An Adaptive Updating Protocol for Reducing Moving Object Database Workload

Su Chen Beng Chin Ooi Zhenjie Zhang
School of Computing
National University of Singapore
{chensu,ooibc,zhenjie}@comp.nus.edu.sg

ABSTRACT

In the last decade, spatio-temporal database research focuses on the design of effective and efficient indexing structures in support of location-based queries such as predictive range queries and nearest neighbor queries. While a variety of indexing techniques have been proposed to accelerate the processing of updates and queries, not much attention has been paid to the updating protocol, which is another important factor affecting system performance. In this paper, we propose a generic and adaptive updating protocol for moving object databases with less number of updating messages between the objects and database server, thereby reducing the overall workload of the system. In contrast to the approach adopted by most conventional moving object database systems where the exact locations and velocities last disclosed are used to predict their motions, we propose the concept of *Spatio-Temporal Safe Region* to approximate possible future locations. Spatio-temporal safe regions provide larger space of tolerance for moving objects, freeing them from location and velocity updates as long as the errors remain predictable in the database. To answer predictive queries accurately, the server is allowed to probe the latest status of some moving objects when their safe regions are inadequate in returning the exact query results. Spatio-temporal safe regions are calculated and optimized by the database server with two contradictory objectives: reducing update workload while guaranteeing query accuracy and efficiency. To achieve this, we propose a cost model that estimates the composition of active and passive updates based on historical motion records and query distribution. We have conducted extensive experiments to evaluate our proposal on a variety of popular indexing structures. The results confirm the viability, robustness, accuracy and efficiency of our proposed protocol.

1. INTRODUCTION

Spatio-temporal databases, especially *Moving Object Databases* (MOD), are well studied in the database community. Efficient disk-resident indexing structures [5, 9, 12, 14, 19, 21] have been proposed to support different types of queries for location-based services. Two typical example queries are "*Which are the vehicles remaining in Central Park after 10 minutes?*" or "*Who will be the policeman closest to City Hall after 30 minutes?*". These queries can be formalized with range query and nearest neighbor query on predicted locations of objects moving in a two-dimensional (2D) space. A moving object database keeps track of all objects by receiving occasional location disclosures from objects. In the meantime, the database server also answers all incoming predictive queries. Fig.1 shows a typical architecture of a moving object system.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '10, September 13-17, 2010, Singapore
Copyright 2010 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

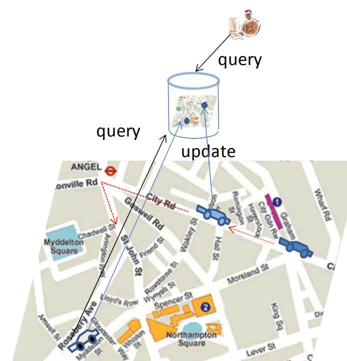


Figure 1: Architecture of a moving object database system

While most existing studies on moving object databases focus on index and query efficiency, less effort has been made to address the issue on the updating mechanism/protocol, which is another crucial factor affecting the system workload and performance even more. Considering the limited room of technical advances on indexing structures and query optimization, it is now important to reconsider the design of the updating protocol. This paper is the first attempt on a systematic investigation on this possibility. In particular, we propose a new adaptive updating protocol, which reduces the object updating frequencies by maintaining only approximate motions instead of exact ones in the database system. The design of the protocol is on the basis of a careful analysis on the advantages and disadvantages of the existing protocols.

The first generation of moving object databases tracks object locations only. In particular, an object reports its location to the database server every (logical) timestamp. It is inadequate to make prediction with location information only. Later, in order to support predictive queries and reduce update frequency, motion models are introduced to approximate object movements. For example, a linear model with object's location and velocity can be used to predict object's location at any future time, assuming the object moves with a fixed speed along a straight line. Currently, object updating protocols follow some simple strategies, such as *Temporal Bounded Strategy* and *Spatial Bounded Strategy*. In the tempo-

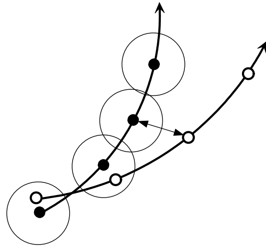


Figure 2: Example of spatial bounded strategy

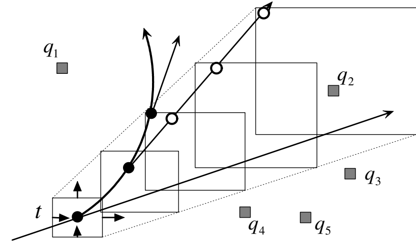


Figure 3: Example of spatial-temporal safe region

ral bounded strategy, moving objects update their motion models periodically after a fixed time interval. This scheme is neither effective nor efficient. An update is issued even if the model does not change since the last update while the movement prediction immediately becomes meaningless when the object changes its velocity dramatically soon after the last update. On the other hand, the spatial bounded strategy [13] adaptively decides the update time, depending on the spatial error incurred by the movement model constructed on the previous update. Specifically, two motion models are stored on both the object and the database server. The motion model at the object side is always updated with current movement, which is supposed to be more accurate. The object regularly measures the error of the motion model which has been sent to the server previously [13]. An immediate update is issued if the prediction error is larger than some specified threshold. In Fig.2, we present an example of the spatial bounded strategy. In the figure, solid points (hollow points resp.) represent the predicted locations of the old model (new model resp.). When the distance between the solid point and hollow point in the near future exceeds the tolerance, the object updates its movement model with the database server. To ensure the accuracy of predictive query, the database server is allowed to actively request the model update from the object, when the candidate object does not necessarily satisfy the query condition.

Compared with the spatial bounded strategy [13] introduced above, our new generic updating protocol is equipped with three major features to enhance the performance. First, we adopt the linear movement model instead of the complex high-order models, which can be seamlessly integrated into database system with optimized index structures [4]. While achieving benefits on computational cost and index efficiency with the simple motion model, our empirical studies show that there is no significant difference on prediction quality even when more complicated motion model is employed instead.

Second, our protocol allows approximation on object motion in both spatial and velocity spaces, providing better flexibility on the tuning of the updating protocol. In particular, our protocol relies on the new concept of *Spatio-Temporal Safe Region*, or STSR in short, which is a rectangle in spatio-temporal space bounding the possible location and velocity. In Fig.3, we illustrate the basic idea of our framework following the example in Fig.2. Our system assigns some rectangle to the object as spatial safe region, as well as some maximal and minimal speeds as velocity safe region. A series of rectangles, called *Predicted Regions*, can then be derived to bound the possible locations of the object on subsequent timestamps. An update is necessary if: 1) the location of the object cannot be bounded by the predicted region at the current timestamp, or 2) the movement inferences on the current location and velocity violate some predicted regions in the future. In Fig.3, the object is "safe" at time $t+1$ since it stays in the predicted region and its updated movement prediction remains bounded by the following predicted regions. At time $t+2$, although its new location is still within the predicted region, updating is invoked to ensure that the prediction error on the server side remains within a tolerable bound.

Third, our updating protocol takes both historical motion records and query distribution into consideration, rendering a cost optimization model and an automatic tuning mechanism highly adaptive to the changing world. Recall the example in Fig.3. Given the locations of the recent queries shown in the figure, $\{q_1, q_2, q_3, q_4, q_5\}$, careful design on the STSR avoids the potential overlaps between the predicted regions and popular querying areas in the spatial space. This enables our framework to outperform existing solutions on the maximization of update savings. We summarize the contributions of this paper as follows:

1. We present a generic and adaptive updating protocol for the purpose of reducing the number of update messages while guaranteeing the efficiency and accuracy of predictive queries.

2. We propose a cost model to estimate the update workload incurred by specific moving object(s).

3. We propose cost-based optimization strategies to reduce the updating frequencies.

4. We evaluate the performance of our proposal with a variety of index structures.

2. PRELIMINARIES

Assume there are n moving objects, i.e., $O = \{o_1, o_2, \ldots, o_n\}$, being monitored in the system. Following the common assumption on spatio-temporal indexing, the time axis is sliced into snapshots at discrete times, i.e., $T = \{0, 1, \ldots, t, \ldots\}$. The exact physical location of object o_i at timestamp t is denoted by a vector $l_i^t = (l_i^t.x, l_i^t.y)$. Similarly, the velocity of o_i at t is denoted by $v_i^t = (v_i^t.x, v_i^t.y)$. With the linear movement model, the predicted location of o_i at time $s \geq t$ is estimated as pl_i^s, i.e., $pl_i^s.x = l_i^t.x + v_i^t.x \cdot (s-t)$ and $pl_i^s.y = l_i^t.y + v_i^t.y \cdot (s-t)$.

Before delving into the details of our updating protocol, we first introduce the concept of *Spatio-Temporal Safe Region*, as defined below.

DEFINITION 2.1. ***Spatio-Temporal Safe Region (STSR)*** *Given a moving object o_i, a Spatio-Temporal Safe Region (STSR) for o_i is represented by a tuple $R(o_i) = (LR, VR, t_r, t_e)$, where LR is a rectangle in the physical space (i.e., the space where the object moves), VR is a rectangle in the velocity space, t_r is the reference time, and t_e is the expiry time.*

Given an STSR $R(o_i)$, the spatial rectangle LR is bounded by $LR.x^{\vdash}, LR.x^{\dashv}$, $LR.y^{\vdash}, LR.y^{\dashv}$ on the two dimensions respectively. Similarly, the velocity rectangle VR of $R(o_i)$ is bounded by

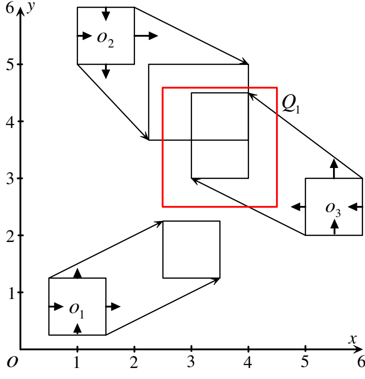


Figure 4: Example of STSR

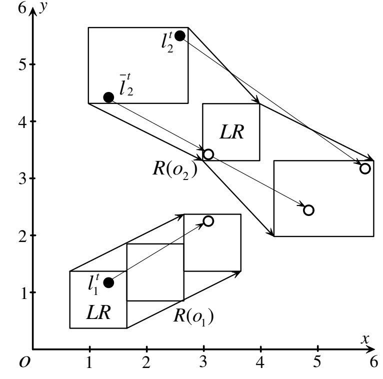


Figure 5: Examples of consistency verification

STSR	LR	VR	t_r	t_e
$R(o_1)$	$[0.5, 1.5] \times [0.2, 1.2]$	$[0.5, 0.5] \times [1, 1]$	1	4
$R(o_2)$	$[1, 2] \times [5, 6]$	$[1.2, 2] \times [-1.4, -1]$	2	5
$R(o_3)$	$[5, 6] \times [2, 3]$	$[-1, -1] \times [0.5, 0.75]$	1	5

Table 1: Details on the STSR in Fig.4

$VR.x^{\vdash}, VR.x^{\dashv}$, $VR.y^{\vdash}, VR.y^{\dashv}$. Intuitively, LR relaxes the location of o_i at reference time t_r, and VR approximates the possible velocities of o_i between t_r and t_e. A predicted region, as defined below, infers the possible locations of o_i at time t before the expiry time t_e.

DEFINITION 2.2. ***Predicted Region***
Given an STSR $R(o_i) = (LR, VR, t_r, t_e)$ and inferring time t ($t_r \leq t \leq t_e$), the predicted region of o_i at time t, $P_i^t =$ $P.x^{\vdash}, P.x^{\dashv}$ $\times$ $P.y^{\vdash}, P.y^{\dashv}$, is the maximal spatial rectangle expanded from LR with respect to VR, where

$$\begin{aligned} P.x^{\vdash} &= LR.x^{\vdash} + VR.x^{\vdash}(t - t_r) \\ P.x^{\dashv} &= LR.x^{\dashv} + VR.x^{\dashv}(t - t_r) \\ P.y^{\vdash} &= LR.y^{\vdash} + VR.y^{\vdash}(t - t_r) \\ P.y^{\dashv} &= LR.y^{\dashv} + VR.y^{\dashv}(t - t_r) \end{aligned}$$

The definition above assumes that the inferring time t is no earlier than the reference time t_r, which can be easily relaxed. When $t < t_r$, the predicted region is calculated with the "reverse" velocity bounding rectangle, which is $VR.x^{\dashv}, VR.x^{\vdash}$ $\times$ $VR.y^{\dashv}, VR.y^{\vdash}$. The predicted region is

$$\begin{aligned} P.x^{\vdash} &= LR.x^{\vdash} + VR.x^{\dashv}(t - t_r) \\ P.x^{\dashv} &= LR.x^{\dashv} + VR.x^{\vdash}(t - t_r) \\ P.y^{\vdash} &= LR.y^{\vdash} + VR.y^{\dashv}(t - t_r) \\ P.y^{\dashv} &= LR.y^{\dashv} + VR.y^{\vdash}(t - t_r) \end{aligned}$$

An STSR $R(o_i) = (LR, VR, t_r, t_e)$ is consistent with the moving object o_i at time $t \leq t_e$, if both of the following two conditions hold: 1) Current location l_i^t remains in the predicted region P_i^t inferred from $R(o_i)$; and 2) The predicted location pl_i^s remains in the predicted region P_i^s for any $t < s \leq t_e$.

Note that the consistency only depends on the locations, which remains valid even when the velocity of o_i at time t is out of the velocity rectangle VR.

As an example, Tbl.1 lists the STSRs of three objects $\{o_1, o_2, o_3\}$ which are illustrated in Fig.4. Based on Definition 2.1 and 2.2, we can derive the predicted regions of the objects at timestamp $t = 3$, as shown in Fig.4.

According to the above definitions, it is straightforward to verify the consistency of the location with the given STSR on the client side, i.e., the moving object, by simply checking the predicted locations of the object at every timestamp before the expiry time t_e. Since we adopt the linear model on deriving the predicted regions, the verification process can be simplified by checking the predicted location (or exact current location) at only three timestamps. In particular, the object first tests if the current timestamp is beyond the expiry time t_e. If the STSR has already expired, the algorithm returns a negative answer immediately. Otherwise, the object then checks if the current location l_i^t and the predicted location $pl_i^{t_e}$ are both covered by the predicted regions P_i^t and $P_i^{t_e}$ respectively. Finally, if the current timestamp t is ahead of the reference time t_r, we also need to see if the predicted location at t_r is adequately covered by the location rectangle LR. The verification algorithm is summarized in Algorithm 1 in Appendix B.

In Fig.5, we give three examples to show why the conditions above are sufficient to prove the validity of a STSR. Two STSRs, $R(o_1)$ and $R(o_2)$, are shown in the figure. The solid dots denote the current locations and hollow dots represent the predicted locations. The reference time of $R(o_1)$ is the current timestamp. Since the current location l_1^t is inside the location rectangle and the predicted location is covered by the predicted region, $R(o_1)$ remains valid. The reference time of $R(o_2)$, on the other hand, is after the current timestamp. The predicted regions between t and t_e are thus extended backwards according to the location prediction formula. If we test only the locations of the current timestamp and expiry time, some false positive STSR may wrongly pass the verification, e.g., the prediction based on l_2^t. However, when the location at the reference time is also verified, all false positives are pruned, as the movement prediction at $\bar{l}_2^t$ implies.

In this paper, we focus on the processing of predictive range queries. Given a querying rectangle QR in location space and the querying time t_q, the predictive range query returns all the objects with predicted locations in QR at time t_q. Although we constrain our discussion in range query throughout the paper, it is easy to see that other queries, e.g., k-nearest neighbor query, can be answered with a series of range queries [9].

3. UPDATING PROTOCOL

Generally speaking, our updating protocol consists of two types of updates: the active update and the passive update. In the database, each object o_i is always associated with one (and only one) STSR

$R(o_i)$, which is kept in the database as well as the memory of the client device. In the following, we discuss the two updates in detail.

Active Update

On each timestamp, the object o_i checks if the new motion prediction model with its current location and velocity remains consistent with its STSR $R(o_i)$, using Algorithm 1. If there is any inconsistency, it issues an active update to the database server consisting of its current location and velocity.

At the server side, the database system continuously listens to any incoming active updates from the objects. If one of the moving objects o_i updates its location and velocity at time t, the system renews the record of the object in the database. The system then calculates a new STSR for o_i based on the the updated record. The new STSR is sent to the object o_i, while the corresponding record in the database is also refreshed accordingly. An outline of the update procedure can be found in Algorithm 2 in Appendix B.

Query Processing and Passive Update

While active updates are initiated by the objects themselves, passive updates are issued when the database processes predictive range queries. Typically, predictive range queries in most moving object databases are processed using a filter-and-refine approach, which determines candidate objects based on their predicted regions and verifies them by probing passive updates if necessary. A candidate set is constructed first by retrieving all objects overlapping with the query range QR at query time t_q, based on the predicted regions of the STSRs. For each candidate object o_i, if the predicted region of o_i is covered by the query range, the object can be safely included into the query result. Otherwise, a request is sent to the object for an update on its current location and velocity, which will be used on the server side to make a more accurate prediction. The object is subsequently listed in the query result if the new predicted location is still in the query range. A general framework for answering range queries based on the concept of STSRs is presented in Algorithm 3 in Appendix B.

Let us recall the example shown in Fig.4, and see how predictive range queries can be answered with STSRs. If a range query is issued in the rectangle region $QR = [2.5, 4.5] \times [2.5, 4.5]$ at querying time $t_q = 3$, the predicted regions can be calculated according to the inference equations above. For object o_3, for example, the predicted region at timestamp $t = 3$, P_3^3, is the rectangle $[3, 4] \times [3, 4.5]$. Since P_3^3 is covered by the query region completely, o_3 is a positive result if it remains consistent with its current STSR. On the other hand, there is no overlap between P_1^3, implying that o_1 is a negative result. Unlike the other o_1 or o_3, the case of o_2 is more complicated, since the predicted region P_2^3 partially overlaps with QR. To clarify if o_2 is in the query result or not, the system sends an update request to o_2 for its current motion parameters, i.e., the location and velocity.

To put the updating protocol in use, there are two issues to resolve. First, to minimize the workload of the system, the STSR calculation algorithm plays an important role in finding an optimal STSR (Step 1 in Algorithm 2). Recall the example in shown Fig.4, which demonstrates that overall update cost, including both active and passive updates, can be reduced if we extend the predicted regions as much as possible without too many overlapping query ranges. We will expand on this idea with further details in Sec.4.1 and the corresponding optimization techniques for better STSR design in Sec.4.3. Second, considering different index structures in moving object databases, objects are stored and searched in different ways, even with the same linear motion model. Therefore, it remains unclear so far how existing database systems support the search for all objects whose corresponding predicted regions overlap with the query range. This is important in query processing for efficient retrieval of candidate objects in the filter step (Step 1 in Algorithm 3). In Sec.C in the appendix, we answer this question by showing that it does not take much effort to modify existing moving object indexing structures to support these queries.

4. OPTIMIZATION TECHNIQUES

4.1 Cost Model

In this section, we present a cost model estimating the probable validity of a given STSR. As introduced in Sec.3, there are two types of updates, namely *Active Update* and *Passive Update*. Either active update or passive update leads to a new STSR. We use $P_a(R(o_i))$ and $P_p(R(o_i))$ to denote the probabilities of active update and passive update happening on $R(o_i)$ before the expiry time t_e. An STSR remains valid until the expiry time t_e with probability:

$$P_{valid}(R(o_i)) = (1 - P_a(R(o_i))) \cdot (1 - P_p(R(o_i))) \quad (1)$$

Obviously, a good STSR should maximize P_{valid}. Intuitively, a larger LR and a larger VR lead to lower probability of $P_a(R(o_i))$ but higher probability of $P_p(R(o_i))$, and vice versa. To optimize Equation 1, it is important to estimate both probabilities first.

Active Update Probability

An active update is issued by object o_i if the previous STSR is no longer consistent with the current location and velocity. Given an STSR $R(o_i)$, $P_a(R(o_i))$ is the probability of inconsistency happening before the expiry time t_e of $R(o_i)$. Without knowing the exact future trajectory of o_i, it is hard to estimate $P_a(R(o_i))$. If we assume the previous motion model doesn't change, $R(o_i)$ will always be valid until t_e; on the other hand, it is hard to indicate the possible changes in the motion without any additional knowledge on the future trajectory. However, if all the similar historical trajectories are recorded in the database, the accumulated statistical information provides probability estimation on the active update. Unfortunately, this solution is impractical due to the high cost in both storage and processing on the trajectories. To facilitate effective and efficient statistical estimation, the database system maintains a set of STSR update records, as defined below, to simulate the historical trajectories of objects.

DEFINITION 4.1. ***STSR Update Record***
An STSR update record is a tuple $(R(o_k), o_k, l_k^{t_u}, v_k^{t_u}, t_u)$ where k is the identity of the associated moving object, t_u is the time when the update record is generated, $l_k^{t_u}$ and $v_k^{t_u}$ are the location and velocity of o_k at t_u, and $R(o_k) = (LR, VR, t_r, t_e)$ is the latest STSR of o_k before the update time t_u.

The STSR update records are maintained in a separate table in the database, called the *Update Record Table*. A record is inserted into the table when: 1) an update from o_k is received due to the violation on $R(o_k)$, or 2) the previous STSR $R(o_k)$ expires. In the first case, the location and velocity at update time are written into the record. In the second case, NULL values are inserted instead. This implies that each STSR issued in the past has a record kept by the database system in the update history table.

Next, we introduce the concept of *Record Coverage* to evaluate the robustness of a new STSR with respect to similar STSR update records in the update record table. Specifically, an STSR $R(o_i)$ covers an STSR update record $U = (R(o_k), o_k, l_k^{t_u}, v_k^{t_u}, t_u)$, if

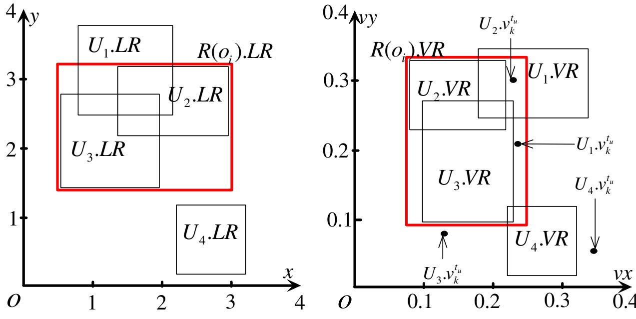


Figure 6: Coverage of STSR on update records

1) $R(o_i).LR$ covers $R(o_k).LR$, and 2) $R(o_i).VR$ covers both $R(o_k).VR$ and $v_k^{t_u}$. Without ambiguity, we use $R(o_i) \supseteq U$ to denote the coverage relationship, in which U is a specific update record. In Fig.6, for example, we present an example on the coverage relationship with an STSR $R(o_i)$ and four update records $\{U_1, U_2, U_3, U_4\}$. The location rectangle LR of the update records are shown with black thin lines in the spatial space on the left. Similarly, the velocity rectangle VR and the velocity at update time $v_k^{t_u}$ are plotted in the velocity space on the right. Given the STSR $R(o_i)$ marked with red thick lines in both spaces, $R(o_i)$ covers U_2 by the definition above. U_3 is not covered by $R(o_i)$ since the updated velocity $v_k^{t_u}$ of U_3 is out of the velocity rectangle of $R(o_i)$.

If the STSR $R(o_i)$ covers an update record U, it is able to remain consistent until the expiry time, if o_i follows the same trajectory of o_k when U was recorded for o_k. This motivates the following definition of *Coverage Rate* to approximate the probability of an active update on the new STSR $R(o_i)$.

DEFINITION 4.2. ***Coverage Rate***
Given an STSR $R(o_i)$ and a group of similar STSR update records U_{NN}, the coverage rate of $R(o_i)$ is measured by $\frac{|\{U_i \in U_{NN} \mid R(o_i) \supseteq U_i\}|}{|U_{NN}|}$

We now discuss the similar update record set U_{NN} as shown in the above definition. To get all update records related to the current moving object o_i, the system retrieves all update records in U_{NN}, if the location rectangle $U_i.LR$ and velocity rectangle $U_i.VR$ of the record U_i cover the location $l_i^{t_r}$ and velocity $v_i^{t_r}$ at reference time t_r respectively. As a summary, we have

$$P_a(R(o_i)) = \frac{|\{U_i \in U_{NN} \mid R(o_i) \supseteq U_i\}|}{|U_{NN}|}$$

Passive Update Probability

A passive update is issued when it is not sufficient to decide if an object meets the query with its current STSR stored in the database, i.e., the predicted region partially overlaps with the query region. To estimate the number of passive updates for a given STSR $R(o_i)$, it is necessary to predict the probability of the event of partial overlap. To simplify the model and save computational cost, we relax the probability by including any overlap event even if the predicted region is completely covered by the query range. This relaxation does not greatly affect estimation error since the query range is usually not large enough to cover many predicted regions.

Following the existing assumptions on the performance analysis of range queries [5, 12, 19], we assume the querying location and querying time follow the uniform distribution in spatial space and temporal space. The probability of a predicted region overlapping any range query at time t is thus proportional to the volume of the predicted region.

For an STSR $R(o_i)$ issued at update time t_u, the total volume of all predicted regions at timestamps between t_u and t_e is denoted by $Vol(R(o_i))$. By using the following notations to replace the side lengths of location rectangle and velocity rectangle, i.e., $LD_x = LR.x^{\dashv} - VR.x^{\vdash}$, $LD_y = LR.y^{\dashv} - VR.y^{\vdash}$, $VD_x = VR.x^{\dashv} - VR.x^{\vdash}$, and $VD_y = VR.y^{\dashv} - VR.y^{\vdash}$, the total volume can be further simplified:

$$Vol(R(o_i)) = \sum_{t=t_u}^{t_e} (LD_x + VD_x(t - t_r)) \cdot (LD_y + VD_y(t - t_r))$$

If the expected number of queries happening at each timestamp is N and the volume of the whole spatial space is S, the probability of not meeting any range query is approximated by the ratio of total volume with respect to the expected query volume.

$$P_p(R(o_i)) = \max \quad 1 - \frac{Vol(R(o_i)) \cdot N}{(t - t_r)S}, 0$$

4.2 Calculation of Optimal STSR

In this section, we present an optimization method to find an STSR $R(o_i)$ to minimize the expected update cost. Given the cost model presented above, the estimation on the active update probability $P_a(R(o_i))$ depends on the number of update records covered by $R(o_i)$. This implies that there are only $2^{|U_{NN}|}$ different values for the possible active update probability for $R(o_i)$. Each possible value is associated with a group of covered records. This motivates our optimization technique of modeling the record covering with a series of discrete events.

To find the optimized STSR, an initial STSR $R(o_i)$ is first created with minimal LR and minimal VR covering only l_i^t and v_i^t of o_i respectively. The optimization procedure executes iteratively. In each iteration, it tries to expand the STSR to cover one more update record from the remaining uncovered records in U_{NN}. If the estimated update cost does not further decrease after some iterations, the optimization procedure stops and returns the final STSR. The detailed algorithm for the optimization procedure is presented in Algorithm 4 in Appendix B.

In Fig.7, we illustrate an example of the optimization algorithm, using the data shown in Fig.6. The red square points are the location and velocity of the object o_i at time t_r. At the beginning of the algorithm, the STSR $R(o_i)$ is initialized with the minimum square covering the red squares in both spaces. Since the inclusion of any update record has the same reduction effect on $P_a(R(o_i))$, the optimal update record to cover next actually has the minimum increase on the passive update probability $P_p(R(o_i))$. By testing all update records, U_2 is selected according to the selection criterion. The STSR $R(o_i)$ grows in both spatial and velocity spaces to cover the update record U_2, as shown in Fig.7(a). In the second iteration, as shown in Fig.7(b), the update record U_1 is picked since the decrease on $P_a(R(o_i))$ is still larger than the increase on $P_p(R(o_i))$. The algorithm terminates after the second iteration, when there is no other expansion that can further reduce the estimated cost. Note that this algorithm works in a greedy manner. Hence, it does not guarantee convergence to the global optimum.

The retrieval of the similar update record set U_{NN} discovers all update records covering both the location and velocity of the current object. To efficiently support such a retrieval process, an index is built on the update records with respect to their location rectangle and velocity rectangle. Given the 4D index structure, U_{NN} is simply retrieved with the issuance of a point query at location l_i^t and velocity v_i^t. The computation of $P_p(R(o_i))$ takes constant time since the total volume $Vol(R(o_i))$ can be summed up quickly by the formula. In each iteration, all the remaining update records are

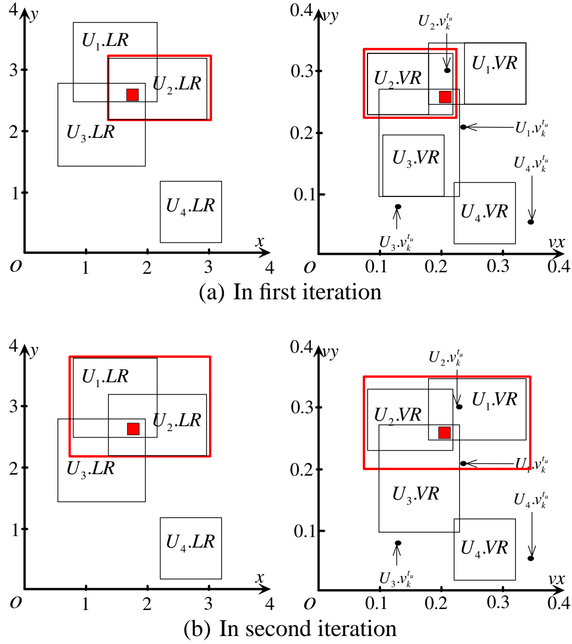


Figure 7: Example of optimization algorithm

tested in sequence. This leads to $O(m^2)$ complexity in the worst case, if m update records are retrieved from the 4D index structure.

4.3 Reducing Computation Cost

The STSR calculation algorithm runs in quadratic complexity in terms of the number of STSR update records. Thus, the computation cost on the STSRs can be very high if every single object update runs the construction method. To reduce computation cost, there are two simple methods, namely *Static STSR*, and *Global Dynamic STSR*. To distinguish from the basic strategy with independent computation of STSR for each object, we call the basic solution *Personal Dynamic STSR*.

With static STSR strategy, there is a group of fixed parameters $\{\Delta_l, \Delta_v, \Delta_t\}$. Δ_l and Δ_v are rectangles in the spatial and velocity space, covering the origins, respectively. Δ_t is a positive constant value that specifies the length between reference time and expiry time. For any updating moving object o_i with location l_i^t, velocity v_i^t and time t, the location rectangle LR for the STSR $R(o_i)$ for o_i is computed by moving Δ_l aligning l_i^t with the origin, i.e.,

$$\begin{aligned} LR.x^{\vdash} &= l_i^t.x + \Delta_l.x^{\vdash} \\ LR.x^{\dashv} &= l_i^t.x + \Delta_l.x^{\dashv} \\ LR.y^{\vdash} &= l_i^t.y + \Delta_l.y^{\vdash} \\ LR.y^{\dashv} &= l_i^t.y + \Delta_l.y^{\dashv} \end{aligned}$$

Similarly, the velocity rectangle VR in $R(o_i)$ is also constructed by expanding the velocity with margins in Δ_v on both dimensions. The expiry time of $R(o_i)$ is $t + \Delta_t$. This strategy is supposed to incur minimal computation cost, since the parameters are never updated after the specification at the beginning. As an example, if the parameter set is $\{\Delta_l = (-1, 1) \times (-2, 1), \Delta_v = (-0.2, 0.1) \times (-0.1, 0.3), \Delta_t = 5\}$, object o_i updates at time $t_r = 10$ with location $l_i^t = (10, 15)$ and velocity $v_i^t = (1, 2)$, the new STSR $R(o_i)$ is constructed with $LR = (9, 11) \times (13, 16)$, $VR = (0.8, 1.1) \times (1.9, 2.3)$ and $t_e = 15$.

With the strategy of global dynamic STSR, each global parameter set $\{\Delta_l, \Delta_v, \Delta_t\}$ is valid only in an interval on the time axis. There is one and only one valid global parameter set at any timestamp t. The object updates are handled with the global parameters valid at the update time, as is done with the static strategy. There is some computation cost incurred by this strategy to calculate a new parameter set when the previous global parameter set is expiring. The computation on parameter re-computation can be run offline when the system has free CPU cycles for use, not affecting the performance of the database system.

In *Global Dynamic STSR Strategy* a global STSR expansion plan is designed and updated from time to time, with evolution on the index structure. To make Algorithm 4 applicable on the search of optimal STSR expansion plan, we present some modifications to the original algorithm.

In the original algorithm, the STSR update records covering the updating location of the current object, U_{NN}, are first retrieved. On the global expansion optimization algorithm, however, there does not exist such similar update record set since the optimization is not location dependent. Thus, instead of finding a group of update records in the table, all records are utilized in the global parameter search by aligning all of the centers of STSRs in the update records to the origin. After the alignment operation, a virtual moving object is created with location at the origin and the velocity at the average speeds of the objects on both dimensions. The original algorithm is then applied on the virtual object and the aligned records. The final expansions on both spatial and velocity spaces are recorded and stored in the parameter set $\{\Delta_l, \Delta_v, \Delta_t\}$.

5. EXPERIMENTAL EVALUATION

We now report experimental results that evaluate effectiveness and efficiency of the STSR based updating protocol. We first studied the performance of the basic STSR protocol under different parameters and then compared it with another state-of-art update mechanism. Finally, we evaluated the performance of different STSR strategies on different indexes.

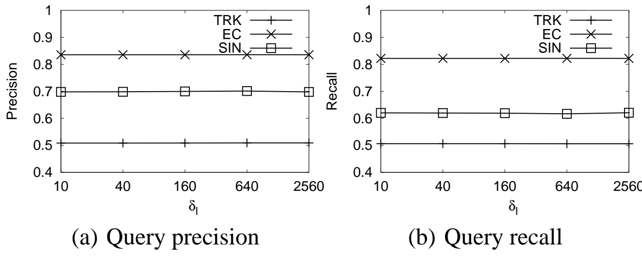
Experimental Settings: Three sources of real and *semi*-real datasets were used in our experiments:

[TRK] is a real dataset provided by R-tree Portal[2], which contains trajectories of 276 trucks moving in Athens metropolitan area (see Fig.16(a) in Appx.D.1). The trucks update at a rate of 30*sec*.

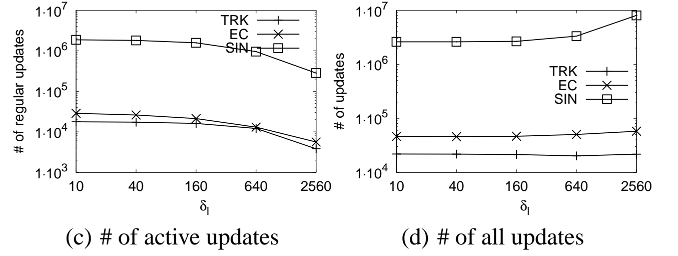
[EC] is another real dataset as a part of the e-Courier datasets[1]. e-Courier keeps track of the movement of all its couriers all over UK (see Fig.16(b) in Appx.D.1). The couriers report their locations (GPS records) every 10*sec*. We crawled e-Courier for one week and extracted 587 objects (i.e., trajectories) that moved nonstop for over 120ts.

[SIN] Due to the lack of large real moving object datasets, we used Brinkhoff generator[3] to generate a set of synthetic movements based on the real road map of Singapore (see Fig.16(c) in Appx.D.1). We generated SIN datasets of different sizes and used the one containing 100K objects by default.

In Appx.D.1, Tbl.2 lists the specifications of the three data sources above, including the data space, maximum object speed and the mapping from physical time to logical time (a timestamp). The query load of the experiments consists of a given number of predictive range queries. Each of these range queries is square-sized, with a preset side length. Since the datasets differ in data space size, we use *qlen* to represent the percentage of the side length of the query over the length of the entire data space. In the experiments, *qlen* is varied from 0.25% to 4% (e.g., 120m to 2058m for SIN datasets). Queries follow the same distribution as the objects. Specifically, the location of a randomly picked object is used as the center of the query. The average predictive time of queries varies

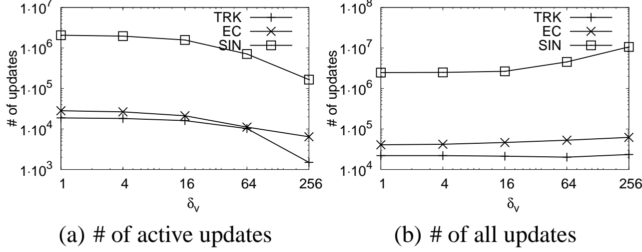


(a) Query precision (b) Query recall



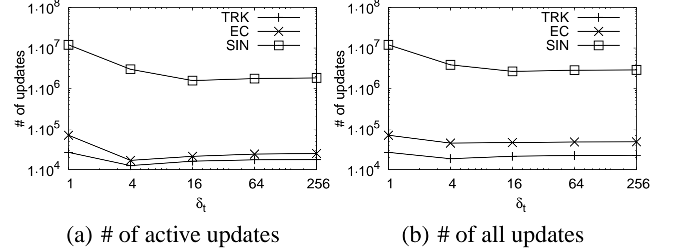
(c) # of active updates (d) # of all updates

Figure 8: Effect of δ_l on STSR performance



(a) # of active updates (b) # of all updates

Figure 9: Effect of δ_v on STSR performance



(a) # of active updates (b) # of all updates

Figure 10: Effect of Δ_t on STSR performance

from 1ts (current query) to 256ts. The query frequency $qfqy$ varies from 1 to 256, which means the corresponding workload contains 0.25 to 4 queries per timestamp. Tbl.3 in Appx.D.1 summarizes the parameters and their values used in the experiments, where default values for variable parameters are shown in bold. All the programs are implemented in C++ and run on a PC with 2.33GHz Intel Core2 Duo CPU, 2.25GB RAM and 200GB SATA disk.

Study of the basic STSR strategy We first studied the performance of the basic STSR strategy without regard of the underlying index. We investigated the effect of the three elements of an STSR: the spatial and velocity rectangles Δ_l and Δ_v; the temporal length Δ_t, i.e., the length of time between expiry time and reference time. A static global parameter set is used in each experiment in this section, where

$$\Delta_l.x^{\vdash} = \Delta_l.y^{\vdash} = -\delta_l$$
$$\Delta_l.y^{\dashv} = \Delta_l.y^{\dashv} = \delta_l$$
$$\Delta_v.x^{\vdash} = \Delta_v.y^{\vdash} = -\delta_l$$
$$\Delta_v.y^{\dashv} = \Delta_v.y^{\dashv} = \delta_l$$

The values of δ_l, δ_v, and Δ_t are listed in Tbl.3.

Fig.8-Fig.10 shows the effect of δ_l, δ_v and Δ_t on update workload and the quality of query result. It is worth noting that query precision and recall are not affected by the size of STSRs. During query processing, an object is added to or excluded from the results if its predicted region is fully contained or disjoint with the query region. Otherwise, a passive update is invoked. The query precision and recall are primarily decided by the predictability of data itself, i.e., predicting based on the motion of the object at query issuing time. As δ_l increases, meaning a larger initial spatial region, the number of active updates decreases at the expense of more passive updates, resulting in an increase in the total number of updates. As shown in Fig.9, δ_v has a similar effect on update times. With a larger δ_v, the predicted region expands faster. STSR is more likely to enclose the location of the object and fewer active updates are incurred. On the other hand, a larger predicted region has higher probability to intersect with the query region and more passive updates are required as a result. The temporal length of STSRs Δ_t affects the update performance differently. The number of active updates and the total number of updates both decrease with a longer temporal length Δ_t. When Δ_t is longer than 16ts, the number of updates do not change much with Δ_t. For a small Δ_t, the STSR expires quickly, resulting in a large number of active updates.

Motion Functions We next investigated the effect of different update policies on update and query performance. We compared the STSR update policy with the recursive function model proposed together with the STP-tree[13]. A client (object) keeps h historical records and derives a recursive motion function from the h records. A D dimensional polynomial function is used to approximate the recursive motion function. For an update, the polynomial function is sent to the server and the system can predict object location using the polynomial function. An active update is issued if the distance between the current location of the object and the location computed from the last reported polynomial function is larger than an error bound de in the next H timestamps. In our experiments, h, D, de and H are set to 8, 5, 160m and 30ts respectively. The query processing of the STP is similar to that of the STSR, i.e., the system asks for an update (passive) if it cannot determine if the object is inside the query region or not.

Fig.11- Fig. 14 shows the results on TRK and EC datasets while varying the query predictive time $qpdt$[1]. For both TRK and EC, the STP method results in a higher query precision, while the STSR policy has a higher query recall. The differences in precision and recall grow with query predictive time. On TRK, when $qpdt = 16$, the precision of STSR is about 10% less than that of STP, however, the recall is about 1.6 times of that of STP. On EC, when $qpdt = 64$, the difference in query precision is less than 0.05, while difference in query recall is as large as 0.5. Considering the update load, the number of active updates are less affected by the query predictive time for both methods and the STSR effectively reduces the number of active updates. The number of total updates increases with the query predictive time. Since the predicted region is larger with longer predictive time, the objects are more likely to issue passive updates. When $qpdt = 16$, the update times of STSR policy is 20% less than that of STP. When $qpdt = 64$, on EC, the update times of STSR is slightly higher than that of STP. As shown in Tbl.2, one timestamp of TRK(EC) corresponds to 30*sec*(10*sec*).

[1]We also tested on SIN datasets and investigated the effect of query side length, query frequency and etc. The results are omitted due to space limitation, which can be found on [6].

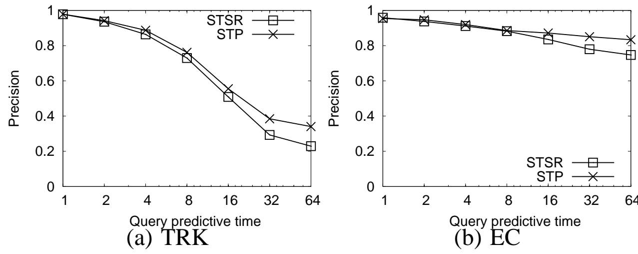


Figure 11: Effect of predictive time on precision

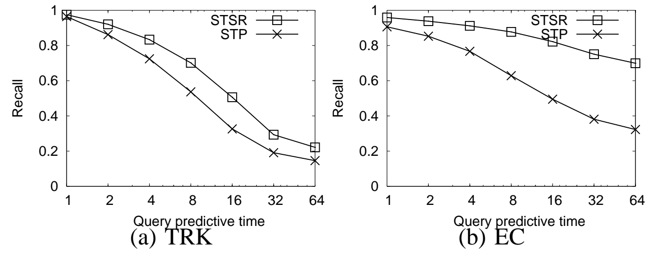


Figure 12: Effect of predictive time on # of active updates

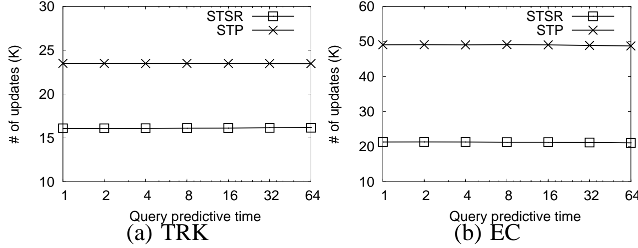


Figure 13: Effect of predictive time on # of active updates

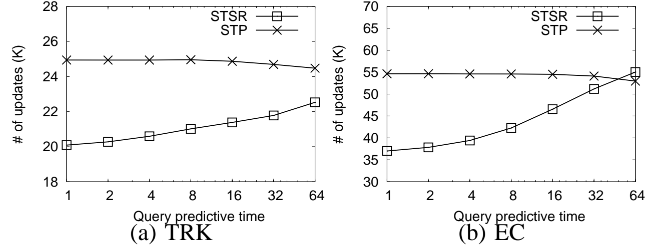


Figure 14: Effect of predictive time on # of total updates

Therefore, a predictive time of $qpdt = 64$ means to find the objects in specific region after $32mins(10mins)$. Based on the reason above, although our experiments show that STSR incurs higher updating costs than STP on EC when query predictive time is larger than 64, we argue that the prediction is meaningful only on a close future.

In summary, our STSR achieves good query precision and outperforms STP update method with regard to query recall. In addition, STSR can effectively reduce update workload when query predictive time is in a reasonable range.

6. CONCLUSION

In this paper, we propose a generic updating protocol for moving object databases that is independent of the underlying index structure. By utilizing the concept of spatio-temporal safe region (STSR), objects actively send motion updates including their locations and velocities to the database server only when the prediction error of their current movement is no longer bounded. To guarantee the accuracy of query prediction, the database server asks objects for their latest motion, if they are potential results of the query. To minimize the number of update messages between moving objects and database system, we present a cost model that analysis the approximate cost depending on the recent update records stored in the system. Based on the cost model, an effective optimization technique is designed to reduce the expected update cost. We carefully evaluate three different implementation strategies, including static STSR, dynamic global STSR and dynamic personal STSR. Experiments on the TPR-tree and the B^x-tree show that our proposed protocol significantly improves the accuracy of query results and reduces the number of update messages.

7. ACKNOWLEDGEMENT

The work was in part supported by Singapore MDA grant R-252-000-376-279.

8. REFERENCES

[1] eCourier. http://api.ecourier.co.uk/.

[2] R-tree Portal, http://www.rtreeportal.org/.

[3] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, 2002.

[4] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008.

[5] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento. St^2b-tree: a self-tunable spatio-temporal b^+-tree index for moving objects. In *SIGMOD*, pages 29–42, 2008.

[6] S. Chen, B. C. Ooi, and Z. Zhang. Capturing the motions with adaptive updating model in moving object database. http://www.comp.nus.edu.sg/ chensu/stsr-tr.pdf.

[7] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[8] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, pages 479–490, 2005.

[9] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B^+-Tree Based Indexing of Moving Objects. In *VLDB*, pages 768–779, 2004.

[10] D. Lin, C. S. Jensen, B. C. Ooi, and S. Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *MDM*, pages 59–66, 2005.

[11] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[12] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, 2000.

[13] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD*, pages 611–622, 2004.

[14] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, pages 790–801, 2003.

[15] Y. Tao, D. Papadias, J. Zhai, and Q. Li. Venn sampling: A novel prediction technique for moving objects. In *ICDE*, pages 680–691, 2005.

[16] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[17] W. Wu, W. Guo, and K.-L. Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *ICDE*, pages 1116–1125, 2007.

[18] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[19] M. L. Yiu, Y. Tao, and N. Mamoulis. The b^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDB J.*, 17(3):379–400, 2008.

[20] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.

[21] M. Zhang, S. Chen, C. S. Jensen, B. C. Ooi, and Z. Zhang. Effectively indexing uncertain moving objects for predictive queries. In *VLDB*, 2009.

[22] Z. Zhang, R. Cheng, D. Papadias, and A. K. H. Tung. Minimizing communication cost for continous skyline maintenance. In *SIGMOD*, 2009.

[23] Z. Zhang, Y. Yang, A. K. H. Tung, and D. Papadias. Continuous k-means monitoring over moving objects. *IEEE Trans. Knowl. Data Eng.*, 20(9):1205–1216, 2008.

APPENDIX

A. RELATED WORK

A.1 Moving Object Indexing

Generally speaking, the index structures for moving objects can be divided into two categories, namely data-partition based structures and space-partition based structures. The TPR-tree [12] and TPR*-tree [14] are typical examples of data-partition based index structures. Given the locations and velocities of moving objects at their respective update times, the objects are inserted into a multi-dimensional index after transformation into some standard reference time. On each intermediate node in the tree structure, the maximal bounding rectangle (MBR) is used to bound the locations of the objects in the subtree at the reference time. The maximal and minimal speeds of the objects along both dimensions are also recorded. Given a range query at the querying time, the query processing algorithm follows the traditional pruning strategies in the R-tree. In particular, when visiting an intermediate node in the index, the MBR expands with respect to its maximal and minimal speeds. If the expanded MBR does not intersect with the query range, this node can be safely pruned. Otherwise, its child nodes are pushed into a queue for further examination. When the querying time is faraway from the reference time of the index, the MBRs have to be expanded accordingly and the resultant MBRs will have a much higher likelihood of intersecting with the query, causing many paths to be traversed. This is the major drawback of data-partition based index structures.

Among space-partition based indexes, the B^x-tree and the B^{dual}-tree are the two representative structures. In the B^x-tree, spatial space is split into small cells, and the cells are mapped to a one-dimensional space with some space filling curve, such as Z-curve or Hilbert curve. To process a range query, the B^x-tree first transforms the query range into a sequence of cells in the space. These cells are used as queries, and during tree traversal, the query cells are expanded since the B$^+$-tree does not make use of any MBRs. There are two logical sub-trees rolling with time, each of which is responsible for the updates happening in an interval time T, with T as the maximal update interval. Unlike the B^x-tree, B^{dual} partitions both spatial and velocity spaces into cells, with a similar index structure built on the spatio-temporal cells. Some extensions have been further developed to enhance the effectiveness and efficiency of these structures. In [10], a forest of B^x-trees is constructed to allow queries on both predicted movements and historical movements. In [5], clusters are dynamically identified in each phase, and different granularity of cells is used improve query efficiency. The index is auto-tuned dynamically based on the object movement. In [21], we show that uncertain models on the moving objects can be easily incorporated into the B^x-tree, returning more meaningful results on predictive queries. Besides predictive queries for range and nearest neighbor search, some research studies are devoted to other queries, e.g. range aggregation [15].

A.2 Continuous Query Optimization

Continuous query processing is also one of the hot topics in the database community. Different from predictive queries, continuous query tries to keep accurate results on range query and nearest neighbor query on the current timestamp, with minimal communication and update costs.

In [16], for example, Wolfson et al. proposed a general framework, providing a mechanism to render approximate trajectories for continuous query processing. In [11], a scalable hash-based framework is proposed for k-Nearest Neighbor (k-NN) and range queries monitoring on moving objects, with shared execution mechanism on incremental evaluations of the queries. In [8], a generic framework is formulated for an energy-efficient monitoring scheme on moving objects for the range query and the k-NN query, with safe regions constructed for each object of all concurrent queries monitored on the server side. In [20] and [18], grid-based techniques are exploited to reduce the number of messages and enhance quick processing of k-NN queries on moving objects. While the problems consider only queries on static location, more recent works study the query processing problem when queries are moving around, e.g. [17, 7]

Besides work that address traditional range queries and k-NN queries, some extensions have been proposed to support more complicated queries on moving objects. In [23], a safe-region based method is proposed to monitor k-means clustering, utilizing some efficient lower bound computation technique on the local optimum of k-means clustering. In [22], cost models and optimization techniques are deployed to evaluate and maintain skyline queries with minimal communication costs between objects and central server.

B. ALGORITHMS

In this appendix, we show the detailed pseudocode for algorithms referred in the paper.

Algorithm 1 shows the consistency verification procedure as introduced in Sec.2.

Algorithm 1 Consistency Verification (timestamp t, current location l_i^t, current velocity v_i^t, current STSR $R(o_i) = (LR, VR, t_r, t_e)$)

1: **if** $t > t_e$ **then**
2: Report to the server and update with new STSR
3: Compute the predicted region P_i^t with respect to $R(o_i)$
4: **if** l_i^t is out of P_i^t **then**
5: Return **FALSE**
6: Compute the predicted region $P_i^{t_e}$ w.r.t. $R(o_i)$
7: Compute predicted location $pl_i^{t_e}$ w.r.t. l_i^t and v_i^t
8: **if** $pl_i^{t_e}$ is out of $P_i^{t_e}$ **then**
9: Return **FALSE**
10: **if** $t < t_r$ **then**
11: Compute the predicted location $pl_i^{t_r}$ w.r.t. l_i^t and v_i^t
12: **if** $pl_i^{t_r}$ is out of LR **then**
13: Return **FALSE**
14: Return **TRUE**

Algorithm 2 outlines the processing of an object update in Sec.3.

Algorithm 2 Object Update (object o_i, current location $l_i^{t_r}$), current velocity $v_i^{t_r}$, reference time t_r)

1: Calculate a new STSR $R(o_i) = (LR, VR, t_r, t_e)$ depending on $l_i^{t_r}$ and $v_i^{t_r}$
2: Send $R(o_i)$ to o_i
3: Renew the record of o_i in the database with $R(o_i)$, $l_i^{t_r}$ and $v_i^{t_r}$

Algorithm 3 shows the processing of predictive range query as presented in Sec.3.

Algorithm 4 summarizes the optimization method for constructing a new STSR for an object (Sec.4).

Algorithm 3 Range Query Processing (Query range QR, query time t_q)

1: Find the object set $O' \subseteq O$ that the predicted region $P_i^{t_q}$ overlaps with QR for any $o_i \in O'$
2: **for** each $o_i \in O'$ **do**
3: **if** $P_i^{t_q}$ is totally covered by QR **then**
4: Include o_i in the query result
5: **else**
6: Send a probe request to o_i for current location and velocity
7: Compute the new $pl_i^{t_q}$ with new location and velocity
8: **if** $pl_i^{t_q} \in RQ$ **then**
9: Include o_i in the query result
10: Return the complete query result

Algorithm 4 STSR Optimization (current location l_i^t, current velocity v_i^t, expiry time t_e)

1: Search all update records covering l_i^t and v_i^t and store them in U_{NN}.
2: Initialize $R(o_i)$ with $LR = \{l_i^t\}$, $VR = \{v_i^t\}$ and t_e
3: Initialize covered update record set $CR = \emptyset$
4: Initialize the cost $Cost(R(o_i)) = \infty$
5: **while** $Cost(R(o_i))$ does not converge **do**
6: Set optimal expansion record U^* as NULL
7: Set optimal expanded STSR R^* as NULL
8: **for** each update record U_j in U_{NN} **do**
9: Construct R' by expanding $R(o_i)$ to cover U_j
10: Estimate $P_a(R')$ and $P_p(R')$
11: **if** the cost of R' is smaller than R^* **then**
12: Replace R^* with R'
13: Replace U^* with U_j
14: **if** U_j is not NULL **then**
15: Replace $R(o_i)$ with R^*
16: Move U^* from U_{NN} to CS
17: Return $R(o_i)$

C. PROTOCOL IMPLEMENTATION WITH EXISTING INDEX STRUCTURES

Our proposed protocol can be seamlessly implemented with almost all existing indexing structures on predictive queries for moving objects. In this section, we focus on incorporating our proposed protocol into two popular data structures: the TPR-tree (and its variants) and the B^x-tree. While these index structures feature in different aspects, e.g., query processing, etc., we discuss primarily the storage issue of STSRs in these structures.

C.1 TPR-tree and its variants

The STSR optimization algorithm (Algorithm 4) can be embedded directly into the TPR-tree. In the TPR-tree and its variants such as the TPR*-tree, spatio-temporal bounding rectangles are used to summarize the possible locations and velocities of a group of moving objects. To replace exact location and velocity with STSR as the underlying object representation in the TPR-tree, we only need to make some minor modifications on the leaf nodes of the TPR-tree. Such changes do not affect the intermediate nodes in the TPR-tree, since STSRs only enlarge the spatial temporal bounding boxes of these intermediate nodes. This enables us to equip general STSRs on TPR-tree, without any constraint on the location rectangles and the velocity rectangles.

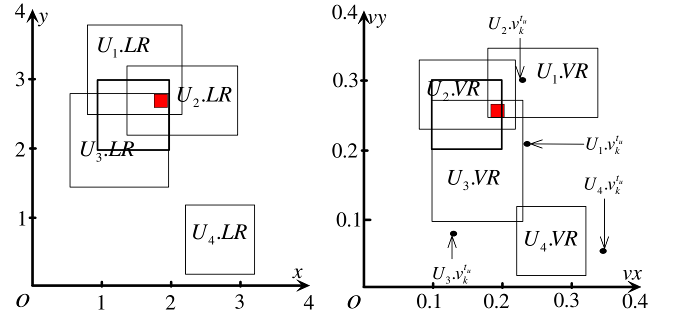


Figure 15: Initial location and velocity rectangle for B^{bdual}-tree

Since the overall structure of the TPR-tree remains the same, the existing update algorithms on the TPR-tree can be reused without any modification. Similarly, the querying algorithm can be left unchanged since we can regard every moving object as the traditional spatial temporal bounding box. The concurrency control mechanism, i.e. the RLink-tree, commonly used by the R-tree indexes remains applicable.

C.2 B^x-tree

To extend the optimization technique from the TPR-tree to other index structures, such as the B^{dual}-tree and the B^x-tree, recall the difference between the TPR-tree and the other two index structures on storage. In particular, the B^x-tree discretizes the spatial space and the B^{dual}-tree discretizes both spatial and velocity space.

To facilitate the extension, the only modification of the algorithm is on the initialization of the STSR. In particular, the STSR at the beginning is initialized by expanding the location and velocity to the minimal cells containing them. In Fig.15, we present the initial STSR for the same moving object update in Fig.7. If the widths of the cells in the spatial space and the velocity space are 1 and 0.1 respectively, the new STSR before the first iteration in Algorithm 4 is constructed with $LR = (1, 2) \times (2, 3)$ and $VR = (0.1, 0.2) \times (0.2, 0.3)$. The subsequent expansion iterations follow exactly the same implementation of the optimization algorithm for the TPR-tree.

In the B^x-tree, spatial space is divided into small cells of equal width on both dimensions. In the original B^x-tree, the coordinates of the object are transformed into the ID of the cell containing it. This implies that the location rectangle in STSRs stored in the B^x-tree must also be discretized before insertion. There are two possible solutions to support our protocol with the B^x-tree. The first option is to allow STSRs to occupy a few spatial cells. With this option, multiple copies of each STSR may be stored in different leaf nodes in the tree, providing more flexible spatial constraints but incurring extra processing costs on queries. The second option requires the location rectangle for every STSR to cover exactly one cell in the partitioned spatial space. This leads to some implementation on the B^x-tree, on which every moving object resides only in one leaf node. However, it sacrifices some of the tuning ability on the location rectangle if this strategy is adopted. In the empirical studies, we employ the second implementation. Given a moving object waiting for a new STSR, the location rectangle is fixed depending on the space partition of the B^x-tree.

With such storage implementation, both the updating and querying algorithms on the B^x-tree are simply adopted without any modification. Other optimizations, such as object grouping and cell size tuning [5], can also be applied directly; the Blink-tree concurrency control that is used by the B^x-tree remains applicable for handling

concurrency operations.

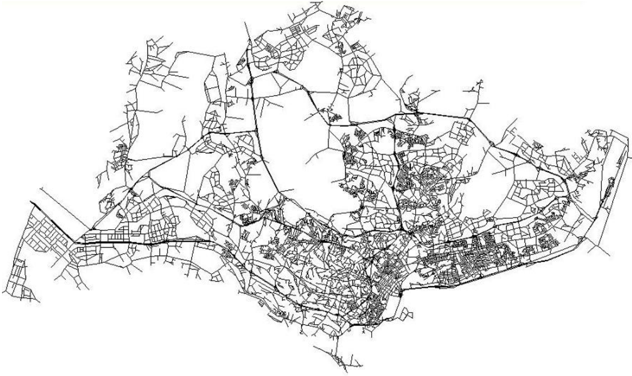
D. EXPERIMENTS

D.1 Experimental Settings

Fig.16 shows the maps of road networks for each datasets.

(a) TRK: Athens metropolitan

(b) EC: UK



(c) SIN: Singapore

Figure 16: Maps of various data sources

Tbl.2 shows some specifications on different data sources.

Tbl.3 lists parameters and there values used in the experiments in Sec.5, where the default values are shown in bold.

D.2 More experimental results

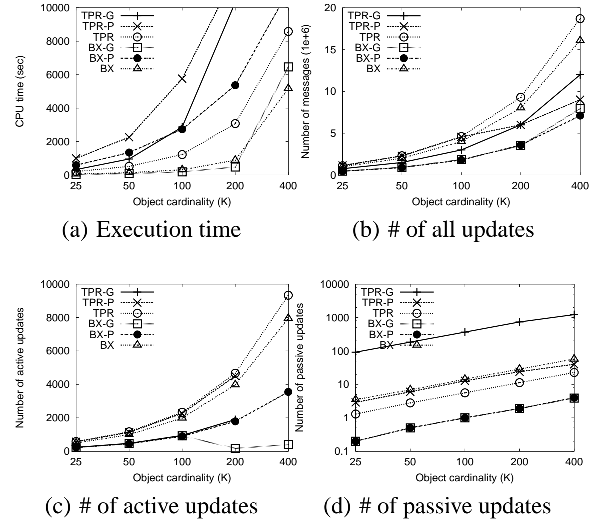
We now proceed to study the performance of different STSR strategies as introduced in Sec.3, including the static STSR, the global dynamic STSR and the personal dynamic STSR. We implement all the three STSR strategies on top of both the TPR-tree and the B^x-tree as explained in Appx.C. In the remaining part of this section, 'BX'/'TPR' denotes using the B^x-tree/TPR-tree with static STSR; 'BX-G'/'TPR-G' denotes using the B^x-tree/TPR-tree with global dynamic STSR; 'BX-P'/'TPR-P' denotes using the B^x-tree/TPR-tree with personal dynamic STSR.

Table 2: Specifics of data sources

Data source	TRK	EC	SIN
Space (long side)	47255.6m	287409.0m	51455.0m
Maximum speed	33.5m/s	54.3m/s	325m/s
Unit of timestamp	30sec	10sec	1sec

Table 3: Experimental parameters and values

Parameter	Setting
Datasets	TRK, EC, **SIN**
Time duration	120ts
Data size	25K, 50K, **100K**, 200K, 400K
Query side length $qlen$	0.25%, 0.5%, **1%**, 2%, 4%
Query predictive time $qpdt$	1ts, 4ts, **16s**, 64ts, 256ts
Query frequency $qfqy$	1, 4, **16**, 64, 256
Δ_t	1ts, 2ts, 4ts, 8ts, **16ts**, 64ts, 256ts
δ_l	10m, 40m, **160**, 640m, 2560m
δ_v	1m/ts, 4m/ts, **16m/ts**, 64m/ts, 256m/ts



(a) Execution time (b) # of all updates

(c) # of active updates (d) # of passive updates

Figure 17: Effect of data size

We first examined the scalability of all update strategies by varying the object cardinality from 25K to 400K. Fig.17 illustrates the total processing time and the number of updates. The total processing includes all computations on queries, STSR computation and updates of the moving objects, in 120 consecutive timestamps.

Comparing to the static and personal dynamic strategies, global dynamic strategy largely decreases the amount of active updates, while adds a number of passive updates. The personal dynamic strategy, on the other hand, reduces the number of passive updates at the expense of more active updates. This is because that the with global dynamic strategy, the size of STSRs is much larger than those generated by personal dynamic strategy.

Second, we examined the effect of various query parameters on the performance of different STSR update strategies. From Fig.17, we have already seen that the 'TPR-G' requires the highest exe-

cution time and is the least scalable in terms of object cardinality, while the 'TPR' always incurs the highest update times. For clear illustration, we leave 'TPR-G' and 'TPR' out. The results of these two methods are omitted in the following figures and analysis.

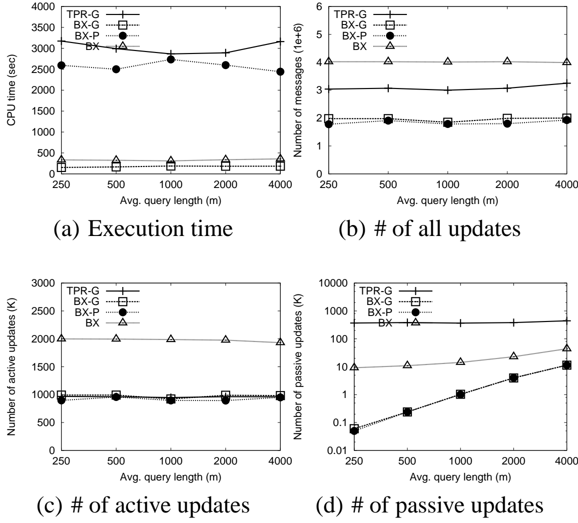


Figure 18: Effect of query side length $qlen$

First, Fig.18 shows the performance of STSR strategies with respect to different query size. Specifically, the query side length varies from 250m to 4000m. In general, the B^x-tree with global dynamic STSR is the best among all considering both total processing time and total number of updates. The B^x-tree with static STSR, although performs good in terms of total proceeding time, sends out the largest total number of updates. For all methods except 'TPR-G', the number of passive updates increases with the query size, since the STSRs of more objects intersects with the query region. For the 'TPR-G', the STSR is relatively large, and thus the performance is less affected by the query size.

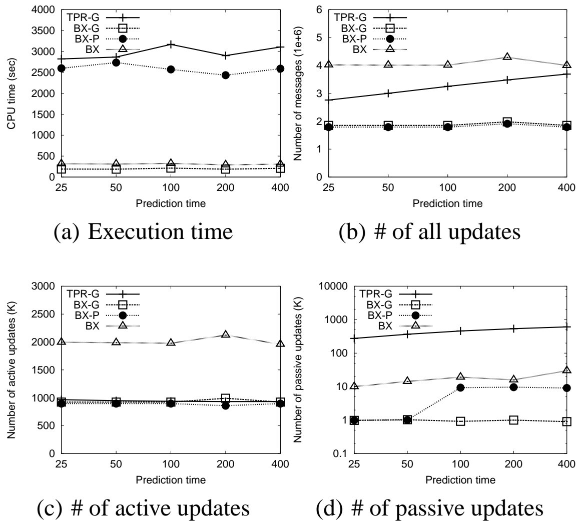


Figure 19: Effect of query predictive time $qpdt$

Fig.19 shows the effect of query prediction time on different STSR strategies. We vary the query prediction time from 0ts (current query) to 120ts. As the prediction time changes, the total processing time shows the similar trends as those of other parameters, i.e., the 'BX-G' and 'BX' both run much faster than the 'GX-P' and 'TPR-G'. As 'BX-G' tunes the parameters for the STSR periodically, it has the best performance in terms of the total processing time. 'BX-P' also minimizes the total number of updates, however, at the expense of longer processing time (for computing the parameters for objects individually). In general, as the predication time increases, more passive updates are incurred.

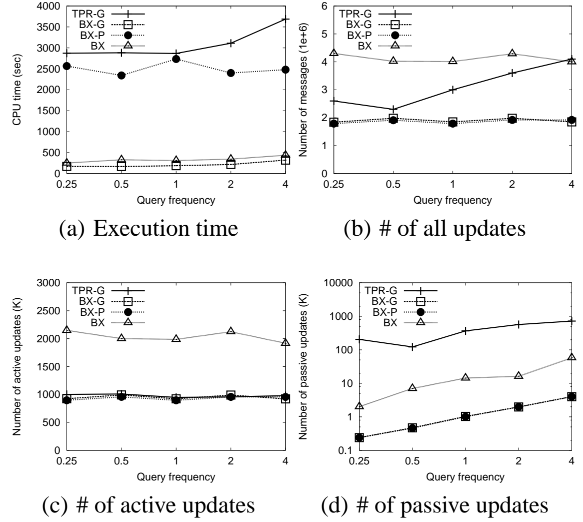


Figure 20: Effect of number of queries per timestamp $pfqy$

Finally, we study the effect of query frequency on the performance Fig.20 shows the effect of query prediction time. For the STSR strategies, with more frequent queries, the number of passive updates increases a lot while the number of active updates decreases slightly. It is worth noticing that the total processing time is not much influenced by the query frequency, although the total number of queries increases a lot (by 16 times). The time for computing STSR dominates the processing time of the whole index.