Noname manuscript No.
(will be inserted by the editor)

*Panda**: A Generic and Scalable Framework for Predictive Spatio-temporal Queries

Abdeltawab M. Hendawi · Mohamed Ali · Mohamed F. Mokbel ⋆

Received: date / Accepted: date

Predictive spatio-temporal queries are crucial in many applications. Traffic management is an example application, where predictive spatial queries are issued to anticipate jammed areas in advance. Also, location-aware advertising is another example application that targets customers expected to be in the vicinity of a shopping mall in the near future. In this paper, we introduce *Panda**, a generic framework for supporting spatial predictive queries over moving objects in Euclidean spaces. *Panda** distinguishes itself from previous work in spatial predictive query processing by the following features: (1) *Panda** is generic in terms of supporting commonly-used types of queries, (e.g., predictive range, KNN, aggregate queries) over stationary points of interests as well as moving objects. (2) *Panda** employees a prediction function that provides accurate prediction even under the absence or the scarcity of the objects' historical trajectories. (3) *Panda** is customizable in the sense that it isolates the prediction calculation from query processing. Hence, it enables the injection and integration of user defined prediction functions within its query processing frame-

⋆The research of these authors is supported in part by the National Science Foundation under Grants IIS-0952977 and IIS-1218168

A. M. Hendawi
Department of Computer Science
University of Virginia - Charlottesville
85 Engineer's Way, Charlottesville, VA 22904-4740
E-mail: hendawi@cs.virginia.edu

M. Ali
Center for Data Science, Institute of Technology
University of Washington Tacoma
1900 Commerce Street, Tacoma, WA 98402-3100
E-mail: mhali@uw.edu

M. F. Mokbel
Department of Computer Science and Engineering
University of Minnesota - Twin Cities
200 SE Union Street, Minneapolis, MN 55455
E-mail: mokbel@cs.umn.edu

work. (4) *Panda** deals with uncertainties and variabilities in the expected travel time from source to destination in response to incomplete information and/or dynamic changes in the underlying Euclidean space. (5) *Panda** provides a controllable parameter that trades low latency responses for computational resources. Experimental analysis proves the scalability of *Panda** in evaluating a massive volume of predictive queries over large numbers of moving objects.

1 INTRODUCTION

The massive proliferation of GPS devices, along with the wide usage of mobile phones, and the easy accessability of wireless technologies have led to a number of new-location based services applications [13, 23, 24]. These applications support queries like "what are the five nearest restaurants", or "give me the list of pharmacies within one mile of my current location". A considerable amount of research has been introduced to handle different types of queries on spatial data, such as range queries [6, 8, 33], and k-NN queries [6, 26].

Another important set of location-based services focuses on *predictive* queries [14, 15, 18, 21] in which a user asks the same previous queries but for a *future* time instance rather than the current time instance. Examples of such services include *predictive* range queries in a smart advertising system, e.g., "send e-coupons to all customers that are expected to show up around a store location in the next 30 minutes" , and *predictve* k-NN queries in ride sharing systems, e.g., "find the three vehicles expected to pass by a waiting rider's location in the next 5 minutes". *Predictive* queries are crucial to many applications. In traffic management systems, predictive queries identify the regions that are expected to be congested even before traffic builds up. In weather alarming systems, predictive queries notify the commuters about severe weather conditions in advance [22]. In service finding applications, predictive queries find the taxi that is expected to be the nearest to my location within the next few minutes. In location-based advertising, predictive queries target customers who are expected to be nearby in the next hour with sales coupons.

Most of the existing work focuses on short-term predictive queries with a single query type [15]. Short term predictions aims at predicting only the next turn, segment, and/or junction of the moving object's trajectory. This short-term prediction is not very useful in many real world applications that require prediction to be farther in the future, as described in the application scenarios above. The majority of existing techniques utilize spatio-temporal index structures to speed up the retrieval of the object's locations. However, these techniques suffer from a significant update overhead [34]. The frequent movement of an object in real life generates a stream of location updates and puts these index structures under performance pressure. Furthermore, the long-term prediction models employed by existing work are mostly based either (1) on historical data modeling or (2) on a linearity movement assumption. From practical experience [1, 9, 10], historical data is not always available in rural areas and/or not available due to the users' privacy concerns. Moreover, the linearity movement assumption is not realistic in real world. Moving objects usually have a complex movement pattern rather than moving in a straight line.

In this paper, we extend our previous work, *Panda* [11, 12]. *Panda* evaluates predictive spatial queries in Euclidean space with a focus on (i) handling predictive range queries, (ii) assuming all objects are moving objects, and (iii) considering a fixed travel time between any given two points in the space regardless of the time of the day. In this work, we introduce the extended framework *Panda**. In this extension, (1) We give *Panda** the ability to process predictive KNN query and predictive aggregate query in addition to the existing predictive range query capabilities. (2) We enable *Panda** to handle predictive queries over a mix of stationary points of interests, POIs, (e.g., restaurants) and moving data (e.g., vehicles). (3) We provide *Panda** with the ability to handle time uncertainty by considering the dynamic change in the travel time between points in the space. Thus, *Panda** reacts to changes in the underlying space based on the time of day. For example, travel time cost between two locations is cheap in the morning while it is expensive in the afternoon. To achieve that, *Panda** introduces the *travel time structure (TTS)* as a multi dimensional grid structure. The *TTS* stores the travel time cost between each pair of locations in the space at various time slots of the day. (4) We conduct comprehensive experimental evaluation based on real and synthetic data to examine the performance of *Panda** under the newly added query types and under time uncertainty.

In more elaboration, we introduce *Panda**, a system designed to efficiently support a wide variety of *predictive* spatio-temporal queries that include predictive range queries, predictive k-NN queries, and predictive aggregate queries, for stationary and moving objects. *Panda** distinguishes itself from previous attempts in predictive query processing [15, 38] in the following aspects: (1) *Panda** has ability to evaluate long-term as well as short-term prediction queries. Hence, it supports prediction up to tens of minutes, (2) *Panda** scales up to answer heavy workloads with tens of thousands of queries over a large number of moving objects in the order of tens of thousands of objects. The scalability of *Panda** is attributed to the adoption of a prediction function that filters out objects with no possibility of appearing in the query result at the a specified time window. Moreover, it prunes out the object movements that have no effect on the result, (3) *Panda** does not only answer current queries, but it also precomputes the results of frequent queries and/or frequently-queried regions in advance. This result precomputation dramatically reduces the query response time, (4) *Panda** is generic in the sense that it does not address a single type of predictive queries. Instead, it provides a generic infrastructure for a wide variety of predictive queries over both stationary and moving data, and (5) *Panda** deals with time uncertainty by modeling the travel time between different locations as a range of time with lower and upper bounds, e.g., from 10 to 15 minutes. This time range enables *Panda** to handle dynamic changes in the underlying space based on the time slot of the day.

The main idea of *Panda** is to monitor those space areas that are highly accessed using predictive queries. For such areas, *Panda** precomputes the likelihood of objects being in these areas beforehand. Whenever *Panda** receives a predictive query, it checks if parts of this predictive query are included in these precomputed space areas according to the overlap between the query region and the underlying space. If this is the case, *Panda** retrieves parts of the query answer from the precomputed areas with a very low response time. For other parts of the incoming predictive query that are not included in the precomputed areas, *Panda** has to dispatch the full predic-

tion module to find out the answer, which will take more time to compute. Worthy to mention here that *Panda** does not apply the prediction module on the whole space, instead, it limits the computation to a clipped space, since some areas are filtered out if they are not under investigation by any standing query. This filtration is a basic key of the scalability of *Panda**. Then, the overlap between the incoming query and the precomputed areas controls how efficient the query would be.

The isolation between the precomputed area and the query area presents the main reason behind the generic nature of *Panda** as any type of predictive queries (e.g., range, k-NN, aggregate) can use the same precomputed areas to serve its own purpose. Another main reason for the isolation between the precomputed areas and queries is to provide a form of *shared execution* environment among various queries. If *Panda** would go for precomputing the answer of all incoming queries in separation manner, there would be significant redundant computations among overlapped query areas.

Panda* provides a tunable threshold that provides a trade-off between the predictive query response time and the overhead of precomputing the answer of selected areas. At one extreme, we may precompute the query answer for all possible areas, which will provide a minimal response time, yet, a significant system overhead will be consumed for the precomputation and materialization of the answer. On the other extreme, we may not precompute any answer, which will provide a minimum system overhead, yet, an incoming predictive query will suffer the most due to the need of computing the query answer from scratch without any precomputations.

The underlying prediction function used in *Panda** utilizes a *long-term* prediction function, designed to predict the *final* destination of a *single* user based on the trajectory [7, 20] of his current trip. Clearly, a direct deployment of such a *long-term* prediction function does not satisfy our purpose of predictive queries that are concerned with the moving object location after some time rather than its final destination. Accordingly, *Panda** alters the prediction function to provide a location prediction after a specified future time interval (e.g., after 20 minutes). This future time interval can represent both a short-term and a long-term prediction rather than the final destination. Moreover, *Panda** considers the travel time between two points variable over time, probably due to the variability and uncertainty in the traffic patterns over the day. To handle travel time variability and uncertainty, *Panda** stores the travel time between any pair of locations in the space as time interval rather than exact value. The boundaries of the interval represent the minimum and maximum time it takes from an object to move from one location to another in the space.

To evaluate the performnace of *Panda**, its query processor is implemented and compared against two other baseline algorithms. The experiments are based on two groups of data, a synthetic data set [5] and a real data set about GPS readings collected by Microsoft [35,36]. The experiments results prove that *Panda** is scalable, efficient, and as accurate as its underlying prediction function. *Panda** achieves a workload that is at least four times bigger than the baseline algorithm without scarifying the response time.

The rest of the papers is organized as follows; Section 2 reviews related work. Section 3 gives an overview of the *Panda** system architecture, and its prediction function. Section 4 presents the generic framework for predictive query processing in

Panda* including its data structures and algorithms. Section 5 describes how *Panda** can be extended to support common predicative spatio-temporal queries. Section 6 provides the experimental analysis and performance evaluation of *Panda**. Finally, the paper is concluded in Section 7.

2 RELATED WORK

The work related to predictive query processing can be classified into three broad categories based on the underlying prediction function into: (1) predictive queries using linearity-based prediction models, (2) predictive queries using historical-based prediction models, and (3) predictive queries using other prediction models. In this section, we give an ovefrview of each category.

(I) **Predictive queries using linearity-based prediction:** [3, 25, 27, 31, 32]. The main idea of predictive query processing in this category is that their underlying prediction models are based on a simple assumption that objects move in a linear function in time. So, the query processor takes into consideration the position of a moving point at a certain time reference, its direction and the velocity, then compute and store the future positions, (using a liner function in time), of that object in a TPR-tree based index. When a predictive query is received the query processor retrieves the anticipated position in the given time [27]. The work in this category concerns with the applications of the linearity-based prediction models to answer nearest neighbor queries [25], k nearest and reverse k nearest neighbor queries [3], or to estimate the query selectivity [32]. Some of these applications attach the expiry time interval to the KNN query result [31].

(II) **Predictive queries using historical-based prediction:** [4, 7, 15, 18, 19, 28]. The main idea of this category is that the prediction models mainly rely on objects historical trajectories. Existing work in this category is either based on mobility model [15], or based on ordered historical routes [4, 7, 19] for predicting the object next trajectory. The main concern of the mobility model [15] is to answer predictive range query by focusing on the prediction of the object behavior in junctions. In the ordered historical routes, the stored historical routes are ordered according to the similarity with the current time and location of the object and the top route is considered the most possible one [4, 7, 18, 19]. In [28] the historical data is employed to approximately answer aggregate spatio-temporal queries.

(III) **Predictive queries using other predictions:** [14, 29, 37, 38]. The main idea of this category is to use more complicated functions to achieve better prediction accuracy. Some of the existing work in this category either exploit a single function [29, 38], or mix between two or more functions to form a hybrid prediction model [14, 37]. A Transformed Minkowski Sum [38] is used to answer circular region range and K-NN queries. Recursive Motion Function (RMF) [29] is used to predict a curve that best fits the recent locations of a moving object and accordingly answer range queries. In the hybrid functions category, two methods [14, 37] are combined to evaluate range and nearest neighbor queries in highly dynamic and uncertain environments.

Moreover, the related work to predictive query processing can be classified in terms of type of queries it supports. Most existing algorithms for predictive query processing have focused only on one kind of predictive queries, or two at most. These algorithms can be classified as follows:

(I) **Predictive Range queries**, i.e., [15, 29, 38]. A predictive range query has a query region R and a future time t, and asks about the objects expected to be inside the R after time t. For example, a mobility model [15] is used to predict the coming path of each of the underlying objects and employ the prediction results to evaluate predictive range queries. Most of existing work considers query region as a rectangle, whoever the Transformed Minkowski Sum is used to answer circular region range [38]. This is done by determining whether a time parameterized bounding rectangular, as a moving object, intersects a moving circle that represents range queries. The initial rectangle of the object and the velocity of each edge in this rectangle are considered to compute the position and the rectangle after a certain duration of time in the future. The transformed Minkowski sum in this method is obtained by doing two steps: (i) a coordinate transformation based on the query region an its movement, then (ii) the Minkowski enlargement in the transformed coordinates system.

(II) **Predictive K-Nearest-Neighbor queries**, i.e., [3, 16, 25, 38]. A predictive K-nearest-neighbor query has point location P, a future time t, and asks about the K objects expected to be closest to P after time t. For example, two algorithms, RangeSearch, KNNSearchBF, [38] are introduced to traverse spatio-temporal index tree (TPR/TPR*-tree) to find the nodes that intersect with the query circular region for Range and KNN queries respectively. Sometimes the expiry time interval is attached to a kNN query result [30, 31]. Thus, the kNN query answer is presented in the form of ¡*result*, *interval*¿, where the interval indicates the future interval during which the answer is valid.

(III) **Predictive Reverse-Nearest-Neighbor queries**, i.e., [3, 17]. Unlike the predictive KNN query which finds the objects expected to be the nearest to a given query region, predictive reverse nearest neighbor, RNN, query finds out the objects that expected to have the query region as their nearest neighbor. This query is useful in service distribution applications such as ad-hoc networking to assign mobile devices to the nearest communication service point. For example, the IGERN algorithm [17] is used to evaluate continuous reverse nearest neighbor queries. We can report that the area of RNN is relatively unexplored and needs more investigation.

(IV) **Predictive Aggregate queries**, i.e., [28]. A predictive aggregate query has a query region R and a future time t, and asks about the number of objects $\mathcal{N}$ predicted to be inside R after time t. For example, a comprehensive technique [28] that employs an adaptive multi-dimensional histogram (AMH), a historical synopsis, and a stochastic method to provide an approximate answer for aggregate spatio-temporal queries for the future addition to the past, and the present.

3 PANDA: SYSTEM OVERVIEW

This section define our research problem and provides an overview of the *Panda** system by briefing the system architecture which includes the main modules and

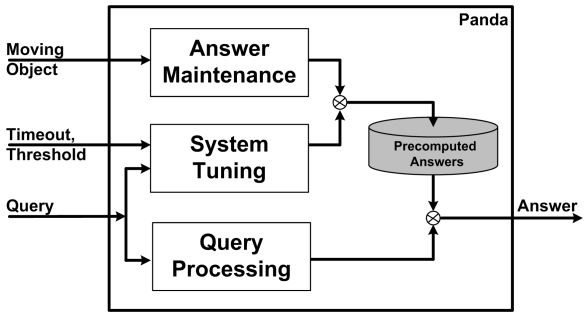


Fig. 1 The *Panda** System Architecture

events, explaining the long-term prediction function [7, 20] and our adaptation on it to be employed in the *Panda** framework.

3.1 Problem Statement

Our problem statement can be formalized as; "Given a set of moving objects sequences S, in a space partitioned into a set of grid cells $\mathcal{C}$, a prediction function $\hat{F}$, a predictive query defined by a region r, and a time period t, we need to find out the objects predicted to be inside r after the time t. This version of the problem statement is about a predictive range query where the query location is expressed as a region. However, this problem and the proposed solution are easily customizable to include other query types. For example, for predictive KNN queries, we consider the query region as a point location L rather than a region and specify a number K to guide the query processor to retrieve the K objects with highest probabilities to show up around L after t time unites. Our objective here is to introduce an efficient predictive query processor that reduces the computation time.

3.2 System Architecture

The *Panda** system consists of three main modules, namely, answer maintenance, statistics maintenance, and query processing, Figure 1. Each module is dispatched by an event, namely, an object movement, a trigger for statistic maintenance, and a query arrival, respectively. As a shared storage, a list of precomputed answers is maintained, which is frequently updated offline and used to construct the final query answer for received predictive queries. Below is a quick overview of each of these three events along with its associated event handler module. Details of these actions are discussed in Section3.

Object movement. Whenever *Panda** receives an object movement, it dispatches the answer maintenance module to check if this movement affects any of the precomputed answers. If this is the case, the affected precomputed answers are updated accordingly.

Tuning trigger. This trigger consists of a tunable threshold and/or a specified timeout interval. Whenever a change happens to the system threshold or at the end of a

timeout, the *system tuning* module is fired to prompt *Panda** that the current set of statistics that judge on which answers to precompute need to be reset. Consequently, the updated statistics affect which parts of query answers will be precomputed which in turns control the whole performance of *Panda**.

Query arrival. Once a query is received by *Panda**, the query processor divides the query area into two parts based on the answer precomputation. The first part is already precomputed where its answer is just retrieved from the precomputed storage. The second part is not precomputed and needs to be evaluated from scratch through the computation of the prediction function against a candidate set of moving objects.

3.3 Prediction Function

The long-term prediction function deployed in *Panda** is mainly an adaptation of the one introduced by Microsoft Researchers [7, 20] to predict the final destination of a single object.

F is applied to any space that is partitioned into a set of grid cells $\mathcal{C}$. It takes two inputs, namely, a cell $C_i \in \mathcal{C}$ and a sequence of cells $O_s = \{C_1, C_2, \cdots, C_k\}$ that represents the current trip of an object O. Then, F returns the probability that C_i will be the final destination of O, Equation 1.

$$F \leftarrow P(C_i|O_s) = \frac{P(O_s|C_i)P(C_i)}{\sum_{j=1}^{N} P(O_s|C_j)P(C_j)} \tag{1}$$

The term *P($O_s|C_i$)* in the numerator is the possibility of the sequence O_s of the current traversed cells by the object O given the destination cell C_i. This term can be computed using the traveling efficiency parameter *E*, Equation 2 which measures to what extend objects on the space follow the shortest path in its movements from sources to destinations.

$$P(O_s|C_i) = \prod_{k=2}^{n} \begin{cases} E & \text{if cell } C_k \text{ in } O_s \text{ is closer to } C_i \text{ than the cell } C_k - 1 \\ 1 - E & \text{otherwise} \end{cases} \tag{2}$$

This parameter *E* varies from object to another and it can be obtained by examining the most recent bunch of trajectories of each object. It is also possible to get one value for the system as whole. This is done by taking the average of traveling efficiency of all objects on the space. For example, the analysis performed by John Krumm in [20] found that *E* is around 0.68. In our set of experiments, we set one *E* value for the *Panda** system. *P(C_i)*, the second term in the numerator, is the previous probability of C_i to be a destination for O_s. Initially this term is set to 1/n, where n is the number of cells in the grid. The denominator is a normalization factor to sum up all probabilities for all cells in the grid to one, given the recent rout of an object.

The way the prediction function works is demonstrated in Figure 2, where the given space in which the objects move is partitioned into 6×6 squared cells numbered from 1 to 36. The current trajectory of the moving object O_1 is drawn as a line

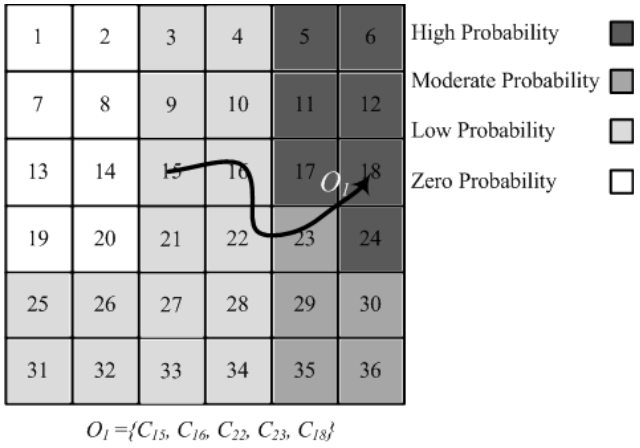


Fig. 2 Destinations Probabilities Based on Object Sequence

started at cell C_{15} and headed to cell C_{18}. The sequence of cells representing O_1 in its current trip is $S_{O_1} = \{C_{15}, C_{16}, C_{22}, C_{23}, C_{18}\}$. The color of a cell indicates its probability of being a destination to the object O_1 given its sequence S_{O_1}, the darker the cell color, the higher the probability. As the object moves toward its final trip destination, the prediction function updates its computation. So, some of the grid cells become more probable destination (e.g. C_{24}), and others become less probability, (e.g. C_{31}).

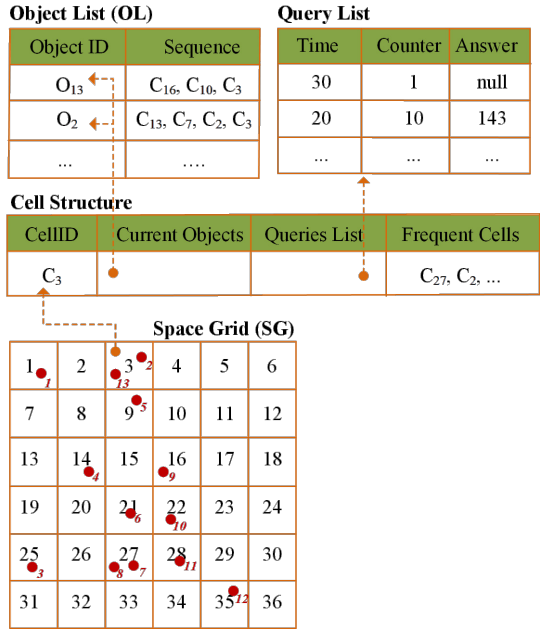
As F only predicts the destination of an object, it does not have the sense of time. In other words, F cannot predict where an object will be after time period t. Since this is a core requirement in *Panda**, we adapt F to be able to compute the probability that object O will be passing by the given cell C_i after time t, where t is specified in the predictive query. The adaptation results in the function $\hat{F}$, Equation 3, which is a normalization of the results from the original prediction function F using the set of cells D_t that could be a possible destination of an object O after time t.

$$\hat{F} \leftarrow P(C_i|O_s, t) = \frac{P(C_i|O_s)}{\sum_{d \in D_t} P(C_d|O_s)} \tag{3}$$

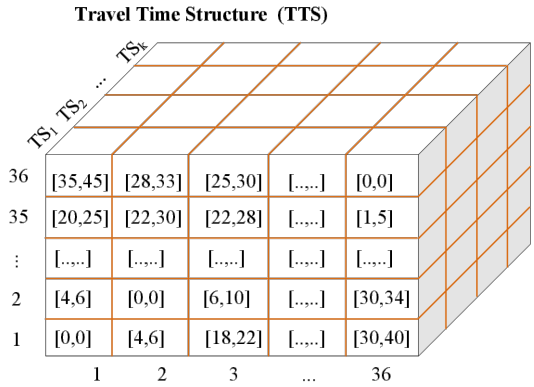
Here, the numerator is the output of the original prediction function *F*, and the denominator is the summations of the probabilities of all grid cells in D_t, also computed from F.

4 Panda*: A Predictive Spatio-Temporal Query Processing

A salient feature of *Panda** is that it is a generic framework that supports a wide variety of predicative spatio-temporal queries. *Panda**'s query processor can support range queries, aggregate queries, and k-nearest-neighbor queries within the same framework. In addition, *Panda**'s query processor can support stationary as well as moving data. Finally, *Panda** is easily extensible to support continuous queires. This generic feature of *Panda** make it more appealing to industry and easier to realize in



(a) Grid Index Structure



(b) Travel Time Structure

Fig. 3 Data Structures in *Panda**

real commercial systems. This is in contrast to all previous work in predictive spatio-temporal queries that focus on only one kind of spatio-temporal queries. As described in Figure 1, *Panda** reacts to three main events, namely, query arrival, object movement, and a trigger for statistics maintenance. Each event prompts *Panda** to call one of its three main modules to take the appropriate response. This section discusses the details of each main module. The section first starts by describing the underlying data structure of *Panda** (Section 4.1). Then, the generic query processor, answer maintenance, and system tuning are described in Section 4.2, 4.3, and 4.4, respectively.

Following the spirit of *Panda**, the discussion in this section is made generic without referring to a particular predictive query type, except when giving examples. The extensibility of *Panada** to support various predictive query types will be described in next section (Section 5).

4.1 Data Structure

Figure 3 depicts the underlying data structure used by *Panda**. A brief overview of each data structure is outlined below:

Space Grid SG. *Panda** partitions the whole space into $N \times N$ grid cells. For each cell $C_i \in SG$, we maintain: (1) *CellID* as an identifier, (2) *Current Objects* as the list of moving objects located inside C_i, (3) *Query List* as the list of predictive queries issued on C_i. Each query Q in this list is presented by the tuple ($Time$, $Counter$, $Answer$), where $Time$ is the future time included in Q, $Counter$ is the number of times that Q is issued to *Panda**, $Answer$ is the precomputed answer for Q which may have different format based on the type of Q, e.g., for predictive range query, it carries the list of objects expected to compose the answer, while in predictive K-NN, it contains the k objects anticipated to satisfy the query conditions., (4) *Frequent Cells* as the list of cells that one of their precomputed answers should be updated with the movement of an object in C_i. For example, as provided in Figure 3, the cell C_3 has a precomputed answer for the future time t = 20 minutes, while the answer for t = 30 is not be precomputed, and hence, it should be computed from scratch when a query with the same future time is received. It is important to notice here that N is tuned based on the application requirements. When N is large, this means the application requires the prediction to be more precise as the cells size will be smaller, and vice versa. For example, an advertising application would choose a bigger N to make sure to allocate the user's future location around a store area. In a severe weather management application, the N could be smaller as it will be enough to forecast the tornado , as a moving object, future location at the level of a city.

Object List OL. This is a list of all moving objects in the system. For each object $O \in OL$, we keep track of an object identifier and the sequence of cells traversed by O in its current trip. For example, as illustrated in Figure 3, O_2 in its current trip, has passed through the sequence of cells $\{C_{13}, C_7, C_2, C_3\}$ that means it started at C_{13} and it is currently moving inside C_3.

Travel Time Structure TTS. This is a three-dimensional array of $N^2 \times N^2 \times TS$ cells where each cell $TTS[i, j, k]$ has the travel time interval between space cells C_i and C_j, at time slot TS_k where C_i and $C_j \in SG$ and TS is the time slots of the day. TTS is fully pre-loaded into *Panda** and is a read-only data structure. For example, as illustrated in Figure 3(b), the travel time from C_1 to C_{36} takes from 35 to 45 minutes while it takes 22 to 30 minutes to travel from C_2 and C_{35} at time slot TS_1. According to the underlying set of moving objects, the value inside a cell in the TTS might be stored as an exact value, e.g., average, or interval of time, e.g., [min,max]. By visiting this travel time structure, we find out the set of possible destination cells D_t, mentioned in Section 3.3, to a specific cell C after time t. This is done by reading the array of time intervals corresponding to the current cell C in the present time slot

TS. Then, we check if the future time t intersects with any of these time intervals, the cell of this interval will be added to the possible destinations D_t.

4.2 Generic Query Processing in Panda

The generic query processing of *Panda** does not only predict the query answer, but it also prepares partial results of the incoming queries before hand. In general, *Panda** does not aim to predict the whole query answer, instead, it predicts the answer for certain areas of the space. Then, the overlap between the incoming query and the precomputed areas controls how efficient the query would be. If all the query is precomputed, the query will have best performance in terms of lower latency, however, the *Panda** system will encounter high overhead of maintaining the precomputed answer. This isolation between the precomputed area and the query area presents the main reason behind the generic nature of *Panda** as any type of predictive queries (e.g., range and k-nearest-neighbor) can use the same precomputed areas to serve its own purpose. Another main reason for the isolation between the precomputed areas and queries is to provide a form of *shared execution* environment among various queries. If *Panda** would go for precomputing the answer of incoming queries, there would be significant redundant computations among overlapped query areas.

The *Panda** query processor utilizes its grid structure $\mathcal{G}$ to decide on precomuting the answer for some specific cells of the $\mathcal{G}$. Upon the arrival of a new predictive spatio-temporal query Q, with an area of interest R, requesting a prediction about future time t, *Panda** first divides Q into a sets of grid cells C_f that overlap with the query region of interest R. For each cell $c \in C_f$, *Panda** goes through two main phases, namely, *result computation* and *statistic maintenance*.

The result computation phase (Section 4.2.1) is responsible on getting the query result from cell c either as a precomputed result or by computing the result from scratch. The *statistic maintenance* phase (Section 4.2.2) is responsible on maintaining a set of statistics that help in deciding whether the answer of cell c, for a future time t, should be precomputed or not.

The precomputation at cell c will significantly help for the next query that asks for prediction on c with the same future time t, yet, precomputation will cause a system overhead in continuously maintaining the answer at c. Throughout this section, Algorithm 1 gives the pseudo code of the *Panda** query processor where the first three lines in the algorithm finds out the set of cells C_f that overlaps with the query region R, and start the iterations over these cells.

4.2.1 Phase I: Result Computation.

Phase I, *result computation*, receives a predictive query Q, either as range, aggregate, or k-nearest-neighbor, asking about future time t and a cell c_i that overlaps with the query region of interest R. The output of this phase is the partial answer of Q computed from c_i. The following describes the main idea, algorithm, and an example of Phase I.

Algorithm 1 *Panda** Predictive Query Processor

Input: query region R, future time t

```
1: QueryResult ← null, CellResult ← null
2: C_f ← the set of grid cells intersecting with (R)
3: for each cell c_i ∈ C_f do
4:     /* Phase I: Result Computation */
5:     if there is an answer in c_i at time t then
6:         CellResult ← read answer from c_i
7:     else
8:         C_R ← the set of grid cells reachable to c_i in time t at the current time slot
9:         for each cell c_j ∈ C_R do
10:            for each object O ∈ current objects in c_j do
11:                ObjectPrediction ← Compute F̂ = P(c_i|O, t)
12:                UpdateResults (CellResult, ObjectPrediction)
13:            end for
14:        end for
15:    end if
16:    UpdateResults (QueryResult, CellResult)
17:    /* Phase II: Statistics Maintenance */
18:    e ← the entry in the query list of c_i at time t
19:    if e is NULL then
20:        e ← Insert a new blank entry e to the query list of c_i with e.Counter=0 and e.Answer is Null
21:    end if
22:    e.Counter ← e.Counter + 1
23:    if e.Counter ≥ System threshold T AND e.Answer is NULL then
24:        e.Answer ← CellResult
25:        C_R ← the set of grid cells reachable to c_i in time t at the current time slot
26:        Add c_i to the list of frequent cells in all cells in C_R
27:    end if
28: end for
29: Return QueryResult
```

Main idea. The main idea of Phase I is to start by checking if the query answer at the input cell c_i is already computed. If this is the case, then Phase I is immediately concluded by updating the query result Q by the precomputed answer of c_i. If the answer at c_i is not precomputed, then, Phase I will proceed by computing the answer of c_i from scratch. Phase I avoids the trivial way of computing the prediction function of all objects in the system to find which objects can make it to the query answer at future time t. Instead, Phase I applies a smart *time filter* to limit its search to only those objects that can possibly reach to cell c_i within the future time t. Basically, Phase I utilizes the *Travel Time Structure (TTS)* to find the set of cells C_R that may include objects reachable to c_i within time t. Then, we calculate the prediction function for only those objects that lie within any of the cells in C_R. The result of these prediction functions pile up to build the answer result produced from c_i.

Algorithm. The pseudo code of Phase I is depicted in Lines 4 to 16 in algorithm 1. Phase I starts by checking if the answer of c_i at time t is already precomputed in its own *Query List* entry in the grid data structure $\mathcal{G}$. If this is the case, we just retrieve the precomputed answer as the complete cell answer (Line 6 in Algorithm 1), and conclude the phase by using the cell result to update the final query result (Line 16 in Algorithm 1). Updating the result is done through the generic function *UpdateResults*

that takes two parameters, the first is the result to be updated, and the second is the value to be used to update the result. The operations inside this functions depend on the underlying query type, e.g., aggregate, range, or k-nearest-neighbor queries. Details of this generic function will be described in Section 5.1. In case that the answer of cell c_i is not precomputed, we start by computing this answer from scratch (Lines 8 to 14 in algorithm 1). To do so, we apply a *time filter* by retrieving only the set of cells C_R that can be reachable to c_i within the future time t by checking the *Travel Time Structure (TTS)* and find out which cells have the potential to send objects to c_i within time t at the current time slot TS_k of the day. We visit just the slice for the current time slot of the day in the TTS. Only those objects that lie within any of the cells of C_R may contribute to the final cell answer, and hence the query answer. For each object O in any of the cell of C_R, we utilize our underlying prediction function, described in Section 3.3, to calculate the predicted value of having O in c_i within time t (Line 11 in Algorithm 1). We then use this predicted value to update the result of cell c_i using the generic *UpdateResults* function. Once we are done with computing all the predicted values of all objects in any of the cell of C_R, we again utilize the generic function *UpdateResults* to update the final query result by the result coming from cell c_i (Line 16 in Algorithm 1).

Example. Figure 4 gives a running example of Phase I in *Panda** where there are 19 objects, O_1 to O_{19} laid on a 6 $\times$ 6 grid structure of 36 cells. Figure 4(a) indicates the arrival of a new predictive range query Q_{30}, a shaded rectangle in cell C_{19}, that asks about the set of objects that will be in the area of Q_{30} after 30 minutes. Though we are using a range query as a running example, all ideas here are applied to aggregate and k-nearest-neighbor queries as well. In Figures 4(b), we find out all the cells that overlap the area of query Q_{30}. For ease of illustration, we intentionally have Q_{30} covering only one cell, C_{19}, in which we are going to carry on for the next steps. If Q_{30} covers more than one cell, then, the next steps will be repeated for each single cell covered by Q_{30}. Figure 4(b) also gives the *Query List* structure of C_{19}, where two previous predictive queries came at this cell before; a query that asks about 30 minutes in future, and it came only one time before (*counter* = 1) and another query that asks about 20 minutes in the future and were issued 10 times before. By looking at this data structure, we find that the answer of the future time t is set to *null*, i.e., it is not precomputed. In this case, we need to compute the answer for this cell from scratch. Note that if this query was asking about the set of objects after 20 minutes, we would just report the answer as $\{O_1, O_8\}$ as it is already precomputed. Unfortunately, for the case of t = 30, we need to proceed for more computations.

Figure 4(c) starts the process of computing the answer of cell C_{19}. As a first step, we utilize the *Travel Time Structure (TTS)* to find out the set of cells that are reachable to C_{19} within 30 minutes. We find that there is only three cells that can contribute to the answer of C_{19}, namely, C_9, C_{16}, C_{33}. This means that objects that are not located in any of these cells are not going to make any contribution to C_{19} within 30 minutes. For example, an object O_3 in C_{25} is likely to be far away from C_{19} in 30 minutes, (i.e., assuming it keeps moving), and thus there is no need to consider it in computation at all. The travel time filter plays an important role in filtering out large number of objects that are not going to contribute to the query result. Then, we can only focus on the objects located in C_9, C_{16}, C_{33}, where there are only four objects O_5, O_9, O_{18},

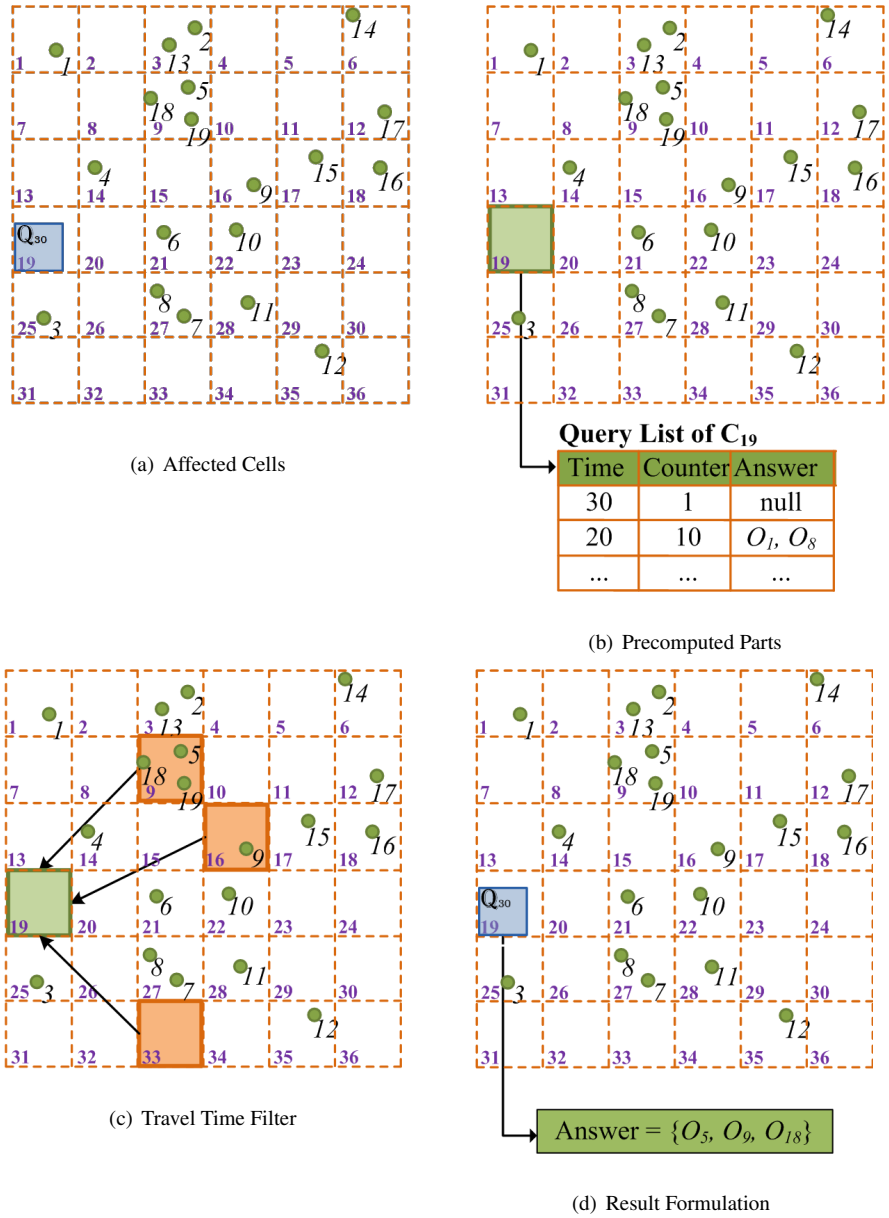
and O_{19}. For each of these four objects, we calculate the prediction function $\hat{F}$ to find out the probability that these objects can be in C_{19} in 30 minutes. With probability calculation, we find out that O_{19} has a zero probability of being in C_{19} in 30 minutes, while the other three objects have a non-zero probability. We finally report the answer in Figure 4(d) as $\{O_5, O_9\, O_{18}\}$ along with the probabilities of these objects being in C_{19} in 30 minutes.

4.2.2 Phase II: Statistics Maintenance

Phase II, *statistics maintenance*, does not add anything to the query answer. Instead, Phase II updates a set of statistics that help in deciding what parts of the space and queries need to be precomputed. The input to this phase is the cell c_i and its answer list, computed in Phase I. Then, Phase II uses this information to update the statistics maintained by *Panda**.

Main idea. The main idea of Phase II is to employ a tunable threshold, $0 \leq \mathcal{T} \leq \infty$, that provides a trade-off between the predictive query response time and the overhead for precomputing the answer of selected areas. At one extreme, $\mathcal{T}$ is set to 0, which means that all queries will be precomputed beforehand. Though this will provide a minimal response time for any incoming query, yet, a significant system overhead will be consumed for the precomputation and materialization of the answer. On the other extreme, $\mathcal{T}$ is set to ∞, which means that nothing will be precomputed at all and all incoming queries need to be computed from scratch. This will provide a minimum system overhead, yet, an incoming predictive query will suffer from high latency. Given a fixed value of $\mathcal{T}$, *Panda** smartly decides which parts of the space should be precomputed. To efficiently utilize the tunable threshold $\mathcal{T}$, Phase II keeps a *counter* for each kind of predictive query arriving at each cell. If this *counter* exceeds the threshold value $\mathcal{T}$, then, this query is considered frequent, and the answer of this query in cell c_i is precomputed. In addition, we add cell c_i to the list of frequent cells in all cells that are reachable to c_i within time t. This is mainly to say that any object movement in any of these reachable cells will affect the result computed (and maintained) at cell c_i. This list of reachable cells to c_i within time t can be directly obtained from the *Travel Time Structure (TTS)*. It is worthy to mention here that there are two things to consider when visiting the travel time structure. (1) We need to read the slice of the data related to the current time slot of the day. (2) We do not search for those cells that are exactly reachable within the specified time slot, rather, their travel time intervals just need to contain the future time in the query. For example, if the travel time between the c_i and the query cell c_q at the current time slot is [8,12] minutes and the query future time is 10 minutes, this means c_i is reachable to c_q.

Algorithm. The pseudo code of Phase II is depicted in Lines 18 to 27 in algorithm 1. Phase II starts by retrieving the entry e from the *query list* of c_i that corresponds to the querying time t. If there is no such prior entry, i.e., e is *NULL*, we just add a new blank entry in the *query list* of c_i for time t, with *counter* set to zero, and *answer* set to null (Lines 18 to 21 in algorithm 1). Then, we just increase the *counter* of e by one to update the number of times that a query arrives at cell c_i with time t. Then, we check the *counter* of this incoming query against the system threshold and the value of the current cell *Answer*. This check may result in three difference cases



(a) Affected Cells

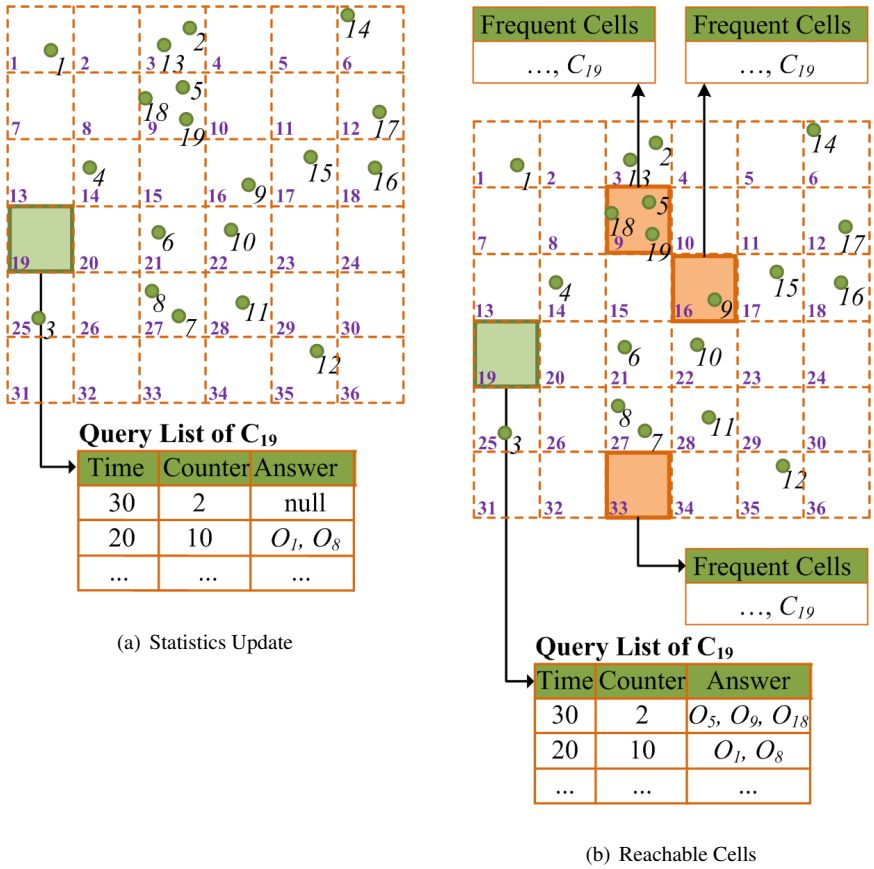
(b) Precomputed Parts

(c) Travel Time Filter

(d) Result Formulation

Fig. 4 Phase I Example.

as follows: (1) $e.counter < \mathcal{T}$, i.e., the *counter* is less than the system threshold $\mathcal{T}$. In this case, Phase II decides that it is not important to precompute the result of this query, as it is not considered as a frequent query yet. So, Phase II is just concluded. (2) $e.Answer \neq$ NULL. In this case, the query time t is already considered frequent and the answer is already precomputed. In this case, Phase II will also just conclude as there is no change in status here. (3) $e.counter \geq \mathcal{T}$ AND $e.Answer$ is NULL. This



(a) Statistics Update

(b) Reachable Cells

Fig. 5 Phase II Example.

case means that the query time t has just become a frequent one, and we need to start precomputing the result for t at cell c_i. In this case, we first add the computed cell result from Phase I to the the answer of e. Then, we find out the set of cells C_R that are reachable to cell c_i within time t. For these cells, we add cell c_i to their list of frequent cells. This is mainly to say that any object movement of any cell $c_j \in C_R$ will affect the result computed at cell c_i (Lines 18 to 23 in algorithm 1).

Example. Figure 5 gives a running example of Phase II continuing the computations of Phase I on the example of Figure 4. Figure 5(a) shows that the *counter* of the time entry 30 is updated to be 2. Assuming the time threshold $\mathcal{T}$ is set to 2. Then, the time t is now considered frequent. Figure 5(b) depicts the actions taken by Phase II upon the consideration that the incoming query with time t becomes frequent. First, the query list of C_{19} is updated to be the computed answer from Phase I. Second, the cell C_{19} is added to the list of frequent cells for C_9, C_{19}, and C_{33} to indicate that any movement in these three cells may trigger a change of answer for cell C_{19}.

Algorithm 2 Answer Maintenance

Input: *Object* O, *Cell* C_{old}, *Cell* C_{new}

1: **if** C_{old} =C_{new} **then**
2: **Return**
3: **end if**
4: Add O to the set of *current objects* of C_{new}
5: $ts \leftarrow$ current time slot of the day
6: $\mathcal{C} \leftarrow$ The set of *frequent cells* of C_{new}
7: **for each** cell $C_i \in \mathcal{C}$ **do**
8: $t \leftarrow$ travel time from C_{new} to C_i from $TTS[new, i]$ at ts
9: ObjectPrediction $\leftarrow$ Compute $\hat{F}$ = $P(C_{new}|O, t)$
10: **UpdateResults (CellResult, ObjectPrediction)**
11: **end for**
12: Remove O from the set of *current objects* of C_{old}
13: $\mathcal{C} \leftarrow$ The set of *frequent cells* of C_{old}
14: **for each** cell $C_i \in \mathcal{C}$ **do**
15: **UpdateResults (CellResult,** O**)**
16: **end for**
17: **Return**

4.3 Answer Maintenance

As has been discussed in the previous section, the efficiency of the *Panda** generic query processor relies mainly on how much of the query answer is precomputed. Though we have discussed how *Panda** takes advantage of the precomputed answers, we did not discuss how *Panda** maintains those precomputed answers, given the underlying dynamic environment of moving objects. This section discusses the *answer maintenance* module in *Panda**, depicted in Figure 1, which basically triggered with every single object movement.

Main idea. The main idea behind the *answer maintenance* module is to check if this object movement has any effect on any of the precomputed answers. If this is the case, then *Panda** computes this effect and propagates it to all affected precomputed answers. If *Panda** figures out that this object movement has no effect on any of the precomputed answers, then, it just does nothing for this object movement. As our underlying prediction function $\hat{F}$ mainly relies on the sequence of prior visited cells for a moving object, an object that moves within its grid cell will have no effect on any of the precomputed answers. Basically, movement within the cell does not change the object predication function, and hence will not have any effect on any of the precomputed answers. It is important to note that the *answer maintenance* module does not decide upon which parts of the queries/space to be precomputed, as this decision is already taken by the statistics collected in the generic query processor module. Instead, the *answer maintenance* module just ensures efficient and accurate maintenance of existing precomputed answers.

Algorithm. Algorithm 2 gives the pseudo code of the *Panda** *answer maintenance* module. The algorithm takes three input parameters, the moved object O, its old cell C_{old} before movement, and its new cell after movement C_{new}. The first thing we do is to check if the new cell is the same as the old cell. If this is the case, the algorithm immediately terminates as this object movement will not have any effect

on any of the precomputed cells. On the other side if the new cell is different form the old one, the algorithm proceeds in two parts. In the first part (Lines 5 to 11 in Algorithm 2), we first add O to the set of current objects of C_{new}. Then, we retrieve the set of *frequent cells* of C_{new}, i.e., those cells that have precomputed answers and may be affected by any change of objects in C_{new}. For each cell C_i in the set of *frequent cells*, we do: (a) retrieve the travel time t from the new cell to C_i from the *Travel Time Structure*, (b) compute the predicted value of O being in C_i after t time units, and (c) update the precomputed result at cell C_i by the predicated value, using the generic function *Update Results*. The second part of the algorithm (Lines 12 to 16 in Algorithm 2) is very similar to the first part, except we are working with C_{old} instead of C_{new}, where we remove O form the set of objects of C_{old}, we update all the precomputed frequent cells of C_{old}. A major difference here is that we update the precomputed result by removing O and its probability from it. It is important to notice here that we do not need to compute the object prediction as it is already stored in the precomputed answer at C_i.

Example. Back to our running example in Figure 5(b) that illustrates the precomputed answer for the query Q_{30} in cell C_{19}. Assume that object O_9 moves out from its cell C_{16} to C_{17}. So, we add O_9 to the list of *current objects* in C_{17}, and get its list of *frequent cells*, only C_1 is there. Then, we obtain the time t between C_1 and C_{17} as 40. We then compute $\hat{F} = P(C_1|O_9, 40)$ which gives the probability that O_9 will be in C_1 after 40 time units. We then incrementally update the answer at C_1 by the value of $\hat{F}$. We do the same for C_{16}, the cell that O_9 has just departed. We delete O_9 from the list of *current objects* in C_{16} as this object is no longer inside it. Then, we read the list of *frequent cells* of C_{16} which returns C_{19}, and we get the time t between C_{16} and C_{19} as 30. At this point, we do not need to compute $\hat{F} = P(C_{19}|O_9, 30)$ because it is already stored in the *query list* of C_{19}. All what we do here is updating the answer in C_{19} by removing O_9 and its probability.

4.4 System Tuning

In the previous two sections, we discussed the first two modules in the *Panda** system, the *query processor* and the *answer maintenance*. As provided, the *query processor* is responsible for processing the incoming predictive queries and deciding which parts to precompute in advance based on the collected statistics and a system threshold $\mathcal{T}$. The *answer maintenance* concerns with maintaining the answers in those precomputed parts such that it always fresh and ready for retrieval.

At this point, if we leave *Panda** to run forever and to precompute a query answer when its frequency exceeds a threshold $\mathcal{T}$, we will end up to precompute the answer for all queries in advance. In this situation, a significant computational overhead will be added as every single object movement will cause updates to all *query lists* in all grid cells which makes *Panda** having the worst possible efficiency. Another problem is that some queries show up with high frequency rate during a certain time duration (i.e. one hour) then disappear or come rarely for long period (i.e. hours or days). Accordingly, it is meaningless to keep precomputing and updating the answer for queries with this behavior for long time. Obviously, it will be better to stop this pre-

Algorithm 3 System Tuning

Input: Threshold $\mathcal{T}$

```
1: for each cell c_i ∈ the Grid G do
2:     for each entry e ∈ c_i.'Query List' do
3:         if e.Counter ≥ T AND e.Answer is NULL then
4:             e.Answer ← Compute the predicted answer for c_i after t = e.Time
5:             Add c_i to the list of frequent cells in each of its reachable cells
6:         else if e.Counter < T AND e.Answer ≠ NULL then
7:             e.Answer ← NULL
8:             Remove c_i from the list of frequent cells in each of its reachable cells
9:         end if
10:        if e.Answer is NULL then
11:            Delete e from c_i.'Query List'
12:        else
13:            e.Counter ← 0
14:        end if
15:    end for
16: end for
17: Return;
```

computation and even forget about those queries and when they come, just compute their answers. A third reason behind the need for the *system tuning* module is that it is infeasible to have an accurate, and a detailed workload information before putting *Panda** in real execution environment. Consequently, it must own a mechanism to be sensitive to the changes in the workload patterns. For these reasons, this section introduces the *system tuning* module that allows *Panda** to periodically analyze the queries behavior during a recent timeout (i.e. one hour) to predict the coming queries for the next timeout, and yet refine its decision about which parts to precompute and which to stop their precomputing.

Idea. The idea of the *system tuning* module is that it periodically prompts *Panda** to adapt its decision toward the precomputed parts by examining the collected statistics about the received queries during the past timeout (i.e. hour or day). If a query had high frequency rate , then we predict it will be frequent during the next timeout too. Intuitively, *Panda** keeps precomputing its answer and makes it fresh for queries in the coming timeout. On the other side, if a query was initially precomputed and the collected statistics during the previous timeout indicate that it became non frequent, we anticipate that it will show up rarely during the next timeout. Yet, *Panda** must refines its decision and stops the precomputation for this query. Actually, we can use more intensive analysis and prediction model here to predict the coming queries for the next timeout, however, we want to keep it simple to avoid adding extra overhead which will downgrade the whole system efficiency.

Algorithm. Algorithm 3 gives the pseudo code for the *system tuning* module used to control the system efficiency by sustaining the used statistics to reflect the recent picture of the system. The $counter$ and the $answer$ fields are the most important ones that need to be kept up to date instantly. Thus, they identify which space areas will be precomputed and which will not be during the next query processing timeout period T_{out}. The algorithm takes the recent value of the threshold $\mathcal{T}$ as an input. This value is used to control the efficiency during the coming timeout T_{out}. The algorithm

navigates through the grid data structure $\mathcal{G}$ and examines the entries in the *query list* in each grid cell c_i. For each entry e that represent a query future time value t in the $e.time$ field, we check both the $e.counter$ and the $e.answer$ fields (Lines 2 to 9 in algorithm 3). This check has four possible alternatives.

(1) $e.counter < \mathcal{T}$ and $e.answer$ is NULL. In this case, the null value in the $answer$ filed means that the query with time t was not frequent (has no precomputed answer) during the ended timeout T_{out}, while the $counter$ value less than the threshold means that it is predicted not be a frequent query in the next T_{out} too. Accordingly, it will not be useful any more to keep the entry for t, yet, this entry e is deleted from the *query list* of c_i (Lines 10 to 12 in algorithm 3).

(2) $e.counter \geq \mathcal{T}$ and $e.answer \neq$ NULL. In this case, the query with the time t is already considered frequent and the answer is already precomputed, and it will remain frequent during the coming timeout T_{out}. Consequently, no action is required rather than resetting its counter to zero (Line 13 in algorithm 3).

(3) $e.counter \geq \mathcal{T}$ and $e.answer$ is NULL. This case means that the query time t has switched its status to be a frequent one during the next timeout T_{out}. Instantly, we precompute the result for t at cell c_i and store it in the $answer$ field.Then, we populate the cell c_i to the list of *frequent cells* in its reachable cells within time t (Lines 3 to 5 in algorithm 3).

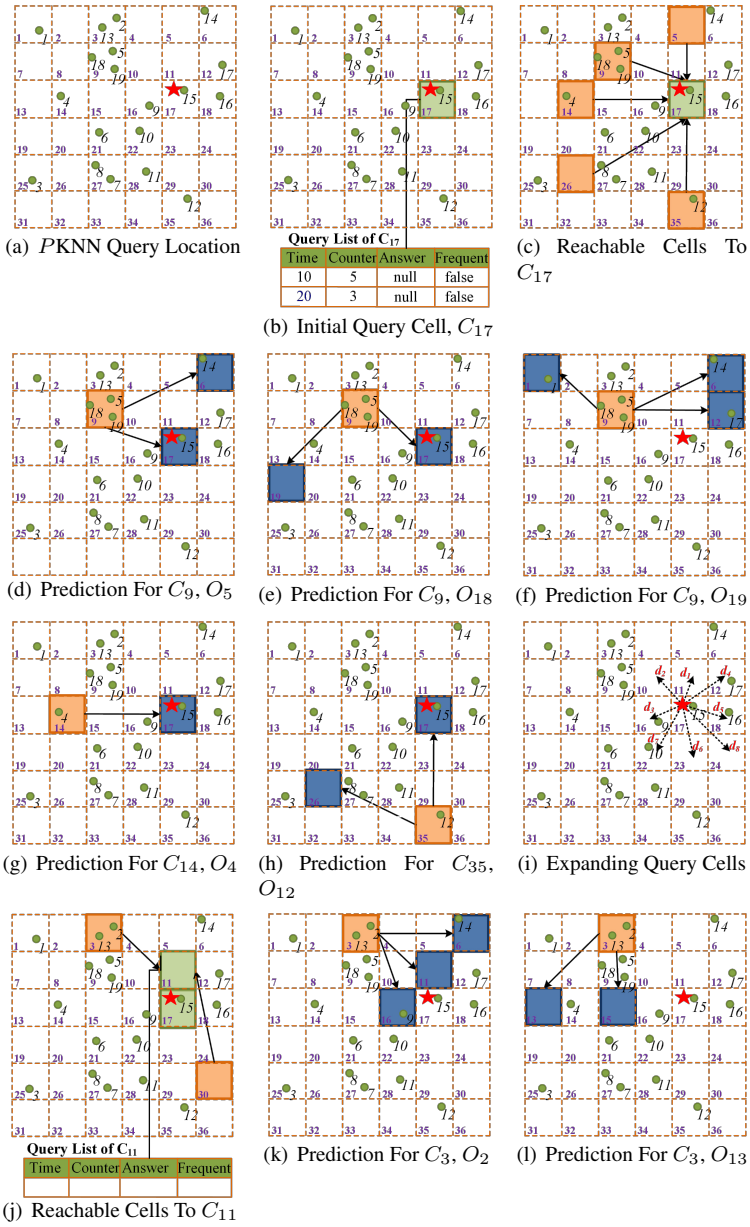
(4) $e.counter < \mathcal{T}$ and $e.answer \neq$ NULL. This case is the opposite to the previous which means that the query time t has switched its status to be non frequent during the next timeout T_{out}. Yet, we empty its $answer$ field by setting it to NULL, and delete the cell c_i from the list of *frequent cells* in its reachable cells within time t.

In all cases, the $e.counter$ fields of the remaining frequent queries are set to zero such that the decision at the end of each timeout T_{out} is affected only by the number of queries received during that recent T_{out} (Line 13 in algorithm 3).

Example. After running *Panda** for one hour, we had the precomputed answer in the cell C_{19} in our example in Figure 5(b). Assuming the used threshold $\mathcal{T}$ value is three, then we check the entries in the query list in the cell C_{19}. The $counter$ value in the first record (where $time = 30$) is less than 3 and its $answer$ is $\neq$ NULL, then we set this precomputed answer to NULL and remove C_{19} from the list of *frequent cells* of of its reachable cells, ($\{C_9$, C_{16}, and $C_{33}\}$), then the $counter$ is set to zero. For the second record (where $time = 20$), the $counter$ is greater than $\mathcal{T}$ and the $answer$ has a non NULL value, therefore, we keep everything as it is except the $counter$ which will reset to zero too. For the next timeout, the query list of C_{19} will not contain the record for $t = 30$ since it is not frequent.

5 Extensibility of Panda*

In the previous section, we have discussed the generic framework of *Panda** as a predictive spatio-temporal query processor, and we have elaborated its main modules that compose its core. In this section, we illustrate how *Panda** can be extended to support a wide variety of predicative spatio-temporal queries and how it can be harmonized according to the nature of the underlying data. Basically, this is accom-



(a) PKNN Query Location (b) Initial Query Cell, C_{17} (c) Reachable Cells To C_{17}

(d) Prediction For C_9, O_5 (e) Prediction For C_9, O_{18} (f) Prediction For C_9, O_{19}

(g) Prediction For C_{14}, O_4 (h) Prediction For C_{35}, O_{12} (i) Expanding Query Cells

(j) Reachable Cells To C_{11} (k) Prediction For C_3, O_2 (l) Prediction For C_3, O_{13}

Fig. 6 Predictive KNN Query Illustrative Example

plished through the optimized implementation of the generic function *UpdateResults* to serve the needs of the underlying *query type*, e.g., aggregate, range, or k-nearest-neighbor queries, and the underlying *data nature*, e.g., moving data, or stationary data.

5.1 Query Type

In this section we study the extensibility of *Panda** to support the evaluation of three main predictive query types, namely range, K-NN, and aggregate query. However, this extensibility is not limited to these types only because the universal nature of the aforementioned data structures and algorithms that can be easily tailored to other feasible predictive spatio-temporal queries.

5.1.1 Range Query Processing

Idea. A predictive range query is defined by two elements, a rectangular query region R and a future time t, and asks about the objects expected to be inside the determined query region after the specified future time. *Panda** starts the range query processing by getting the grid cells that overlap the query region. Those cells are divided into two groups. The first one contains the cells that already have precomputed answers that we need to retrieve them only without further processing, while the second group contains the overlapped cells that their answers have to be computed from scratch. For each cell c_i in the second group, *Panda** visits the travel data structure *TTS* and gets the list of the reachable cells C_R to the cell in hand c_i. For each object in a reachable cell, *Panda** applies the prediction function and checks if that object is predicted to arrive at c_i after the desired future time, if this is the case, this object with its probability is appended the result of c_i. To do that, the *UpdateResult* function is implemented to precisely serve the predictive range query processing such that it is able to build up the result of the cell in hand as well as the final query result by continue appending the objects identifiers and with their probabilities. Also, the *UpdateResult* function is used to form the final result of a range query by merging the objects in the precomputed cells and those that have just been computed. Yet, the returned answer encompasses the list of objects expected to be inside the overlapped cells after the desired time unites in the future.

Example. The illustrated example in the two phases in Section 4.2 is sufficient to explain the processing of predictive range query.

5.1.2 K-NN Query Processing

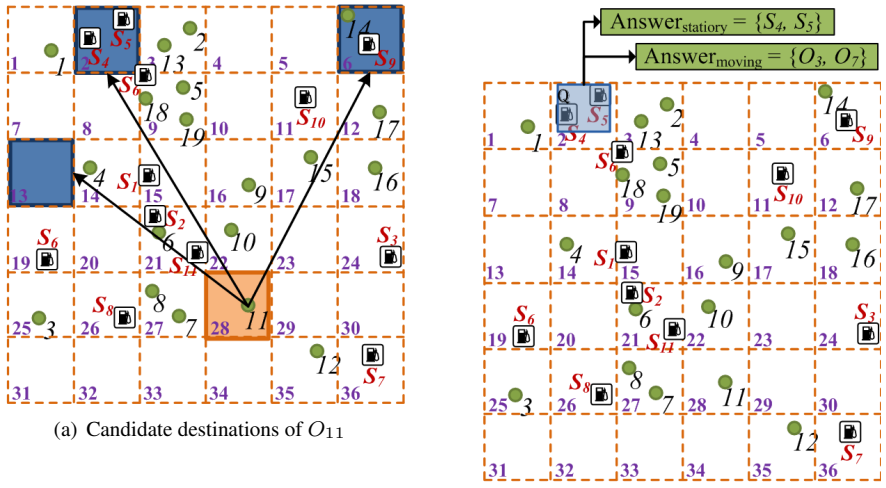
Idea. A predictive K-NN query has two parameters, a specified location point and a future time, and enquires about the K objects expected to be the nearest to that location after the given time. Initially, *Panda** locates that point into a corresponding grid cell which will be checked if having a precomputed answer for the K or more objects expected to appear at this cell after the determined future time. If this is the case, the precomputed answer is simply returned without extra computation. Otherwise, in the case that there is no precomputed answer, *Panda** computes it in similar way to range query in order to find the satisfactory K objects. In the case that the precomputed answer or the yet computed answer has objects less than the desired K, *Panda** expands the query location by adding the nearest adjacent cell and recall the original computation steps on this recently added cell. This process is repeated until the query is satisfied. Definitely, *Panda** has a different interpretation of the term

nearest here, which reflected as the objects with the highest probability to be within the nearest cell(s) of the query point. This interpretation inherited from the nature of the underlying prediction function which locates the anticipated location of an object in a cell size area rather than predicting its exact future point. The *UpdateResult* function is exactly implemented as in the range query without difference. The results for both queries are lists of objects.

Example. If we consider a predictive K-NN query with K = 5, t = 20, and location point = L which is represented by a star in Figure 6(a), *Panda** will start by locating L in C_{17}, Figure 6(b). Since C_{17} has no precomputed answer for t = 20, as shown in the query list in Figure 6(b), then *Panda** has to prepare the answer from the ground. Accordingly, we find out the set of cells reachable to C_{17} in 20 minutes. By examining the travel time grid, we get $\{C_5, C_{35}\}$ are within [18,22] minutes, and $\{C_9, C_{14}, C_{26}, C_{35}\}$ are within [15,25] minutes, based on the current status of the space. Other cells have travel time either completely smaller than 20 minutes or larger than 20 minutes. For example, cells with travel time [25,35] to C_{17} will not contribute in the query answer as they do not intersect with the query time, 20 minutes. This means those five cells might contribute to the predicted results at C_{17}, Figure 6(c). It is easily noticeable that C_5 and C_{26} do not have objects moving inside their boundaries, thus, no need to call the prediction function here. For each object in the other three cells, $\{C_9, C_{14}, C_{35}\}$, we need to predict their possible destinations. For C_9, Figures 6(d)(e)(f) give the predicted destinations for objects O_5, O_{18}, and O_{19}, respectively. This leads to, objects O_5 and O_{18} are likely to show up at the query cell C_{17}, but O_{19} will not as C_{17} is not among its future destinations. By performing the prediction for objects in C_{14} and C_{35}, we infer that both will contribute by objects O_4 and O_{12}, Figures 6(g)(h), respectively. At this moment, we are done with the five reachable cells to the query initial region. But what in-hand as a predicted result is just four objects, $\{O_5, O_{18}, O_4, O_{12}\}$ along with their probabilities, which is less than the desired K. So, this is not sufficient as a final query result. Therefore, we need expand the initial query region by adding one of its neighbor cells. To do so, we measure the distance from the query location L, (the star symbol), to the center of the eight vicinity cells, Figure 6(i). This results in adding C_{11}, the nearest cell, to the query region, Figure 6(j). Nothing is precomputed for predictive queries in this C_{11}, yet, another round of prediction is required. The set of reachable cells to C_{11} includes $\{C_3, C_{30}\}$, with travel time [20,22] and [15,21] minutes, respectively, Figure 6(j). As the later has no objects at the present time, we predict the future destinations of the former cell's objects. Fortunately, O_2 is likely reach the new query cell C_{11} in 20 minutes, Figure 6(k). Finally, we have five objects predicted to show up around the query location L in 20 minutes from the present time. That fulfills the query requirements and the returned answer will be $\{O_5, O_{18}, O_4, O_{12}, O_2\}$. In the case that we find more than K objects, we sort them based on their computed probabilities and pick up the highest K ones.

5.1.3 Aggregate Query Processing

Idea. A predictive aggregate query consists of a query region R and a future time t, and it finds out the number of objects $\mathcal{N}$ predicted to be inside that region after the



(a) Candidate destinations of O_{11}

(b) Query Results

Fig. 7 Query processing on (Stationary vs Moving) Data

given time. Clearly, it looks similar to the range query, hence, it follows the same exact steps. However the format of the final result is different where the aggregate query wants the number of objects $\mathcal{N}$ rather than the objects themselves. Therefore the *UpdateResult* function is customized to sum up the number of objects instead of appending objects to the result list. So it is employed to compute the the expected number objects to be inside each single cell in R after time t, and also aggregate those numbers to obtain the final query result.

Example. Considering Q_{30} as an aggregate query, *Panda** will apply the same steps in Figures 4 and Figures 5, and the final result will give the number of objects expected to arrive at the query region after 30 minutes which will be $\{3\}$.

5.2 Data Nature

The shared data structure, the isolation between the precomputed areas and the query region, and the generic query processor frame the infrastructure that allows *Panda** to support a wide variety of predictive queries including range queries, k-NN queries , and aggregate queries. Apparently, each query type can operate the same data structure in a different way to compute its own answer. As described in the mentioned algorithms, to handle the case of predictive range or K-NN queries, the precomputed answer in the *query list (QL)* maintains a list of objects expected to be inside a query region R after future time t, while in the case of predictive aggregate queries, the same list is used to carry a number.

In this section, we explain the ability of *Panda** to act suitably according to the nature the underlying data whether it represents stationary or moving objects. We provide some examples to illustrate this flexibility feature.

5.2.1 Stationary Data

In this class of data, the points of interest that a user query questions about are static objects that almost do not have mobility nature. Examples for stationary data include gas stations, restaurants, theaters, cinemas,...etc. We refer to a stationary object i by S_i. Basically, *Panda** can preserve information about the underlying stationary objects in addition to the moving objects using its grid data structure where each object is linked to its corresponding cell. The predictive query in this case is tight to a certain moving object not to any of the stationary objects. For example, as illustrated in Figure 7, a moving object O_{11} sends a range query to find out the gas stations within half mile of its future location after 40 minutes, of course without releasing its intension. To process this query, *Panda** initially finds the current cell, C_{28}, in which O_{11} is currently moving. Thus, C_{28} is added to the current trajectory of O_{11}. Then it obtains the list of its candidate destination cells after 40 minutes which will include $\{C_2, C_6, C_{13}\}$. This is achieved by accessing the travel time structure under the column of C_{28}. After that, it determines the highly anticipated destination in a granularity equivalent to a grid cell area. That is accomplished by calling our prediction function feeded with the current trajectory of O_{11}, Figure 7(a). Since C_2 is the predicted destination, a half mile query region is placed starting from its center. Therefore, the set of the stationary objects that intersect with that region is returned as the query results which will be $\{S_4, S_5\}$, Figure 7(b).

5.2.2 Moving Data

This class of data has both the spatial and temporal features, so they dynamically change their locations. Predictive query on this class of data has two options. The first option, is to be connected to a specified moving object which imposes *Panda** to do a preprocessing step by calling our prediction function $\hat{F}$ to identify the possible destination cell c_d for that object at the given time t. Accordingly, the query is located at that destination cell c_d. Then it ordains a customized version of the predictive query processor according to the query type as explained in the previous section. For example a moving object O_{11} asks about the moving objects, (i.e., friends with mobile phones, some moving service like police cars), expected to show up within half mile of its future location after 40 minutes. To process this query, we find that C_2 is the most predicted target cell after 40 minutes based on the recent trajectory of O_{11}. After that, we dispatch *Panda** query processor to evaluate a range query positioned at the center of C_2. The expected answer will be $\{O_3, O_7\}$, Figure 7(b).

The second option for predictive query on moving data, is to be connected a static area or location in the given space, for example a store wants to notify the cars expected to be within two miles of its location about offers after 3o minutes. In this situation, the query processor is fired directly without any additional preparation steps. The example illustrated in Section 4.2 fits in this class of data.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency and the scalability of our proposed system *Panda** for processing predictive queries. We compare *Panda** with two other baseline algorithms namely *Precomputed* and *Instant*, introduced for performance comparison.

In the first baseline algorithm, *Precomputed*, all possible queries results are precomputed before they are issued in a behavior likes a kind of brute force algorithm. Once a snapshot query is received, the answer is read and returned to the user without any further computation. It is obvious here that this baseline algorithm will be having the faster response time, since no computation happens after receiving a query. Just the precomputed result is accessed and returned directly as a final query answer.

The second baseline algorithm, *Instant*, stands at the opposite side of the previous one in terms of precompuation for the query answer. In the *Instant* algorithm, there is no precompuation at all for any query answer. Instead, all results are instantly computed from scratch once the query arrives for processing. Clearly, queries evaluated using this algorithm will wait the longest time until getting their results ready. However, it will save the computation overhead required by the *Precomputed* algorithm for keeping all queries answers up-to-date.

In all experiments, the evaluation and comparison are in terms of, (a) average response time per query, which means the average CPU time it takes to return the answer to the issued query since the query is received by the underlying algorithm, (b) average updating cost, which is the average system overhead measured by the CPU time consumed for updating the precomputed results according to the objects movements between different cells, and (c) total processing time, which is equivalent to the sum of the CPU time consumed for preparing the precomputed parts of a query beforehand and the CPU time to complete and the rest of computation after the query arrival.

Since *Panda** and the other two baseline algorithms employ the same prediction function to compute the probability of an object being in a certain query region after some time duration, we will not compare the accuracy among them. Worth to remind here that we discussed the overall idea of our adjusted prediction function $\hat{F}$ with respect to its underlying base prediction function F in section 3. The accuracy of the employed prediction function is examined and reported here in section 6.5.

6.1 Experiment Setup

In our performance evaluation experiments, we use two data sets.

Synthetic Data. We use the Network-based Generator of Moving Objects [5] to generate large sets of synthetic data of moving objects. A real road network map is used for our experiment setting and for the generator as an input. The road map is extracted from the shape files of Hennepin County in Minnesota, USA. Then, the shape file are converted to network files as required by the moving objects generator. The output of the generator contains different sets of moving objects that move on the given road

network map. The generated objects are assumed to be uniformly distributed over the spatial space.

Real Data. This is also a real data containing the GPS trajectories for more than 10,000 taxis within Beijing [35, 36]. In this data, each taxi has an identifier taxi id, and its movements that are defined by three fields namely, date time, longitude, and latitude.

Both real data sets require some data preprocessing to remove the outliers caused by GPS reading errors, and to partition these readings into realistic objects trajectories. The space in which objects move is virtually partitioned into $N \times N$ squared grid cells of width relative to the minimum and the maximum step taken by any of the underlying moving objects. Our *space grid* data structure mirrors the space partitions by storing an identifier for each cell C_i and updatable list of objects moving within that cell C_i. To have the travel time data structure TTS filled before starting the experiment, the travel time between any pair of cells, C_i and C_j, is obtained by taking the average minimum and the average maximum time it takes from the underlying set of objects to move from C_i to C_j at time slot TS_k.

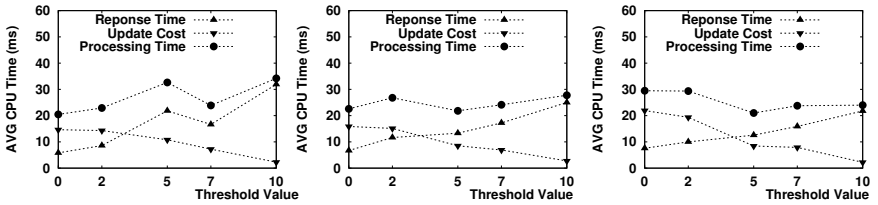
In fact, it might be more practical to store the travel time between various cells as a range of time rather than an exact value. Therefore, a user can issue predictive queries on objects anticipated to be inside a query region after a future time specified as an interval. At the query processing, to deal with this kind of time uncertainty, we use the weighted probability that varies according to the size of the intersection between the travel time interval from a cell to the query region and the query future time interval. For example, if a query asks about predicted number of moving objects in R_q after [10,20] time unites as the future time interval, and if the cell C_i is far from R_q by a travel time interval [19, 28], then we weigh the participation of C_i by 20 percent of the number of objects predicted to come from it to R_q.

To have the algorithms tested against different workload rather than single queries, a query workload generator is built to obtain workloads of predictive queries that vary in the number of queries, the query region size, and the query future time. The number of queries in the generated workloads starts at 1K queries per batch, and increases by 10K until reaches 100K queries in a batch file. The generated queries regions are squares and their locations are uniformly distributed over the space. The size of the generated queries vary from 0.01 to 0.06 of the total space size.

All experiments are based on an actual implementation of *Panda** and the two baseline algorithms, *Instant* and *Precomputed*. All the behaviors of the generated objects, query workload generator, and query processing algorithms are implemented on a Core(TM) i3 4GB RAM PC running Windows 7 with C++.

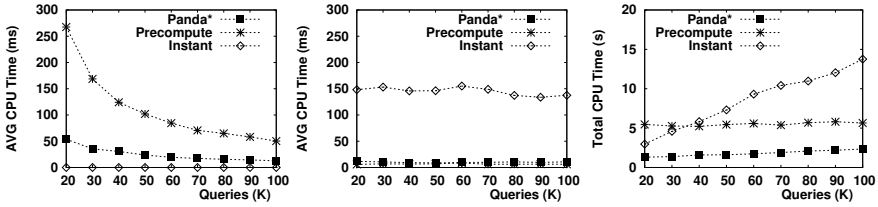
As the $Panda^*$ system deals with moving objects, the objects movements are supposed to be streamed to the system directly. So, data is handled in the memory as it comes. Therefore, we focus on the performance from the I/O perspective. For the purpose of the experiments, normally, the data is stored on files. However, we upload all in the memory at the warming up phase of the experiments. Then, we start measuring our performance parameters after we make sure everything is in memory.

In the following sections, we study the effect of threshold value on the performance of *Panda**. Then we compare the efficiency of *Panda** *to the other two approaches and provide their evaluation with different query workloads. After that, we*



(a) 5K (b) 10K (c) 20K

Fig. 8 Effect of Threshold Tuning



(a) Update Cost (b) Response Time (c) Processing Time

Fig. 9 Efficiency of *Panda** vs Precompute, Instant

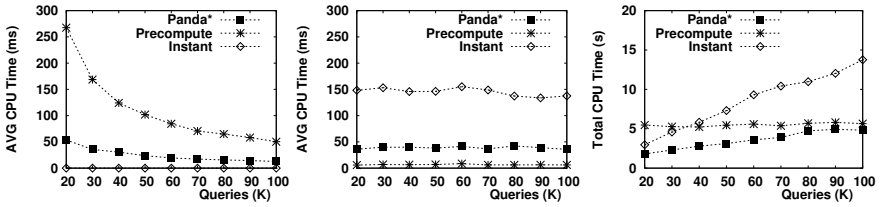
explain how Panda* *can scale up with large number of objects and with outsized queries.*

6.2 Impact of Threshold Tuning

In the first set of experiments, we study the impact of different threshold $\mathcal{T}$ values on the performance of Panda* *with different data sets. The minimum value that the $\mathcal{T}$ can take in this experiment is 0 which means any query on cell C_i with time t that appears at least one time on C_i will be precomputed in advance, and the maximum is set to 10 which means any query with reappearance rate less than 10 times will not be precomputed at all. The maximum threshold value is decided based on the number of queries in a workload, and the number of different sizes and the number of different future times for the generated queries. $\mathcal{T}_{max} \geq N_{Queries} / (N_{DistinctSizes} * N_{DistinctTimes})$, while $\mathcal{T}_{min}$ is always zero.*

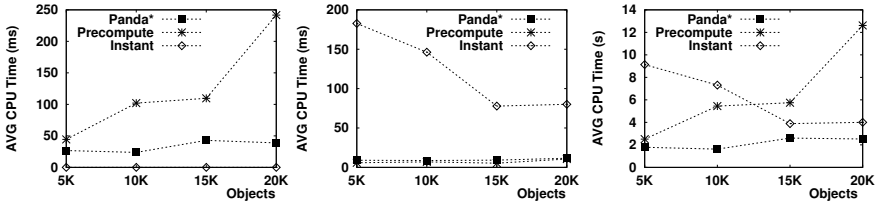
As given in Figure 8, in all data sets, Panda* *gives its best response time when $\mathcal{T}$ = 0 and the lowest update cost when $\mathcal{T}$ = 10, while it gives its worst response at $\mathcal{T}$ = 10 and highest update cost at $\mathcal{T}$ = 0. Between the minimum $\mathcal{T}$ and the maximum $\mathcal{T}$, the threshold value can be tuned to provide the required balance between the time a user has to wait to receive a query result and the overhead cost used to prepare this answer in advance.*

The effect of the threshold value on the overall performance of Panda* *varies according to the underlying data set. For example, in the first data set, Figure 8(a), the lowest processing time required to evaluate a query is achieved when $\mathcal{T}$ = 0 and the highest is at $\mathcal{T}$ = 10, and the trend of the curve is to increase when the threshold values increases. The matter is different in the second and the third data sets, where the lowest processing cost is at $\mathcal{T}$ = 5 and the highest is at $\mathcal{T}$ = 10 in the second data*



(a) Update Cost (b) Response Time (c) Processing Time

Fig. 10 Efficiency of *Panda** at $\mathcal{T} = 2$



(a) Update Cost (b) Response Time (c) Processing Time

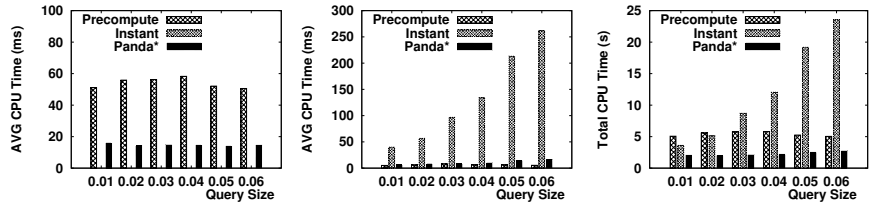
Fig. 11 Scalability with Number of Objects

set, Figure 8(b), while at $\mathcal{T} = 5$ and $\mathcal{T} = 0$ Panda* *achieves its lowest and highest processing time respectively in the third data set with a decreasing trend of curve, Figure 8(c). To sum up, it is noticeable that the effect of the threshold depends on the behavior of the moving objects in the underlying data set. So, when the overall objects movements can trigger the* answer maintenance *module frequently, the update cost increases. In this case, large threshold values help* Panda* *to achieve better performance.*

6.3 Efficiency Evaluation

To evaluate the efficiency of Panda**, we processed workloads of predictive queries with varying the number of received queries from 20K to 100K. Figures 9 provides a comparison between* Panda* *and the other two algorithms $w.r.t$ number of queries in terms of update cost, Figure 9(a), response time, Figure 9(b), and processing time per query, Figure 9(c), with the average CPU time as a measure.* Panda* *and* Precompute *provide the best possible response time which almost equals to zero waiting, while the* Instant *gives the worst response time with a significant difference, Figure 9(b).*

As explained earlier and as depicted in Figure 9(a), the Instant *requires no update cost as it does not prepare any answers in advance, while the* Precompute *does, thus, it costs the most update time. On the* Panda* *side, it consumes some of the CPU time for precomputation during the early phases of the experiment then decreases to a low update cost when the number of queries equals 100K. In Figure 9, the efficiency of* Panda* *is recorded when the threshold is set to 0, however, it still obtains the best overall performance when $\mathcal{T}$ equals 2, Figure 10. In a nutshell, those figures provide*



(a) Update Cost (b) Response Time (c) Processing Time

Fig. 12 Scalability with Query Size

that Panda* *achieves a dramatic high performance which is up to four orders better than the* Precompute *algorithm and even much better when compared to the* Instant *algorithm.*

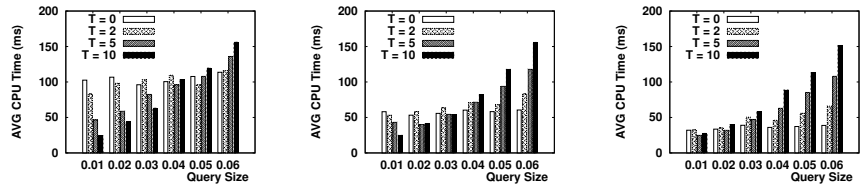
6.4 Scalability Evaluation

We proceed to study the scalability of Panda* *with large number of moving objects and large query sizes. In the first set of experiments, Figure 11, we evaluate the scalability of the* Panda* *query processing performance when the number of moving objects increases from 1k to 25k. With respect to the query processing time, Figure 11(c),* Panda* *performs the best, regardless the increasing in the number of objects. In addition to its high performance in terms of the average query processing time,* Panda* *also gives a response time as low as the one in the* Precompute *algorithm which guaranteed to give the lowest query response, Figure 11(b), while saving a lot of the required update cost consumed by the* Precompute *algorithm, Figure 11(a). The next set of experiments, Figure 12, illustrates that* Panda* *can save about 50% of the CPU time required to answer a user query while preserving its response to be almost equals to the fastest one with a vast dropping in the update cost. The last set of experiments, Figure 13, evaluates the influence of the threshold tweaking on the scalability of* Panda* *with respect to the query size in three different query workloads. The used workloads are 20K queries, 40K queries, and 80K queries. The provided figures suggest the use of a large threshold value with small sized queries and a small threshold value with the ones with larger query size.*

In one word, we can establish that the main reason behind the capability of Panda* *to achieve smooth scalability comes from its ability to adapt according to the nature of the moving objects behavior and the heaviness of the query workload.*

6.5 Accuracy Test

To examine the quality of the underline prediction function, we employed a another real data set of GPS trajectories collected by Microsoft researchers around the area of Seattle, Washington, USA [2]. The area is divided in 1 KM of squares to form our grid area. Then, we compute the accuracy based on the probability given by the



(a) 20 Queries (b) 40 Queries (c) 80 Queries

Fig. 13 Effect of Threshold Tuning on Scalability w.r.t Query Size in Different Workloads

prediction function for the prediction of the next cell. As shown in Figure 14, we vary the percentage of trip completion on x axis, and we measure the accuracy of the prediction on y-axis. Rationally, the quality of the prediction improves while the object moves forward. It is obviously provided that we can achieve between 70% to 90% for next destination prediction.

7 CONCLUSION

This paper introduces Panda*; *a system for evaluating predictive spatio-temporal queries.* Panda* *enables users to request a location-based service using the predicted locations of moving objects in a future time instance.* Panda* *has three main modules that are tasked with* query processing*,* system performance tuning*, and* query answer maintenance*. Each module is triggered by a firing event; a* query arrival*, a* trigger for system tuning*, and an* object movement*, respectively. The* query processing *component is responsible for computing the query results or accessing the precomputed ones from previous query processing cycles. The task of the* answer maintenance *component is to update the materialized precomputed answers according to the effect of an object movement. The third component,* system tuning*, is dispatched to periodically adapt the performance of* Panda* *according to the nature of the given workload.*

Panda* *supports a variety of predictive queries including predictive range queries, predictive aggregate queries, and predictive k-NN queries, on both stationary and moving objects.* Panda* *also supports dealing with time uncertainty by modeling the travel time between different locations in the space as intervals.* Panda* *introduces the* Travel Time Structure *(TTS), a time varying multidimensional grid, coupled with a long term prediction function to achieve its prediction goals. Extensive experimental evaluation using large groups of real and synthetic data proves the efficiency and scalability of* Panda*. Panda* *has reduced the processing time by up to* 42% *compared to its precomputed baseline algorithm and by up to* 17% *compared to the instant baseline algorithm (at 100K queries).*

While this paper addressed time uncertainty*,* location uncertainty *is another interesting dimension of uncertainty. Location uncertainty comes up due to the noisy acquisition of a GPS reading and/or due to a privacy preserving layer that dilutes the user's location from a point into a region. We expect future research directions to expand* Panda* *along the location uncertainty direction.*

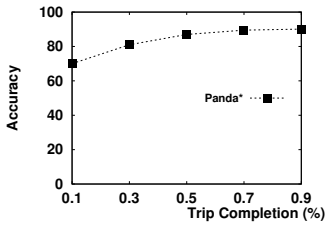


Fig. 14 Accuracy of Destination Prediction

References

1. *M. Ali and A. M. Hendawi. Spatial Predictive Queries. In* MDM*, Pennsylvania, USA, June 2015.*
2. *M. Ali, J. Krumm, and A. Teredesai. ACM SIGSPATIAL GIS Cup 2012. In* ACM SIGSPATIAL GIS*, pages 597–600, California, USA, Nov. 2012.*
3. *R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and Reverse Nearest Neighbor Queries for Moving Objects.* VLDB Journal*, 15(3):229–249, 2006.*
4. *A. Brillingaite and C. S. Jensen. Online Route Prediction for Automotive Applications. In* ITS*, London, United Kingdom, Oct. 2006.*
5. *T. Brinkhoff. A Framework for Generating Network-Based Moving Objects.* GeoInformatica*, 6(2):153–180, 2002.*
6. *H. D. Chon, D. Agrawal, and A. E. Abbadi. Range and kNN Query Processing for Moving Objects in Grid Model.* MONET*, 8(4):401–412, 2003.*
7. *J. Froehlich and J. Krumm. Route Prediction from Trip Observations. In* Society of Automotive Engineers (SAE) World Congress*, Michigan, USA, Apr. 2008.*
8. *Y. Gu, G. Yu, N. Guo, and Y. Chen. Probabilistic Moving Range Query over RFID Spatio-temporal Data Streams. In* CIKM*, pages 1413–1416, Hong Kong, China, Nov. 2009.*
9. *A. Hendawi.* Scalable Spatial Predictive Query Processing for Moving Objects*. PhD thesis, University of Minnesota, Twin-Cities, 2015.*
10. *A. M. Hendawi. Predictive query processing on moving objects. In* In proceedings of the Data Engineering Workshops (ICDEW)*, Illinoi, USA, Apr. 2014.*
11. *A. M. Hendawi, M. Ali, and M. F. Mokbel. A Framework for Spatial Predictive Query Processing and Visualization. In* MDM*, pages 327–330, Pennsylvania, USA, June 2015.*
12. *A. M. Hendawi and M. F. Mokbel. Panda: A Predictive Spatio-Temporal Query Processor. In* ACM SIGSPATIAL GIS*, California, USA, Nov. 2012.*
13. *H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In* SIGMOD*, pages 479–490, Maryland, USA, June 2005.*
14. *H. Jeung, Q. Liu, H. T. Shen, and X. Zhou. A Hybrid Prediction Model for Moving Objects. In* ICDE*, pages 70–79, Cancn, Mxico, Apr. 2008.*
15. *H. Jeung, M. L. Yiu, X. Zhou, and C. S. Jensen. Path Prediction and Predictive Range Querying in Road Network Databases.* VLDB Journal*, 19(4):585–602, Aug. 2010.*
16. *Z. Jinghua, W. Xue, and L. Yingshu. Predictive Nearest Neighbor Queries over Uncertain Spatial-Temporal Data. In* WASA*, pages 424–4359, Harbin, China, June 2014.*
17. *J. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors. In* ICDE*, pages 806–815, Istanbul, Turkey, Apr. 2007.*
18. *H. A. Karimi and X. Liu. A Predictive Location Model for Location-Based Services. In* GIS*, pages 126–133, Louisiana, USA, Nov. 2003.*
19. *S.-W. Kim, J.-I. Won, J.-D. Kim, M. Shin, J. Lee, and H. Kim. Path Prediction of Moving Objects on Road Networks Through Analyzing Past Trajectories. In* KES*, pages 379–389, Vietri sul Mare, Italy, Sept. 2007.*
20. *J. Krumm. Real Time Destination Prediction Based on Efficient Routes. In* SAE*, Michigan, USA, Apr. 2006.*
21. *K. C. K. Lee, H. V. Leong, J. Zhou, and A. Si. An Efficient Algorithm for Predictive Continuous Nearest Neighbor Query Processing and Result Maintenance. In* MDM*, pages 178–182, Ayia Napa, Cyprus, May 2005.*

22. *Y. Li, S. George, C. Apfelbeck, A. M. Hendawi, D. Hazel, A. Teredesai, and M. Ali. Routing Service With Real World Severe Weather. In* ACM SIGSPATIAL GIS*, pages 585–588, Texas, USA, Nov. 2014.*
23. *M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In* SIGMOD*, pages 443–454, Paris, France, June 2004.*
24. *M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous Query Processing of Spatio-temporal Data Streams in PLACE. In* STDBM*, pages 57–64, Toronto, Canada, Aug. 2004.*
25. *K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast Nearest-Neighbor Query Processing in Moving-Object Databases.* GeoInformatica*, 7(2):113–137, June 2003.*
26. *C. Shahabi, L.-A. Tang, and S. Xing. Indexing Land Surface for Efficient kNN Query. In* VLDB*, pages 1020–1031, Aucklan, New Zealand, Aug. 2008.*
27. *A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In* ICDE*, pages 422–432, Birmingham U.K, Apr. 1997.*
28. *J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal. In* ICDE*, pages 202–213, MASSACHUSETTS, USA, Mar. 2004.*
29. *Y. Tao, C. Faloutsos, D. Papadias, and B. L. 0002. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In* SIGMOD*, pages 611–622, Paris, France, June 2004.*
30. *Y. Tao and D. Papadias. Time-parameterized Queries in Spatio-temporal Databases. In* SIGMOD*, pages 334–345, Wisconsin, USA, June 2002.*
31. *Y. Tao and D. Papadias. Spatial queries in dynamic environments.* TODS*, 28(2):101–139, 2003.*
32. *Y. Tao, J. Sun, and D. Papadias. Analysis of predictive spatio-temporal queries.* TODS*, 28(4):295–336, Dec. 2003.*
33. *H. Wang, R. Zimmermann, and W.-S. Ku. Distributed Continuous Range Query Processing on Moving Objects. In* DEXA*, pages 655–665, Krakow, Poland, Sept. 2006.*
34. *M. L. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: Indexing Moving Objects by Space Filling Curves in the Dual Space.* VLDB Journal*, 17(3):379–400, May 2008.*
35. *J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In* KDD*, pages 316–324, California, USA, Aug. 2011.*
36. *J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In* GIS*, pages 99–108, California, USA, Nov. 2010.*
37. *M. Zhang, S. Chen, C. S. Jensen, B. C. Ooi, and Z. Zhang. Effectively Indexing Uncertain Moving Objects for Predictive Queries.* PVLDB*, 2(1):1198–1209, 2009.*
38. *R. Zhang, H. V. Jagadish, B. T. Dai, and K. Ramamohanarao. Optimized Algorithms for Predictive Range and KNN Queries on Moving Objects.* Information Systems*, 35(8):911–932, Dec. 2010.*