the DEBS challenge, but also allows an easy migration of previously written MapReduce application to modern ESP systems for low latency data processing. In fact, many problems such as authors have presented in [8] are only efficiently solvable using ESP systems that support UDFs rather than CQL-based ones as those applications require custom data structures to maintain the complex application state in an efficient manner.

In addition to the operators, developers have to specify the order events should traverse the set of previously defined operators. Such an event flow is often expressed using an operator topology that is primarily used by the ESP system internally, in order to route the output of one operator to its successor according to the definition of the provided topology.

In order to achieve scalable event processing, the majority of those ESP systems supports data partitioning using a simple hash or some custom partitioner implementation. The partitioned input stream is then passed to multiple instances of the same operator where each operator instance processes only a subset of events from the original data stream in parallel.

Since ESP applications operate on unbounded streams of events, many of those applications require the use of state in order to aggregate or correlate multiple events within some user-specified (time or count-based) window. In order to prevent of loss of the accumulated state upon a shutdown, crash, or migration of the system, explicit state support in those ESP systems is essential.

In the following, we will provide detailed information about the architecture and the programming interface of StreamMine3G, a distributed ESP system used to solve this year’s DEBS challenge.

### 2.1 StreamMine3G System Architecture

StreamMine3G is a distributed ESP system which offers a MapReduce-like programming interface to its users. However, in contrast to the original MapReduce interface, users are also provided with a state object which allows the implementation of stateful operations in a convenient way. In order to free users from the burden of having to implement a complex and bridle locking scheme themselves in order to ensure consistent modifications of the provided state object, StreamMine3G guarantees that only accesses to disjoint parts of the state are executed in a concurrent fashion while accesses to the same state partition are strictly sequentialized. This is especially beneficial for data analysts as they can entirely focus on the data mining aspect of their application while leaving the parallelization aspect of the application to the ESP system itself in order to fully harness nowadays multi-core machines.

Once the operators are implemented, developers have to specify the desired flow of events by providing an operator topology to the system. In StreamMine3G, an operator can consume input from one or multiple upstream operators and produce output that is consumed by one or multiple downstream operators depending on the provided topology. This allows the composition of complex queries where intermediate results of one operator can be freely reused by multiple downstream operators saving costs for re-computation.

In order to achieve scalability, StreamMine3G supports stream partitioning where users can either opt to use the provided key-range partitioner, or alternatively implement their own partitioning scheme using a similar interface as in Hadoop [4]. Operator partitions in StreamMine3G are called slices and process only a subset of the incoming data as determined by the partitioner. However, it is also possible to provide a partitioner that broadcasts certain events to all downstream slices in order to implement a publish-subscribe system on top of StreamMine3G, or to implement a notification mechanism for downstream operator partitions about the closing of a window as we will describe more in detail in Section 3.

A StreamMine3G cluster comprises of a set of nodes where each node runs a single StreamMine3G instance. Such an instance can be envisioned as a container that can host an arbitrary number of slices. In addition to the slices, a StreamMine3G instance can also be assigned to host the so-called manager component. The manager is the central component in a StreamMine3G cluster and responsible for taking actions such as deploying and placing operator slices across the cluster, wiring slices based on the provided topology and monitoring the overall health status.

In order to monitor the health of the cluster, instances in a StreamMine3G cluster send periodic heartbeat messages to the manager component. Those messages contain performance related information of the underlying physical or virtual machine such as the CPU and network utilization as well as slice related performance metrics such as the current event throughput of incoming and outgoing events, and the accumulated state size of each slice an instance hosts. Using the provided information, the manager component can also detect overload or under-load situations and trigger compensation actions such as migrating slices to spare nodes in order to expand or contract the cluster in an elastic fashion. Furthermore, system recovery actions such as re-deploying of lost slices due to a node crash can also be triggered by the manager if desired.

In order to benefit from the elasticity and fault tolerance mechanism provided by StreamMine3G, users are required to provide appropriate code for serialization and de-serialization of the state object as the object can comprise arbitrary data structures. In case the user opted to use standard data structures, off-the-shelf frameworks such as boost.serialization can be used in order to provide the necessary code.

### 3. SOLUTION

In the following, we will provide implementation details of the two queries used to solve the challenge. The objective of
the queries is to provide timely information for cab drivers about the top ten:

1. most frequently driven routes within the past 30 min s, and
2. most profitable areas within the past 15 mins,

where the profit for an area is defined by the median of all fares (i.e., fare + tip) for rides that started or ended in an area divided by the number of empty taxis in that area. A taxi is considered empty if it had a drop off but no follow-up pick up within 30 minutes at its drop off location.

Both queries have in common to provide a top-k, however, the top-k computation is only supposed to produce an output if a change occurred with regards to contents or order of elements in the top-k set. Thus, the operators have to track changes, either for routes that can be uniquely identified using the pair: \((x_1, y_1), (x_2, y_2)\) where \((x_1, y_1)\) is the coordinate of the cell the ride departed and \((x_2, y_2)\) the destination cell, or for areas which can be defined just using: \(\{(x, y)\}\). Furthermore, routes and areas must be sorted according to the number of rides or profit in order to determine the desired top-k, respectively. Hence, the problem can be divided into two sub problems: change tracking and top-k computation.

In order to achieve scalability, the change tracking computation for both queries can be partition in the following way: For the first query, the pair: \(\{(x_1, y_1), (x_2, y_2)\}\) can be used as key to partition the routes used in the first query whereas \(\{(x, y)\}\) will serve as key to partition the areas of the second query. Data partitioning serves the following two purposes: First, it allows to scale the system without bounds, i.e., an almost infinite number of routes and areas can be tracked using an arbitrary number of processing nodes (i.e., horizontal scalability), and second, data can be processed in parallel increasing the overall throughput of the system as well as harnessing the power of multi-core machines (i.e., vertical scalability).

Figure 1 depicts the resulting query graph for the application at hand. As mentioned previously, the change tracking and top-k computation can be considered as independent problems and have been therefore carried out as separate operators in each query. Furthermore, the tracking operators (routes and profitable areas) are partitioned in order to achieve horizontal and vertical scalability. However, since the challenge requires to output a global top-k, only a single non-partitioned top-k operator instance is used in each query serving as sink as shown in Figure 1. In the following, we will describe the task of each operator used in the queries.

### 3.1 Source Operator

The purpose of the source operator is to perform a transformation of the incoming data originating from various sources such as sensors or mobile devices in a StreamMine3G compatible event format for processing. The input can be either provided via a simple network socket or a file stream.

Since the data set for the challenge is provided through a 32 GB large csv-file, the primary task of the source operator is to parse the records stored in the file and perform a data conversion to a format that can be consumed by both downstream operators, i.e., the routes and profitable areas tracker.

The conversion comprises of a discretization of the pick up and drop off location provided in latitude and longitude notation to coordinates of a predefined grid. Two grids are used, matrices sized 300 × 300 and 600 × 600 with cells measuring 500 or 250 meters in length and width for the first and second query, respectively. The grid covers an area of 22,500 square kilometers in total.

Since the two queries require different discretization for the given geographic coordinates, the source operator has to generate two output events per input event: One event with a transposition of the given coordinates to squares with 500 meters in length and width (query #1), and one using a more fine grained resolution with smaller squares measuring only 250 meters on each side (query #2). In order to avoid redundant and unnecessary network traffic, we use a custom partitioner that selectively routes events carrying the coarse grain discretized coordinates only to the routes tracker operator of the first query while the remaining events are transparently routed to the profitable areas tracker operator of the second query instead.

### 3.2 Query #1 - Route Frequencies

In the following, we will provide details with regards to the algorithms we used in order to carry out the functionality of the tracker and top-k operator of the first query. Listing 1 depicts the pseudo-code for the routes tracker operator.

#### 3.2.1 Tracker Operator

Contrary to the source operator which is implemented as a stateless operator as it solely performs a transformation of the input stream, a stateful implementation is required for the tracker operator, as it operates on a 30 mins sliding window. Therefore, the process() method is provided with a state object in addition to the incoming event as input as shown in Line 1 of Listing 1.

In a first step, the partitioning key is computed where we use a simple hash (Line 2) which comprises all four dimensions to uniquely identify a route in the given grid. Alternatively, a string can be used which however requires the usage of other hashing algorithms such as a Murmur Hash in order utilize data structures such as hash maps later on.

Although data partitioning allows a system to scale, it requires synchronization mechanisms in order to ensure semantic transparency, i.e., the output of the partitioned execution must be equivalent to the non-partitioned one. In order to ensure such semantic transparency in our application, we need to broadcast source events to all partitions of the tracker operator so that all tracker partitions are synchronously notified about a move forward in time in order to purge potentially outdated events from the predefined 30 mins sliding window. However, in order to track a route only at a single partition, such notifications events must be filtered and may not be considered for tracking at that spec-
Listing 1 Query#1 Routes Tracker (Worker) Operator

1: function process(rec, state)  
2:  key ← rec.xPp * 300^2 + rec.yPp * 300^2 + rec.xDf * 300 + rec.yDf  
3:  modifiedKeys ← 0  
4:  if rec.XPickup mod slices = sliceld then  
5:    ride ← new Ride(rec.key, rec.tsPp, rec.tsDf)  
6:  state.rides.append(ride)  
7:  rRec ← state.map[key]  
8:  rRec.count ← rRec.count + 1  
9:  rRec.tsPp ← rec.tsPp  
10:  rRec.tsDf ← rec.tsDf  
11:  modifiedKeys.insert(ride.key)  
12: if rRec.tsDf ≠ state.oldTsDf then  
13:   state.oldTsDf ⟨= tsDf  
14:   cutofftime ← rec.tsDf - 30 * 60  
15:   while cutofftime > state.rides.front().tsDf do  
16:     ride ← state.rides.front()  
17:     rRec ← state.map[ride.key]  
18:     rRec.count ← rRec.count - 1  
19:     rRec.tsPp ← ride.tsPp  
20:     rRec.tsDf ← ride.tsDf + 30 * 60  
21:     state.rides.pop()  
22:     modifiedKeys.insert(ride.key)  
23: for each key in modifiedKeys do  
24:   rRec ← state.map[key]  
25:   emit(rRec.key, sliceld))  
26: emit((SILENCEPROPAGATION, rec.tsDf))

cific partition. The filtering of such notification events is performed at Line 4 in Listing 1 where only events are considered for tracking whose x-component matches the following equation: \( x \mod \text{slices} = \text{sliceld} \) where \( \text{slices} \) is the number of partitions used for the tracker operator and \( \text{sliceld} \) the unique partition identifier. As an example consider the case where only two partitions are used, i.e., \( \text{slices} = 2 \). Using the previous equation, routes with an even x-component would be filtered by the partition with \( \text{sliceld} = 0 \) while odd ones by the partition with \( \text{sliceld} = 1 \).

In the case the x-component of the event satisfies the equation in Line 4, a Ride object is created which is composed of the unique key to identify the route and the timestamps of the corresponding incoming event, i.e., pickup (tsPp) and drop off time (tsDf) in Line 5. The Ride object is then appended to the tail of a linked list which is part of the state object (Line 6). The purpose of the linked list is to maintain all rides that occurred within the past 30 mins in the order of occurrence where the head of the list represents the oldest ride while the tail contains the most recent one. In addition to the linked list, the state object consists of a hash map (state.map) which keeps track of the number of rides for each route in the grid, and what incoming event (using the timestamp pair tsPp and tsDf) caused an update to the counter for the route using the route record (rRec) object.

The hash map entry is updated right after adding a new ride to the linked list. In order to do so, the corresponding route record object is retrieved from the hash map first using the route’s unique key, or transparently created if it did not exist (Line 7). In a next step, the counter is incremented (Line 8) and the timestamps from the corresponding event that modified the counter are copied (Line 9-10). Finally, the key of the route that has been updated is added to a modifiedKeys set (Line 11) in order to notify the downstream top-k operator about this update later.

While the first part of the tracker operator (Line 2-11) adds information based on the incoming event, the second part of the operator (Line 12-21) is solely responsible for updating information based on events leaving the window due to a time shift triggered by the new incoming event.

In order to determine if the incoming event triggered a move of the sliding window, the drop off time of the previous and the current event are compared first (Line 12). In case they differ, the new drop off time is stored in the state object to indicate the move forward in time (Line 13), and the new cutoff time is computed by subtracting 30 mins from the drop off timestamp of the incoming event (Line 14). Using the cutoff time, the linked list can now be traversed starting from the head in order to purge outdated rides (Line 15-21). In case the traversal encounters an expired element, the counter as well as the timestamps for corresponding route in the hash map are updated (Line 16-19). In a similar fashion as in the first part of the operator, the key for the route that has been updated is added to the set of modified keys (Line 22).

In a final step, the route records carrying the frequencies which have been recently updated are sent downstream to the top-k operator (Line 24). In case the event triggered no updates at all, a silence propagation event is sent in order to notify the downstream operator that the current event has been fully processed but did not generate an update.

### 3.2.2 Top-k Operator

The previously generated frequency updates are then processed using the top-k operator as depicted in Listing 2. In a first step, the type of the event is examined. In case the event is a silence propagation event and originated from the last partition of the upstream operator (i.e., holding the largest sliceld) (Line 2), the generateOutput method is called in order to determine if an output can be generated or not (Line 3). This constraint is an optimization to reduce the number of modification checks of the top-k set due to the partitioning of the upstream tracker operator. Note that the events arriving at the top-k operator are processed in a strictly deterministic fashion in StreamMine3G using a round-robin merge scheme so that the event with the largest sliceld represents the last event from the same epoch of the previously broadcasted source event.

In order to determine if the update (routeUpd) received from the tracker operator belongs to a route that has not yet registered at the top-k operator, a boolean flag (newRoute) is set to false first (Line 4).

The state for the top-k operator comprises of three different data structures: (i) a hash map (state.map) which contains a mapping of keys to route records in order to keep track of the frequencies, (ii) a custom linked list implementation (allRoutes) which store the previously computed top-k. The first linked list is used maintain an order across all elements required to determine the top-k. The advantage of our custom implementation is that insertions are of cost \( O(n) \) in worst case while removal of elements \( O(1) \). Moreover, since updates to list elements cause only marginal changes, i.e., counts are only incremented or decremented by 1, an update of an element requires only a single swap with its neighboring elements in order to maintain order which is far less costly than performing a whole sort on each update.

Once the top-k operator receives an update, a lookup in order to update the route frequency based on the incoming
event is performed first (Line 5). In case the route has not yet been registered, a new RouteRec object is being created and added to the hash map (Line 6). In addition to the creation, the newRoute flag is set to true indicating that the update corresponds to a newly tracked route. In a second step, the new information carried by the incoming event is stored by updating the fields of the RouteRec object appropriately (Line 9-11). This implies also the copying the timestamps from the event causing the update.

In case the incoming event reveals that no rides occurred on that specific route within the past 30 mins which is indicated by a count of 0 (Line 12), the route record (rRec) is purged in order to free memory where the rRec removes itself from the custom linked list allRoutes (Line 13), and from the hash map (Line 14).

For all other cases, the update method of the route record (rRec) object is called (Line 18) which causes a swap down or up in the custom linked list in order to re-establish order. In case the route did not exist before, the route is also prepended to the head of the custom linked list (Line 17) where it transparently moves the element up to its correct position.

Listing 3 shows the pseudo-code of the GENERATEOUTPUT method: In order to track if there were modifications to the top-k that has been outputted previously or not, a boolean flag is set to false first (Line 2). In a next step, an iteration over both data structures is performed, i.e., the oldTopK linked list as well as the allRoutes custom linked list (Line 3-8). If the traversal detects a mismatch for at least one of the first k-elements (Line 5-6), the modification flag is set to true (Line 7).

In case modification flag was set to true, the oldTopK list is cleared (Line 10) and filled with the current top-k from the up-to-date allRoutes custom linked list (Line 11-14). Finally, the current top-k is outputted to screen or file in case k or more elements exists in the allRoutes custom linked list (Line 18-21) including the latency (Line 22). In case the top-k does not contain at least k elements, NULL is outputted instead (Line 16).

3.3 Query #2 - Area Profit

In the following, we will describe the algorithm used in order to track the profit for an area. Since the computation of the area profit is way more complex than the tracking of route frequencies as presented in the previous query, additional data structures are required as the area profit is composed of (i) the amount of empty taxies for an area and (ii) the median of all fares for drop offs and pick ups that occurred in an area. An area is therefore represented using an area record (aRec) which contains a profit field, holding the value of the most recently computed median of all fares, the number of empty taxies, and the timestamp ((bsPd and tsDf) of the relevant event that caused a modification of one of the two fields, i.e., the profit or the number of empty taxies.

In order to efficiently compute the median of fares collected for an area, we use two linked list implementations: (i) a self-balancing double linked list named faresVal, and (ii) a regular linked list named faresTs which are both associated with an area record. While the first list is used to efficiently perform updates and lookups for the median value in an area, the second one is solely used to efficiently remove elements from a time window due to their expiration. Hence, the first one orders elements according to their fare amount (fare-t-tip) while the second list maintains its elements according to the drop off time.

The self-balancing double linked list maintains a pointer to the median element in the following way: An insertion of an element with a lower value than the current median one causes a shift of the pointer to the median to its left neighbor while an insertion to the left causes a shift to the right. Elements “bubble” up in the same ways as described for the allRoutes custom linked list where an element swaps the position with its neighbor to the right or left depending on the value used for comparison. Hence, the insertion of an element in the list is of cost $O(n)$ (worst case) while deletions and median lookups are of cost $O(1)$. Only if one of the median’s neighbors is being removed that shared the same value as the current median, the new median and its position within the list must be determined again which is of cost $O(n/2)$.

The amount of empty taxies is tracked using a one-dimensional hash map (taxiesId) where the taxi’s unique identifier is used as key and the timestamps of the pick up and drop off time as value. We will now describe how those data structures are updated for an area using Listing 4.
The previously described methods are used in the process method of the area tracker operator. The pseudocode for the operator is depicted in Listing 5 and consists mainly of two parts: In the first part (Line 2-16), new information from the incoming event is added to the appropriate data structures of the areas affected while the second part (Line 17-50) handles the removal of outdated events due to a window shift caused by a move forward in time.

Contrary to the first query where an incoming event causes an update only to a single route, an input event in the second query can cause an update to two areas at ones, i.e.,
the area where a customer was picked up and dropped off. We handle those two cases separately as depicted in the code of Listing 5: First, the drop off is handled in Line 4-9 while the pickup is handled in Line 11-16. In both cases, we first determine if the operator partition that received the incoming event is responsible for handling the area or not using the modulo computation in a similar fashion as in the first query (Line 3 & 10 in Listing 5). As a reminder, events are broadcast’ed to all tracker partitions, hence, areas which are handled by different partitions will still receive the same event in order to update the records appropriately ensuring correctness of the computation through semantic transparency.

Taxi drop offs and pick ups are handled as follows: First, the key/hash for the area is computed as shown in Line 4 & 11. Second, the associated area record (aRec) is retrieved from the hash map (state.map), or transparently created if not existing (Line 6 & 13). Next, the fare is added to the two linked lists faresTs and faresVal of the area using the ADD-TOAREA() method (Line 7 & 14) followed by an update of the number of empty taxis using the UPDTAXI_DROP() and UPDTAXI_PICKUP() methods (Line 8 & 15), respectively. In case those updates caused a modification of the previously recorded area profit, i.e., the median fare or the number of empty taxi changed, the keys of the corresponding areas are added to the modified keys set (Line 9 & 16).

As previously mentioned, the second part of the area tracker operator is responsible for updating the area profit based on expired events: First, the timestamp of the current event is compared with the previously recorded one in order to determine if a time shift occurred or not (Line 17). If so, the timestamp is updated (Line 18) followed by an iteration over all area records maintained by the respective operator partition (Line 19-50).

During the iteration, expired fares from the 15 mins window are removed first (Line 23-29) while in a second step, expired taxi drop off recordings from the 30 mins window are removed (Line 38-44). In case expired fares were encountered, the new fare median is compared with the previously recorded value and the profit field in the area record updated in case they do not match (Line 30-36). In a similar fashion, the record is updated if the amount of empty taxis changed due to expired taxi drop offs (Line 45-50). In case an area was updated, the corresponding key is added to the modifiedKeys set in order to send updates downstream to the top-k operator. After sending an update, area records that contain an empty count of taxis and a median fare of zero are removed from the hash map in order to free resources (Line 54-55). For the sink, we use the same implementation as for the first query, and hence omit the pseudo-code here.

4. EVALUATION

In this section, we present the results of various micro benchmarks we executed in order to assess the performance of our proposed solution. For the evaluation, we deployed the query using the single-node as well as the distributed execution mode option the StreamMine3G runtime offers.

In single-node setup, the query can be executed in a completely lock free manner as only a single instance per operator is used for carrying out the processing, hence, neither data partitioning nor a parallel execution of operators takes place. Furthermore, event dissemination across network as well as event ordering mechanisms, i.e., a deterministic merge, can be disabled which allows an easy assessment of the achievable baseline performance of the query at hand. The single-node execution mode can therefore be envisioned as a naïve implementation of the query which does not scale as no parallelization techniques are applied.

In contrast to the single-node setup, multiple operator partitions are deployed on a set of either physical or virtual machines in the distributed node with the benefit of a scalable data processing. However, the distributed setup comes with a price as data must be exchanged using TCP connections introducing additional latency. Moreover events originating from multiple upstream partitions must be merged in a deterministic fashion prior passing them to the next operator in order to ensure semantic transparency which adds additional overhead.

4.1 Single Node Execution

In the first set of experiments, we assessed the performance of our implementation using the single-node setup. For the experiment, we used different configurations in order to determine the overhead of individual operators and sub-queries as shown in Figure 2. The configurations are as follows: top-k freq + top-k area represents the execution of both queries, i.e., all operators while source only represents the execution using only the source operator. In top-k freq + top-k area w/o sinks is the execution of the full query except the sink operators that maintain the top-k sets while in top-k freq only and top-k area only, only one of the two queries is executed. For the last two configurations we disabled the sink operator either of the first or second query. Furthermore, we executed each configuration in single and multi-threaded mode. In multi-threaded mode, operators are executed in a pipelining fashion using producer-consumer queues rather than in a chained fashion.

The results of the experiments are depicted in Figure 2: When executing both queries, we can achieve a throughput of roughly 13.8 kEvents/s using a single-node and single-threaded execution whereas only 10.4 kEvents/s can be achieved using pipelining. In order to determine how quickly events can be read and parsed from the provided csv-file, we used solely the source operators in the second configuration. As shown in the Figure, roughly 260 kEvents/s can be processed which therefore represents the upper bound of the performance that can be achieved.

An interesting effect can be observed when disabling the sink operators for the two queries since we can achieve a noticeable higher throughput when using the multi-threaded version, hence, pipelining of the source and worker operators increases the throughput. In the following two configurations, we used only one of the two queries for execution. As shown in the graph, an order of magnitude higher throughput can be achieved when solely executing the top-k profitable areas query (78.2 kEvents/s) while only 16.3 kEvents/s are processed when running the top-k frequent routes query. As shown in the last two configurations, the sink operator for both queries have also different runtime behaviors: Disabling the top-k profitable area sink operator increases the throughput by an order of magnitude compared to having both queries running as shown in the first configuration. This can be easily explained as with every event, two areas are updated while in the first query only one element in the top-k set is updated at a time.
The focus of our work is to provide a scalable solution right from the beginning which can cope with arbitrary amounts of data. We achieve scalability through data partitioning and sub queries or individual operators.

Figure 2: Throughput for different configurations, i.e., sub queries or individual operators.

4.2 Distributed Execution

In the second set of experiments, we used the distributed setup in order to assess the scalability of our system. For those experiment, we partitioned the source operator in order to prevent the source being the bottleneck in our system, i.e., eight source operator instances are emitting events to the worker operators of the two queries concurrently.

In the first experiment, we varied the number of partitions used for each of the worker operators. The results are depicted in Figure 3. As shown in the graph, the throughput increases with the number of partitions. This is an effect of data partition since each partition is only responsible for a subset of routes/areas, i.e., hence with every event arrival, less routes/areas must be traversed in order to check for expired events due to a window move which therefore increases the throughput. Unfortunately, with an increase of partitions, also an increasing amount of events are sent to the sink operators which then becomes a bottleneck as one can see in the graph where the throughput saturates and finally decreases when using more than six partitions.

In the last experiment, we varied the size of StreamMine3G’s thread pool, i.e., the number of threads used for parallel execution. As shown in Figure 4, the throughput increases with the number of threads. However, as in the previous experiment, the single instance of the sink prevents further scaling of the system which even lowers the throughput beyond six threads due to heavy contention. However, contrary to the multi-threaded single node execution, multi-threading when applied on partitioned data can fully harness nowadays modern multi-core architectures.

5. CONCLUSION

In this paper, we presented the implementation of the DEBS 2015 challenge on top of StreamMine3G, a distributed, scalable, elastic and fault tolerant ESP system. The main focus of our work is to provide a scalable solution right from the beginning which can cope with arbitrary amounts of data. We achieve scalability through data partitioning and the use of the deterministic execution properties offered by StreamMine3G which allows us to provide semantic transparency regardless of the used degree of parallelism.

The evaluation of our approach confirms our believe that our solution is highly scalable and can efficiently use nowadays many core architectures with latencies under 1 ms for a single-node and around 20 ms in a distributed execution.

Acknowledgment

The research leading to these results has received funding from the European Community’s Seventh Framework Program (FP7/2012-2015) under grant agreement number 318809 (LEADS) and from the CAPES, DAAD and GIZ through the NoPa program (TruEGrid project, 2011-2013).

6. REFERENCES