Poet: an Interactive Spatial Query Processing System in Grab

Johns Paul, Jie Liang Ang, Tianyuan Fu, Bingsheng He, Shengliang Lu
National University of Singapore

Sien Yi Tan, Feng Cheng
Grab Taxi Holdings Pvt. Ltd.

ABSTRACT

Interaction-based systems have been widely used in many enterprises like Grab to enable quick and easy analysis of large-scale spatial data. Unlike traditional instruction-based query processing systems, modern interaction-based systems allow users to issue complex queries through simple interactions with a Graphical User Interface (GUI). While such systems have significantly transformed the process of spatial query processing, they still rely on a *process-after-query* approach for executing the queries. Even though the user is continuously interacting with the GUI, the actual processing is only initiated after the user completes their interactions, thus wasting the opportunities to reduce the response time of query processing.

Inside Grab, we develop Poet, a progressive execution framework to continuously analyze user interactions and to perform progressive execution as soon as the system gains *reasonable confidence* regarding the user intentions. By integrating Poet, the interaction-based system can begin processing before the query is expressed in its whole by the user. The user interactions are captured and modelled in Markov chains, which guide the probability of progressive execution. For handling large-scale trajectory data in Grab, the progressive execution engine of Poet has been designed on top of Apache Flink. Our experiments show that Poet is able to reduce the latency in generating the output, providing a more interactive experience. Our experiments find that Poet helps reduce the query execution latency by up to 25x.

CCS CONCEPTS

• **Information systems** → **Spatial-temporal systems**; • **Human-centered computing** → *Visual analytics.*

KEYWORDS

progressive execution, spatial operators, interactive systems

ACM Reference Format:
Johns Paul, Jie Liang Ang, Tianyuan Fu, Bingsheng He, Shengliang Lu and Sien Yi Tan, Feng Cheng. 2020. Poet: an Interactive Spatial Query Processing System in Grab. In *28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '20), November 3–6, 2020, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3397536.3422230

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

ACM ISBN 978-1-4503-8019-5/20/11…$15.00
https://doi.org/10.1145/3397536.3422230

1 INTRODUCTION

The explosive growth in ride-hailing and food delivery services [2–4] in the last decade has lead to the generation of large amounts of trajectory data from GPS-enabled devices like smartphones. Among these services, Grab Holdings Pte. Ltd. (Grab) is the largest ride-hailing and food delivery platform in Southeast Asia. Like other similar companies, efficient analysis of spatial data is key to improving vehicle routing, minimizing congestion, and improving service quality at Grab.

To allow data scientists and data engineers to easily and efficiently explore spatial data sets at a large scale, numerous interaction-based systems [8, 16, 26, 35] have been introduced in the past. Unlike traditional instruction-based systems (e.g., those with operational commands in SQL and Python), which require the user to spend considerable amounts of time expressing their trajectory queries as precise instructions with geographical coordinates, modern interaction-based systems make use of user-friendly GUIs for issuing queries. Response time is an important design consideration in those interaction-based systems.

Many existing systems still rely on traditional *process-after-query* based frameworks for executing the queries and processing the underlying data, which cause long response time. The system simply keeps track of the user interactions until an explicit signal to begin query execution is issued by the user (either through a button press or other explicit interactions). However, such an approach is inefficient since these systems fail to take advantage of the opportunity to process data as the user is interacting with the system. This is especially problematic, since issuing complex queries that requires the execution of a sequence of operators can be time-consuming, even on an interaction-based system. There are many opportunities to take advantage of this time period for useful computation. Even worse, this approach often defeats the purpose of interactive GUIs for analytical query processing, due to the high latency between query issue and the generation of the output data.

To address these inefficiencies, we have developed Poet, an interaction-based progressive execution system for analyzing large scale spatial data sets in Grab. Poet has addressed two major technical challenges. First, it is challenging to efficiently predict user intentions [9, 10]. Second, the progressive execution of the query plan is carefully optimized to minimize the overhead. There is a trade-off between the frequency of execution plan updates and the opportunities for progressive execution.

Poet keeps track of user interactions and predicts user intentions (i.e., in the form of *query template*) and execution parameters during the user expresses the query. For this, Poet makes use of an n-order discrete-time Markov Chain [9], taking into account past user interactions and the underlying data set. The execution engine in Poet then begins execution based on the predictions. The query plan and parameters derived based on the predictions are constantly

evaluated against new user interactions (i.e., feedback), and the execution is refined progressively. For this, the spatial operators in Poet are designed to ensure that progressive updates to the input data set or query parameters lead to a minimal amount of additional computation and data accesses. To handle large data sets in Grab, we have developed Poet on top of Apache Flink, a popular high-performance data stream engine.

The major contributions of this paper are as follows:

- We study existing interaction-based query processing systems and detail their inefficiencies when interactively processing large scale spatial data sets.
- We propose Poet, an interaction-based query processing system that supports the progressive execution of spatial queries. Poet makes it possible to progressively execute spatial queries through efficient prediction of user intentions and through the use of progressive designs for spatial operators.
- We present case studies to demonstrate the improvement in the interactivity and overall query execution performance of Poet when compared to existing interaction-based systems [8] for different use cases in Grab. Our experiments find that, Poet helps reduce the query execution latency by up to 25x.

The remainder of this paper is organized as follows. We present the background and related work in Section 2. We present the design and implementation of Poet in Section 3, followed by details of our prediction engine and progressive execution unit in Sections 4 and 5, respectively. We present detailed experiments in Section 6, and conclude in Section 7.

2 BACKGROUND & RELATED WORK

2.1 Interactive Spatial Systems

Interaction-based systems improve the user experience for data scientists and engineers by allowing them to issue complex analytical queries through simple interactions with the system. Numerous interactive systems have already been proposed for non-spatial data sets [17, 20, 21, 25]. However, by design, these systems are not suitable for efficient progressive processing of spatial data objects.

For processing spatial data sets, interactive systems (e.g., [6, 16, 27]) typically adopt a two-layer design consisting of 1) an interactive visualization layer that allows users to issue queries and handles the presentation of the output, and 2) an analytical layer that is responsible for executing queries issued by the user. The interactive layer keeps track of user interactions and builds a query/execution plan after the query is entirely expressed. Despite the interactive nature of those systems, they still adopt a process-after-query execution approach and lack any form of predictive or progressive execution of queries. In the following, we review some of those systems, and refer readers to a survey [23] for more related work.

DisDC [27] proposed a new computing model to improve the performance of interactive spatial data analysis by mapping pixels on display to spatial objects and then performing the computation based on the pixels on the screen. Xin et al. [16] proposed Viptra, an interactive front-end framework providing visualization on top of UltraMan [15], a distributed in-memory system for big trajectory data. Semantic Traj [6] proposed the integration of semantic information along with the visualized trajectory data to promote an easy understanding of the trajectory objects and enable efficient query issue. Wang et al. [31] proposed an interactive visualization platform for analyzing traffic jam data. Further, interaction-based systems like Vaite [35] and Vaud [13] were proposed specifically for the analysis of urban spatial data.

2.2 Progressive Query Execution

There have been numerous studies on adopting progressive execution for many applications like image retrieval [29], online advertisements [11] and relational entity resolution [7]. However, these studies often focus on designing progressive versions of specific non-spatial operators, which is non-trivial to be extended to spatial processing. Further, while progressive optimization of relational queries has been explored in literature [28, 33, 37], they are not capable of executing the operators progressively.

Progressive versions of spatial operators like trajectory similarity search [12, 14, 34], spatial join [19, 24, 36] and others [5, 30] have also been explored in the literature. However, these operator implementations are designed for non-interactive spatial processing systems and are hence not suitable for highly interactive systems like Poet.

There have been implementations that steer users towards interesting queries and operator parameters [10, 21] or pre-fetch data based on a predictive understanding of user intentions [9]. However, these systems either do not support progressive execution of queries/operations or their progressive execution is limited to specific operations like pre-fetching of data tiles and not complex spatial operators.

2.3 Spatial Data Analysis at Grab

Grab is now the largest ride-hailing and food delivery platform in Southeast Asia. The spatial data set that needs to be processed at Grab consists of the following two objects: GPS record and Trajectory. A GPS record is simply a 3-tuple consisting of latitude, longitude and the timestamp collected from GPS enabled client devices (smartphones or tablets owned by drivers). The latitude and longitude values in the GPS record together help identify the drivers' location and the timestamp represents the time at which the data was collected from the client device. Further, a trajectory is simply a time ordered collection of GPS records along with other information that is relevant to Grab's business. Overall, the data set used in study consists of 418,403 trajectories and 112,569,492 GPS records. Further, the GPS records were collected from the users in Singapore and Indonesia at a sampling rate of 1 second. The data set includes records collected as recently as December 2018.

There are many scenarios for interactive spatial processing in Grab. We use the following two to motivate our studies.

Scenario 1: *Understanding the trajectory/trip pattern to/from specific areas of interest.*

Data scientists at Grab frequently analyze trajectory/trip patterns to/from major areas of interest (e.g., airports and shopping malls) over specific periods of time (e.g., peak hours and non-peak hours). The insights gained from this analysis often helps optimize their services. To execute this query on an interactive system like Poet, a data scientist would first zoom and pan over the map to focus in

on the specific region they are interested in. The user would then draw a polygon around the region, which triggers a spatial join between the trajectory objects and the polygon object expressed by the user. The user may then use the timeline to filter out only the trajectories/trips that were generated during specific time periods. Overall, the need to perform a spatial join over a large trajectory data set makes this query time consuming, leading to a significant delay between the query issue and the generation of the output. Figure 1a shows the typical interactions of a user to issue the above query in Poet.

Scenario 2: *Identifying missing road network data.*

Despite the improvements to the mapping services in the last decade, there are still regions where the underlying road network data is incomplete. For example, information like the direction of travel allowed along a road segment or sometimes even the entire road segment itself could be missing in the data provided by the mapping services. This can be detrimental to services like Grab as it leads to increased operational costs. Hence, data engineers in Grab makes use of past trip data to identify such missing information. To identify missing road network data, engineers would simply draw a trajectory in the relevant region and query the system to find trajectories similar to the one expressed by the user. The presence or absence of trajectories similar to the one expressed by the user would help them to identify the missing road network data. Figure 1b shows the typical interactions of a user to issue the above query in Poet.

(a) Scenario 1

(b) Scenario 2

Figure 1: User interactions when issuing queries.

3 SYSTEM DESIGN OF POET

In this section, we present the design overview of Poet, the interaction-based progressive query execution system proposed in this study. As shown in Figure 2, Poet consists of two major components: 1) an interactive client application running on a touch- or pointer-enabled device (e.g., smartphone, tablet, and personal computers) and 2) a high-performance query execution unit running on remote clusters. Note, touch- or pointer-enabled client devices often have limited storage and computation capabilities. Hence, the client application is only responsible for managing the user interactions, and all processing of the underlying spatial data set is carried out by the execution unit running on powerful remote clusters.

The key design goal of Poet is to make efficient use of the opportunities for the progressive execution of spatial queries as the query is being expressed by the user. We achieve this in the following ways. First, by efficiently predicting user intentions, Poet begins

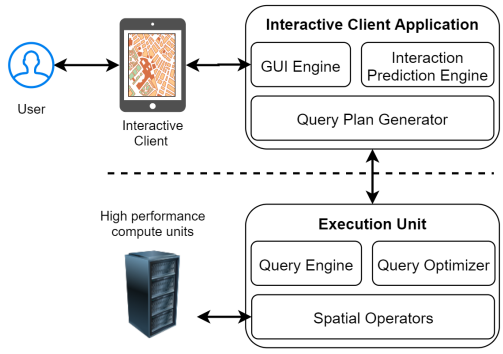

Figure 2: Architecture design of Poet.

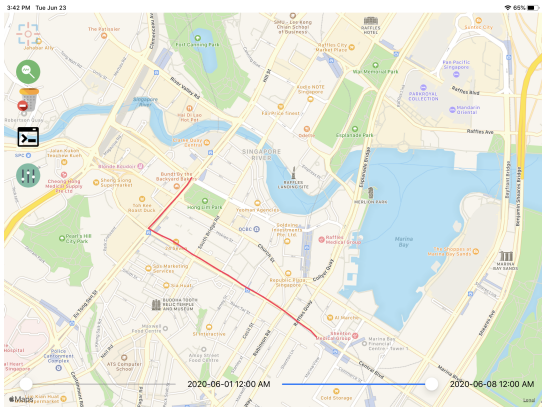

Figure 3: The GUI of Poet running on an iPad.

processing of the underlying data set before the query is expressed in its entirety by the user. Second, by designing efficient progressive versions of the commonly used spatial operators, Poet minimizes the cost of miss-prediction of user intentions.

3.1 Interactive Client

The interactive client application of Poet allows users (data scientists and engineers) to issue complex spatial queries through simple interactions. The client application is designed specifically for touch or pointer enabled devices like smartphones, tablets, and personal computers. As shown in Figure 2, the interactive client application consists of the following key components: 1) a graphical user interface (GUI), 2) an interaction prediction engine (IPE), and 3) a query plan generator.

Graphical User Interface (GUI). As shown in Figure 3, the GUI component of the interactive client application consists of the map of a geographical region (Singapore in Figure 3) with which the user can interact using a pointing device (e.g., fingers, cursor, and stylus, to name a few). Currently, our client application supports the following set of user interactions: zoom in/out, pan left/right/up/down, draw, annotate (marking up a geographical region), and timeline update (using the timeline shown on the right in Figure 3). These interactions are mapped to standard gestures available in the specific input device (touch or non-touch). For

example, the zoom-in/out interaction can be issued using two-finger gestures on a touch-enabled device or the mouse wheel in non-touch enabled devices. The GUI keeps track of user interactions and feeds them into the IPE as the user interacts with the system. Note, when an interaction is fed into the IPE, the GUI attaches a set of interaction specific parameters along with it (e.g., the start and end point of the drawing for the draw interaction). That means, each state in IPE has parameter values associated with it. These parameters are later used to derive query/operator execution parameters (performed by the query plan generator).

Interaction Prediction Engine (IPE). The IPE helps Poet to understand the user intentions before it is expressed in its entirety by the user. The IPE achieves this by predicting future user interactions based on the user interactions already issued by the user. Like past studies [9, 10], the IPE uses an n-order Markov Chain to predict future user interactions. Markov chains assume that the users' past moves can be good indicators of future actions. Hence, an n-order Markov chain can be used to predict the next user interaction, given past n user interactions. More details regarding the training of the Markov model and prediction of user intentions can be found in Section 4. Finally, the sequence of the past user interactions and the predicted future interactions are fed into the query plan generator by the IPE, for generating the query plan.

Query Plan Generator. The query plan generator in Poet translates the sequence of interactions received from the IPE ($I_1, I_2, ..., I_n$) into a query plan or a sequence of spatial operations ($Q = O_1, O_2, ..., O_l$). Note, the parameters associated with each operator (e.g. predicates in a filter operation) in the query plan is derived based on the parameters associated with the interactions. More details regarding how a sequence of interactions are converted to a sequence of operators can be found in our past study [8]. The generated query plan is then sent to the query execution unit running on a remote cluster for execution.

Finally, since the probabilistic prediction of future user interactions using a Markov chain can have errors, the IPE updates the query plan generator with a new sequence of interactions whenever wrong predictions are detected. The IPE constantly evaluates its past predictions with the new set of user interactions generated by the GUI. The query plan generator, in turn, updates the query plan and the query execution parameters based on the new sequence of interactions. The updated query plan is then relayed to the query execution unit for execution.

3.2 Query Execution Unit

The query execution unit of Poet is responsible for executing the query plan ($Q = O_1, O_2, ..., O_l$) in the cluster. After the client application submits the query to the query engine, the execution unit first optimizes the query plan for the underlying data set (using the query optimizer) and then dispatches the operators in the query plan for execution. To enable progressive execution, the query execution unit maintains a context for each user session. The context keeps track of the query plan under execution, the execution parameters of the operators and the current set of intermediate as well as output data. Further, to ensure high performance computation, the spatial operators in Poet are designed on top of Apache Flink [1]. This means that 1) the execution unit is capable of distributing the execution of its spatial operators over a number of processing nodes and 2) all spatial operators in Poet operate on data loaded on to main memory during the system initialization. The current system focuses on optimizing individual queries. We are interested in exploring multi-query optimizations like [38].

In a progressive execution system based on prediction of user intentions, the query plan or the query execution parameters submitted to the execution unit at any given time. There is a trade-off between the frequency of execution plan updates and the opportunities for progressive execution.

Poet addresses this inefficiency and minimizes the amount of unnecessary computation and data accesses in the following ways. First, to efficiently handle cases where the query plan itself needs to be updated through the addition or removal of spatial operators, Poet maintains a snapshot of the output generated by the operators that were executed speculatively (those operators given by our predictions, which have not yet been expressed by the user). We do not maintain the output snapshot for all the operators in the query plan because the updates to the query plan resulted from the wrong predictions will be limited to the operators that were executed speculatively. Second, to handle the updates to the execution parameters of an operator in the query plan, Poet makes use of progressive implementations of the spatial operators that are capable of handling updates to the query execution parameters. These operators ensure that the amount of additional computations required on update to the query execution parameters are kept to a minimum. More details regarding the implementation of the progressive spatial operators can be found in Section 5.

3.3 Case Studies using Poet

For the two case studies presented in Section 2.3, we discuss how progressive execution help improve their query response time and overall query execution performance, thereby improving the productivity of data engineers and data scientists at Grab.

Scenario 1: Poet helps improve the overall execution performance of the above query in the following ways. First, based on the pan and zoom interactions issued by the user when trying to focus in on the region of interest, the IPE predicts the users interest in a particular region. This initiates a pre-fetching of the spatial data in the region. Second, as the system recognizes that the user has focused their attention on a specific region, the IPE predicts the users intention to markup a region using a polygon. Now, it is not feasible to accurately predict the exact region that will be annotated by the user. Hence, the IPE predictively sets *good enough* polygon parameters based on past user interactions. Third, if the IPE predicts that the user would further perform a temporal filtering on the data, then Poet further prepares the output of the spatial join for progressive filtering of the trajectory data.

Scenario 2: First, similar to scenario 1, Poet predicts the users' interest in a specific region as they zoom and pan around on the GUI. This helps pre-fetch spatial data in the region before the user begins expressing the trajectory. Second, as the user begins expressing the reference trajectory by drawing on the GUI, the system progressively computes the similarity of the pre-fetched trajectories with respect to the trajectory being expressed by the user. In

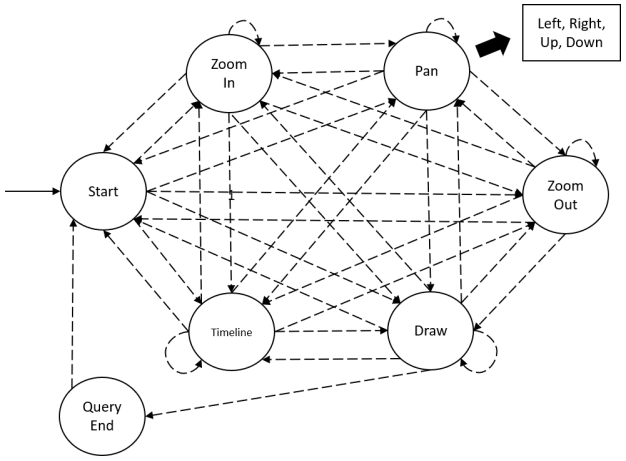

Figure 4: Simplified version of the user interaction model used by the IPE.

the ideal case, only the similarity for the last segment of the trajectory expressed by the user will need to be computed after the user completes their drawing.

4 PREDICTING USER INTERACTIONS

Predicting user intentions is key to ensure effective progressive execution of queries as the user is interacting with the system.

4.1 User Interaction Models

To enable efficient prediction, the IPE develops a *user interaction model* (UIM) trained for each user in the system. The UIM is essentially a Markov chain describing a sequence of possible events in which the probability of the next event depends past n events. Figure 4 represents a simplified version of the Markov chain used by the IPE. Each state in the UIM represents a unique interaction issued by the user and the arcs between the states represent the transition from one interaction to another by the user. Note, the actual UIM used in our system is much more complicated than the one shown in Figure 4, which is simplified for presentation purposes.

Poet uses the same structure (the states and transitions) of the UIM for different users, since each user has access to the same set of interactions. However, the transitions probabilities in UIM are generated separately for each individual user. This is done during the calibration phase of the IPE, which helps more accurately predict the individual user intentions. In the remainder of this section, we detail how the UIM is calibrated for each individual user and the prediction process that helps predict the user interactions based on the UIM.

4.2 Calibration Stage

The calibration stage of the IPE is responsible for generating the probability values associated with each arc/transition in the UIM. Since these transition probabilities must take into account the unique characteristics and interactions patterns of each user, the calibration process is executed separately for each user. Currently, the calibration process in Poet is invoked for only three scenarios: new users, periodical model maintenance or when requested by the user.

The calibration process for each user happens in two phases. In phase 1, the user will be asked to issue a pre-determined set of queries using the front-end GUI. These are often simple queries consisting of 1 or 2 spatial operators and are mostly used to tune the parameters associated with each individual interactions (e.g., users typical zoom level). In phase 2, the user is asked to issue a set of frequently used query templates using the GUI. This is important because, based on their job role, different teams at Grab often focus on a different set queries/interactions. The system keeps track of the user interactions during the phase and generates a sequence of interactions corresponding to each query issued by the user. Once the user finishes expressing the queries, the set of interaction sequences generated by the user is then used to generate the transition probabilities in the UIM using Algorithm 1.

Algorithm 1: Markov chain probability computation.

Input: L, sequences of past interactions and n, order of Markov Model.
Output: *nextProb*, the next interactions' probabilities.

```
1  Function GenerateMarkovProbabilities(L, n)
2      stateFreq = []; nextFreq = [][]; nextProb = [][];
3      for each sequence S in L do
4          for each arr in GetAllSubarrays(S, n + 1) do
5              state = GetSubarray(arr, 1, n);
6              next = GetSubarray(arr, n + 1, 1);
7              stateFreq[state] += 1;
8              nextFreq[state][next] += 1;
9          end
10     end
11     for each state in nextFreq do
12         for each next in nextFreq[state] do
13             nextProb[state][next] = nextFreq[state][next] /
                 stateFreq[state];
14         end
15     end
16     return nextProb
17 end
18 Function GetAllSubarrays(S, n)
19     A = [];
20     for i from 1 to len(S)-n do
21         A.add(Subarray(i, n));
22     end
23     return A;
24 end
25 Function GetSubarray(arr, start, length)
26     return arr[start : start + length];
27 end
```

Algorithm 1 generates the transitions probabilities between interactions (or the weights of the edges in the UIM). In our n-th order Markov model, each state is defined to be a sequence of interactions of length n. Given a UIM(U), a list of sequences ($L = S_1, S_2, ..., S_i, ...$)

of past interactions the algorithm works as follows. For each sequence S_i, we obtain all sub-arrays of length n within the sequence (Line 5), along with the next succeeding the state (Line 6). We then update the frequencies of each sub-array of n-states as well as the frequency of the succeeding (Lines 7 and 8). We are then able to compute the state transition probabilities of each interaction based on the previous n-interactions/states as shown in Lines 11 - 14. Note, the model that is generated from the training process is similar to n-gram models applied in computational linguistics [22], which is sufficiently efficient and accurate in our experiments.

4.3 Prediction Stage

In the prediction stage, the IPE is responsible for predicting users intention (or the query the user is trying to express using the GUI) based on the user's past interactions.

Each spatial operator in Poet is expressed as an ordered sequence of basic interactions (or states in UIM). Hence, the user intentions can be determined by simply predicting a sequence of future user interactions. Overall, the prediction problem can be expressed as follows.

Problem Definition. Given a UIM(U), an ordered sequence of past interactions ($I_c = I_1, I_2, ..., I_n$) and a probability threshold (P), generate the longest sequence of future interactions ($I_p = I_1, I_2, ..., I_m$) that has a probability of occurrence higher than P.

Algorithm 2: Interaction prediction based on Markov model.

Input: I_c: the sequence of past n interactions.
P: threshold probability.
U : The user interaction model.
Output: I_p : The predicted future interaction(s)

```
1  Function PredictInteractionSequence(I_c, P, U)
2      return FutureIntrs(I_c, P, 1)
3  end
4  Function FutureIntrs(I_c, P, curProb)
5      I_p = []; maxProb = 0;
6      for intr in U.NextStates(I_c) do
7          p = curProb × U.TransitionProbability(I_c, intr);
8          if p > P then
9              future = [intr, FutureIntrs([I_c[1 : n], intr], P, p)];
10             if len(I_p) < len(future) or (len(I_p) == len(future)
                 and p > maxProb) then
11                 I_p = future;
12                 maxProb = prob;
13             end
14         end
15     end
16     return I_p;
17 end
```

Solution. To predict the sequence of future user interactions (I_p), the IPE makes use of Algorithm 2. Note, if there is no sequence with a probability of occurrence higher P then an empty set is returned. Given past n interactions, Algorithm 2 aims to predict the longest sequence of future interactions with a probability $> P$. Based on the n-order Markov model built by Algorithm 1, we assume that the future interactions depends only on the past n interactions.

The IPE invokes Algorithm 2 with the set of past n interactions (I_c), the probability threshold (P) and the user interaction model (U) containing the transition probabilities, as inputs. The prediction algorithm then recursively enumerates a tree of interactions where the probability of reaching each new level is determined based on the past n-levels. Further, each branch in the tree is only enumerated until the probability of reaching the last level falls below the probability threshold. The algorithm then returns the longest sequence of future interactions with a probability higher than the threshold. This algorithm is very lightweight, with minimal runtime overhead. Note, a lower threshold would lead to more frequent and early predictions but will increase the rate of mis-predictions and vice versa. For our experiments in Section 6, we set this threshold value to be 0.7. Finally, when two or more sequences with the same length satisfy the probability threshold, we choose the sequence with highest probability.

5 PROGRESSIVE EXECUTION

The use of progressive execution helps Poet minimize the impact of miss-predictions by the client application. Progressive operators in Poet are designed with the ability to progressively process large scale spatial data sets and handle repeated updates to the operator parameters. Note, in Poet, a progressive spatial operator is triggered by a sequence of interactions. Now, any update to the operator parameters (e.g., filter predicate) may require 1) processing of an additional set of input tuples (e.g., more trajectory objects from a different region) and 2) additional computation on the spatial objects which are in the current set of output objects.

To ensure efficient progressive execution with minimal overhead, we adopt the following design rationales for the progressive spatial operators. First, we make sure that the additional input data objects that are chosen for processing by the operators as a result of an update to the operator parameters does not include any tuples that are currently in the set of output data objects. This helps avoid unnecessary repeated processing of the same spatial objects. Second, the spatial objects in the current output data set that needs to be processed again as a result of an update to the operator parameters should be kept to a minimum.

In the remainder of this section, we present the implementation details of two of the key progressive spatial operators in Poet: trajectory similarity search (TSS) and spatial Join. Poet supports other operators that are relevant to Grab's data analysis tasks, such as filtering (e.g., invoked using the timeline in the GUI) and aggregation. For each operator, we develop techniques to minimize the overhead of updates to the operator parameters.

5.1 Trajectory Similarity Search (TSS)

The TSS operator identifies all trajectories that are *similar* to a given reference trajectory. Note, two trajectories are considered to be *similar* if the similarity metric (e.g., Fréchet distance [18]) of the two trajectories is above a specified threshold. As noted by scenario 2 in Section 3.3, the TSS operator often helps data engineers in Grab to identify missing road network data.

The user can trigger the TSS in Poet by simply focusing in on a specific geographical region and then drawing a trajectory (referred to as the *reference trajectory*). Due to the progressive nature of Poet, the operator will begin computing the similarity between the reference trajectory and the trajectories in the underlying data set as soon as the drawing state in the UIM is activated/predicted by the IPE. The computation continues until the user finishes expressing the reference trajectory (i.e., on exit from the drawing state).

Numerous TSS implementations have been proposed in the literature [12, 14, 34]. The progressive TSS implementation used in Poet was designed based on the more recent algorithm [14]. Unlike the original implementation, the implementation in Poet is capable of progressively computing the similarity between the reference trajectory and the trajectories in the underlying data set as the reference trajectory is being expressed by the user.

The pseudo-code for the progressive TSS implementation used in Poet is presented in Algorithm 3. To enable a progressive execution, the reference trajectory is broken down into segments and sent to the execution unit by the client application at the granularity of a single segment. In addition to this, as soon the user focuses in on a specific region and the IPE predicts the future possibility of the user drawing a trajectory, the trajectory objects in the underlying data set will be pre-fetched by the execution unit to help improve the performance of the subsequent operators. For reducing the disk I/O, we make use of a virtual page table to enable pre-fetching by keeping track of the physical locations of each page of spatial objects. The page table is checked before a new page is accessed for processing. Anytime a page is pre-fetched, the page table is updated to reflect this.

Now, the progressive TSS operator in Algorithm 3 is invoked as soon as the first segment is expressed by the user. The operator is also provided with a similarity threshold (S_T) and a minimum bounding rectangle (*mbr*). The values for these parameters are either estimated by the IPE or expressed by the user. The *mbr* is simply a rectangle around a geographical region and the progressive TSS implementation limits the number of trajectory objects that are processed to those that lie within the *mbr* for each input segment. The intuition here is that the users are often only focused on the spatial objects in the region that is visible on the screen.

For each new segment received by the operator, the code in Lines 17 - 22 retrieves the trajectories in the underlying data set that fall within the *mbr* around the segment and computes their similarity with the received segment (Line 18). Note, we adopt the same least common sub-sequence approach as in the case of the previous study for similarity computation [14]. The computed similarity value of each candidate trajectory within the *mbr* is stored in *sim_list*. When a trajectory achieves a cumulative similarity (combined similarity with all the segments) higher than the threshold, it is added to the output data set and is not considered for future segments.

As mentioned before, due to the predictive and highly interactive nature of Poet, parameters like the S_T and *mbr* could be later updated (either by the user or as the result of a wrong prediction). Lines 5 - 9 and 10 - 16 handle the updates to the S_T and *mbr* parameters respectively. An update to the S_T would require a revisiting of the trajectories identified for the previous segments, and trajectories will have to be added or removed from the output set depending on the new values of S_T. Similarly, an update to the *mbr* would lead to more trajectory objects being added or removed to or from the *sim_list* depending on the new value of *mbr*.

Algorithm 3: Progressive trajectory similarity search.

```
Input: T : The underlying trajectory data set.
S_T : The similarity threshold.
seg : The first segment of the reference trajectory.
mbr : The minimum bounding rectangle.
Output: output : The set of output trajectories.
1  Function ProgressiveTSS(T, S_T, seg, mbr)
2      output = []; count = 0;
3      segs[count] = seg;
4      while segs[count] != Nil do
5          S_T^n = NewThreshold();
6          if S_T^n != S_T then
7              UpdateTrajectorySet(sim_list, output, S_T, S_T^n);
8              S_T = S_T^n
9          end
10         new_mbr = NewMBR();
11         if new_mbr != mbr then
12             for seg in segs do
13                 UpdateTrajectorySet(sim_list, output,
                     new_mbr, mbr);
14             end
15             mbr = new_mbr;
16         end
17         for trajectory in T.GetTrajectories(mbr, segs[count])
            do
18             sim_list[trajectory][count] =
                 ComputeSimilarity(segs[count],
                 LCSS(trajectory, segs[count]));
19             if CumSimilarity(sim_list[trajectory]) > S_T and
                 trajectory not in output then
20                 output.append(trajectory);
21             end
22         end
23         count++;
24         segs[count] = NextTrajectorySegment();
25     end
26     return output;
27 end
```

5.2 Spatial Join

The spatial join operator in Poet identifies all the trajectories passing through a polygonal region expressed by the user. As noted by scenario 1 in Section 3.3, the spatial operator helps data engineers in Grab to study/analyze trajectory/trip patterns to/from specific regions of interest.

The user can trigger the spatial join operator in Poet by simply focusing in on a specific geographical region and then annotating a region using a polygon. Due to the progressive nature of Poet, the operator will begin performing the spatial join between the

polygon and the trajectories in the underlying data set as soon as the region annotation state is activated/predicted by the IPE.

Numerous studies on spatial joins exist in literature [19, 24, 36]. Specifically, we design our progressive spatial join implementation based on the study by Oje et. al. [24], which has been widely used in spatial systems. However, unlike the original study, our implementation is designed to support more complex updates to the level of detail of the polygon being used for the join operation.

Algorithm 4: Progressive spatial join.

```
   Input: T : The underlying trajectory data set.
   P : The reference polygon.
   Output: output : The set of output trajectories.
1  Function ProgressiveSpatialJoin(T, P)
2      output = [];
3      region_list = [];
4      while P != nil do
5          P_N = GetPolygon();
6          if P_N != P then
7              UpdateTrajectorySet(P, P_N, output);
8          end
9          for region in P.index_regions()
10             if region not in region_list then
11                 if region.isEnclosed(P) then
12                     output.append(T.GetTrajectories(region));
13                 end
14                 else
15                     for t in T.GetTrajectories(region)
16                         if t.isInside(ref_polygon) then
17                             output.append(t);
18                         end
19                 end
20                 region_list.append(region);
21             end
22     end
23 end
```

The pseudo-code for the progressive spatial join implementation used in Poet is presented in Algorithm 4. To enable progressive execution, an operator is designed to support progressive update to the polygonal structure being expressed by the user. In addition to this, as soon as the user focuses in on a specific region and the IPE predicts the future possibility of the user annotating the region using a polygon, the trajectory objects in the underlying data set will be pre-fetched by the execution unit to help improve the performance of the subsequent operators.

Initially, the spatial join operator in Algorithm 4 is invoked with a simple rectangular region estimated/predicted by the IPE as the user zooms into the region of interest. As the user begin annotating the region, the changes to the polygonal structure is progressively updated by the client application. The operator makes use of aggressive indexing to determine the trajectory objects that pass through the polygon expressed by the user. For this, we make use of a grid index where the entire region is divided into small rectangular

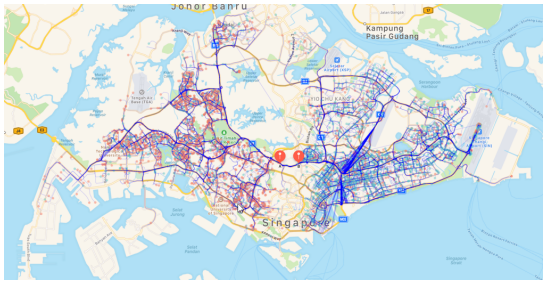

Figure 5: Example query execution in Poet.

grids and each index entry has pointers to the coordinates of the trajectories that lie within the region. Hence, the operator first decomposes the polygon in to a set of small index regions (Line 9) which are completely or partially enclosed within the polygon. Now, if a region is completely enclosed within the polygon then all the trajectories that it points to will be added to the output list (Lines 11 - 13). Further for partially enclosed regions, each individual trajectory is checked to ensure that they lie inside the polygon (Lines 14 - 19). To minimize the execution cost, we also make sure that a region that is explored once is not explored again (Line 10)

Now, for each update to the polygon (Lines 5 - 8) the following actions are executed by the *UpdateTrajectorySet* function. First, the trajectories in the regions that are partially or completely enclosed by the new polygon but not by the original polygon is added to the output set if they are inside the new polygon. Second, the trajectories in the regions that are partially or completely enclosed by the original polygon but not by the new polygon are removed from the output set if they are not inside the new polygon. This helps ensure the correctness of execution.

6 EXPERIMENTS

In this section, we evaluate the efficiency of Poet in analyzing massive spatial data sets at Grab. As mentioned before, data scientists and engineers at Grab make use of Poet for issuing complex queries by simple interactions with the system. Figure 5 shows an example query issue and the presentation of the output to the user by Poet.

Hardware. While Poet supports distributed execution of spatial operators, our experiments were conducted on a powerful server equipped with a Xeon E5-2698 v4 CPU with 20 cores, operating at 2.20 GHz. The node is also equipped with 256 GB of main memory.

Workload. For evaluating the efficiency of Poet we make use of taxi trip data of Grab users. As detailed in Section 2.3, the data set consists of over 418K trajectory objects. To effectively evaluate the ability of Poet to help improve the data analysis process at Grab, we make use of the two scenarios described in Section 3.3. In Grab, Scenario 1 (**S1**) helps data scientists to understand trajectory/trip patterns to/from a specific area of interest at a given time; while Scenario 2 (**S2**) helps data engineers to identify missing road network data by searching for trajectories similar to a reference trajectory at specific periods of time.

Experimental Outline. In Section 6.1, we evaluate the efficiency of the IPE in Poet to efficiently predict the user interactions. In Section 6.2, we evaluate the efficiency of the progressive spatial operators in executing the query while it is being expressed by the

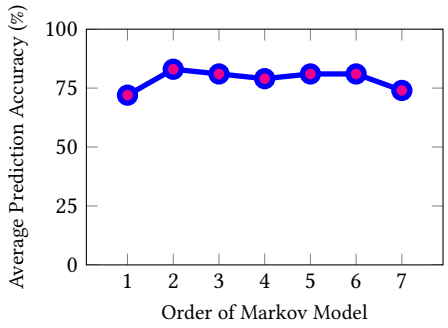

Figure 6: Impact of order of Markov model on prediction.

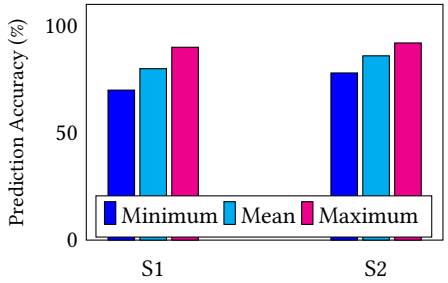

Figure 7: Prediction accuracy of IPE for the two case studies.

user. Finally, in Section 6.3, we detail how Poet helps improve the overall query execution performance at Grab.

6.1 Impact of Predictive Execution

As mentioned before, the IPE in Poet helps predict user intention based on an n-order Markov model. Now, the order of the Markov model (i.e., the value of n) can have significant impact on the quality of prediction and performance of the IPE. To demonstrate this, we present the average prediction accuracy of the IPE for different values of n in Figure 6. The results show that, too small values of n achieve poorer prediction accuracy as it fails to take into account sufficient number of past user interactions. Further, increasing the value of n beyond a value of 3 leads to minimal improvement in prediction accuracy (and sometimes a degradation in accuracy), while significantly increasing the space and time complexity of the training and prediction algorithms. Hence, for the remainder of our experiments, we use a value of n = 3, which achieves a reasonable balance between prediction accuracy and the cost of the prediction.

To further understand the ability of the IPE in predicting user intentions, we conduct a user study with the help data scientists at Grab. The minimum, maximum and mean prediction accuracy achieved by the IPE for scenarios S1 and S2 during the study is presented in Figure 7. Note, the accuracy values are based on a sample of 30 query issues each for S1 and S2 by the users. Further, a prediction by the IPE is considered to be accurate, if it is able to interpret the next spatial operator that will be issued by the user.

As shown in Figure 7, the IPE is able to predict the next spatial operator with an accuracy above 80% for both S1 and S2. This helps Poet to pre-fetch data objects and begin processing of the underlying data set as the query is being expressed by the user. The runtime overhead of IPE should be quite minimal, since it runs very smooth on an iPad Pro.

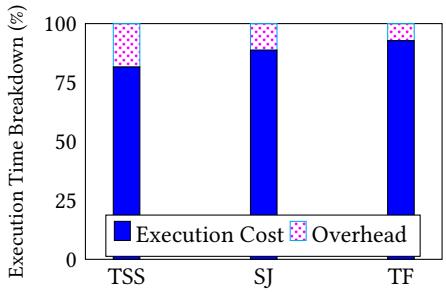

Figure 8: Progressive execution overhead in operators.

6.2 Impact of Progressive Execution

To demonstrate additional computation and data accesses (during each execution parameter update) as opposed to the traditional process-after-query based approaches, we present the execution time breakdown of the progressive version of three key spatial operators in Poet: trajectory similarity search (TSS), spatial join (SJ) and temporal filter (TF). Note that in Figure 8, the overhead is measured as the additional cost of updating query execution parameters as opposed to the case when there are no progressive updates to the execution parameters. Since progressive execution does not reduce the total amount of work that needs to be done, the implementation does encounter some amount of overhead resulted from mis-predictions and updates to parameters. However, the results in Figure 8 clearly show that in spite of frequent updates to the query execution parameters, the operator implementations in Poet is able to keep the overhead of mis-predictions and updates to parameters to a minimum (< 15% of the total execution time).

Now, to understand the ability of a progressive system in reducing the execution latency of spatial queries, we present the normalized execution latency of TSS, SJ, and TF in Poet with respect to an existing system named TraV [8] in Figure 9. The latency is defined as the time between the user finishing their interactions with the GUI for a specific query/operator and the time when the output data is generated by the server. We define the normalized latency improvement as the ratio of the latency of TraV over that of Poet. The results in Figure 9 clearly show that a progressive execution scheme can help reduce the execution latency of spatial operators in an interaction-based system by up to 10x.

In summary, the results in Figures 8 and 9 together shows that, while progressive execution does lead to some level of additional computation and data accesses, it can help significantly reduce the execution latency by allowing the system to process the underlying data set as the user is interacting with the system.

6.3 Overall Comparison

We demonstrate the effectiveness of Poet in improving the overall query execution performance for some application scenarios at Grab. For the comparison, we use TraV [8] as baseline. Besides S1 and S2, we also use two scenarios S1' and S2', which are the same as S1 and S2 except that they do not perform a temporal filter. The results are in Figure 10, which clearly show that 1) Poet helps improve the overall query execution performance at Grab by up to 1.7x and 2) progressive execution scheme can help improve the performance even when expressing relatively simpler queries like S1' and S2'. The improvement in the overall query execution performance is

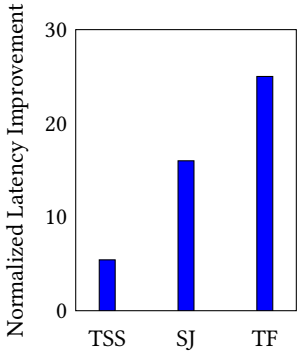

Figure 9: Normalized improvement in query execution latency due to progressive execution.

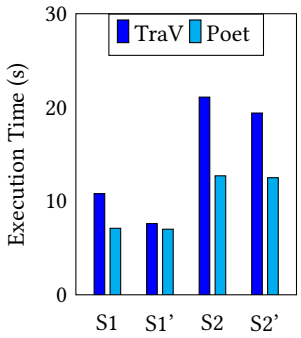

Figure 10: Performance evaluation of Poet against TraV.

significantly smaller than the improvement in latency (Figure 9), since the overall processing time includes the time taken by the user to express the query. Nevertheless, Poet offers much better response time and interactiveness for spatial analysis in Grab.

7 CONCLUSION

In this paper, we propose Poet, an interaction-based progressive spatial data processing system at Grab. Poet keeps track of user interactions and predicts user intentions with a Markov Chain model. Further, Poet supports the progressive processing of the underlying data set, allowing its spatial operator to process the data as the query is being expressed by the user. The experiments show that Poet helps reduce the query execution latency by up to 25x and improve the overall query execution performance by up to 1.7x. As for future work, we are interested in exploring GPU accelerations like [32] for achieving more real-time spatial data analytics.

8 ACKNOWLEDGEMENT

This work was funded by the Grab-NUS AI Lab, a joint collaboration between GrabTaxi Holdings Pte. Ltd. and National University of Singapore. We thank See-Kiong Ng, Teddy Sison David Wenceslao, Xiang Hui Nicholas Lim, and Yong Liang Goh for their support.

REFERENCES

[1] 2018. *Apache Flink*. https://flink.apache.org/
[2] 2019. Didi Chuxing Technology Co. https://www.didiglobal.com
[3] 2019. Grab transport, food delivery and payment solutions. https://www.grab.com/sg/
[4] 2019. Uber Technologies Inc. https://www.uber.com
[5] Tinghua Ai and Xiang Zhang. 2007. *The Aggregation of Urban Building Clusters Based on the Skeleton Partitioning of Gap Space.*
[6] S. Al-Dohuki, Y. Wu, F. Kamw, J. Yang, X. Li, Y. Zhao, X. Ye, W. Chen, C. Ma, and F. Wang. 2017. SemanticTraj: A New Approach to Interacting with Massive Taxi Trajectories. *TVCG* 23, 1 (2017), 11–20.
[7] Yasser Altowim, Dmitri V Kalashnikov, and Sharad Mehrotra. 2018. ProgressER: adaptive progressive approach to relational entity resolution. *TKDD* (2018).
[8] J. Ang, T. Fu, J. Paul, S. Zhang, B. He, T. S. D. Wenceslao, and S. Y. Tan. 2019. TraV: An Interactive Exploration System for Massive Trajectory Data. In *BigMM (demo).*
[9] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *SIGMOD.*
[10] Ugur Cetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alexander Kalinin, Olga Papaemmanouil, and Stanley B Zdonik. 2013. Query Steering for Interactive Data Exploration.. In *CIDR.* Citeseer.
[11] Badrish Chandramouli, Jonathan Goldstein, and Abdul Hussain Quamar. 2016. Deterministic progressive big data analytics. US Patent 9,514,214.
[12] Lei Chen, M. Tamer Özsu, and Vincent Oria. 2005. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD.*
[13] W. Chen, Z. Huang, F. Wu, M. Zhu, H. Guan, and R. Maciejewski. 2018. VAUD: A Visual Analysis Approach for Exploring Spatio-Temporal Urban Data. *TVCG* 24, 9 (2018), 2636–2648.
[14] Jiafeng Ding, Junhua Fang, Zonglei Zhang, Pengpeng Zhao, Jiajie Xu, and Lei Zhao. 2019. Real-time trajectory similarity processing using longest common subsequence. In *HPCC*. IEEE, 1398–1405.
[15] Xin Ding, Lu Chen, Yunjun Gao, Christian S Jensen, and Hujun Bao. 2018. UlTraMan: A unified platform for big trajectory data management and analytics. *Proceedings of the VLDB Endowment* 11, 7 (2018), 787–799.
[16] X. Ding, R. Chen, L. Chen, Y. Gao, and C. S. Jensen. 2018. VIPTRA: Visualization and Interactive Processing on Big Trajectory Data. In *MDM*. 290–291. https://doi.org/10.1109/MDM.2018.00055
[17] Roee Ebenstein, Niranjan Kamat, and Arnab Nandi. 2016. FluxQuery: An Execution Framework for Highly Interactive Query Workloads. In *SIGMOD.*
[18] Sariel Har-Peled and Benjamin Raichel. 2011. The Fréchet distance revisited and extended. In *ASCG*. ACM, 448–457.
[19] Edwin H. Jacox and Hanan Samet. 2007. Spatial Join Techniques. *ACM Trans. Database Syst.* 32, 1 (March 2007), 7–es. https://doi.org/10.1145/1206049.1206056
[20] Lilong Jiang, Michael Mandel, and Arnab Nandi. 2014. GestureQuery: A Multitouch Database Query Interface. *Proceedings of the VLDB Endowment* (2014).
[21] Lilong Jiang and Arnab Nandi. 2015. SnapToQuery: providing interactive feedback during exploratory query specification. *Proceedings of the VLDB Endowment* (2015), 1250–1261.
[22] D. Jurafsky and J. H. Martin. 2000. *Speech and language processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice-Hall, Englewood Cliffs, NJ.
[23] R. R. N. Kanapaka and R. K. Neelisetti. 2015. A survey of tools for visualizing geo spatial data. In *2015 ICCICCT*. 22–27.
[24] Oje Kwon and Ki-Joune Li. 2011. Progressive Spatial Join for Polygon Data Stream. In *SIGSPATIAL.*
[25] Erietta Liarou and Stratos Idreos. 2014. DbTouch in action database kernels for touch-based data exploration. *ICDE* (2014), 1262–1265. https://doi.org/10.1109/ICDE.2014.6816756
[26] Chen Lisi, Shang Shuo, and Guo Tao. 2020. Real-Time Route Search by Locations. *AAAI* (2020).
[27] M. Ma, Y. Wu, L. Chen, J. Li, N. Jing, and C. Du. 2020. Interactive Analytics of Massive Spatial Vector Data via Display-Driven Computing. In *BigComp*. 311–314.
[28] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *SIGMOD*. 659–670.
[29] Deepika Nagthane. 2013. Content based image retrieval system using k-means clustering technique. *Int J Comput Appl Inf Technol* 3, 1 (2013), 22–29.
[30] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2003. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD.*
[31] Z. Wang, M. Lu, X. Yuan, J. Zhang, and H. v. d. Wetering. 2013. Visual Traffic Jam Analysis Based on Trajectory Data. *TVCG* 19, 12 (2013), 2159–2168.
[32] Z. Wen, J. Shi, B. He, J. Chen, K. Ramamohanarao, and Q. Li. 2019. Exploiting GPUs for Efficient Gradient Boosting Decision Tree Training. *IEEE TPDS* 30, 12 (2019), 2706–2717.
[33] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *SIGMOD*. 1721–1736.
[34] Dong Xie, Feifei Li, and Jeff M. Phillips. 2017. Distributed Trajectory Similarity Search. *Proc. VLDB Endow.* (2017).
[35] C. Yang, Y. Zhang, B. Tang, and M. Zhu. 2019. Vaite: A Visualization-Assisted Interactive Big Urban Trajectory Data Exploration System. In *ICDE*. 2036–2039.
[36] S. You, J. Zhang, and L. Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *ICDEW*. 34–41.
[37] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. 2016. Non-Invasive Progressive Optimization for in-Memory Databases. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1659–1670. https://doi.org/10.14778/3007328.3007332
[38] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *ICDE*. 1213–1224.