Panda: A Predictive Spatio-Temporal Query Processor *

Abdeltawab M. Hendawi Mohamed F. Mokbel

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
{hendawi, mokbel}@cs.umn.edu

ABSTRACT

This paper presents the *Panda* system for efficient support of a wide variety of *predictive* spatio-temporal queries that are widely used in several applications including traffic management, location-based advertising, and ride sharing. Unlike previous attempts in supporting predictive queries, *Panda* targets long-term query prediction as it relies on adapting a well-designed long-term prediction function to: (a) scale up to large number of moving objects, and (b) support large number of predictive queries. As a means of scalability, *Panda* smartly precomputes parts of the most frequent incoming predictive queries, which significantly reduces the query response time. *Panda* employs a tunable threshold that achieves a trade-off between query response time and the maintenance cost of precomptued answers. Experimental results, based on large data sets, show that *Panda* is scalable, efficient, and as accurate as its underlying prediction function.

1. INTRODUCTION

The emergence of wireless communication networks and cell phone technologies with embedded global positioning systems (GPS) have resulted in a wide deployment of location-based services [8, 15]. Common examples of such services include range queries [7, 25], e.g., "find all gas stations within three miles of my *current* location" and K-nearest-neighbor (kNN) queries, e.g., "find the two nearest restaurants to my *current* location". However, such common examples focus on the *current* locations of moving objects. Another valuable set of location-based services focuses on *predictive* queries [9, 10], in which the same previous queries are asked, yet, for a *future* time instance, e.g., "find all gas stations that will be within three miles of my *future* location after 30 minutes". *Predictive* queries are extremely beneficial in a wide variety of applications that include traffic management, e.g., predict congested areas before it takes place, location-based advertising, e.g., predict the customers who are expected to be nearby in the next hour, and ride sharing, e.g., find those riders who are likely to share their route with me.

In this paper, we present the *Panda* system, designed to provide efficient support for *predictive* spatio-temporal queries. *Panda* provides the necessary infrastructure to support a wide variety of *predictive* queries that include predictive spatio-temporal range, aggregate, and k-nearest-neighbor queries as well as *continuous* queries. *Panda* distinguishes itself from all previous attempts for processing predictive queries [10, 27] in the following: (1) *Panda* targets *long-term* predication in the order of tens of minutes, while existing attempts mainly target short-term prediction in terms of only minutes and seconds, (2) *Panda* smartly precomputes parts of the frequent incoming queries, which significantly reduces the query response time, and (3) *Panda* is generic in the sense that it does not only address a certain type of predictive queries, as done by previous work, instead, it provides a generic infrastructure for a wide variety of predictive queries.

The main idea of *Panda* is to monitor those space areas that are highly accessed using predictive queries. For such areas, *Panda* precomputes the prediction of objects being in these areas beforehand. Whenever a predictive query is received by *Panda*, it checks if parts of this predictive query are included in those precomputed space areas. If this is the case, *Panda* retrieves parts of its 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 prediction module to find out the answer, which will take more time to compute. It is important to note here that *Panda* does not aim to predict the whole query answer, instead, *Panda* 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. 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 kNN) 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.

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

*The work of this paper is supported in part by the National Science Foundation under Grants IIS-0811998, IIS-0811935, CNS-0708604, IIS-0952977, and the Egyptian Ministry of Higher Education under Grant GM-887.

ACM SIGSPATIAL GIS'12 November 6-9, 2012. Redondo Beach, CA, USA

predictive query will suffer the most due to the need of computing the query answer from scratch without any precomputations. The underlying prediction function deployed by *Panda* mainly relies on a *long-term* prediction function, designed by John Krumm [6, 13] to predict the final destination of a single user based on his/her current trajectory. Unfortunately, a direct deployment of such *long-term* prediction function does not scale up for large numbers of moving objects nor it serves our purpose for predictive queries that are concerned with the moving object location in a future time rather than its final destination. *Panda* adapts such well-designed prediction function to: (a) scale up with the large number of users through a specially designed data structure shared among all moving objects, and (b) provide the prediction for a future query time (e.g., after 30 minutes) rather than only the prediction for the final destination.

The rest of this paper is organized as follows. Section 2 highlights related work. Section 3 gives an overview of the *Panda* system. The generic framework for *Panda*'s query processor is given in Section 4, while its extensibility to a wide variety of predictive queries is presented in Section 5. Section 6 provides experimental evaluation of *Panda*. Finally, Section 7 concludes the paper.

2. RELATED WORK

In terms of supported predictive queries, existing algorithms for predictive query processing have focused only on one kind of predictive queries, e.g., *range queries* [10, 20, 27], *k-nearest-neighbor queries* [1, 16, 27], *reverse-nearest-neighbor queries* [1], *continuous-nearest-neighbor queries* [14], *aggregate queries* [19], or *predictive join* with estimating the expected query selectivity [4, 5, 24, 23]. Some of this work attaches the expiry time interval to a kNN query result [21, 22]. 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.

In terms of the underlying prediction function, existing algorithms for predictive query processing can be classified into three categories:

(1) *Linearity-based prediction*, where the underlying prediction function is based on a simple assumption that objects move in a linear function in time along the input velocity and direction. So, query processing techniques in this category, e.g., [1, 16, 18, 22, 23], take into consideration the position of a moving point at a certain time reference, its direction, and the velocity to compute and store the future positions of that object in a TPR-tree-based index [17]. When a predictive query is received, the query processor retrieves the anticipated position in the given time [18]. The work in this category is concerned with the applications of the linearity-based prediction models to answer nearest neighbor queries [16] and reverse nearest neighbor queries [1], and to estimate the query selectivity [23]. Some of these applications attach the expiry time interval to the kNN query result [22].

(2) *Historical-based prediction*, where the predication function uses object historical trajectories to predict the object next trajectory. Then, query processing techniques in this category, e.g., [2, 6, 10, 11, 12, 19] are applied to trajectory of location points. Existing work in this category is based on either mobility model [10], or ordered historical routes [2, 6, 12]. The mobility model [10] is used to capture the different possible turning pattern at different roads junctions, and the travel speed for each segment in the road network for each single object in the system. Then, the model is used to predict the future trajectory of each object, and based on that they can answer predictive range queries. The main concern of that model is to put more focus on the prediction of the object behavior in junctions based on historical data of objects trajectories. In the ordered historical routes, the stored past trajectories 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 [2, 6, 12]. Some of the existing work in this category is employed for predicting the current object trajectory in non-euclidian space [11] such as road-level granularity. For example, a Predictive Location Model (PLM) [11] is proposed to predict locations in location-based services. The model considers the start point as the object current location while the end point could be any of the possible exit points. PLM computes the shortest path trajectory between the current location and each of the exit points, then the trajectory with the highest probability is considered the predicted path.

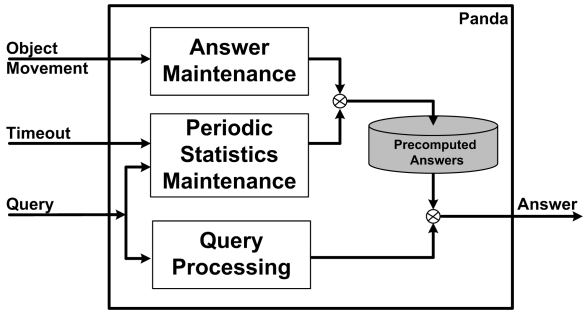


Figure 1: The *Panda* System Architecture

(3) *Other prediction functions*, where more complicated prediction functions are employed to realize better prediction accuracy. Query processing techniques in this category, e.g., [9, 20, 26, 27], are adjusted based on the outcome of the prediction function. Existing work in this category either exploits a single function [20, 27], or mixes between two or more functions to form a hybrid prediction model [9, 26]. As an example for a single function, a Transformed Minkowski Sum [27] is used to answer circular region range and K-NN queries, while Recursive Motion Function (RMF) [20] 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 [9, 26] are combined to evaluate predictive range and nearest neighbor queries in highly dynamic and uncertain environments. Unfortunately, all the employed prediction functions can support only short-term prediction in terms of seconds or minutes.

Panda distinguishes itself from all the above work in the following: (1) *Panda* is a generic framework that is not only applicable to one kind of predictive queries. Instead, a wide variety of predictive queries, e.g., range and nearest-neighbor queries can be supported within one framework, (2) *Panda* relies on a long-term prediction function applicable to tens of minutes, and (3) *Panda* is a scalable framework that supports large number of predictive snapshot and continuous queries.

3. SYSTEM OVERVIEW

This section gives an overview of the *Panda* system outlining the system architecture and underlying prediction function.

3.1 System Architecture

Figure 1 gives the system architecture of the *Panda* system, which includes three main modules, namely, *query processing*, *periodic statistics maintenance*, and *answer maintenance*. Each module is dispatched by an event, namely, *query arrival*, *periodic statistics maintenance trigger*, and *object movement*, 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 brief overview of the actions taken by *Panda* for each event. Details of these actions are discussed in Section 4.

Query arrival. Once a query is received by *Panda*, the query processor divides the query area into two parts. The first part is already precomputed where this part of the 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.

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.

Periodic statistics maintenance trigger. System statistics that decide on which parts of the space to precompute for potential incoming frequent queries need to be updated periodically using the statistics maintenance module. The module basically reset the statistics to ensure the accuracy and recency of collected statistics.

3.2 Prediction Function

The long-term prediction function deployed in *Panda* is mainly an adaptation of the one introduced by John Krumm [6, 13] to predict the final destination of a single object, F= $P(C_i|S_o)$. F is applied to any space that is partitioned into a set of grid cells $\mathcal{C}$. F takes two inputs, namely, a cell $C_i \in \mathcal{C}$ and a sequence of cells S_o = $\{C_1, C_2, \cdots, C_k\}$ that represents the current trip of an object O. Then, F returns the probability that C_i will be the final destination of O.

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 $\hat{F}$ 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.

$$\hat{F} = \frac{P(C_i|S_o)}{\sum_{d \in D_t} P(C_d|S_o)} \quad (1)$$

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. D_t is the set of possible destination cells of object O after time t, computed based on the travel time distance. It is important to mention here that the recomputation of this prediction function is triggered only when an object O changes its location from cell to another rather than from point to another point within the same cell.

Panda also has another adaptation of F to scale it up to support large numbers of moving objects as F is mainly designed to support single object prediction. The scaling up is mainly supported by the underlying data structure, discussed in the next section, which gives an infrastructure to share by large numbers of moving objects.

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* can support stationary as well as moving objects. Finally, *Panda* is easily extensible to support continuous queries. This generic feature of *Panda* makes it more appealing to industry and easier to realize in 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 *periodic statistics maintenance trigger*. Each event prompts *Panda* to call one of its three main modules to take the appropriate response. The section first starts by describing the underlying data structure of *Panda* (Section 4.1). Then, the reaction of *Panda* to the events query arrival, object movement, and periodic statistics maintenance trigger 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 2 depicts the underlying data structure used by *Panda*. A brief overview of each data structure is outlined below:

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 2, O_2 in its current trip, has passed through the sequence of cells $\{C_{13}, C_7, C_2, C_3\}$. This means that O_2 has started at C_{13} and it is currently moving inside C_3.

Space Grid SG. *Panda* partitions the whole space into $N \times N$ equal-size grid cells. For each cell $C_i \in SG$, we maintain four pieces of information as: (1) *CellID* as the cell identifier, (2) *Current Objects* as the list of moving objects currently located inside C_i, presented as pointers to the Object List OL, (3) *Query List* as the list of predictive queries recently issued on C_i. Each query Q in this list is presented by the triple (*Time*, *Counter*, *Answer*), where *Time* is the future time included in Q, *Counter* is the number of times that Q is recently issued, *Answer* is the precomputed answer for Q which may have different format based on the type of Q, and (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.

Travel Time Grid TTG. This is a two-dimensional array of $N^2 \times N^2$ cells where each cell $TTG[i, j]$ has the average travel time between space cells C_i and C_j, where C_i and $C_j \in SG$. TTG is fully pre-loaded into *Panda* and is a read-only data structure.

4.2 Query Processing in Panda

Panda receives a predictive spatio-temporal query, e.g., range or nearest-neighbor query, that asks about the query answer after a future time t. The main idea behind efficiency and scalability in *Panda* is that *Panda* precomputes partial results of the frequent incoming queries beforehand. In general, *Panda* does not aim to precompute the whole query answer, instead, it precomputes 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, *Panda* 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

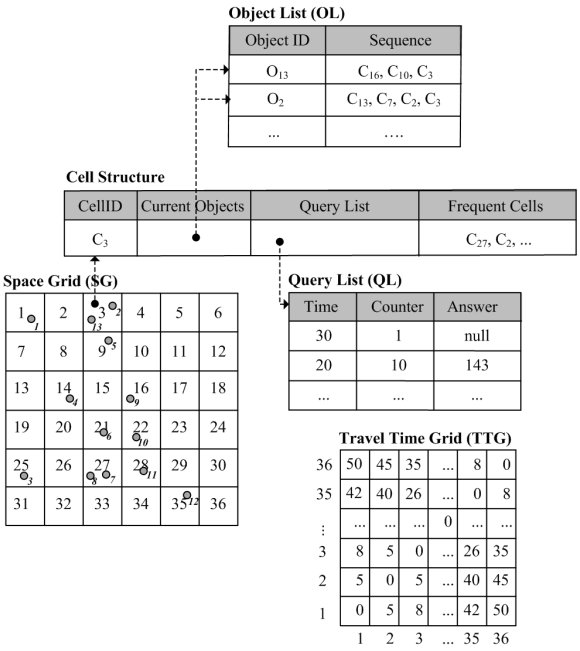


Figure 2: Data Structures in *Panda*

answer of incoming queries, there would be significant redundant computations among overlapped query areas.

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 find 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 receives: (a) a predictive query Q, either as range, aggregate, or k-nearest-neighbor, asking about future time t, and (b) a cell c_i that overlaps with the query area of interest R. The output of Phase I is the partial answer of Q computed from c_i.

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

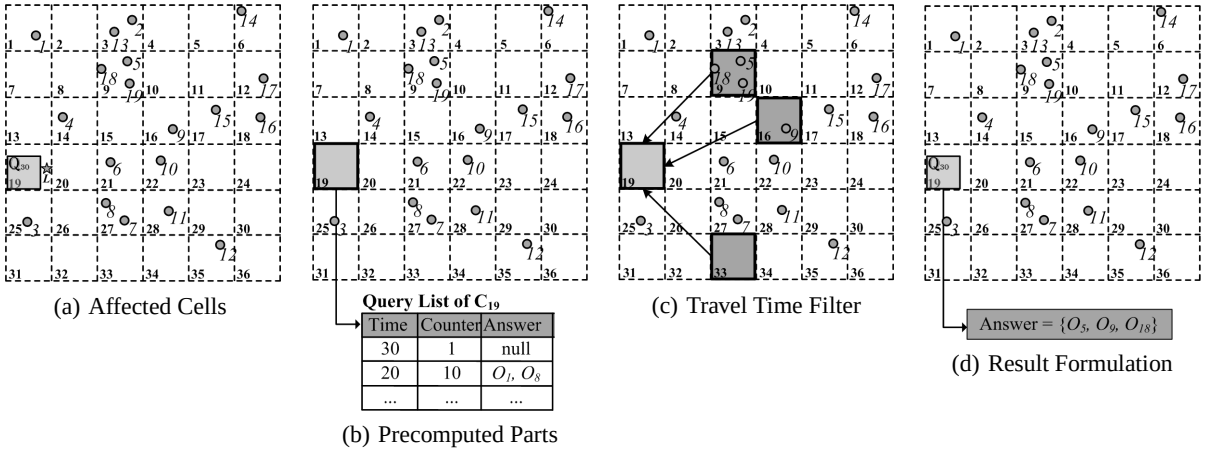
Algorithm 1 *Panda* Predictive Query Processor

Input: Region R, time t, Threshold $\mathcal{T}$

```
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
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 ≥ 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
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
```

that can possibly reach to cell c_i within the future time t. Basically, Phase I utilizes the *Travel Time Grid (TTG)* data structure 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 SG. 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. As will be detailed in Section 5, the operations inside this functions depend on the underlying query type, e.g., aggregate, range, or k-nearest-neighbor queries. 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 are reachable to c_i within the future time t by checking the *Travel Time Grid (TTG)*. Only those objects that lie within any of the cells in C_R may contribute to the final cell answer, and hence the query answer. For each object O in any of the cells of C_R, we utilize our underlying prediction function (Section 3.2) 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 cells of C_R, we again utilize the generic function *UpdateResults* to update the final query result by the result coming from cell c_i



(a) Affected Cells (b) Precomputed Parts (c) Travel Time Filter (d) Result Formulation

Figure 3: Phase I Example.

(Line 16 in Algorithm 1).

Example. Figure 3 gives a running example of Phase I where 19 objects, O_1 to O_{19} are laid on a 6 $\times$ 6 grid structure. Figure 3(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 idea here are applied to aggregate and k-nearest-neighbor queries. First, 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 3(b) 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.

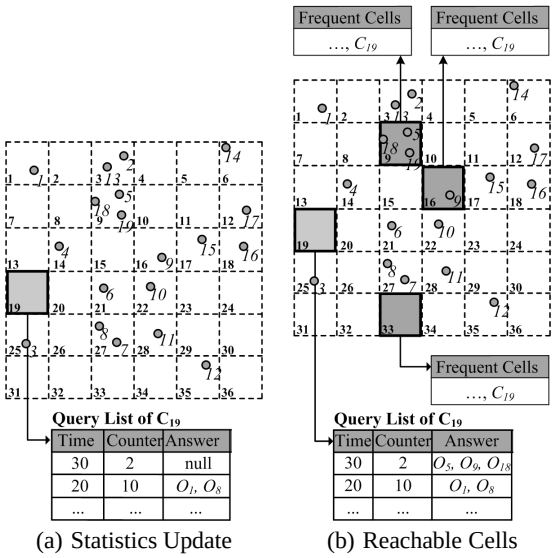
Figure 3(c) starts the process of computing the answer of cell C_{19}. As a first step, we utilize the *Travel Time Grid (TTG)* data structure to find out the set of cells that are reachable to C_{19} within 30 minutes. We find that there are only three cells that can contribute to the answer of C_{19}, 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, which filters out large number of moving objects that. 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 3(d) as $\{O_5, O_9\, O_{18}\}$ along with their probabilities of being in C_{19} in 30 minutes (Probabilities are not shown in the figure as it is an illustrative example).

4.2.2 Phase II: Statistics Maintenance

Phase II does not add anything to the query answer. Instead, it 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 a 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, 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, 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. To efficiently utilize the tunable threshold $\mathcal{T}$, Phase II utilizes the *counter* information in the *Query List* data structure of cell c_i (described in Section 4.1). 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, i.e., stored in the *Query List* data structure. 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. Such list of reachable cells can be directly obtained from the *Travel Time Grid (TTG)* data structure.

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 this query is issued at cell c_i with time t. Then, we check the *counter* against the system threshold T and the value of the current cell *Answer*. This check may result in three different cases 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 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 answer of e.



(a) Statistics Update (b) Reachable Cells

Figure 4: Phase II Example.

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 4 gives a running example of Phase II continuing the computations of Phase I in Figure 3. Figure 4(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 4(b) depicts the actions taken by Phase II upon the consideration that Q_{30} becomes frequent. First, the *Query List* of C_{19} is updated to hold the computed answer from Phase I. Second, the cell C_{19} is added to the list of frequent cells for C_9, C_{16}, and C_{33} to indicate that any movement in these three cells may trigger a change of answer for cell C_{19}.

4.3 Object Movement in Panda

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 is basically triggered with every single object movement in any cell c_i in the space grid SG.

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 does nothing for this object movement. As the underlying prediction function $\hat{F}$ mainly relies on the sequence of prior visited cells for a moving object, an object 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 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: $\mathcal{C} \leftarrow$ The set of *frequent cells* of C_{new}
6: **for each** cell $C_i \in \mathcal{C}$ **do**
7: $t \leftarrow$ travel time from C_{new} to C_i from $TTG[new, i]$
8: ObjectPrediction $\leftarrow$ Compute $\hat{F}$ = $P(C_{new}|O, t)$
9: **UpdateResults (CellResult, ObjectPrediction)**
10: **end for**
11: Remove O from the set of *current objects* of C_{old}
12: $\mathcal{C} \leftarrow$ The set of *frequent cells* of C_{old}
13: **for each** cell $C_i \in \mathcal{C}$ **do**
14: **UpdateResults (CellResult,** O**)**
15: **end for**
16: **Return**

Example. Back to our running example in Figure 4(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}. In this case, we add O_9 to the list of *current objects* in C_{17}, and get its list of *frequent cells*, which only includes C_1. 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 first delete O_9 from the list of *current objects* in C_{16}. Then, we read the list of *frequent cells* of C_{16} which returns only C_{19}. At this point, we do not need to compute the prediction function $\hat{F}$ as it should be already stored in the *query list* of C_{19}. So, we just update the answer at C_{19} by removing O_9 and its probability.

4.4 Periodic Statistics Maintenance

As has been seen earlier, the *query processing* module (Sec-

Algorithm 3 Periodic Statistics Maintenance

Input: System Threshold $\mathcal{T}$

```
1: for each cell C_i ∈ the Space Grid SG do
2:    for each entry e ∈ C_i.QueryList do
3:       if e.Counter ≥ T then
4:          e.Counter ← 0
5:       else
6:          Remove C_i from the list of frequent cells in each of its
            reachable cells within e.Time
7:          Delete e from C_i.QueryList
8:       end if
9:    end for
10: end for
11: Return;
```

tion 4.2) mainly relies on simple maintained statistics, namely, the *counter* in the *Query List* data structure, to decide on which parts of the space to precompute for which future query time. However, the *counter* information just keeps increasing while frequent queries may no longer be frequent any more, yet, as they still keep their *counter* value intact, their answers may still be unnecessarily precomputed, causing extra system overhead. It is the job of the *periodic statistics maintenance* module, discussed in this section, to ensure that current statistics information is accurate and updated.

Main Idea. The main idea behind this module is to run periodically each t units to sweep over current statistics and update it. For a query Q to be considered frequent, it has to appear at least $\mathcal{T}$ times in the last time period t, where $\mathcal{T}$ is the system threshold, described in Section 4.2. In the mean time, a frequent query Q, who failed to appear at least $\mathcal{T}$ times in the last time period t is demoted to be infrequent.

Algorithm. Algorithm 3 gives the pseudo code for the *periodic statistics maintenance* module. The algorithm sweeps over all grid cells in the grid data structure SG. For each cell C_i, the algorithm goes through every single entry e in C_i.*QueryList*. For each entry e, we compare its *counter* against the system threshold $\mathcal{T}$, which will result in one of these two cases: (1) *e.Counter* $\geq \mathcal{T}$. In this case, e represents a frequent query, and thus we just reset its counter to 0 to restart its statistics with the next time period t. (2) *e.Counter* $< \mathcal{T}$, in which e represents a query that failed to appear more than $\mathcal{T}$ times in the last time period t. In this case, we remove e from the list C_i.*QueryList* while doing a clean up by removing the entry for C_i from the list of frequent cells in each of its reachable cells within *e.Time*.

5. EXTENSIBILITY OF PANDA

This section discusses the extensibility of *Panda* framework to support various kinds of predictive spatio-temporal queries along with continuous queries.

5.1 Extensibility in Predictive Queries

Algorithms 1 and 2, that give the *Panda* framework for query processing and the maintenance of precomputed answers, respectively, were described in terms of a generic function, termed *UpdateResults(Result,List)*, called twice in each algorithm. The *UpdateResults* function takes two inputs, a (partial) query result and a list of objects along with their prediction functions, and its objective is to update the given query result by the given list of objects. The specific implementation of the *UpdateResults* generic function depends on the underlying type predictive query. Below is a brief description of the functionality of the *UpdateResults* function for predictive spatio-temporal range, aggregate, and k-nearest-neighbor queries.

Range Queries. 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. In this case, the *UpdateResults(Result,List)* implementation is very simple as it just adds the set of objects in in *List*, along with the values of their prediction functions to the current result. The running example of Figure 3 was described in terms of range queries.

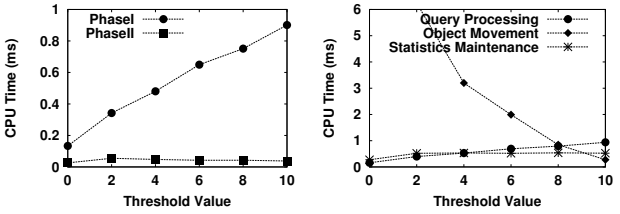
Aggregate Queries. A predictive aggregate query has a query region R and a future time t, and asks about the number of objects $\mathcal{N}$ predicted to be inside R after time t. In this case, the *UpdateResults(Result,List)* implementation just adds the number of objects in *List* to the current value of *Result*. In Figure 3, if Q_{30} is an aggregate query, its answer will be stored as 3.

K-Nearest-Neighbor Queries. 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. To fit the *Panda* framework, the point P is translated to an area R, which basically includes the grid cell that contains P. Then, the *UpdateResults(Result,List)* implementation basically add the objects in *List* to the current *Result*. If *Result* ends up to have more than K objects, then, only the closest k objects to P are kept at *Result*, while all other objects are removed from *Result*. If the final query result ends up to have less than K objects, the query is issued again with larger R that includes the cell that contains P and all its neighbor cells. Similarly, if the final query result has K objects, yet, a circle C centered at P with radius d (the distance from P to its furthest Kth object) overlaps with any grid cell that is not covered by R, we will issue the query again with larger R that includes all the cells covered by circle C. In Figure 3, consider that Q_{30} is a K-nearest-neighbor query with location point P, located in C_{19} and depicted by a star in Figure 3(a), with K = 5 and t = 30. In this case, *Panda* will form a range query that includes cell C_{19}. This gives an answer with only three objects, which is less than K. Then, the query is issued again with an area R that covers the cells C_{13}, C_{14}, C_{19}, C_{20}, C_{25}, and C_{26}, which will return more than five objects as an answer. Only the closest five will be selected.

5.2 Extensibility to Continuous Queries

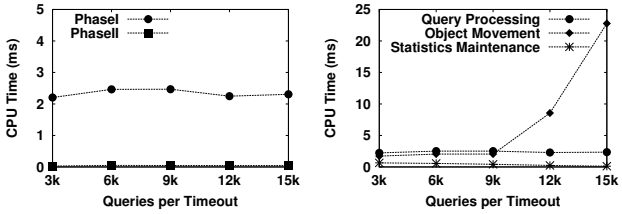
Unlike snapshot predictive spatio-temporal queries that are issued once, and are terminated immediately once a query answer is returned to the user, continuous predictive spatio-temporal queries stay active in the server for long intervals, e.g., hours or days, where updates in the query answer are continuously sent to the user. Examples of continuous predictive spatio-temporal queries include: "Continuously, monitor all moving objects within five miles of my location", "Alert me if any moving object is expected to be within one mile of my area in the next 30 minutes", or "Send E-coupons to my closest three customers, 10 minutes before they become my closest ones".

Unlike previous techniques for predictive spatio-temporal queries, it is a silent feature in *Panda* that it lends itself to continuous query execution. Once a continuous query is issued to *Panda*, it is immediately registered as a frequent one, regardless of the number of times it is issued. To this end, the answer of the continuous query is always precomputed and stored in the *Query List* data structure of the cells overlapping the query region. The only additional component here is to push the precomputed answer to the user upon any change. As continuous queries could be range, aggregate, or k-nearest-neighbor queries, they follow the same requirements of each query type as described in the previous section. Once a continuous query is terminated, we return to the normal processing mode where we take care of the *counter* of each query to determine the query parts to precompute.



(a) Query Processing Phases (b) Main modules

Figure 5: Effect of Threshold Tuning



(a) Query Processing Phases (b) Main modules

Figure 6: Effect of Timeout Value

6. EXPERIMENTS

In this section, we evaluate the efficiency and the scalability of our proposed system *Panda* for processing predictive queries. We start by explaining the environment of the conducted experiments in Section 6.1. Then, we study the impact of threshold tuning on the performance of the main modules of *Panda* in Section 6.2. Section 6.3 provides the effect of different timeouts on *Panda* efficiency. The behavior of *Panda* with different query workloads is given in Section 6.4. Finally, we examine the scalability of *Panda* with respect to outsized queries and large number of objects in Sections 6.5 and 6.6, respectively.

6.1 Experiment Setup

In our performance evaluation experiments, we use the Network-based Generator of Moving Objects [3] to generate large sets of synthetic data of moving objects. The input to the generator includes a real road network map for Hennepin County, Minnesota, USA. The output of the generator includes different sets of moving objects that move on the given road network map. The used data sets require some data preprocessing to partition the space in which objects move is 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 grid TTG filled before starting the experiment, the travel time between any pair of cells, C_i and C_j, is obtained by taking the average time it takes from the underlying set of objects to move from C_i to C_j.

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 10K queries per batch, and increases by 10K until reaches 100K queries in a query workload. The generated query regions are squares and their locations are uniformly distributed over the space. The size of the generated queries vary from 0.01 to 0.16 of the total space size. The future times for the generated queries vary from 10 to 80 time unites. A 3k query workload is used for warming up the system before we start to measure the experimental results.

All experiments are based on an actual implementation of *Panda*. 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++. In all experiments, the evaluation and comparison are in terms of CPU time cost.

6.2 Impact of Threshold Tuning

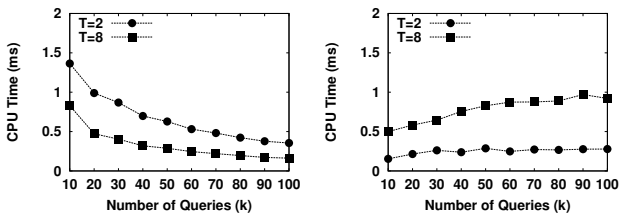
In the first set of experiments, we study the impact of different threshold $\mathcal{T}$ values on the efficiency of *Panda*. The minimum value that the $\mathcal{T}$ can take in this experiment is zero, which means that the results for any possible query will be precomputed in advance. Accordingly, once a predictive query is received, the answer is read and returned to the user without any further computation. It is obvious here that at this threshold value, we will have the fastest response time, since no computation happens after receiving a query. Just the precomputed result is accessed and returned directly as a final query answer. However, it is expected to have the most significant overhead for updating those precomputed answers. The maximum threshold value is ten which decided based on the number of queries in a timeout divided by the number of cells in our space grid.

Figure 5 illustrates the effect of choosing different threshold values on the performance of *Panda*. In this experiment we run 5k queries on 10K moving objects at $\mathcal{T}$ = 0, 2, 4, 5, 6, 8, 10 shown in the x-axis and y-axis represents the average CPU cost per query in milliseconds. In Figure 5(a), we study the influence of threshold tuning on the two phases of the *query processing* module. As the value of $\mathcal{T}$ increases, the cost of *Phase I* increases. The justification is that when $\mathcal{T}$ has large value, this means that most of the answers in the cells intersecting with the query region need to be computed from scratch. With large threshold value, only queries with frequency rate above that value are precomputed. *Phase II* is not sensitive to threshold tuning. The reason is that *Phase II* concerns mainly with updating the statistics inside the cells affected by received queries, and those cells are only sensitive to the size of the query region rather than the threshold value.

Figure 5(b) depicts the sensitivity of the main modules in *Panda* to different threshold values. The cost of *query processing* module increases when $\mathcal{T}$ increases. We justified that in the previous sub-figure. The *periodic statistics maintenance* module, *statistics maintenance* for short, is not sensitive to $\mathcal{T}$ at all. The third module, *object movement* is significantly affected by threshold value, as with small values, more answers are precomputed, hence more updates are triggered with each single object movement, and vise versa. In a nutshell, 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.

6.3 Impact of Timeout

In this set of experiments, we study the effect of timeout values on the efficiency of *Panda*. We run a workload of 30K queries on 20K objects with threshold $\mathcal{T}$ = 8, while varying the timeout values from 3k queries every timeout to 15K queries per timeout. In Figure 6, the x-axis represents the number of queries per timeout and the y-axis represents the average CPU cost per query in milliseconds for processing the given workload at that timeout value. Figure 6(a) shows that there is a slim effect of selecting different



(a) Update Cost (b) Response Time

Figure 7: Efficiency Evaluation with Different Workloads

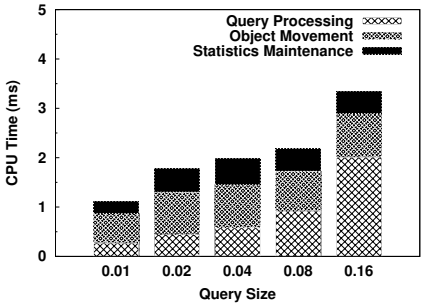


Figure 8: Scalability With Query Size

timeouts on the two phases of the *query processing* module. At larger timeouts, 12K and 15K, there is a higher opportunity for queries counters to exceed the given threshold. However, it takes most of the experiment time before the counter passing that threshold. So the effect of timeout is not significant on the two phases of this module. In all cases, more portions of the queries answers are expected to be precomputed, which in turn starts to decrease the cost of computing answers from scratch at timeout 12K in *phaseI*. Figure 6(b) assures that timeout values have different impact on the three main modules of the *Panda* system. With smaller timeouts, the *periodic statistics maintenance* module is dispatched more times which gives it higher cost than with bigger timeouts. The curve of the statistics maintenance cost has a decreasing trend from 0.65 milliseconds/query at timeout = 3k to 0.12 milliseconds/query at timeout = 15k. As depicted in the previous figure, there is a slim effect of timeout tuning on the average query cost of the *query processing* module. With an opposite impact, the cost of the *object movement* module sharply increases with higher timeouts as a result of more queries parts are being precomputed which means each single object movement triggers a possible update.

6.4 Efficiency Evaluation

To evaluate the efficiency of *Panda*, we processed workloads of predictive queries while varying the number of received queries from 10K to 100K. We compare the response time and update cost of *Panda* at two different threshold values, two and eight respectively. The response time is the time a user has to wait to get the query result, and it is equal to *Phase I* of the *query processing* module. The update cost is the time consumed to update the precomputed answer when a change happens, and it is equivalent to the cost of the *object movement* module. In this experiment, we use *timeout* equals to the time required to process 3K queries which means after every 3k queries, we call the *statistics maintenance* module to adjust *Panda* decision about which parts to precompute. As we mentioned in the setup, we use a warmup workload with 3k queries before we write down the experimental results. The used data file contains 20k of moving objects.

Figures 7 provides a comparison between *Panda* response time and update cost with different queries workloads at the two selected threshold values. The horizontal axis represents the number of queries in thousands in each workload file, and the vertical axis measures the average CPU cost per query in milliseconds. Figure 7(a) studies the update cost at different workloads. As expected, the update costs at $\mathcal{T}$ = 2 are much bigger than the ones at $\mathcal{T}$ = 8 in all workloads, while the behavior of the response time, Figure 7(b), is the opposite. It is also remarkable from the decreasing trend of the average update cost per query that *Panda* acts efficiently even with heavy workloads. The reason for that is the same object update serves many queries with the same update cost. On the other hand, there is a very slight increase in the response time then it almost remains steady from 60K to 100K.

6.5 Scalability with Outsized Queries

We proceed to study the scalability of *Panda* with large query sizes. In this set of experiments, Figure 8, we show that *Panda* can scale up with outsized queries, where we run a workload of 20k queries on 20K moving object at $\mathcal{T}$ = 5 and timeout = 5K. The region size of the given queries vary from 0.01 to 0.16 of the total space. From this figure, we notice that when the size increases, *Panda* still behaves efficiently without significant increase in the total processing cost, for example when the query size increased by 16 times, from 0.01 to 0.16 of the total space, the average processing time per query increases only by three times, from 0.11 to 0.34 milliseconds/query. We also notice that the most affected parts are the *query processing* and *object movement* modules because when query size increases, the number of cells intersecting with query region increases. Accordingly, more cells need to compute their results from scratch and more are already precomputed and need to be updated with objects movements. We can conclude that increasing query size affects the response time without significant overhead on update cost.

6.6 Scalability with Number of Objects

In our second set of scalability experiments, Figure 9 depicts the behavior of the main components of *Panda* when the number of moving objects increases from 5K to 80K. We run these experiments with $\mathcal{T}$ = 5, timeout = 3K, and 20K queries. As seen in this figure, the *object movement* is the most affected module as there is a positive relationship between the number of objects and update overhead cost results from their movements. It is also observed that there is almost no impact of increasing the underlying number of objects on both the *statistics maintenance* and the *query processing* modules. As noticed from the average CPU cost per query when the number of objects increases by 16 times, from 5K to 80K, the average cost per query increases only by less than four times, i.e., from 0.7 to 2.7 milliseconds/query. Also, the average cost per object decreases from 2.95 milliseconds/object at 5K objects to 0.67 milliseconds/object at 80K objects. This shows that *Panda* can scale up with large number of moving objects without scarifying neither the response time nor the overall performance.

7. CONCLUSION

We have presented *Panda* as a scalable, efficient, and generic framework for supporting a wide variety of predictive spatio-temporal queries. Unlike previous attempts in supporting predictive queries, *Panda* targets long-term query prediction as it relies on adapting a well-designed long-term prediction function to: (a) scale up to large number of moving objects, and (b) support large number

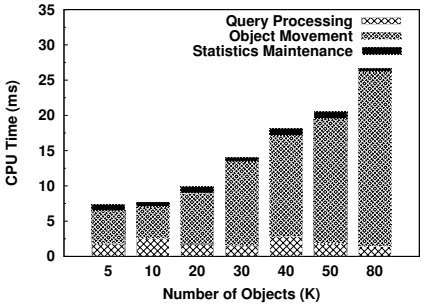


Figure 9: Scalability With Objects Number

of predictive queries. The main idea of *Panda* is to precompute the answer for certain areas of space that are likely to be hit by predictive queries. *Panda* employs a tunable threshold $\mathcal{T}$ that achieves a trade-off between the computational overhead required for precomputation and the query response latency. *Panda* is extensible to support various kinds of predictive spatio-temporal queries including range, aggregate, and k-nearest-neighbor queries. Finally, *Panda* lends itself to support continuous queries, which is very common in spatio-temporal databases. Experimental results, based on large data sets, show that *Panda* is scalable, efficient, and as accurate as its underlying prediction function.

8. REFERENCES

[1] 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.

[2] A. Brillingaite and C. S. Jensen. Online Route Prediction for Automotive Applications. In *Proceedings of teh World Congress and Exhibition on Intelligent Transport Systems and Services, ITS*, London, United Kingdom, Oct. 2006.

[3] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *International Journal on Advances of Computer Science for Geographic Information Systems, GeoInformatica*, 6(2):153–180, 2002.

[4] Y.-J. Choi and C.-W. Chung. Selectivity Estimation for Spatio-temporal Queries to Moving Objects. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 440–451, Wisconsin, USA, June 2002.

[5] A. Corral, M. Torres, M. Vassilakopoulos, and Y. Manolopoulos. Predictive Join Processing between Regions and Moving Objects. In *Advances in Databases and Information Systems Lecture Notes in Computer Science, ADBIS*, volume 5207, pages 46–61, 2008.

[6] J. Froehlich and J. Krumm. Route Prediction from Trip Observations. In *Society of Automotive Engineers (SAE) World Congress*, Michigan, USA, Apr. 2008.

[7] Y. Gu, G. Yu, N. Guo, and Y. Chen. Probabilistic Moving Range Query over RFID Spatio-temporal Data Streams. In *Proceedings of the International Conference on Information and Knowledge Managemen, CIKM*, pages 1413–1416, Hong Kong, China, Nov. 2009.

[8] H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 479–490, Maryland, USA, June 2005.

[9] H. Jeung, Q. Liu, H. T. Shen, and X. Zhou. A Hybrid Prediction Model for Moving Objects. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 70–79, Cancún, México, Apr. 2008.

[10] 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.

[11] H. A. Karimi and X. Liu. A Predictive Location Model for Location-Based Services. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS*, pages 126–133, Louisiana, USA, Nov. 2003.

[12] 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 *Proceedings of the International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, KES*, pages 379–389, Vietri sul Mare, Italy, Sept. 2007.

[13] J. Krumm. Real Time Destination Prediction Based on Efficient Routes. In *Proceedings of the Society of Automotive Engineers World Congress, SAE*, Michigan, USA, Apr. 2006.

[14] 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 *Proceedings of the International Conference on Mobile Data Management, MDM*, pages 178–182, Ayia Napa, Cyprus, May 2005.

[15] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous Query Processing of Spatio-temporal Data Streams in PLACE. *GeoInformatica*, 9(4):343–365, Dec. 2005.

[16] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast Nearest-Neighbor Query Processing in Moving-Object Databases. *GeoInformatica*, 7(2):113–137, June 2003.

[17] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 331–342, Texas, USA, May 2000.

[18] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 422–432, Birmingham U.K, Apr. 1997.

[19] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 202–213, MASSACHUSETTS, USA, Mar. 2004.

[20] Y. Tao, C. Faloutsos, D. Papadias, and B. L. 0002. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 611–622, Paris, France, June 2004.

[21] Y. Tao and D. Papadias. Time-parameterized Queries in Spatio-temporal Databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 334–345, Wisconsin, USA, June 2002.

[22] Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM Transactions on Database Systems, TODS*, 28(2):101–139, 2003.

[23] Y. Tao, J. Sun, and D. Papadias. Analysis of predictive spatio-temporal queries. *ACM Transactions on Database Systems, TODS*, 28(4):295–336, Dec. 2003.

[24] Y. Tao, J. Sun, and D. Papadias. Selectivity Estimation for Predictive Spatio-Temporal Queries. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 417–428, Bangalore, India, Mar. 2003.

[25] H. Wang, R. Zimmermann, and W.-S. Ku. Distributed Continuous Range Query Processing on Moving Objects. In *Proceedings of the International Conference on Database and Expert Systems Applications, DEXA*, pages 655–665, Krakow, Poland, Sept. 2006.

[26] 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.

[27] 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.