Geoinformatica
DOI 10.1007/s10707-007-0029-9

Efficient Maintenance of Continuous Queries for Trajectories

Hui Ding · Goce Trajcevski · Peter Scheuermann

Received: 23 October 2006 / Revised: 10 March 2007
Accepted: 3 April 2007
© Springer Science + Business Media, LLC 2007

Abstract We address the problem of optimizing the maintenance of continuous queries in Moving Objects Databases, when a set of pending continuous queries need to be reevaluated as a result of bulk updates to the trajectories of moving objects. Such bulk updates may happen when traffic abnormalities, e.g., accidents or road works, affect a subset of trajectories in the corresponding regions, throughout the duration of these abnormalities. The updates to the trajectories may in turn affect the correctness of the answer sets for the pending continuous queries in much larger geographic areas. We present a comprehensive set of techniques, both static and dynamic, for improving the performance of reevaluating the continuous queries in response to the bulk updates. The static techniques correspond to specifying the values for the various semantic dimensions of trigger execution. The dynamic techniques include an in-memory shared reevaluation algorithm, extending query indexing to queries described by trajectories and query reevaluation ordering based on space-filling curves. We have completely implemented our system prototype on top of an existing Object-Relational Database Management System, Oracle 9i, and conducted extensive experimental evaluations using realistic data sets to demonstrate the validity of our approach.

Keywords moving object database · continuous queries · triggers · context-aware reevaluation

Research supported by the Northrop Grumman Corp., contract: P.O.8200082518.

Research supported partially by NSF grant IIS-0325144.

H. Ding (✉) · G. Trajcevski · P. Scheuermann
Northwestern University, Evanston, IL 60208, USA
e-mail: hdi117@eecs.northwestern.edu

G. Trajcevski
e-mail: goce@eecs.northwestern.edu

P. Scheuermann
e-mail: peters@eecs.northwestern.edu

1 Introduction

Recent advances in wireless communication, miniaturization of computing devices and Global Positioning Systems (GPS) have resulted in a large number of novel application domains in which an important aspect is the mobility of the end users. Location-Based Services (LBS) integrate a mobile device's location or position with other information, so as to provide added value to a user, and their crucial functionalities depend on the management of the location-in-time information of the participants [30]. The problems related to efficient storage and query processing of the moving objects have recently spurred a tremendous amount of research in the emerging field of Moving Objects Databases (MOD) [14], [16], [19], [22], [35], [36].

One of the peculiarities of the MOD settings is that, due to the constantly evolving nature of the users' location-in-time information, many of the queries of interests are *continuous* [35], i.e., their answers change over time and, consequently, may need to be reevaluated. In order to efficiently maintain the correctness of the answer sets of such queries, it is desirable to develop specialized data storage/access mechanisms and processing algorithms [22], [24], [36]. The main distinguishing properties of both the indexing structures and the algorithms are a consequence of the model adopted for representing the spatio-temporal nature of the moving objects [12], [22]. When it comes to representing and reasoning about the future states of the mobile entities, relative to the ever-evolving *now* time, three models are most frequently used in the existing MOD literature:

1. Stream-like updates of (*location*, *time*) tuples whose values are obtained, for example, via on-board GPS devices, and are periodically reported to the MOD by the moving objects (model *I* in Fig. 1). Due to the high frequency of updates, intelligent methodologies are necessary to avoid constant reevaluation of the pending continuous queries while still ensuring the correctness of their answers [24].
2. A sequence of (*location*, *time*, *velocity*) updates which are sent to the MOD only when an object deviates from its expected motion according to the previous update (model *II* in Fig. 1) [35]. One objective of this model is to reduce the

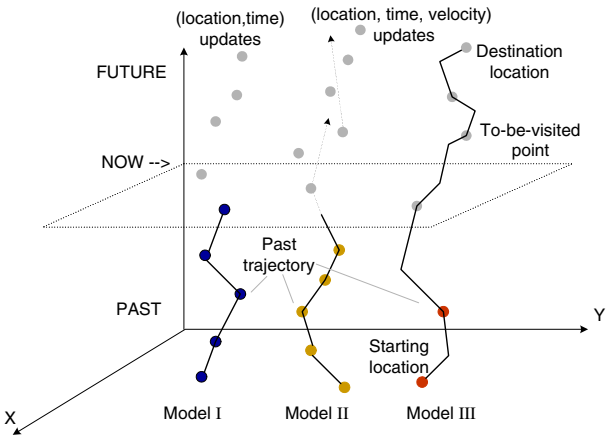


Fig. 1 Modelling the motion of moving objects

communication overhead between the moving objects and the server, while ensuring a bounded error on the MOD representation of the objects' motion [14], [19], [41].

3. A *trajectory*, which is a sequence of 3D points (x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n), $(t_1 < t_2 < \ldots < t_n)$, is used to represent the entire future motion plan of an object (model *III* in Fig. 1). Between any two consecutive points, the object is assumed to move along a straight line with a constant speed. This is a realistic model for many entities in practical settings when representing future motion plans, e.g., public transportation vehicles, police patrol cars, delivery trucks and individuals travelling back-and-forth between home and work. Furthermore, many individuals rely on certain online trip planning services, such as Mapquest, Yahoo! Maps and Google Maps,[1] to provide driving directions(essentially future trajectories). It has been reported by Forbes.com [3] that the number of distinct users that request future trajectory generation monthly reaches 46.4 million for Mapquest, 20 million for Yahoo! Maps and 19.1 million for Google Maps. A distinguishing feature of the trajectory model is that it enables the MOD to answer predictive queries pertaining to the locations of the moving objects in future time intervals, which can be useful for various planning purposes.

Note that when it comes to the past location-in-time information, all three models converge in typical MOD settings, in the sense that the historical data is often stored as trajectories.

1.1 Motivation

In this article we assume that the future motion plans of the moving objects are represented as trajectories and consider a problem that is inherent to continuous query management in such settings, which has not been addressed formally before. Namely, when constructing a trajectory for the future movement of a given object, the MOD relies on certain speed-distribution information regarding the individual road segments. The MOD constantly manages a large number of such trajectories and answers predictive continuous queries like, for example: Q_1*: retrieve all the moving objects that will be within the Northwestern University campus (sometimes/always) between 4 pm and 5 pm*. When an unexpected traffic abnormality happens, e.g., an accident or a road work, it may affect the future portions of the trajectories passing through the abnormality region throughout its duration. This, in turn, affects the correctness of the pending continuous queries in a much larger geographic area. The key problem is how to efficiently reevaluate such a set of pending queries so that their answer sets can be brought up to date with the new traffic conditions.

To better illustrate the main aspects of the problem addressed in this article, let us consider the scenario in Fig. 2. It consists of six trajectories and two static regions. In particular, trajectories $Tr_1, \ldots, Tr_5$ depicted by solid polylines represent the expected motion plans of five moving objects. Q_{R_1} and Q_{R_3} are two static range queries represented as prisms in the three dimensional space (two spatial dimensions plus the temporal dimension). On the other hand, Q_{R_2} is a dynamic range

[1]Mapquest is an online map service provided by Time Warner through its AOL service, Yahoo! Maps is a similar service provided by Yahoo! Inc. and Google Maps by Google Inc.

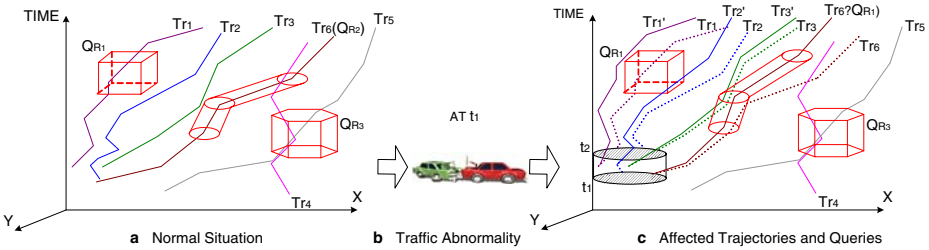


Fig. 2 Trajectories, updates and continuous queries

query defined with respect to trajectory Tr_6, its query volume is represented by the two sheared cylinders over the time interval of interest (a formal definition of the continuous queries will be presented in Section 2).

In this example, we assume that the preliminary query answers have been computed and the queries are being monitored until some future time of interest. Suppose an accident occurs at some time t_1 on a particular road segment after the initial answer sets of the queries have been computed (cf. Fig. 2b), which affects the traffic patterns in a neighboring region (the base of the shaded cylinder in Fig. 2c), and its impact on the traffic persists until time t_2 (the top of the shaded cylinder in Fig. 2c). The objects moving through the abnormality zone during this time period will have to be slowed down from their original expected speed used as a parameter when constructing their trajectories. Hence, the future portions of the affected trajectories need to be updated, which is the case for Tr_1, Tr_2, Tr_3 and Tr_6. As illustrated in Fig. 2c, the dashed lines depict the original trajectories while the solid lines depict the updated trajectories. Now it becomes obvious that the updates affect the answers to the pending queries. First, observe that Tr_1 is no longer an answer to Q_{R_1}, but Tr_2 becomes an answer to Q_{R_1}. In addition, since trajectory Tr_6 for query Q_{R_2} is delayed by the accident, the whole volume of interest (the two sheared cylinders) changes. Hence, Q_{R_2} also needs to be reevaluated, upon which it turns out that Tr_4 is not its answer any more, but Tr_3 becomes an answer. Another important observation is that Q_{R_3} does not need to be reevaluated at all, since the accident has no impact on its answer set. Clearly, it is necessary to reevaluate only the affected pending queries upon updating the trajectories. The key issue of efficient maintenance of these queries is minimizing the cost of various overheads, e.g., disk I/Os, context-switching among processes etc.

1.2 Main contributions

One may observe that the aforementioned query maintenance in MOD exhibits a reactive pattern that can be accommodated by the *event-condition-action* (ECA) rules of active database systems [26], [40], which has been well-studied over the last decade and implemented in existing commercial Object-Relational Database Management Systems (ORDBMS). However, a MOD-like functionality that couples the reactive behavior with the spatio-temporal data management is still missing. One of the main contributions of this work stems from the fact that we demonstrate that a MOD system can be integrated into a traditional ORDBMS, where many benefits of mature and reliable technologies are readily available.

In this article we present an integrated approach towards the optimization of continuous query reevaluation in MOD that covers two popular categories of queries: range queries and k-Nearest Neighbor (kNN) queries, and we create a seamless framework which facilitates scalable execution when a set of mixed continuous queries are posed to the system. Upon a detection of an abnormality, we reduce the unnecessary database table accesses and computations for each type of the query and, moreover, we share the spatio-temporal context information among the queries to increase the overall efficiency of the reevaluation. As we will demonstrate, further improvements can be achieved when the process of updating the affected trajectories is intelligently intermixed with that of updating the answers to the pending continuous queries. For this purpose, we propose a set of specifications for the triggers that perform the desired reactive behavior along with techniques for ordering their execution. Our main goal is to reduce the processing time spent from the moment a particular abnormality is presented to the MOD, up to the point when all the pending continuous queries' answer sets are brought up-to-date. In summary, the contributions of this article are as follows:

1. We provide a framework for efficiently maintaining continuous queries on future trajectories when a subset of them are subject to bulk updates.
2. We identify three distinct phases required for the complete reevaluation of the pending queries and propose appropriate techniques that can improve the reevaluation performance in each phase, by intelligently combining the updates to the trajectories with the updates to the query answers. In particular, we: (a) propose an in-memory shared reevaluation scheme that interleaves mapping on the spatial dimensions with a sweep-plane approach on the temporal dimension to avoid unnecessary computation, (b) adapt the query indexing technique to the trajectory data to limit the search space and (c) utilize ordering based on space-filling curves to improve the I/O efficiency.
3. As a proof of concept, we have fully implemented our system on top of an existing commercial ORDBMS—Oracle 9i [5].
4. We have conducted an extensive set of experiments to validate our approach and obtained quantitative analysis of the benefits of the proposed optimization techniques.

The rest of this article is organized as follows: Section 2 provides an overview of the problem and reviews the preliminary background. Section 3 presents the architecture of our system prototype and explains the essential features of the reevaluation procedures for continuous queries. Section 4 and section 5 elaborate on our optimization techniques and their impact on query reevaluation. Section 6 discusses the experimental evaluation results and Section 7 positions our work with respect to the related literature. Section 8 concludes the article and outlines directions for future work.

2 Preliminaries

In this section we formally introduce the trajectory data model and the semantics of the continuous queries considered in this work. We also examine some of the semantic dimensions of the triggers in active database systems.

2.1 Trajectory model for moving objects

As mentioned in Section 1, there are three frequently used models for representing the objects' future motion plans: a sequence of (*location*, *time*) updates, a sequence of (*location*, *time*, *velocity*) updates and a full trajectory. In this article, we adopt the trajectory model and handle the problems for query maintenance that are specific under this model.

A *trajectory* [39], is a continuous piece-wise linear function $f : T \rightarrow \mathbb{R}^2$ from the temporal dimension to the two dimensional Euclidean space, connecting a sequence of points (x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n) $(t_1 < t_2 < ... < t_n)$. For a given trajectory Tr, its projection on the X-Y plane is called the *route* of Tr. This representation entails that the object is at position (x_i, y_i) at time t_i, and that during each time interval (t_i, t_{i+1}), the object moves along a straight line from (x_i, y_i) to (x_{i+1}, y_{i+1}) and with a constant speed. The *expected location* of the object at any time $t \in (t_i, t_{i+1})$ $(1 \leq i < n)$ is obtained by a linear interpolation between (x_i, y_i) and (x_{i+1}, y_{i+1}). Throughout the text, we will also refer to the portion of a given trajectory between two consecutive points as a *line segment*.

Relative to now, a trajectory can represent both the past and the future motion of objects. In this article, we are only interested in trajectories that pertain to the future. Each trajectory corresponds to the *motion plan* of a moving object and is constructed as follows. We assume that before travelling, each moving object submits to MOD its *start location*, *start time*, *destination location* and, optionally, a sequence of other to-be-visited points. Besides these information, the MOD also utilizes two additional sources [39]:

- Information available from the electronic maps, representing the road segments and intersections as a connected graph. Such maps are available from various sources like, for example, the Geographic Data Technology Co. [4];
- Knowledge about the time-varying distribution of the traffic patterns for each road segment during a given time of the day.

With these data, the MOD server can apply an A*-like [10] variant of the time-aware Dijkstra's algorithm to generate the shortest path for the moving object where the cost of an edge in a graph depends on the start time to travel along that edge. Observe that the shortest path can be defined in terms of travel-time or travel-distance. When it comes to minimizing the total travel time, the time-dependency is due to the fact that road conditions may vary during different times of a day. For each straight line segment of the constructed trajectory, the object's expected arrival times at its end points are computed. This pre-computed trajectory is then sent back to the moving object as travelling instructions and is used by the MOD for generating the initial answer sets of the pending queries [39].

2.2 Continuous queries for trajectories

A continuous query in LBS is logically associated with two distinct categories of entities:

- *Querying object*, which is the object that defines the semantics of the query and is denoted by Q_i. Note that a querying object may be different from the entity that actually submits the query, e.g., a web-based user or another moving object,

which determines where and how the query answer set is to be transmitted. However, we do not consider this aspect of the problem in this article.
– *Candidate objects*, which include all the objects that are relevant to the particular query reevaluation and are evaluated against the query predicates. All the candidate objects that satisfy the query predicate of a given query Q_i constitute the answer set to the query, denoted as A_Q_i.

Each querying object and candidate object has its future motion described by a trajectory. The trajectory of a querying object Q_i is simply called a *querying trajectory*. The trajectory of a candidate object is called a *candidate trajectory*. An object can be either static or dynamic (mobile). In case the object is static, its location is a static point which implies a trivial trajectory consisting simply of a vertical line. In the following sections, when no ambiguity arises, we may use the terms querying object and query trajectory interchangeably and denote both of them as Q_i. Similarly, we may interchange the terms candidate object and candidate trajectory and refer to them as Tr_i.

The two most common types of continuous queries in MOD that we consider are *range queries* and *k-nearest neighbor queries*:

– A range query Q_R retrieves all objects that are within a specified region with respect to the querying object, during a given time interval of interest. The starting and ending time stamps of interest of the query are defined by t_{si} and t_{ei}, respectively. A range query can be further classified as either *static* or *dynamic*. A static range query is specified with a fixed spatial region of interest r_R. For example, query Q_1 in Section 1.1 represents a static range query where the region r_R for this query is the geographic extent of the campus, t_{si} equals 4 pm and t_{ei} equals 5 pm. On the other hand, if the querying object is dynamic, then its motion is described by a non-trivial trajectory Tr_R. For example, a police patrol car may ask for other patrol cars within 1 mile of its trajectory in the next hour, when there are suspicious criminal activities. In real applications, a user may be interested in whether the objects are *sometimes* or *always* inside the region r_R [39]. Without loss of generality, in the rest of this article we assume that the *sometimes* temporal semantics is used for generating the query answer sets.
– A k-Nearest Neighbor query Q_{kNN} retrieves the k closest objects to a given querying object, between the time stamps t_{si} and t_{ei}. As an example of this type of query, a tourist who is visiting attractions in downtown Chicago, may ask for the 5 nearest buses to his planned route, within the next 1 hour from now.

In this article, we consider finding time-parameterized query answers [36], i.e., each query answer is a triple of the form (oid, t_{as}, t_{ae}) where *oid* is the id of the object, t_{as} is the time when the object starts to become an answer and t_{ae} is the time after which the object is no longer an answer to the query.

2.3 Triggers, semantic dimensions and context awareness

Triggers in active databases couple the database technology with rule-based programming in order to achieve reactive capabilities in response to various database (and possibly external) events. In this work, we will make extensive use of such reactive behavior to maintain the correctness of the answers to continuous spatio-temporal queries. A trigger is specified in the form of an event-condition-action rule,

which stipulates that an action will be executed upon the occurrence of a particular event, provided that a given condition holds [2]. One of the early observations in the active database literature [13], [26] was that in many prototype systems, triggers with similar specifications were exhibiting different executional behavior. In order to better categorize the declarative aspects of the triggers' specifications and enable more precise reasoning about their behavior, researchers have identified a number of so-called *semantic dimensions* and a set of values that may be chosen for each dimension, many of which have become part of the latest SQL standard [2]. For example, one semantic dimension is the *granularity of updates*: upon detection of certain event, the reaction of the ORDBMS can be in a per-tuple manner, i.e., execute the action part of the trigger for each individual tuple that has been modified, or per-set manner, i.e., execute the action only once for the entire set of the modified tuples. As another example, one can specify different *coupling modes*: e.g., should the condition evaluation immediately follow the detection of the event, or should it be deferred until the end of the current transaction. A total of 13 different semantic dimensions have been identified in [13], and later in the corresponding sections we will elaborate in detail on the semantic dimensions of the triggers that we use in this work.

Analogous to trigger semantic dimensions, we can also speak of query semantic dimensions. These include the motion patterns of queries (dynamic or static), semantics of query predicates (range or kNN), locations of queries, spatio-temporal extents of queries, etc. Together with the semantic dimensions of the triggers, the correlation among these context dimensions can highly impact the efficiency of the reevaluation and should be orchestrated to obtain further performance gains. The values of some of these dimensions can be determined at specification time, whereas the others need to be determined at run-time. In subsequent sections we will discuss how to decide the best values for these semantic dimensions to improve the performance of query maintenance.

3 Trajectory processing of continuous queries

The trajectory model, when applicable, alleviates the problem of frequent updates to the MOD and enables currently available answers for spatio-temporal queries which pertain to any future time interval. However, as we discussed in Section 1, the main problem here is due to the fact that the changes to the motion plans of individual objects, which may occur at any time between requesting an answer set to a particular query from the MOD and the end time of interest for that query in the future, could change the initially calculated answer set. This raises the question of how to maintain efficiently the answer sets to the queries on trajectories against changes to the future motion plans of the moving objects. In this section, we first discuss the relevant aspects of the reactive behavior of MOD, then introduce the overall architecture of our system. Based on these we outline the basic procedure and methodology for maintaining continuous queries.

3.1 Reactive query maintenance

Once the MOD server receives a request for processing a new query, its interface extracts the elements of the query syntax necessary for query maintenance, e.g.,

t_{ei} (end time of interest) that specifies the time after which the query no longer needs to be monitored and can be purged from the system. The basic paradigm for maintaining the answer set to a given continuous query can be described as follows:

1. Upon receiving a new query Q_{new}, process Q_{new} and set its answer set A_Q_{new};
2. Transmit A_Q_{new} to the user who posed the query;
3. Create a trigger TR_Q_{new} of the form:

   EVENT: on update to MOD
   CONDITION: if A_Q_{new} affected
   ACTION: reevaluate Q_{new} and update A_Q_{new}

In terms of the actual execution model, the step of checking if A_Q_{new} was affected requires reevaluating the query on the modified trajectories. A_Q_{new} is subsequently updated with the new results, and the user is notified of the changes.

Since we assume that the sources for the reactive behavior are bulk updates to the trajectory data in the database, we also set up a trigger TR_TAT that continuously monitors for any new traffic abnormality:

EVENT: on insert/update of a new traffic abnormality event
CONDITION: if a moving object trajectory Tr is affected
ACTION: update Tr to reflect the traffic abnormality

The problem of spatio-temporal query processing on trajectories for the purpose of generating the initial answer set has already been addressed in the literature [16], [22], [23] and is not the main topic of this work. Our objective is to reduce the processing time spent from the moment a particular abnormality is presented to the MOD up to the point when all the pending continuous queries' answer sets are brought up-to-date. Towards this goal, we focus on orchestrating the execution of the respective triggers, based on intelligent usage of the dependencies among various context/semantic dimensions. When applicable our system also interleaves the evaluation of condition parts and the execution of action parts of different triggers, in order to reduce the overall response time.

3.2 System architecture

The main components of our system are depicted in Fig. 3:

– *Moving Object Table* (MOT) stores information about the moving objects: their spatio-temporal trajectories plus some other non-spatial attributes of interest such as name, license plate number, model etc. The main attribute *traj_shape* is a *User Defined Type* (UDT) [2] conforming to the object-relational model. It contains a handle to the actual trajectory coordinates. The actual trajectories are stored as Large Object (LOB) data [2] outside the MOT and can be accessed via the handles in the respective tuples of the table. More specifically, every trajectory is uniformly split into a number of *sub-trajectories*, each of which contains a fixed number of line segments, and these sub-trajectories are organized as a linked list. The handle in the MOT table points to the first sub-trajectory of the trajectory, from which the entire trajectory can be visited following the subsequent links. A spatio-temporal variant of the R-tree [17] index such as the STR tree or TB-tree [28], [16] is created on the set of

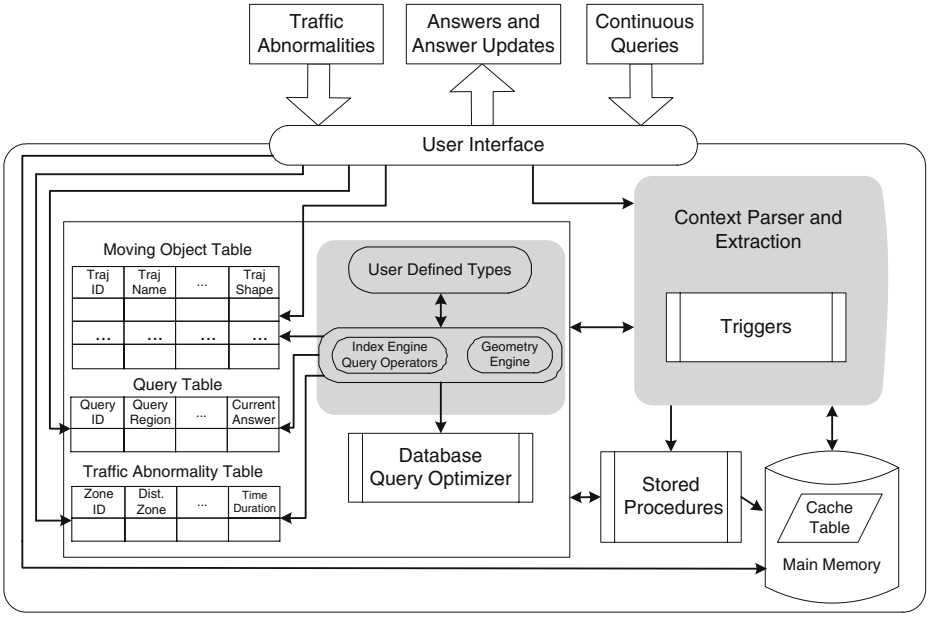


Fig. 3 System architecture

sub-trajectories independently, based on their minimum bounding boxes (MBBs), in order to facilitate efficient accesses for updates and query reevaluation. As such, the uniform splitting strategy of the trajectories aims at striking a balance between the selectivity of the index and the size of the index structures, which may lead to a better overall performance [32]. More complex splitting schemes for trajectory data may be employed to improve the performance of the index for certain types of queries [18], [32]. However, the problem of optimal splitting of the trajectories is beyond the scope of this work.

– *Traffic Abnormality Table* (TAT) stores information about the effects of the traffic abnormalities. The *disturbance_zone* attribute records the spatial region affected by an abnormality, while the *time_duration* attribute records the start and end time of an abnormality. The *disturbance_type* attribute is also a UDT that specifies the impact that a particular disturbance has on the road segments in the region of its *disturbance_zone*, in terms of the relative slow down effect on each road segment.
– *Query Table* (QT) stores information about the pending queries posed to the MOD, and contains the attributes such as query id, start time, end time of the query, etc. The two important attributes are the *query_region* and the *current_answer*. According to the semantics of the various queries, the *query_region* attribute represents the spatial area for a static range query or the querying trajectory for a dynamic range query or a kNN query. It is organized in a manner similar to the MOT table: each tuple only contains a handle to the actual data which is stored as a LOB outside of QT. The *current_answer* attribute is a UDT storing the list of the time-parameterized (oid, t_{as}, t_{ae}) answer tuples for a particular query.

3.3 Procedural framework

In a MOD, the trajectories affected by a traffic abnormality must be retrieved into main memory and updated, which includes both the trajectories of candidate moving objects as well as the trajectories of querying objects. Hence, upon a new traffic abnormality, at any point of the query reevaluation there are four distinct categories of entities to be considered, as shown in Fig. 4.

In particular, the four categories are: the unaffected pending queries and the unaffected candidate objects which reside on the hard disk, the affected queries and the affected candidate objects that have been retrieved into main memory for update. In order to bring the answers of the pending queries up-to-date again, the query reevaluation is carried out in three phases among these entities:

- In phase 1, the updated queries are reevaluated against the updated candidate objects;
- In phase 2, the updated candidate objects are checked against the unaffected pending queries;
- In phase 3, the updated queries are reevaluated against the unaffected candidate objects;

Algorithm 1 describes the basic procedural semantics of our approach, and outlines the sequence of procedures invoked, once the trigger TR_TAT is fired (recall that TR_TAT is fired on an insert or update of a new traffic abnormality). First, function `FindAffectedTrajectories` and `FindAffectedQueries` are called to retrieve the trajectories from the databases that intersect with the current traffic abnormality region, throughout the specified duration (line 1–2). All these trajectories are subsequently updated in procedures `UpdateTrajectory` and `UpdateQuery` using the new speed profile for their future portions, taking the effects of the abnormality into account (line 3–8).

At this point, the query maintenance proceeds in the three phases sequentially. Each phase is handled by the respective `ReevaluateQuery` procedure. One may observe that the signatures of all three procedures: `ReevaluateQuery_i` (*i=1,2,3*) are quite similar. However, as we will elaborate in the following sections, they

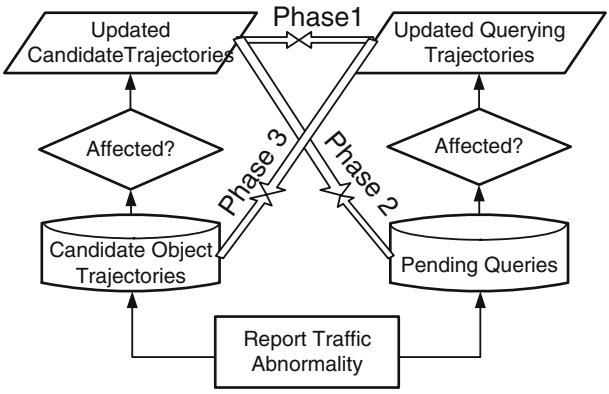


Fig. 4 The three phases of query reevaluation

Algorithm 1 Reactive Behavior to TAT Update

Input: Traffic abnormality TA_{new}, moving objects table MOT, query table QT

1: $\mathbb{T} \Leftarrow$ select $FindAffectedTrajectories(TA_{new}, MOT)$ from MOT
2: $\mathbb{Q} \Leftarrow$ select $FindAffectedQueries(TA_{new}, QT)$ from QT
3: **for all** $Tr_i \in \mathbb{T}$ **do**
4: $\quad UpdateTrajectory(TA_{new}, Tr_i)$
5: **end for**
6: **for all** $Q_j \in \mathbb{Q}$ **do**
7: $\quad UpdateQuery(TA_{new}, Q_j)$
8: **end for**
9: **for all** $Tr_i \in \mathbb{T}$ **do**
10: $\quad$ **for all** $Q_j \in \mathbb{Q}$ **do**
11: $\quad\quad ReevaluateQuery\_1(Q_j, Tr_i)$ `//phase 1`
12: $\quad$ **end for**
13: **end for**
14: **for all** $Tr_i \in \mathbb{T}$ **do**
15: $\quad ReevaluateQuery\_3(Tr_i, QT)$ `//phase 2`
16: **end for**
17: **for all** $Q_j \in \mathbb{Q}$ **do**
18: $\quad ReevaluateQuery\_2(Q_j, MOT)$ `//phase 3`
19: **end for**
20: **return**

are different with respect to the applicable techniques that are specific to the context/semantic dimensions of the triggers and queries.

The algorithm for finding the trajectories that need to be updated due to the new traffic abnormality is relatively straightforward. First, the index on the MOT table is accessed and the set of MBBs of the sub-trajectories that spatially intersect the abnormality region throughout the time interval of the abnormality are subsequently retrieved. Next, each of these sub-trajectories is checked against the abnormality region to validate if they are actually affected. If so, the entire future portion of the trajectory to which this sub-trajectory belongs is retrieved into main memory for update.

In the remainder of this section, we briefly describe the reevaluation semantics for the types of queries considered and their impact on the bottlenecks in the reevaluation. A more detailed description is available in [6].

– *Range Query* If the range query is a static one, only the second phase need to be considered for its reevaluation. This involves checking whether each updated trajectory intersects with the spatial query region during the time interval of interest. The process is similar to finding the trajectories affected by the traffic abnormality, except that now the updated trajectories reside in main memory and no index structure is available.
  When the range query is a dynamic one, all three phases in Algorithm 1 must be executed for the complete query reevaluation, since a querying trajectory may be affected by the traffic abnormality as well. Given a querying trajectory and a candidate trajectory, in order to reevaluate the spatio-temporal predicate of

the query, the two trajectories need to be broken down into segments and the calculation is performed at the line segments level using the linear functions of the segments [23].

– *kNN Query* Similar to dynamic range queries, kNN queries are reevaluated at the line segments level. The main difference is that for each kNN query, we must check the current answer set list against every new candidate object. To reduce the number of comparisons made, we adopt the idea of windowing the kNN query [19] by maintaining a *roof* value for each querying trajectory segment, which is obtained during the initial query evaluation. The roof value is the maximum distance of the *k*th nearest neighbor to that line segment. It minimizes the number of candidate trajectories/segments that need to be examined in order to find out the answer set of the kNN query. During the reevaluation, we retrieve the candidate segments which are within roof distance of the querying segment. In case we cannot find enough nearest neighbors within the retrieved data, we increase the roof value by certain percentage to retrieve more candidate segments. This iteration continues until we have completely updated the answer set.

4 System-level context-aware optimization

In this section, we present our *static* optimization techniques at the system level, i.e., how to select the proper values for the respective semantic dimensions of the triggers to minimize the context-switching overhead.

One of the semantic dimensions that may have different values in a particular trigger is the granularity of execution, i.e., whether to execute the action part of the trigger at *set level* or at *instance level* [26]. Recall from our discussion in Section 2.3, to achieve the behavior corresponding to the values of update granularity options, generally one needs to use one of the clauses: `FOR EACH ROW` or `FOR EACH STATEMENT` respectively in the specification of TR_Q_i [2].

If the executional granularity specified is at instance level, whenever a particular trajectory Tr_j is updated, the corresponding instance of TR_Q_i will evaluate it against Q_i, in order to check if the modifications to Tr_j affect the answer set A_Q_i. If so, A_Q_i will be updated accordingly. This cycle will be repeated for each of the subsequent trajectories that are affected by the update to the TAT. However, at each cycle, we have at least three major context-switchings that the ORDBMS[2] needs to do, and these will be reflected at the Operating System level:

1. From the update mode of MOT to checking if A_Q_i is affected;
2. From checking if A_Q_i is affected to eventually updating the answer set;
3. From updating A_Q_i, back to the update mode of MOT for processing and completing the modification to another trajectory Tr_k.

On the other hand, if TR_Q_i is specified to execute at the set level, all the trajectories affected by the traffic abnormality will be updated in the MOT first before

[2]The specification of the SQL99 standard distinguishes between *statement execution context*, *routine execution context*, and *trigger execution context* [2].

the ORDBMS proceeds towards evaluation of the condition part of the TR_Q_i for all the updated rows. This achieves some savings in the context-switching overhead which are far from negligible, as will be illustrated by our experimental results.

From the perspective of the user who submitted a given query Q_i, the most important issue is that the answer set A_Q_i be brought to current state as soon as possible. Part of that race can be won if one observes that the modifications of the trajectories that are affected by the traffic abnormality need not be actually completed in the database table MOT, in order to bring the queries' answers up-to-date. Once the trajectories that are affected are identified and brought into the main memory and their new shapes are calculated, the information needed to reevaluate Q_i is already available.

The context dimension of the triggers that will determine whether this kind of behavior can be utilized to optimize the response time of the pending queries in the MOD is the time coupling of the event, condition and action part of the triggers. More specifically, one can specify whether the update of the tables will take place `BEFORE` or `AFTER` the action part of the triggers [2]. Both specifications will generate correct updates of A_Q_i. However, the ordering of the operations that will take place inside the MOD is different, and the `BEFORE` option yields much faster response time. This is due to the savings in context-switching as the `BEFORE` mode avoids an extra cycle of access to the disk for update that could potentially affect some of the pending queries.

5 Context-aware query reevaluation

In this section, we present our proposed techniques for optimizing each of the three phases of the query reevaluation process at *run-time*. Based on the discussion from Section 4, we assume that the `FOR EACH STATEMENT` and `BEFORE` values have been selected as the static/non-runtime options. In the following, we first describe how to perform shared in-memory execution in phase 1, how to apply query indexing in phase 2 to restrict the search space and how to decide on the query reevaluation order in phase 3 to reduce disk accesses. We then address the extension needed to handle the case when the data processed during the reevaluation cannot fit into the main memory.

5.1 Phase 1: In-memory shared reevaluation

When considering the phase of the reevaluation where both the querying objects and the candidate objects are affected by the traffic abnormality, the trajectories of both types of objects must be updated with respect to the impact of the abnormality. The query reevaluation is carried out for these two groups of updated trajectories in memory. We assume for now that these updated trajectories can fit in the main memory and we consider the optimization of the query reevaluation in this phase.

Given a set of affected querying trajectories and a set of affected candidate trajectories that have been updated in memory, a naive approach is to perform the reevaluation in a nested loop where each affected query is checked against each affected candidate object. However, a great deal of the computation involved in the nested loop is unnecessary and can be avoided by utilizing the spatio-temporal

context information embedded in the individual querying and candidate objects. We propose an in-memory shared reevaluation scheme that can handle a simultaneous reevaluation for a group of various types of queries and is scalable to large data sets.

5.1.1 The in-memory shared reevaluation algorithm

The main idea is to exploit the spatio-temporal relationships between the querying and candidate objects by applying two filtering techniques, a mapping step based on the spatial dimensions and a sorting step based on the temporal dimension, followed by a refinement step. The mapping step utilizes a uniform grid structure that is virtually imposed over the entire region of interest [18], which maps objects from a spatial region to a particular grid cell. Hence, only queries and objects that are spatially close to each other will be mapped to the same cell and considered as candidates in the refinement step. The sorting step orders the objects based on their temporal attributes and uses a sweep-plane approach [10] to perform the reevaluation sequentially. The important observation is that these two steps are interleaved to effectively reduce a large number of false hits. The refinement step applies computational geometry algorithms on the objects that passed the two filtering steps to obtain the exact final answer set. For example, to determine whether a candidate line segment is within a given distance to a dynamic range query line segment, a quadratic equation based on their time-parameterized linear functions is solved to obtain the answer [15].

The in-memory reevaluation phase is carried out at the segment level and the mapping function for the segments is as follows:

- Each candidate trajectory segment will be mapped to the grid cells that intersect with the segment.
- For each querying trajectory segment, we consider the semantics of the query and one of the following actions is performed:
  - For a static range query, a *Minimum Bounding Rectangle* (MBR) is created for the spatial region of the query and the query will be mapped to all the grid cells that intersect with the MBR.
  - For a dynamic range query, each querying segment will be mapped to the grid cells that are within the distance specified by the query.
  - For a kNN query, each querying segment will be mapped to the grid cells that are within the distance specified by the roof value, which is the maximum distance of the kth nearest neighbor calculated during the initial query evaluation.

By performing the grid mapping, we only need to consider the reevaluation of a particular querying segment against the candidate segments that are mapped to the same grid cells. One important issue is how to decide upon the size of the grid cells. If the size of the cells is too large, many querying and candidate trajectory segments will be mapped to the same cell, resulting in a large number of false hits. On the other hand, if the size is too small, every segment is likely to be mapped to many cells and this will incur an extra computational overhead. The optimization of the grid cell size for an efficient mapping is also related to the problem of segmenting a given trajectory for the purpose of minimizing the volume of dead space in the index structure and improving the query performance, which has been addressed

in the literature [18]. However, a detailed investigation of this topic is beyond the scope of this article. In this work, we set the width/height of the cells equal to two times the average length of the trajectory segments. According to the results of the Buffon–Laplace needle problem [9], the probability of a segment with length l landing on at least one of the grid cell boundaries whose width and height are a and b respectively, is $\frac{2l(a+b)-l^2}{\pi ab}$. In our case, this means that an average-length segment with randomly chosen coordinates, has less than 48% chance of intersecting two grid cells, and moreover, it has less than 8% chances of intersecting more than two grid cells.

There is another context dimension that can be exploited to further improve the reevaluation performance: the time interval of each segment. A candidate trajectory segment only needs to be considered for a querying trajectory segment if the time intervals of the two intersect. Based on this observation, we sort the segments according to their begin times and process the sorted segments using the sweep-plane approach [10]. The sorting further reduces the number of false hits along the temporal dimension.

The main data structure involved in this algorithm is an in-memory grid table with N^2 elements corresponding to the N^2 grid cells, where N is the number of grid cells along each spatial dimension. Each element in the table points to two linked lists, one for the querying trajectory segments and the other for the candidate trajectory segments that are mapped to it, denoted as $QList$ and $OList$, respectively. We also maintain an event list E which is essentially a priority queue that keeps the starting and ending times of the segments as events to be processed. The processing procedure is illustrated in Algorithm 2.

The algorithm takes as input the segments of the updated querying trajectories and candidate trajectories. The starting times of all segments are treated as event points and initially inserted into the event list E (line 1). These events are sorted based on their time of occurrence. Note that since the starting times of the segments on the same trajectory are naturally sorted based on their time attribute, the creation of the event list can be accomplished by merging all the affected trajectories. Next, each event on the list is examined sequentially and the algorithm differentiates among the following three cases:

1. When a starting time event of a querying trajectory segment, say seg_Q, is encountered, we use the mapping function described earlier in this section to identify the grid cells that are of interest to the segment. seg_Q is then inserted into the respective $QList$s of the grid cells. All candidate trajectory segments that are currently on the $OList$s of the respective cells are then evaluated against seg_Q to produce the query answers (line 3–8), by invoking the function $EvaluateQ(seg_Q, c_{ij}.OList)$. Lastly, the ending time event of seg_Q is inserted into the event list (line 9).
2. When a starting time event of a candidate trajectory segment, say seg_{Tr}, is encountered, the grid cells with which it intersects are identified. Consequently, seg_{Tr} is considered a potential answer to all querying trajectories segments currently on the $QList$s of those cells identified by the mapping. The refinement step is then carried out by invoking the function $EvaluateTr(seg_{Tr}, c_{mn}.QList)$ in order to determine whether any new answer will be produced. Finally, seg_{Tr} is also inserted into the respective lists of the grid cells it intersects and the ending time event of seg_{Tr} is inserted into the event list (line 10–16).

Algorithm 2 In-memory Shared Reevaluation

Input: $\mathbb{S}_{Tr} \sim$ segment set of affected candidate trajectories, $\mathbb{S}_Q \sim$ segment set of affected querying trajectories

Output: $\mathbb{A}_{inmem} \sim$ Partial answer set of in-memory reevaluation phase

1: $E \Leftarrow$ the starting time events of segments in $\mathbb{S}_{Tr} \cup \mathbb{S}_Q$, in sorted order
2: **for** each event e in E **do**
3: **if** e is the starting time of a querying trajectory segment seg_Q **then**
4: $\mathbb{C}_Q \Leftarrow$ the set of grid cells that seg_Q is mapped to
5: **for** each cell $c_{ij} \in \mathbb{C}_Q$ **do**
6: $\mathbb{A}_{inmem} \Leftarrow \mathbb{A}_{inmem} \cup EvaluateQ(seg_Q, c_{ij}.OList)$
7: insert $seg_Q.id$ into $c_{ij}.QList$
8: **end for**
9: insert ending time event of seg_Q into E
10: **else if** e is the starting time of a candidate trajectory segment seg_{Tr} **then**
11: $\mathbb{C}_{Tr} \Leftarrow$ the set of grid cells that seg_{Tr} is mapped to
12: **for** each cell $c_{mn} \in \mathbb{C}_{Tr}$ **do**
13: $\mathbb{A}_{inmem} \Leftarrow \mathbb{A}_{inmem} \cup EvaluateTr(seg_{Tr}, c_{mn}.QList)$
14: insert $seg_{Tr}.id$ into $c_{mn}.OList$
15: **end for**
16: insert ending time event of seg_{Tr} into E
17: **else if** e is an ending time event **then**
18: delete the respective segment from the grid cells where it was mapped to
19: **end if**
20: **end for**
21: **return**

3. When the ending time event of a segment is encountered, the segment is removed from the corresponding grid cells (the $OLists$/$QLists$). This is needed to eliminate reevaluating querying and candidate segments that do not intersect in the temporal dimension (line 17–19).

As an example, Fig. 5a illustrates the reevaluation of three queries against four candidate trajectories, all of which have been updated in memory due to a traffic abnormality TA_{new}. The polylines represent the trajectories, where solid lines highlight the portions of the trajectories considered in this example. The octagon represents the new traffic abnormality. The shaded regions around the querying trajectory segments illustrate the mapping regions for the respective queries. The starting and ending time value of each segment of the trajectories are labelled next to the vertex. For simplicity, we align the time stamps to start from 0. Q_{R_1} and Q_{R_2} are two dynamic range queries whose querying trajectories are Tr_5 and Tr_6, respectively. Q_{kNN_1} is a k-nearest neighbor query described by trajectory Tr_7. We consider only one segment from each querying trajectory and denote them as $Q_i.seg$. The respective distance parameter for Q_{R_1} and Q_{R_2} are d_1 and d_2. The $roof$ value of Q_{kNN_1} shows the maximum distance to the kth nearest neighbor currently maintained. Trajectories Tr_1 through Tr_4 describe the motion of four candidate objects and their segments are denoted as $Tr_i.seg_j$ as shown.

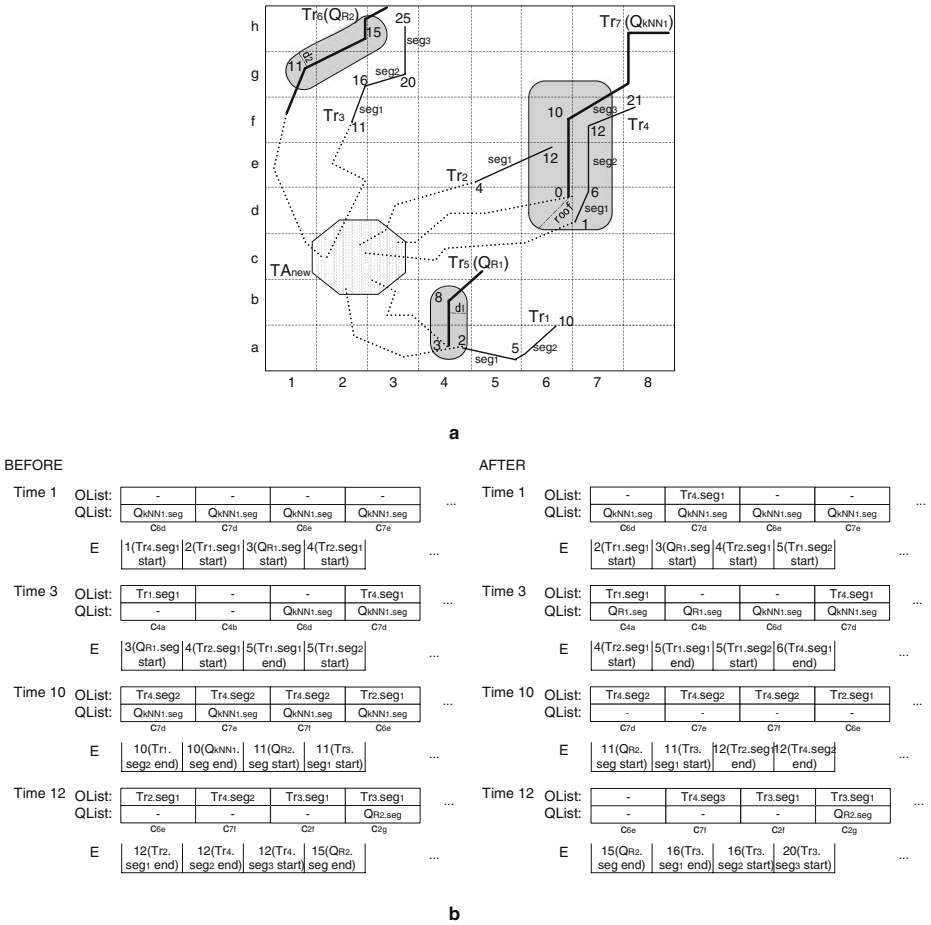


Fig. 5 In-memory shared reevaluation example

First, all starting times of the segments are inserted into the event list E: 0, 1, 2, 3, 4, 5, 6, 8, ... Then the algorithm sweeps over the priority queue and processes each event sequentially. To better illustrate the algorithm, Fig. 5b shows the states of the relevant data structures before and after processing actions are taken at the selected time stamps. For clarity, we only show the grid cells that are of interest to the reevaluation, and the first few elements currently stored in the event list E at each time stamp.

- At time 0, $Q_{kNN_1}.seg$ is inserted into the $QList$s of the grid cells intersecting with the shaded regions for $Q_{kNN_1}.seg$. Since all $OList$s of these cells are empty at time 0, no further evaluation is necessary.
- At time 1, segment $Tr_4.seg_1$ is inserted into the $c_{7d}.OList$. Since $c_{7d}.QList$ contains $Q_{kNN_1}.seg$, the segment-wise computational geometry calculation is carried out to determine whether Tr_4 is an answer to Q_{kNN_1}.

- At time 3 $Q_{R_1}.seg$ is inserted into the $QList$s for grid cell c_{4a}, c_{4b}. At this time $Tr_1.seg_1$ is in $c_{4a}.OList$, thus it is evaluated against Q_{R_1}. Note also that the ending time event for $Tr_1.seg_1$ has already been inserted into E.
- At time 10, $Q_{kNN_1}.seg$ and $Tr_1.seg_2$ (not shown) are removed from their respective grid cells ($QList$s/$OList$s).
- At time 12, $Tr_2.seg_1$ and $Tr_4.seg_2$ are removed from the $OList$s of the intersecting cells. $Tr_4.seg_3$ is mapped to $c_{7f}.OList$ and $c_{8f}.OList$. However, since $Q_{kNN_1}.seg$ has been removed from $c_{7f}.QList$ at time 10, no further reevaluation is needed.
- Finally, the algorithm stops at time stamp 25 when the event list E becomes empty.

To determine the time complexity of Algorithm 2, observe that when initially creating the event list in which the segments are sorted according to their starting time stamps, the total cost is linear in the total number of segments since each individual trajectory already has its segments sorted. Let k denote the number of querying trajectory segments and let n denote the number of candidate trajectory segments, the worst case complexity of Algorithm 2 is bounded by $O(kn)$. In practice, however, one may expect a better bound because as a consequence of the sorting of the segments and the mapping of the segments into the grid cells (which is also linear in the number of segments), the pairwise evaluation of many segments will be avoided.

5.2 Phase 2: Selecting the relevant subset of triggers

In phase 2, we need to reevaluate the updated moving object trajectories that are in the main memory against the unaffected queries that are on the hard disk. Let us recall our example from Section 2: query Q_{R_3} did not need to be reevaluated since none of the trajectories in $A\_Q_{R_3}$ were affected by the traffic abnormality, and vice-versa, none of the trajectories affected by the given traffic abnormality impact the correctness of $A\_Q_{R_3}$. In order to utilize this kind of intelligent behavior in our settings we perform some extra preprocessing work when a particular query is submitted, which will enable our system to behave in an output sensitive manner when executing the respective triggers.

The basic idea is to also maintain an index on the queries, as they are being posed to the system. In our system we implement an R-tree like index on the query_region attribute of the query table. After updating the TAT due to a traffic abnormality, as each trajectory is being accordingly modified, we maintain on the fly a two dimensional MBR of the updated portions of the trajectory routes, as well as a bounding time interval during which updates to the trajectories occur. When determining the triggers that need to be fired in order to reevaluate their respective pending queries, we only choose a subset of them. This subset consists of the triggers associated with the queries whose regions of interest actually intersect the MBR constructed when updating the trajectories during the bounding time interval. If a particular trajectory is not affected, it will not contribute to the construction of the MBR. This will prevent reevaluation of queries like Q_{R_3} in our motivational scenario. The query reevaluation only needs to be performed between a subset of the unaffected queries and the affected trajectories that will lead to query answer updates.

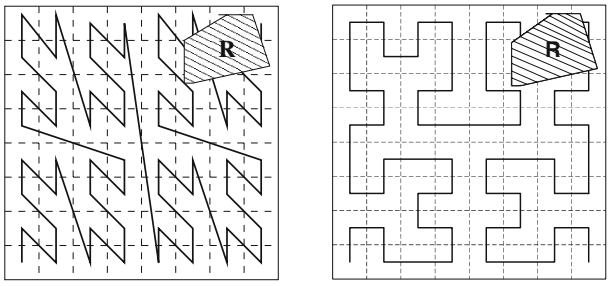


Fig. 6 Space-filling curves

a Peano curve (Z-curve) **b** Hilbert-curve

Note that the queries indexed are a mixture of spatial regions for static range queries and querying trajectories for dynamic range queries and kNN queries. For the latter two types of queries, when constructing the index, the querying trajectories are uniformly split into sub-trajectories using the exact same strategy for splitting the trajectories in MOT (cf. Section 3).

5.3 Phase 3: Ordering among the triggers

Ordering can be specified for a set of triggers that will fire upon the same enabling event, i.e., one can express when a particular trigger will have its condition evaluated and its action executed. [3] In phase 3 where we reevaluate updated queries in the main memory against unaffected candidate object trajectories on the hard disk, we propose to order the triggers (and consequently the reevaluation of the set of queries) based on space-filling curves, in order to further reduce the total reevaluation time by exploiting the spatial proximity of the corresponding queries' geometries. This is due to the fact that the LRU policy is usually used for swapping the pages between the disk and the main memory. By enforcing an order based on the space-filling curves, we increase the chances that the same pages of data be utilized for the reevaluation of several consecutive queries, and minimize the page swapping overhead [20], [25].

As discussed before in Section 5.1, we utilize the same grid structure that is conceptually imposed over the entire region of interest and order the cells according to their positions in the sequence based on the space-filling curve chosen (cf. Fig. 6). In our study, we have considered the Peano curve (also referred to as Z-curve) [25] and the Hilbert curve, since it has been shown in previous studies that the orderings generated by the Z-curve and the Hilbert curve outperform the others in preserving spatial proximity [20], [25]. To determine the position of a particular trigger in the precedence order, we use the cell to which the geometric centroid of the region R of the associated query belongs. In case that the centroids of two (or more) query regions fall in the same cell, we order these queries based on their starting time of interests. For dynamic range queries and kNN queries, the ordering is based

[3] The detailed analysis of the impact of the *coupling* modes [26] of the different parts of a particular trigger (Event, Condition or Action) among themselves and with the transaction that generated its enabling Event is beyond the scope of this article.

on the individual sub-trajectories, i.e., each sub-trajectory forms a "sub-query" and its reevaluation ordering is determined by the centroid of the MBR of this sub-trajectory. Hence, the reevaluation of a number of queries is actually an intermixed process: part of one query is reevaluated followed by the reevaluation of part of another query which is spatially close based on the space filling curve.

In order to effectively benefit from the ordering of the triggers' execution, we enforce an ordering on the candidate objects data as well. The basic idea is to organize the LOB data for the trajectories in the MOT table according to their spatial proximity. This guarantees that the trajectories that are close to one another will be stored in the same or consecutive disk pages. Hence, portions of different trajectories that are relevant to a particular query will be retrieved with fewer page accesses. In addition, as we order the triggers such that consecutive queries will pertain to spatial regions close to each other, pages that were recently read into memory can be fully utilized before swapped back to disk and thus can further reduce the number of I/Os.

More specifically, when the trajectories of the moving objects are initially bulk loaded into the MOD, they are split into sub-trajectories using the uniform splitting strategy introduced in Section 3.2. These sub-trajectories are then ordered based the same space-filling curve used for ordering the queries. We use the centroid of each sub-trajectory's 2D projection to decide their relative ordering. Subsequently, the sorted sub-trajectories are stored into MOT in this order. This ensures that the data in the MOT is more or less "spatially clustered" on the hard disk, i.e., a disk page is filled with sub-trajectories that are spatially close according to the space-filling curve. Note that the ordering of the sub-trajectories takes place during initial bulk loading, and hence it will not incur extra overhead to the reevaluation of the queries.

5.4 Scalable reevaluation for large data sets

The techniques presented in the previous sections utilize the spatio-temporal context information embedded into the trajectories to minimize the unnecessary pairs of (querying trajectory segment, candidate trajectory segment) that are reevaluated, in each of the three phases. However, since the techniques are main memory based, they cannot handle the cases when the data set is too large to fit entirely into the main memory for the reevaluation. This will cause extra data transfer due to page swapping and can become a source of performance degradation.

Strictly following the phases of Algorithm 1 and converting it into a secondary storage based algorithm is not a suitable choice. To say the least, this may cause some data items to be purged back to the disk during the reevaluation of a particular phase. However, the same data may be needed from the disk, when executing the next phase of reevaluation. Aside from the page swapping penalty, the observation that we just made has another consequence: in effect, we have lost the benefit of the `BEFORE` option for the semantic dimension of the triggers.

To address this issue, we adapt the divide-and-conquer paradigm and partition the entire MOD data into smaller portions where each one of them alone can be handled using Algorithm 1. In our system, this is achieved via *table partitioning* [2]. More specifically, we utilize *a priori* knowledge about the distribution of the trajectory data, which we collected by performing a preprocessing step that samples over the large data set, to determine the number of partitions along each spatial dimension. We ensure that the trajectory data contained in each resulting spatial partition can be

handled using Algorithm 1 without causing extra memory page swapping. Once we obtain the spatial partitioning, we will partition the MOT such that each partitioned table contains only data that spatially falls into a single partition. Trajectories that intersect with more than one partition are split and stored separately in each individual partition. The spatial partitioning need not be uniform in size, i.e., the data distribution may be skewed and partitions may have a smaller spatial extent where the trajectory distribution is dense, as illustrated in Fig. 7. Observe that due to the density of the data, the lower-left portion of the region of interest is further partitioned with a higher granularity than the rest. Furthermore, Tr_1 and Tr_2 intersect with more than one cell. Consequently, they are split and stored in each of the intersected cells respectively. The query table QT is partitioned using the same spatial partitioning.

Both the MOT table that stores the candidate trajectories and the query table QT are specified to be partitioned following the same spatial partitioning upon creation. The partitioning is decided by mapping based on the route of the trajectories (spatial projection of the trajectories). Two trajectories whose routes belong to the same spatial partition will be mapped to the same partitioned table.

Now when a new traffic abnormality is inserted into the database, we first identify the spatial partitions that are affected by the abnormality. As illustrated in Fig. 7, these are the partitions which intersect the shaded ellipse (the dashed lines represent its MBR) representing the abnormality region. We consider the fractions of the abnormality region that is contained in each individual partition region, and perform the reevaluation sequentially on each table partition. Within each partition, we use the particular fraction of the abnormality region to identify the affected queries and candidate objects, and perform the three phase reevaluation over the data belonged to that partition. This ensures that no memory page swapping will occur while reevaluating the portion of continuous queries pertaining to a given partition and guarantees the query reevaluation is scalable to very large data sets.

6 Experimental evaluation

In order to assess the effectiveness of the various aspects of our proposed methodology, we have fully implemented our system on top of Oracle 9i, a mature and

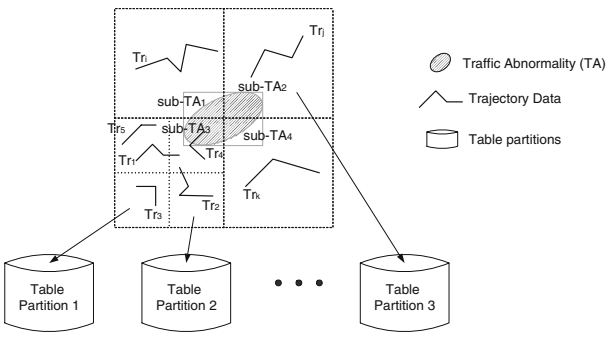


Fig. 7 Table partitioning based on the grid structure

commercially available ORDBMS which, among the others, offers the following features:

- *Triggers*, the mechanism which enables declarative specification of reactive behavior in ORDBMS for the maintenance of continuous queries and part of the SQL99 standard [2].
- *PL/SQL*, a programming language that enables the specification of the procedural aspects of the applications. Our query evaluation and reevaluation algorithms have been completely implemented using PL/SQL.
- *Oracle Spatial* [33], which defines what is essentially an abstract data type that conforms to the object-relational model and is used for modelling various spatial objects in the database. In our system, we have utilized and extended the data type to model and represent spatial-temporal trajectories and continuous queries. More details about our implementation can be found in [6], [11].

6.1 Experimental setup

The experiments were performed on a PC with an Intel Pentium IV CPU 3.6 GHz processor, 1 GB of DDR2 memory, an 80 GB SATA hard disk, and with *Windows XP Professional* installed. The ORDBMS system that we used is *Oracle Release 2, version 9.2.0.1.0*.

The data set used in our experiments contains 5,000 trajectories. They were generated using 1,000 real trajectories obtained from the DOMINO project [1], which were constructed using the Chicagoland (Cook County) electronic map [4]. On the basis of these trajectories, we generated another 4,000 trajectories, using road segment endpoints from the source data set and adding randomized perturbations to each endpoint. The distribution of trajectory length in terms of number of segments per trajectory is shown in Fig. 8. Assuming that the average length of a block is about 1/8 mile, the length of the trajectories in our data set varied between 1 and 60 miles. In our experiments, the data actually spanned an area that is bounded by a rectangle of approximately 1,780 miles^2. The vertices of the trajectories were sampled every

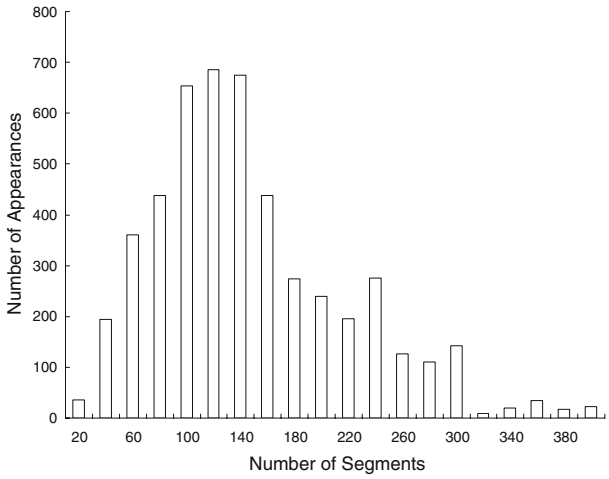


Fig. 8 Trajectories length distribution (1 segment corresponds to 1 block, which is approximately 1/8 mile)

minute, and the duration of the trajectories varied between 20 and 400 min. In our experiments, we transform the time measure into discrete units, where 10 time units correspond to 1 min. Thus, the lifetime of MOD is 4, 000 time units.

For static range queries, the query regions were randomly generated within the area of interest to the MOD, and the time interval spans were distributed evenly throughout the lifetime of MOD. For dynamic range queries and kNN queries, the querying trajectories were randomly selected from the 5, 000 trajectories in the data set, and the radius of the circles for the range queries varied from 1.0 to 5.0 miles. Traffic abnormalities had a duration of 200 time units and could cause the traffic delay by up to 50% of the normal speed.

The metric that we used in the experimental evaluation is the *response time* for reevaluating a set of pending continuous queries. We defined the response time to be the duration from the time instance at which an INSERT request is specified to the TAT table (reflecting a traffic abnormality), until the answer set (`Current_Answer` attribute in the query table) of every pending query in the MOD is brought up-to-date. The performance is measured using Oracle's `DBMS_PROFILER` package.

6.2 Experimental results

First we demonstrate the effectiveness of each of our context-aware optimization techniques in isolation. We then present a combined approach to compare the overall performance of our optimized query reevaluation approach against a naive approach, which works in a brute force manner without utilizing any of our proposed techniques.

6.2.1 Semantic dimensions of the triggers

When it comes to the different semantic dimensions used in the specification of a particular trigger, we investigated two of them: (1) `BEFORE` *vs.* `AFTER` execution, which specifies the mode in which the modifications to the database are applied with respect to the action part of the trigger; (2) `SET` *vs.* `TUPLE` execution, which specifies the level of granularity of applying the modifications to the database and its relationship with the trigger. Combining each of the values, we have four possible options and we measured the running times for each of them. In the experiments, we focused on one single query (consequently, one trigger) and we varied the number of trajectories affected by the traffic abnormality from 8 to 200, by changing the spatial location and time interval of the disturbance zone. For a static range query, the average response times are shown in Fig. 9. The cases for the dynamic range queries and kNN queries are shown in Fig. 10a and b, respectively. The dotted lines represent the case when the querying trajectory is not affected by the abnormality, where only the second phase of reevaluation need to be executed. The solid lines represent the case when the query trajectory is also affected by the abnormality and need to be updated, and all three phases of reevaluation are necessary. All three groups of experiments have provided consistent outcomes. As expected, the best performance is obtained when the `BEFORE` trigger executes in a `SET` oriented manner, improving the response time by up to 85% compared with the worst case running time. A noteworthy observation is that loading the *Oracle Spatial* package incurs a substantial context-switching time penalty and this is part of the reason why

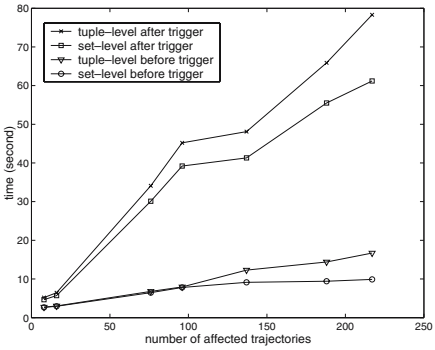


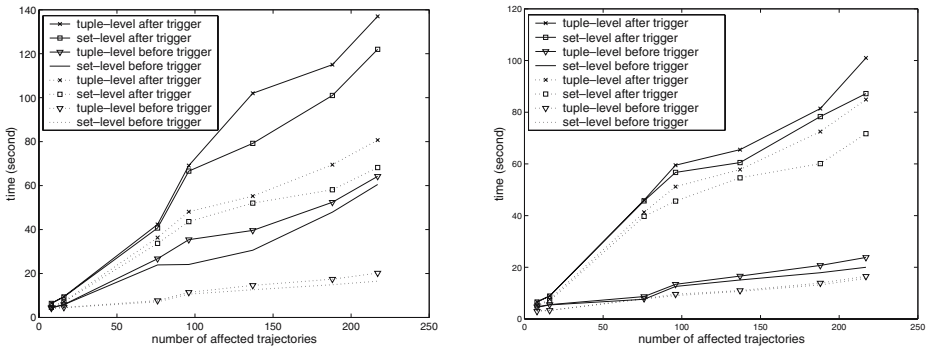
Fig. 9 Semantic dimensions of the triggers for static range query

the discrepancy between the set-based manner and the tuple-based manner in the `BEFORE` context is somewhat diminished. However, as we will demonstrate shortly, a significant contribution in the overall improvement of the response time yields from combining the benefits of the `SET` and `BEFORE` semantic dimensions as we will discuss later. All the following experiments on the run-time optimization techniques assume the `SET` and `BEFORE` options are used.

6.2.2 Query indexing and query ordering

We first evaluated the impact of using query indexing on static range queries. We randomly selected a number of queries that are distributed evenly in the area of interest. We conducted two groups of experiments, where the number of affected trajectories was fixed at 100 and 200, respectively. The number of pending queries increases from 4 to 100 in both group of experiments. The result is shown in Fig. 11.

As indicated in Fig. 11, the usage of the query index for intersecting the MBR of the affected trajectories achieves significant improvements in the response time. This is due to the elimination of the triggers whose associated queries need not be reevaluated. Observe that the improvements are greater as the number of the pending queries increases.



a Dynamic Range Query

b kNN Query

Fig. 10 Semantic dimensions of the triggers for dynamic range and kNN queries

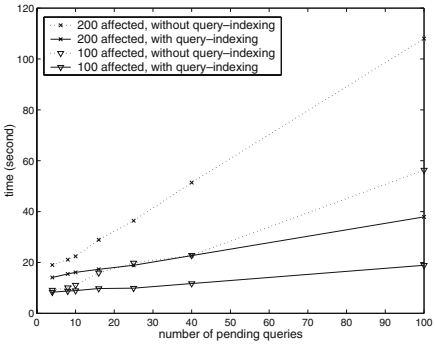
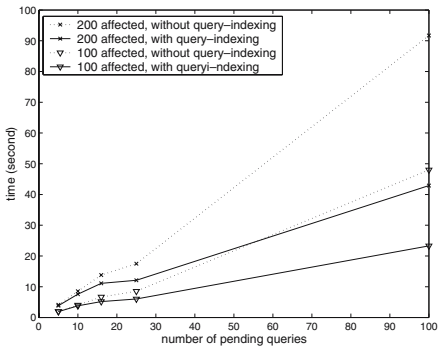


Fig. 11 Query indexing for static range queries

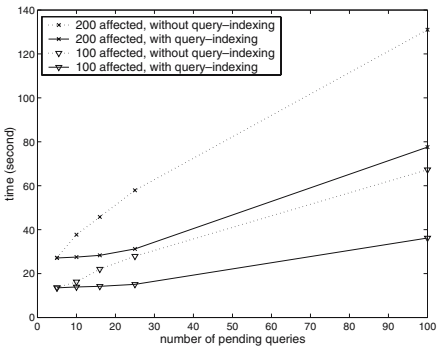
Next we evaluate the case when query indexing is applied to dynamic range queries and kNN queries. When reevaluating the queries whose trajectories are not affected against the updated moving object trajectories, we use the MBR of the updated trajectories constructed on the fly to retrieve intersecting sub-trajectories from the query table. Subsequently we check each of these "sub-queries" against the updated trajectories. The experimental results for dynamic range queries and kNN queries are shown in Fig. 12a and b, respectively. The two figures suggest a similar pattern of improvement when indices are created on the pending queries, by up to 75% compared to the case when query indexing is not utilized.

Our next group of experiments evaluates the impact of using query ordering. Recall that query ordering applies to the third phase when the updated querying trajectories are reevaluated against the unaffected candidate trajectories residing on the disk. Hence only dynamic range queries and kNN queries are considered in this step. We fix the values for the trigger semantic dimensions as suggested in Section 4, to exclude their impact. Figure 13a and b verifies our hypothesis by showing that using ordering among the triggers improves the response time, both for dynamic range queries and kNN queries, the more than 10% improvement steadily comes from the reuse of the data loaded into memory for reevaluating consecutive queries.

We have also studied the effects of the choice of different space-filling curves in our experimental evaluation, by comparing the Hilbert curve with the Peano curve.



a Range Queries



b kNN Queries

Fig. 12 The impact of query indexing

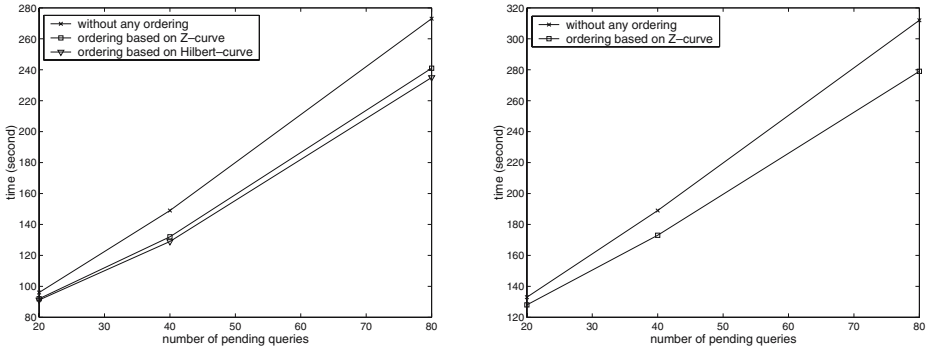
We observed that the Hilbert curve consistently outperforms the Peano curve during the experiments, conforming with the observations in [20]. Geometry-based intuition would tend to explain these observations as a consequence of the fact that the Hilbert curve has fewer large space jumpings and irregularities than the Peano curve. Part of the reason is likely to be due to the fact that the iterative process of constructing the Hilbert space-filling curves maps the intervals of length 2^{-2n} into squares of size $2^{-n}x2^{-n}$, while the one for Peano's curve is equivalent to mapping the interval of length 3^{-2n} into squares of size $3^{-n}x3^{-n}$ [34]. However, a further analysis of the properties of different space filling curves is beyond the scope of this article. The experimental results are shown in Fig. 13a as well.

6.2.3 In-memory shared reevaluation

In this section, we examine the performance of our in-memory shared reevaluation approach for reevaluating a group of range queries and kNN queries, whose querying trajectories are affected by the traffic abnormality. We compare the performance of our in-memory shared reevaluation algorithm with the naive approach, which uses a nested loop without any grid mapping and sorting based on the spatio-temporal attributes of the trajectories. For both groups of experiments, we use the BEFORE trigger processing semantics and the results of the evaluation are shown in Fig. 14. It can be seen that the in-memory shared reevaluation approach which, viewed in isolation, does not yield as much performance gain as the other optimization methods that we have considered. The reason for this is that this algorithm optimizes the first phase of the whole reevaluation process, which takes a relatively smaller portion of the total reevaluation time. Hence it is harder to achieve a significant performance gain.

6.2.4 Overall performance improvement

So far, we analyzed the impact on the response time for each semantic/context dimension in isolation. Figure 15 presents our experimental observations when



a Ordering Range Query Reevaluation

b Ordering kNN Query Reevaluation

Fig. 13 The pact of ordering query reevaluation

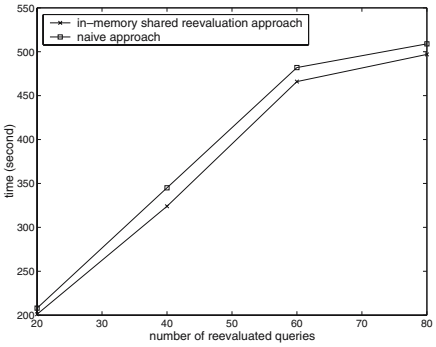


Fig. 14 Performance of in-memory shared reevaluation

comparing our combined approach against the naive maintenance approach, i.e., TUPLE-level and AFTER trigger execution, no query indexing, no ordering among triggers and no shared in-memory reevaluation. The experimental results show that the optimized query reevaluation can execute as much as 4 times faster than the naive approach.

Next, we studied the impact of the partitioning based approach for the data sets, which are too large to be handled in memory all at the same time. To observe the impact of the large size trajectory data, we conduct the following case study: (1) we modified the Oracle database system parameter `db_cache_size`, which determines the size of memory that is allocated to hold the data blocks after they are retrieved from disk. In our simulation, `db_cache_size` is set to 1 MB. (2) furthermore, we generated another 30, 000 trajectories, such that the total data size became 33 MB. We fixed the number of pending queries to 80 and measured the total reevaluation time when varying the number of affected trajectories. We compared the performance when the relevant data tables are partitioned *v.s* the case when they are not partitioned. We apply all the optimization techniques in both cases. The naive approach is not fully shown simply because it takes too long for the reevaluation process to terminate. The experimental results are shown in Fig. 16.

From the figure it can be seen that when the affected data set is small the reevaluation performance of the two approaches are almost identical. However, as the number of affected trajectories increases, the partitioned approach outperforms the non-partitioned approach by up to 22%, when the size of the relevant data

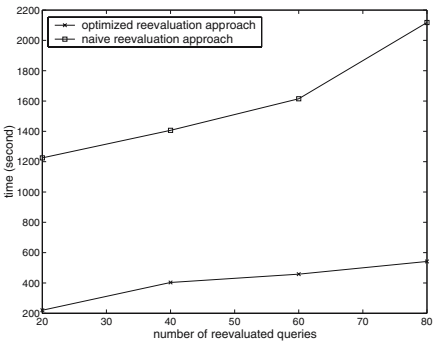


Fig. 15 Performance of optimized query maintenance

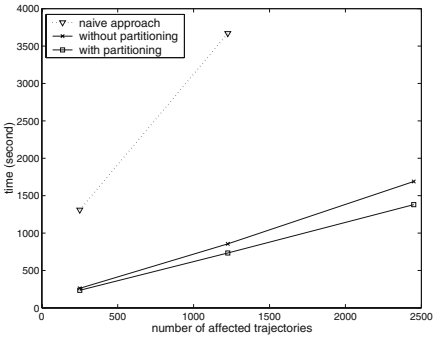


Fig. 16 Performance of table-partitioning approach

is relatively large, i.e., 2, 500 affected trajectories. Observe that the non-partition approach is not scalable, e.g., when the number of affected trajectories increases from around 250 to 2, 500, the reevaluation time increases by more than 10 times.

7 Related work

An important aspect of our work is related to the well-studied field of active databases [26], [40]. Besides the variety of prototype systems, triggers have become a common feature of many commercial ORDBMS that comply with the SQL standard [2]. In this work, we utilized the triggers for the particular problem of maintaining continuous spatio-temporal queries for trajectories. More specifically, we combined the values of various semantic dimensions [13] and focused on orchestrating the triggers' execution for the purpose of efficient reevaluation of a given set of pending queries.

The field of MOD has generated a large body of research results in the past few years, and the existing works address the problems of modelling and representation of moving objects, algorithms for efficient management of spatio-temporal queries as well as access/indexing methods. Most of these works use one of the three data models that we presented in Section 2: a sequence of (*location*, *time*) updates, a sequence of (*location*, *time*, *velocity*) updates or a full trajectory. Our work belongs to the third category: we use trajectories to represent the future motion plans of the mobile entities. The peculiarities of this model have been thoroughly investigated [12], [15], and the efficient algorithms for processing continuous queries for the historical trajectories have been proposed in [23]. When it comes to the portions of the trajectories representing the future motion plans of the objects, the algorithms in [23] can be carried over verbatim for the initial evaluation of the queries' answers. However, a perturbation in the parameters that were used to generate the future trajectories, even if applied to relatively small geographic area, can affect the correctness of the pending queries' answer sets pertaining to the future in a much larger geographic region. This is precisely the problem that we addressed: how to efficiently reevaluate the set of pending continuous queries when traffic abnormalities cause updates to the future motion plans of the moving objects.

Perhaps the work that comes closest in spirit with ours is the SINA project [24], [42], which has investigated the problem of efficient maintenance of spatio-temporal queries for the model of continuous (*location*, *time*) updates. As a result of the

different models, the underlying algorithms for maintaining the continuous queries are different. In SINA, query maintenance has to be performed periodically at a high frequency, whenever updates are available. Furthermore, the query maintenance exhibits some spatial locality because a subsequent update is in general close to the previous one, while in our case a local update may impact queries that span over a much larger geographic regions. Its recent extensions have also considered the lazy buffering approach for balancing the trade-offs between data updates and query reevaluation [43]. If the adopted model is the (*location*, *time*, *velocity*) updates to the MOD [35], the system can afford to "see" a bit further into the future and avoid constant reevaluation of the pending queries. However, when updates arrive, a larger set of spatio-temporally correlated queries may need to be reevaluated in a bulk manner [19]. In this sense, our work covers the "far-end of the spectrum": with the trajectories pertaining to the future motions of the moving objects, one can pose and obtain the answer sets to the various continuous queries pertaining to any future time interval. However, the main penalty is the reevaluation due to the impact of some potential traffic abnormality that may affect a large number of queries which need not be in a close geographic proximity.

Investigation into the essential properties of efficient indexing structures in spatio-temporal settings were presented by Theodoridis et al. [37]. It has also been observed that straightforwardly using an R-tree [17] or its variants, e.g., the R∗ tree [7] on spatio-temporal trajectory data will introduce a large amount of dead space. Several indexing methods have been proposed that try to reduce the impact of the dead space for trajectories. For example, Pfoser et al. [28] attempt to preserve the segments that belong to the same trajectory, and the problem of obtaining an efficient splitting of the trajectory segments has been further investigated in [8], [18], [32]. In our work, we relied on the R∗ tree indexing mechanism that is readily available in Oracle 9i [33], for indexing the two dimensional routes of the trajectories. However, we have adopted it for various needs, e.g., indexing the trajectory segments, using the time information of trajectories in a refinement step etc.

The idea of indexing the queries instead of moving objects, for the purpose of efficient spatio-temporal query processing was presented in [31], and was further extended for in-memory evaluation in [21]. However, the motion model used there—a sequence of (*location*, *time*) updates as the objects are moving—is different from our data model of the trajectories. This in turn requires a different reactive behavior and consequently processing algorithms. The utilization of space-filling curves to impose a linear order on the spatial-temporal objects has been investigated by Jensen et al. [20], which reports access efficiency obtained from the linear ordering. On the other hand, we adopted the space-filling curves for sequencing the reevaluation of the pending queries, in order to reduce the context switching costs among different queries.

8 Conclusions and future work

The efficient maintenance of continuous queries over moving objects is essential when providing location-based services. In this article, we addressed the issues that arise when reevaluating a set of continuous queries over the trajectories that are used to model the future motion plans of the moving objects, which is an extension of

our earlier framework [38]. Towards this, we proposed a context-aware approach for optimization and identified opportunities for performance tuning at various levels. We proposed several context-aware optimization techniques that can substantially reduce the query reevaluation time, and analyzed the impact that each of them has on various continuous queries, e.g., range queries and kNN queries.

Our query reevaluation begins by setting up relevant triggers that watch the moving objects table and the pending queries table, as new traffic abnormalities are reported. We identified at least three major sources of context-switching overhead and unnecessary disk I/Os incurred at the operating system level, if the triggers are not specified properly. We analyzed the trigger execution process and found that the optimal performance is expected when the triggers are specified to execute at the `SET` level, and in the `BEFORE` manner.

The first step in the query reevaluation is to retrieve the affected querying trajectories and candidate trajectories into main memory, and update them to reflect the impact of the traffic abnormality. The unaffected queries and candidate objects remain on the disk. Then, the query reevaluation takes three phases to complete:

- In the first phase, the affected queries are reevaluated against the affected candidate trajectories. We employed an in-memory shared reevaluation algorithm to utilize the spatio-temporal information embedded in the data and effectively reduce the amount of computation for reevaluating querying trajectory segments against candidate trajectory segments.
- In the second phase, the affected candidate trajectories are checked against the unaffected queries on disk. We utilized an R-tree index built on the queries to reevaluate only the set of relevant queries and thus improve reevaluation efficiency by limiting the search space.
- In the third phase, the affected queries are reevaluated against the unaffected candidate trajectories on disk. We ordered the queries to be reevaluated as well as the on-disk data (the segments of trajectories) based on a space-filling curve to preserve spatial proximity. This reduces the number of disk pages to be retrieved and avoids potential memory page swapping.

We also considered the case where the entire set of affected data cannot be accommodated in the main memory at the same time. We proposed to partition the relevant data tables according to a grid structure, such that each partition can fit into the memory in order to avoid unnecessary memory page swapping. The reevaluation cycles over all the partitions affected by the traffic abnormality executing each of the three phases.

We have built our system prototype on top of an industry-strength ORDBMS—Oracle 9i, making maximum use of the existing commercial database capability. We have conducted extensive experiments to observe the benefits of our proposed techniques. We would like to point out that our approach is not strictly limited to Oracle 9i. In fact, any ORDBMS that conforms to the SQL-99 standard [2] and supports specifying User-Defined Types (UDT) and implementing User-Defined Functions (UDF) can be used as a foundation of our implementation.

There are a few potential opportunities for future research work. Currently we are working on incorporate uncertainty into the query maintenance process, i.e., a user who posed a continuous query may receive answers that come with a probabilistic

attribute, which more realistically reflects the status of moving objects [29]. We are also working on incorporating some other optimization techniques such as shared memory buffer and data prefetching to improve the execution of the set of enabled triggers in the ORDBMS. Another interesting problem is how to utilize in our approach information of the underlying road networks [27] to further improve the efficiency of the reevaluation procedure.

References

1. DOMINO project at DBMC laboratory, UIC. *http://www.cs.uic.edu/~wolfson/html/mobile.html*.
2. ANSI/ISO international standard: Database language SQL. *http://webstore.ansi.org*.
3. Forbes online issue, April 13, 2006. *http://www.forbes.com/home/digitalentertainment/2006/04/13/google-aol-yahoo -cx_rr_0417maps.html*.
4. Geographic Data Technology. *http://www.geographic.com*.
5. Oracle 9i. *http://www.oracle.com/technology/products/oracle9i*.
6. H. Ding. *Context Aware Optimization of Continuous Query Maintenance for Trajectories*. M.S. thesis, Dept. of EECS, Northwestern University, 2005.
7. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. " The R*-Tree: An efficient and robust access method for points and rectangles," in *Proc. of SIGMOD Conference*, pp. 322–331, 1990.
8. V.P. Chakka, A. Everspaugh, and J.M. Patel. "Indexing large trajectory data sets with SETI," in *Proc. of CIDR*, 2003.
9. E.W. Cheney and D.R. Kincaid. Numerical Mathematics and Computing. Brooks Cole, 2003.
10. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. McGraw-Hill, NY, 1990.
11. H. Ding, G. Trajcevski, and P. Scheuermann. "Omcat: optimal maintenance of continuous queries' answers for trajectories," in *Proc. of SIGMOD Conference*, pp. 748–750, 2006.
12. L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. " A data model and data structures for moving objects databases," in *Proc. of SIGMOD Conference*, pp. 319–330, 2000.
13. P. Fraternali and L. Tanca. " A structured approach for the definition of the semantics of active databases," *ACM Transactions on Database Systems*, Vol. 20(4):414–471, 1995.
14. B. Gedik and L. Liu. "Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system," in *Proc. of EDBT*, pp. 67–87, 2004.
15. R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. "A foundation for representing and querying moving objects," *ACM Transactions on Database Systems*, Vol. 25(1):1.
16. R.H. Güting and M. Schneider. Moving Object Databases. Morgan Kaufmann, CA, 2005.
17. A. Guttman. "R-trees: A dynamic index structure for spatial searching," in B. Yormark (Ed.), *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18–21, 1984*, pp. 47–57. ACM Press, 1984.
18. M. Hadjieleftheriou, G. Kollios, V.J. Tsotras, and D. Gunopulos. "Efficient indexing of spatiotemporal objects," in *Proc. of EDBT*, pp. 251–268, 2002.
19. G.S. Iwerks, H. Samet, and K.Smith. "Continuous k-nearest neighbor queries for continuously moving points with updates," in *Proc. of VLDB*, pp. 512–523, 2003.
20. C.S. Jensen, D. Lin, and B.C. Ooi. "Query and update efficient b+-tree based indexing of moving objects," in *Proc. of VLDB*, pp. 768–779, 2004.
21. D.V. Kalashnikov, S. Prabhakar, S.E. Hambrusch, and W.G. Aref. "Efficient evaluation of continuous range queries on moving objects," in *DEXA*, pp. 731–740, 2002.
22. M. Koubarakis, T.K. Sellis, A.U. Frank, S. Grumbach, R.H. Güting, C.S. Jensen, N.A. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Schek, M. Scholl, B. Theodoulidis, and N. Tryfona, (Eds.), *Spatio-Temporal Databases: The CHOROCHRONOS Approach*, volume 2520 of *Lecture Notes in Computer Science*. Springer, 2003.
23. J.A.C. Lema, L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. "Algorithms for moving objects databases," *Computer Journalen*, Vol. 46(6):680–712, 2003.
24. M.F. Mokbel, X. Xiong, and W.G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. in *Proc. of SIGMOD Conference*, pp. 623–634, 2004.

25. B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 13(1):124–141, 2001.
26. N.W. Paton (Ed.), *Active Rules in Database Systems*. Springer, New York, 1999.
27. D. Pfoser and C.S. Jensen. "Trajectory indexing using movement constraints*," *GeoInformatica*, Vol. 9(2):93–115, 2005.
28. D. Pfoser, C.S. Jensen, and Y. Theodoridis. "Novel approaches in query processing for moving object trajectories," in *Proc. of VLDB*, pp. 395–406, 2000.
29. D. Pfoser, N. Tryfona, and C.S. Jensen. "Indeterminacy and spatiotemporal data: Basic definitions and case study," *GeoInformatica*, Vol. 9(3):211–236, 2005.
30. E. Pitoura and G. Samaras. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 13(4):571–592, 2001.
31. S. Prabhakar, Y. Xia, D.V. Kalashnikov, W.G. Aref, and S.E. Hambrusch. "Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects," *IEEE Transactions on Computers*, Vol. 51(10):1124–1140, 2002.
32. S. Rasetic, J. Sander, J. Elding, and M.A. Nascimento. "A trajectory splitting model for efficient spatio-temporal indexing," in *Proc. of VLDB*, pp. 934–945, 2005.
33. R. Kothuri, A. Godfrind and E. Beinat. Pro Oracle Spatial. Apress, 2004.
34. H. Sagan. Space-filling Curves. Springer, 1994.
35. A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. "Modeling and querying moving objects," in *Proc. of ICDE*, pp. 422–432, 1997.
36. Y. Tao and D. Papadias. "Time-parameterized queries in spatio-temporal databases," in *Proc. of SIGMOD Conference*, pp. 334–345, 2002.
37. Y. Theodoridis, T.K. Sellis, A. Papadopoulos, and Y. Manolopoulos. "Specifications for efficient indexing in spatiotemporal databases," in *Proc. of SSDBM*, pp. 123–132, 1998.
38. G. Trajcevski, H.Ding, and P. Scheuermann. "Context-aware optimization of continuous range queries maintenance for trajectories," in *MobiDE*, pp. 1–8, 2005.
39. G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. "Managing uncertainty in moving objects databases," *ACM Trans. Database Syst.*, Vol. 29(3):463–507, 2004.
40. J. Widom and S. Ceri. *Active Database Systems*. 1995.
41. O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha. "Updating and querying databases that track mobile units," *Distributed and Parallel Databases*, Vol. 7(3):257–387, 1999.
42. X. Xiong, M.F. Mokbel, W.G. Aref, S.E. Hambrusch, and S. Prabhakar. "Scalable spatio-temporal continuous query processing for location-aware services," in *Proc. of SSDBM*, p. 317, 2004.
43. X. Xiong and W.G. Aref. "R-trees with update memos" in *Proc. of ICDE*, 2006.

Hui Ding received the B.E. degree in electronic engineering from Tsinghua University, Beijing, China in 2003. He is now a Ph.D. student in the deparment of electrical engineering and computer science at Northwestern University, U.S.A. His research interest is in spatio-temporal databases and data management in mobile computing.

Goce Trajcevski is a researcher at the Dept. of Electrical Engineering and Computer Science at the Northwestern University. His main interests are in the areas of mobile data management and sensor networks. He received a BS from the University of Sts. Kiril & Metodi, and the MS and PhD from the University of Illinois at Chicago. He coauthored over 25 publications, participated as a PC member of several conferences and workshops, and was ACM DiSC associate editor 2003–2005. He is a member of IEEE and ACM.

Peter Scheuermann is a Professor of Electrical Engineering and Computer Science at Northwestern University. He has held visiting professor positions with the Free University of Amsterdam, the University of Hamburg and the Swiss Federal Institute of Technology, Zurich. During 1997–1998 he served as Program Director for Operating Systems at the NSF. Dr. Scheuermann has served on the editorial board of the Communications of ACM, The VLB Journal and IEEE Transactions on Knowledge and Data Engineering. His current research interests are in parallel and distributed database systems, mobile computing, spatial databases and data mining. He has published more than 100 journal and conference papers. Peter Scheuermann is a Fellow of IEEE.