Deterministic Real-Time Analytics of Geospatial Data Streams through ScaleGate Objects

Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou and Philippas Tsigas
Chalmers University of Technology, Gothenburg, Sweden
{vinmas, ioaniko, ivanw, ptrianta, tsigas}@chalmers.se

ABSTRACT
In this work we present the design, implementation and evaluation of our approach to solve the DEBS 2015 Grand Challenge. Our work studies how ScaleGate, a concurrent implementation of a recently proposed abstract data type, that articulates data access in parallel data streaming, can be leveraged to partition the Grand Challenge analysis among an arbitrary number of processing units. ScaleGate aims not only at supporting high throughput and low latency parallel streaming analysis, but also at guaranteeing deterministic processing, which is one of the biggest challenges in parallelizing computation while maintaining consistency.

Our main contribution is a new perspective for addressing the high throughput, low latency and determinism challenges of parallel data streaming by letting such challenges permeate the entire analysis framework, down to its underlying shared data objects. As a result, we propose shared data objects that balance independent actions among processing threads in order to guarantee high throughput, while providing the necessary synchronization for deterministic processing.

1. INTRODUCTION
This paper presents the design, implementation and evaluation of our approach to solve the DEBS 2015 Grand Challenge, which focuses on the analysis of high volume geospatial data (more concretely, on travel reports generated by taxis in New York City during the year 2013).

The motivation behind the challenge, shared by distributed and parallel streaming analysis frameworks, is the need for high processing throughput and low processing latency [7, 13, 6, 1]. Towards this direction; the continuously wider adoption of many-core and heterogeneous systems (spanning from high-end servers to embedded single-board devices) is an enabler for high processing throughput and low processing latency to happen. However, the possibilities enabled by such systems come at a price. More concretely, there are challenges that come along with identifying which parts of the computation can be parallelized efficiently, as well as with ensuring that the analysis’ results are produced deterministically, independently of how a given framework is deployed (e.g., independently of which and how many heterogeneous devices run the analysis). It should be noticed that the need for deterministic analysis is further exacerbated when data sources are distributed (e.g., when each taxi represents a data source, as in the Grand Challenge) as the latter might generate data with different granularities and different speeds [7].

As we discuss in [4, 3, 9], the underlying shared objects, which the different processing units (e.g. data streaming operators) utilize, constitute a key enabler in achieving high throughput, low latency and deterministic execution in a distributed and parallel analysis framework. Such data and computation articulation points can enable fine-grained, non-blocking and consistent synchronization that allows for high parallelism, with improvements in throughput up to one order of magnitude [4]. Quoting from Michael [11], “the choice of data structures is one of the most important decisions in designing a non-blocking environment.”

We recently introduced ScaleGate [4, 9], a new abstract data type together with its concurrent implementation, to facilitate parallel data streaming, allowing both data exchange and synchronization while guaranteeing determinism. In this paper we employ ScaleGate in our solution and study how it can be leveraged to solve the DEBS 2015 Grand Challenge. As we discuss in the following sections, ScaleGate enables for pipeline parallelism in the analysis defined by the challenge and also allows the latter to be partitioned and parallelized among an arbitrary number of processing units. Hence, it can constitute one of the fundamental building blocks of parallel and concurrent data streaming frameworks.

The rest of the paper is organized as follows. We overview the challenge in Section 2. We present briefly the ScaleGate abstract data type in Section 3. We overview the proposed solution in Section 4, we discuss it in detail in Section 5 while we present its evaluation in Section 6. Section 7 concludes the paper.

2. CHALLENGE OVERVIEW
In this section, we overview the DEBS 2015 Grand Challenge (we refer the reader to [5] for a comprehensive discussion). The challenge focuses on the analysis of high volume geospatial data, and more concretely, on the analysis of a stream of travel reports (also referred to as tuples in the following) generated by taxis in New York City. Such tuples are to be analyzed in order to find the top-10 i) most frequent routes, and ii) areas that are the most profitable for taxi drivers. Both queries are continuously computed (also referred to as tuples in the following) generated by taxis in New York City. Such tuples are to be analyzed in order to find the top-10 i) most frequent routes, and ii) areas that are the most profitable for taxi drivers. Both queries are continuously computed (e.g., independently of which and how many heterogeneous devices run the analysis). It should be noticed that the need for deterministic analysis is further exacerbated when data sources are distributed (e.g., when each taxi represents a data source, as in the Grand Challenge) as the latter might generate data with different granularities and different speeds [7].

Tuples share the attributes presented in Table 1. The pick up and drop off locations of each tuple are expressed by means of cell
grids (of different sizes). The metropolitan area covered by the data is partitioned in 300x300 Q1 cells (of 500^2 m^2) and 600x600 Q2 cells (of 250^2 m^2), numbered as presented in Figure 1. The challenge assumes tuples are delivered in timestamp order, based on their drop off time, and consumed by the two following queries.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taxi ts</td>
<td>Unique identifier of the taxi</td>
</tr>
<tr>
<td>Q1_pickupCell</td>
<td>Query 1 pickup cell</td>
</tr>
<tr>
<td>Q1_dropoffCell</td>
<td>Query 1 dropoff cell</td>
</tr>
<tr>
<td>Q2_pickupCell</td>
<td>Query 2 pickup cell</td>
</tr>
<tr>
<td>Q2_dropoffCell</td>
<td>Query 2 dropoff cell</td>
</tr>
<tr>
<td>amount</td>
<td>Fare and tip at travel completion</td>
</tr>
</tbody>
</table>

Table 1: Tuples’ attributes.

Figure 1: Grid partitioning the metropolitan area covered by the data and cells referring to a sample travel report.

**Frequent Routes** This query aims at computing the top-10 most frequent routes during the last 30 minutes (each route defined as a pair (Q1_pickupCell, Q1_dropoffCell)). The score of each route is defined as the number of tuples associated with the specific route within a given time window. If t denotes the tuple being processed, the 30 minute window contains tuples t’ | t.ts ≥ t.ts - 30 minutes.

**Profitable Areas** This query aims at computing the top-10 most profitable areas (each defined as a Q2 cell). The score of each area is defined as its profit divided by its number of empty taxis. Given a Q2 cell C, the profit is the median value of the amount attribute (Fare+Tip) for tuples observed during the last 15 minutes that report C as the value of Q2_pickupCell. At the same time, the empty taxis is the number of tuples observed during the last 30 minutes reporting C as the Q2_dropoffCell value, given that each of the taxis has not appeared in another tuple with a different dropoff location later during the same period.

It is useful to point out that for both queries, the Grand Challenge assumes updated frequent routes and profitable areas should be reported for each individual input tuple that changes them. As we explain subsequently, this requirement introduces an important challenge in the parallelization of the analysis, considering that each input tuple should be observed by all the processing units accessed by the queries (with modified names and merged attributes too, as for Fare and Tip).

The image shows a sample portion of the grid covering the metropolitan area of New York City. The grid is not aligned with the coordinates provided by the Grand Challenge and the sample travel report is not contained in the data.

(e.g., threads or data streaming operators) maintaining the top-10 frequent routes and profitable areas observed in the time window.

3. **BACKGROUND - THE SCALEGATE ABSTRACT DATA TYPE**

As introduced in Section 1, the challenges and motivations behind parallel data streaming are usually driven by the need for high processing throughput and low latency. Nevertheless, with the increasing adoption of heterogeneous multi-core and many-core systems, deterministic processing of data streams is also desirable in order for the analysis result to be (1) independent of the number of sources delivering the streams and (2) independent of the number of processing units operating concurrently and in parallel on the streams. Based on the definitions of deterministic processing provided in [8, 7, 4], we can state that:

**Definition 1.** The property of deterministic processing in data streaming analysis guarantees that, by consuming the tuples delivered by an arbitrary set of input streams, the parallel analysis run on such tuples produces exactly the same output that would be produced by its centralized counterpart.

As discussed in [4], a condition to enforce deterministic parallel analysis is to process the tuples delivered by distinct streams in timestamp order. In the scenario of the DEBS 2015 Grand Challenge, the distinct streams could be represented by streams of travel reports generated by the taxis (or collected from each area). In [3], the authors introduce the concept of ready tuple as follows:

**Definition 2.** Let t_i be the i-th tuple in timestamp-sorted stream j. Tuple t_i is ready to be processed if t_i.ts ≤ merge_i, where merge_i is the minimum among the latest timestamps from each timestamp-sorted stream j, i.e.

\[
merge_i = \min_j \{\max_k (t_{i,k}.ts)\}
\]

Based on the above, deterministic processing can be enforced if operators consume timestamp-sorted ready tuples from their input streams.

In the literature, the merging of distinct input streams into a single stream of ready tuples has been discussed in the context of parallel and distributed SPEs [12, 7, 2]. The STREAM Input Manager [12] performs such merging for multi-source data streaming operators. It relies on heartbeats (sent by each data source at given intervals) to perform the merging periodically. In parallel SPEs like [8, 7], special data streaming operators (Input Mergers) enforce the merging of the incoming timestamp-sorted input streams. Merging of sorted input streams is also employed in the context of replica-based fault tolerant SPEs (with the UNION operator) where primary
and replica instances are enforced to deterministically process tuples in order to have equal states [2]. Independently of whether tuples are sorted periodically or upon reception of a new incoming tuple, these approaches only allow for sequential access patterns, having a single thread retrieving the tuples delivered from the different input streams and merging them. The merging can result in time complexity linear in the number of inputs, thus constituting a potential bottleneck for the analysis scalability.

In order to allow for parallel analysis of timestamp-sorted streams of ready tuples (while preventing the merging of tuples from constituting a bottleneck of the analysis) we introduced in [9] the ScaleGate data structure\(^\text{\textsuperscript{1}}\). ScaleGate guarantees properties essential for concurrently merging streams at the articulation points where data and threads meet, while it integrates the necessary synchronization to allow multiple threads to consume ready tuples concurrently. As presented in Figure 2, ScaleGate allows for an arbitrary number of source threads to deliver the tuples forwarded by an arbitrary number of timestamp-sorted input streams. At the same time, it allows for arbitrary number of reader threads to consume the same sequence of timestamp-sorted ready tuples. The interface of ScaleGate provides the following methods:

- \texttt{addTuple(timestamp, tuple, sourceID)}: which allows a tuple carrying the given timestamp from the source thread sourceID to be merged by ScaleGate in the resulting timestamp-sorted stream of ready tuples.
- \texttt{getNextReadyTuple(readerID)}: which provides to the calling reader thread readerID the next earliest ready tuple that has not yet been consumed by the former.

4. PARALLELIZING THE ANALYSIS BY EXPANDING THE TOP-K QUERIES

In this section, we outline the idea behind our solution, discussing how the analysis for the queries \textit{Frequent Routes} and \textit{Profitable Areas} is distributed among \(n\) Processing Units (PUs), each responsible for approximately \(1/n\) of the overall analysis. As discussed, the ScaleGate data structure is the key enabler for the distribution of the analysis among the \(n\) PUs.\(^\text{\textsuperscript{1}}\)

The idea behind the workload-partitioning scheme that we propose is based on the observation that the top-10 values for a set of elements can be computed by first computing the top-10 values of disjoint partitions of such a set and subsequently choosing the global top-10 by merging the results from the disjoint partitions. In this regard, the parallelism of the analysis can be enhanced by increasing the number of top frequent routes and profitable areas being observed (and thus allowing for such analysis to be carried out by an increasing number of PUs) and subsequently computing the global top-10 frequent routes and profitable areas. It should further be noted that, in order to increase the pipeline parallelism, the top-10 most frequent routes and profitable areas observed for disjoint routes and areas partitions should be merged into the global top-10 only when a change is observed in at least one of the partitions (we discuss how this is done in Section 5.3).

In the proposed solution, each of the PUs computes the top-10 most frequent routes for its share of routes (\(1/n\) of the overall routes), which are merged in sorted order to produce the overall top 10 most frequent routes for the period under consideration. Similarly, each of the PUs computes the top-10 most profitable areas for its share of areas (\(1/n\) of the overall areas), while the overall top-10 most profitable areas are computed by merging them as in the previous query case. Additionally, if the analysis of the overall routes and areas is split among PUs, exactly one PU is expected to maintain the frequency of a given route or the profitability of a given area in order for results to be correct (i.e. it is paramount to avoid a valid partitioning scheme). Furthermore, since the global top-10 frequent routes and profitable areas are computed from those reported by each PU (possibly changing for each new incoming tuple), it is key to feed input tuples and collect PUs’ results correctly. While discussing how such challenges are addressed, we refer the reader to Figure 3 for a sample deployment of the proposed solution. In this example, the overall analysis is distributed among four distinct PUs.

\textbf{Feeding input tuples to PUs.} As discussed in Section 2, the Grand Challenge specifies that updating top-10 frequent routes and profitable areas should be produced upon processing of each input tuple that modifies the values. In this sense, when multiple PUs are in charge of computing the top-10 routes and areas, each PU should process each incoming tuple in a deterministic fashion. To

---

\(^\text{1}\)We refer the reader to [9] for a detailed discussion of ScaleGate design and implementation choices.

Figure 3: Overview of the system architecture
provide an example of why this is required, assume that 2 PUs are in charge of running the analysis and consider tuple \( t \) as the next tuple to be processed by the 2 PUs. Also, assume that \( t \) causes a change in the global top-10 frequent routes and profitable areas. In order for the global top-10 results to merge from the 2 PUs to be correct, each PU’s top-10 frequent routes and profitable areas should only refer to tuples \( t' \) s.t. \( t' \.ts \leq 30 \) minutes at the time \( t \) is processed. To achieve this, the ordered processing of input tuples for an arbitrary number of PUs can be enforced by feeding them with the timestamp-sorted stream of ready tuples generated by ScaleGate. We refer to such a ScaleGate instance feeding the multiple PUs as \( SG_{in} \).

Partitioning routes and areas among PUs. As introduced earlier in this section, exactly one PU should be responsible for each route and area. Based on the grid partitioning into Q1 and Q2 cells, squares composed by four Q2 cells fall within exactly one Q1 cell (in the example of Figure 1, Q2 cells 1.1, 2.1, 1.2, 2.2 fall within Q1 cell 1.1). This observation can help in partitioning the routes and areas among \( n \) PUs. By assigning the routes originating from each Q1_pickupCell to exactly one PU, it is also possible to assign the four Q2 cells (i.e., areas) falling within the Q1_pickupCell to the same PU. Henceforth, we consider a PU as responsible for a Q1_pickupCell if it is maintaining the frequency of the routes originating from that cell and the profitability of the areas within the given Q1_pickupCell. While we focus on such a scheme in the following section, we would like to point out that any partitioning scheme resulting in exactly one PU responsible for a given route or area can be utilized.

Collecting and merging PUs results. By iterating through the same stream of ready tuples provided by the \( SG_{in} \), each PU is able to produce its resulting top-10 frequent routes and profitable areas based on each incoming tuple deterministically. In order to produce the global top-10 most frequent routes and profitable areas for input tuple \( t \), though, all the respective top-10 values from each PU must be gathered and merged. It should be observed that such merging should happen after all the top-10 values for tuples \( t' | t'.ts < t.ts \) have been merged and before the merging of the top-10 values for tuples \( t' | t'.ts > t.ts \). Also in this case, a ScaleGate instance (which we refer to as \( SG_{out} \)) can be leveraged to merge all the top-10 frequent routes and profitable areas produced by an arbitrary number of PUs into a timestamp-sorted stream of ready top-10 values. Subsequently, a dedicated Output Unit (OU) can be defined to merge such values into the queries Frequent Routes and Profitable Areas results. As we discuss in the following, each PU forwards the updated top-10 frequent routes and profitable areas only when the latter change. Since a top-10 value is ready only if each PU forwards at least one top-10 value, heartbeat\(^s \) \[12\] are leveraged to ensure top-10 values become ready within a certain period of time, even if the tuples consumed by one or more PUs do not change their top-10 most frequent routes or profitable areas.

5. SOLUTION IMPLEMENTATION

In this section, we present the implementation details for the proposed solution to the DEBS 2015 Grand Challenge. We first explain in detail the computation performed by each PU in Section 5.1. Subsequently, we discuss how the results produced by each PU are merged into the outcome for the two queries in Section 5.2. Finally, we discuss in Section 5.3 the data structures we designed and implemented for each PU to efficiently maintain relevant information about a time window from which we extract the top most frequent routes, profitable areas and keep track of their changes (in order to forward the latter only when a change in the top-10 values is observed for a new incoming tuple). As we explain in the following sub-sections, each PU is producing the results for both queries Frequent Routes and Profitable Areas concurrently.

5.1 Processing Unit Implementation

Algorithm 1 summarizes the processing steps performed by each PU. Each PU processes incoming tuples and produces results for queries Frequent Routes and Profitable Areas at the same time. For ease of exposition, we present a simplified algorithm, without cov-
erating its runtime optimizations. Also, rather than discussing the algorithm sequentially, we discuss separately how it maintains the tuples contributing to the active window, how it updates the results for query \textit{Frequent Routes} and how it updates the results for query \textit{Profitable Areas}.

\textbf{Maintaining windows’ tuples}. Each PU iterates through all the \textit{ready} tuples in timestamp order and aggregates them into the results of the \textit{Frequent Routes} and \textit{Profitable Areas} queries. Given the queries’ window model, each input tuple contributes to multiple windows (e.g., a tuple received at time 08:00:00 contributes to windows [07:30:00.08:00:00], [07:30:01.08:00:01], and so on). More precisely, if WS denotes the window size (i.e., 30 minutes for query \textit{Frequent Routes}) and WA denotes the window advance (1 second, the smallest time granularity defined by the travel reports), each tuple contributes to \{WS/WA\} windows (i.e., 1800 windows for query \textit{Frequent Routes}). At the same time, given the number of Q1 cells and Q2 cells defined by the queries’ grids, up to 300^3 routes and 600^2 areas could be observed at the same time by the n PUs. Based on this observation, rather than maintaining all the currently active windows (for different starting timestamps, routes and areas) and discarding each input tuple after the windows to which it contributes have been updated, we instead maintain the tuples contributing to the current window (in order to remove their contribution once they expire) and rely on a single window for each route and area, which helps in reducing the PUs’ memory footprint.

In Algorithm 1, we use two distinct lists \texttt{last15Mins} (L. 3) and \texttt{prev15Mins} (L. 4). The former maintains the tuples received in the last 15 minutes. The latter maintains the tuples received from 15 to 30 minutes ago. We use two different lists in order to be able to distinguish the tuples that, while contributing to the \textit{Frequent Routes} query and the empty taxi metric of query \textit{Profitable Areas}, no longer contribute to the profit metric for query \textit{Profitable Areas} (as discussed in Section 2). Upon reception of each tuple, the latter is stored if it contributes to the PU responsible for the route or area it refers to (L. 19 or L. 26). Subsequently, tuples received more than 15 minutes ago are shifted from \texttt{last15Mins} to \texttt{prev15Mins} (L. 38 and 42). Finally, tuples received more than 30 minutes ago are removed from \texttt{prev15Mins} (L. 46).

\textbf{Updating the \textit{Frequent Routes}}. Each PU starts its processing iteration by retrieving the next \textit{ready} tuple from \texttt{SG}\textsubscript{in} (L. 11). Additionally, each PU maintains a tuple counter for the tuples being processed, and increases it for each new \textit{ready} tuple (L. 8,12). Notice that, since all PUs access the \textit{ready} tuples in the same order, such counter is consistent among the different PUs.

In order to update the \textit{Frequent Routes} query, the frequency of the route to which each input tuple \(t\) contributes is increased by one when the latter is processed by the PU responsible for \(t.Q1\)\textsubscript{pickupCell} (L. 15-16). In our implementation, all the tuples’ attributes presented in Table 1 are defined as numeric values (except for the taxi attribute). For this reason, we use the \texttt{modulo} operation (as in L. 15) to check whether a PU is responsible for the drop off (or pick up) cell carried by each incoming tuple.

Subsequently, the contribution of every tuple received more than 30 minutes ago is removed from all the respective routes. Notice that the latter update is performed not only by the PU responsible for \(t.Q1\)\textsubscript{pickupCell}, but by any PU storing tuples \(t'(t'.ts\leq ts-30\) min (L. 45-48). Since a tuple is maintained by a PU if the latter is responsible for its pick up or drop off Q1 cell (as explained when discussing the updating of the \textit{Profitable Areas} query), each PU decreases the routes’ frequencies only for the tuples that previously contributed to them (L. 47). Finally, the updated top-10 routes of each PU are computed and added to \texttt{SG}\textsubscript{out} if a change has been detected in any of the previously maintained top-10 frequent routes or if the heartbeat threshold, the maximum number of consecutive non-changing top-10 frequent routes that are not added to \texttt{SG}\textsubscript{out}, has expired (L. 53-54,56). The tuple counter is used instead of the latest tuple timestamp when adding the result to \texttt{SG}\textsubscript{out}. This is done in order to allow for updated top-10 routes and areas to be produced for each incoming tuple, as requested by the Grand Challenge (discussed in Section 2).

\textbf{Updating the \textit{Profitable Areas}}. The update of query \textit{Profitable Areas} is more challenging than the update of query \textit{Frequent Routes}, because of the different window sizes for the median profit and empty taxis metrics (15 and 30 minutes, respectively) contributing to the score of each area and that, for each incoming tuple \(t\), the profit metric refers to \(t.Q2\)\textsubscript{pickupCell} while the empty taxi metric refers to \(t.Q2\)\textsubscript{dropoffCell}. Upon reception of a new input tuple \(t\), each PU updates the profit metric and the resulting area score if the pickup \(Q2\)\textsubscript{pickupCell} falls within one of the Q1 cells for which the PU is responsible (L. 15,17-18). Similarly, each PU updates the empty taxi metric, storing the last drop off cell of the taxi, and the resulting area score if the dropoff \(Q2\)\textsubscript{dropoffCell} falls within one of the Q1 cells for which the PU is responsible (L. 21-24). If the previous \(Q2\)\textsubscript{dropoffCell} attributed to \(t\) differs from the current one, the contribution of \(t.tazi\) to the empty taxis of the previous \(Q1\)\textsubscript{dropoffCell} must be removed by the PU responsible for such cell (L. 30-33).

Similar to query \textit{Frequent Routes}, once the areas’ profit and empty taxis metrics have been updated given the input tuple \(t\), the contribution of stale tuples is removed by each PU. The profit metric is updated removing the contribution of tuples received more than 15 minutes before (L. 37,39-41). Subsequently, the empty taxi metric is updated removing the contribution of tuples received more than 30 minutes ago (L. 45,49-51). As done for the \textit{Frequent Routes} query, the updated top-10 profitable areas of each PU are computed and added to \texttt{SG}\textsubscript{out} if a change has been detected in any of them or if the heartbeat threshold is exceeded (L. 53,55-56).

\subsection{Output Unit Implementation}

In this section, we present how the top-10 routes and areas computed by the n PUs (based on the respective partitions of routes and areas) for each incoming tuple \(t\) are merged into the results for queries \textit{Frequent Routes} and \textit{Profitable Areas}. The algorithm for such merging, carried out by the Output Unit OU, is presented in Algorithm 2.

\begin{algorithm}
\begin{algorithmic}
\State \texttt{ScaleGate} \texttt{SG}\textsubscript{out}; // Output ScaleGate,
\State \texttt{HashMap Top10s;} // shared with all the PUs
\State long prevCounter; // counter of previous tuple
\State \texttt{run()}
\State \texttt{while\( (\text{running}) \)}
\State \{\texttt{counter,puID,top10routes,top10areas} = \texttt{SG}\textsubscript{out}.\texttt{getNextReadyTuple(0)}\}
\State \texttt{if counter = prevCounter} \texttt{Top10s.get(puID).put((top10routes,top10areas));}
\State else \texttt{output(merge(Top10s));}
\end{algorithmic}
\caption{Output Unit - Implementation}
\end{algorithm}

The OU starts its computation by retrieving the next \textit{ready} top-
10 routes and areas computed by any of the PU from the $SG_{out}$ (L. 68). As presented in Algorithm 1, each PU forwards such information by means of a (counter,puID,top10routes,top10areas) tuple (L. 56). Hence, the OU checks whether the retrieved counter is equal to the previous retrieved one. In such a case, the retrieved tuple is stored in $top10s$ for the corresponding PU identifier puID (L. 70-71). In the opposite case (L. 73), the merged top-10 areas and routes, computed by method merge (not shown in the code) are forwarded to the output (written to disk in our implementation). Since the top-10 frequent routes and profitable areas computed by each PU for a given tuple $t$ are reported if the latter results in a change, the OU relies on the previous top-10 most frequent routes and profitable areas reported by a PU if they are not modified by $t$.

Since $SG_{out}$ delivers ready tuples (based on the counter field in this case), all tuples sharing a counter $c$ have already been delivered when the OU accesses the first tuple with counter $c + 1$. That is, the ScaleGate guarantees that all the partial top-10 routes and areas from each of the $n$ PUs are indeed delivered (and merged deterministically into queries $Frequent Routes$ and $Profitable Areas$ results) before any tuple with a higher counter is delivered (as discussed in Section 4).

### 5.3 Efficient update and retrieval of top-10 frequent routes and profitable areas

In this section, we discuss the mechanism that allows each PU to (1) efficiently update routes’ and areas’ scores based on the tuples being processed (and the ones becoming stale and no longer contributing to the queries’ windows) and (2) efficiently retrieve the top-10 frequent routes and profitable areas (whenever the latter change). We discuss the mechanism for routes and areas separately.

#### A) Maintaining top frequent routes

![Diagram of routes and scores](image1)

- Route 1
- Route 2
- Route n
- Top ten frequencies

- List of route frequencies
- List of routes with score 1
- List of routes with score 2
- List of routes with score $s$

#### B) Maintaining top profitable areas

![Diagram of areas and scores](image2)

- Area 1
- Area 2
- Area n
- Top ten profitable areas

#### Top-10 Most Frequent Routes.** For computing the top-10 frequent routes at each PU, the routes are split into buckets according to their current frequency. In each bucket, the routes (i.e., pointers to the latest tuples kept in the current window) are sorted in timestamp order of their causing effect. The frequency value of each route $R$ maintained by a PU needs to be increased or decreased for each incoming or stale tuple referring to $R$ respectively. A hashable is used in order to locate in amortized constant time a route along with its current frequency value. Thus, to increment or decrement the frequency count of a route, it is enough to query the hash table and then move the route to the next or previous bucket. The top-10 query (or generally top-$k$) can be easily answered by traversing $k$ nodes starting from the highest bucket towards the lower ones. Furthermore, it is easy to keep track of changes affecting the top $k$ elements (by monitoring the buckets being modified) in order to produce updated top-$k$ scores only when incoming or stale tuples change them. Figure 4.A presents the different data structures used to efficiently maintain the routes’ score and compute the top-10 frequent routes.

#### Top-10 Most Profitable Areas.** Similarly as done for the top-10 frequent routes, we look for an efficient way of updating the profitability of each area upon reception of an incoming tuple (or when stored tuples become stale) while keeping track of the top-10 most profitable ones (as discussed in Section 2, the profitability of an area depends on its empty taxis and profit metrics. While the former is similar in spirit to the route’s frequency score (i.e., it can be maintained by a per-area dedicated counter), the latter is challenging to maintain as it depends on the median value of the amount attribute carried by tuple referring to a given area in the last 15 minutes. There is a common algorithm for keeping track of the median of a stream of data, by using two heaps, one for maintaining the items that are greater and one for maintaining the ones that are less than the current median, respectively. Nevertheless, by focusing only on the addition of new values, such an algorithm is not efficient for a sliding window computation, where tuples become stale and have to be removed from any of the two heaps, thus leaving them unbalanced. Based on this observation, our algorithm relies on a skip-list for each area, ordered according to the amount attribute of the tuples, and an index to the current median value. The skip list data structure guarantees with high probability logarithmic time in the insertion of a node. Removing a stale tuple from the skip list can happen in constant time if the address of the node (which is in the skip list) is kept along with each tuple in the current window. Updating the median after an insertion (or removal) is merely a comparison of the new (or removed) node’s amount attribute with the current median and updating the current median appropriately. Overall, this efficient implementation, adapted for sliding-windows, results in a logarithmic time (high probability) update and maintaining of the top-10 profitable areas. Figure 4.B presents the different data structures used to efficiently maintain the areas’ score and compute the top-10 ones. In this case, a priority queue is used to keep track of the changes in the top-10 most profitable areas.

### 6. EVALUATION

In this section, we evaluate our implementation for the 2015 Grand Challenge solution. The challenge requests the proposed solutions to be delivered in a virtual machine with 4 cores and 8GB of memory. Based on this request, we opted for a multi-threaded process implementation of our solution. This implementation choice gave us valuable insights not only on its performance, but also on the best practices to run it both in shared-memory and shared-nothing environments.

We first present the hardware and software setup. Subsequently, we discuss in detail the proposed solution by focusing on (1) the applicability of the proposed parallelization method and (2) its per-
performance in terms of execution time and processing latency, further discussing the scalability of the ScaleGate data type and the tuples’ processing cost for the different threads running within the process.

Hardware and software setup
We run our evaluation on a machine equipped with two 3.40 GHz Intel Xeon Processor E5-2687W v2 (25M Cache and 8 cores each) and 16 GB of memory. The proposed solution is implemented in c++11 and also relies on Intel’s TBB library (Version: 4.0+233-1). In our implementation, a single dedicated converter thread is responsible for reading the input file, converting each line into the corresponding input tuple and adding it to the $SG_{in}$ ScaleGate. Also, each PU is run by a dedicated thread, reading tuples from the $SG_{in}$ and storing results in $SG_{out}$. Finally, a separated thread runs the OU, merging the top-10 routes and areas produced by each PU and storing results to disk.

In the performance evaluation, the execution time is measured from the creation of the process to its end. The per-tuple processing latency is measured as the time interleaving the reading of a tuple to the point in time when the corresponding output tuple is written to disk. Presented results are averaged over 5 runs. In all the experiments, we discuss the performance and scalability up to 8 PUs, as 8 is the maximum number of PUs that can be executed within the same CPU socket. When running 8 PUs, the different process’ threads are actually spanning two CPU sockets, and thus allow us to show the degradation incurred by the inter-socket communication over the intra-socket one.

Applicability of the parallelization approach
In the proposed parallelization approach, the overall computation is split among arbitrary numbers of PUs by assigning distinct routes and areas to them (as discussed in Section 4). In order to evaluate the applicability of this approach, we first analyzed the number of distinct routes and areas that can be observed over the largest window of time defined by the queries (30 minutes). The results are presented in the box plots of Figure 5. As it can be observed, the median value for the distinct number of routes is approximately 5100 while the median value for the distinct number of areas is approximately 880. These numbers suggest that the proposed work-load partitioning scheme can indeed be leveraged. To further evaluate the applicability of this approach, we also counted the number of tuples (e.g., routes or areas) responsibility of each PU for an increasing number of PUs. The results are presented in Figure 6. As it can be observed, each PU is responsible for approximately $1/\sqrt{n}$ tuples when $n$ PUs execute the analysis in parallel. This result is expected, as the partitioning scheme results in each tuple being processed by up to 2 PUs, the one responsible for its pick up area and the one responsible for the drop off one (Section 5.1).

Performance evaluation
In this experiment, we evaluate the performance of our evaluation, in terms of processing throughput and latency, for an increasing number of PUs. The different observed execution times are of 650, 670, 700 and 1170 seconds for 1, 2, 4 and 8 PUs, respectively. The resulting throughput and latency results are presented in Figures 7 and 8. As it can be observed, the throughput is approximately stable around 260,000 t/s for parallel executions up to 4 PUs, while it degrades when running 8 PUs (because of the different instantiated threads spanning across the two CPU sockets). At the same time, while the throughput remains approximately stable, a decreasing latency can be observed both for queries Frequent Routes and Profitable Areas, down to 0.09 seconds\(^4\) (also in this case, a degradation is observed when running 8 PUs).

At a first glance, these results seem not to match the expected scalability in Figure 6. Nonetheless, after a thorough analysis of the different processing costs for the running threads, these results actually indicate that a balanced distribution of the proposed parallelization approach among the available cores of a given shared-memory or shared-nothing setup can be achieved by instantiating 2 PUs for each converter thread. Moreover, they also provide insights about how to distribute threads running either the converter or the PU task among the cores of a given setup. We discuss these claims in the following.

Scalability of the ScaleGate data structure. In this micro-benchmark, we measure the rate at which tuples added to a Scale-
7. CONCLUSION AND FUTURE WORK

In this paper, we present the design, implementation and evaluation of our approach to solve the DEBS 2015 Grand Challenge, by leveraging the ScaleGate data structure. As we discuss, the functionality and API provided by ScaleGate allows for the parallelization of the Grand Challenge’s Frequent Routes and Profitable Areas queries among an arbitrary number of processing units. At the same time, ScaleGate enables for deterministic processing of the input data, independently of the number of streams delivering the latter and independently of the actual number of processing units consuming them. Overall, we show that, even with a machine having access to a reduced number of cores, the data produced by thousands of taxis over a period of one year can be analyzed in approximately 11 minutes, with a processing throughput of approximately 260,000 t/s and processing latencies of 0.09 seconds.

The design and implementation of this year’s Grand Challenge gave us valuable insights with respect to the benefits and limitations of the parallelization of complex queries such as the proposed ones. Among other insights, it is possible to point-out the influence of shared-memory accesses and contention on shared-nothing algorithmic approaches, which shows to be an important issue on itself in data streaming, as well as in databases [10]. Future directions based on this work will further explore trade-offs in such designs and investigate how to achieve a balanced parallelism, mixing at the same time shared-memory and shared-nothing analysis.

Acknowledgments. We would like to thank Zbigniew Jerzak and Holger Ziekow for the organization of this challenge and the support with answering related questions.

References


