On Moving Object Queries*

(Extended Abstract)

Hoda Mokhtar Jianwen Su Oscar Ibarra

Department of Computer Science
University of California
Santa Barbara, CA 93106
{hmokhtar, su, ibarra}@cs.ucsb.edu

ABSTRACT

Database applications for moving objects pose new challenges in modeling, querying, and maintenance of objects whose locations are rapidly changing over time. Previous work on modeling and querying spatio-temporal databases and constraint databases focus primarily on snapshots of changing databases. In this paper we study query evaluation techniques for moving object databases where moving objects are being updated frequently. We consider a constraint database approach to moving objects and queries. We classify moving object queries into: "past", "continuing", and "future" queries. We argue that while traditional constraint query evaluation techniques are suitable for past queries, new techniques are needed for continuing and future queries. Motivated by nearest-neighbor queries, we define a query language based on a single "generalized distance" function f mapping from objects to continuous functions from time to $\mathbb{R}$. Queries in this language may be past, continuing, or future. We show that if f maps to polynomials, queries can be evaluated efficiently using the plane sweeping technique from computational geometry. Consequently, many known distance based queries can be evaluated efficiently.

1. INTRODUCTION

Present technology has made it possible to track down movements of target objects in the air (e.g., airplanes), on the land (e.g., vehicles, wild animals, people, etc.) and ocean (e.g., ships, animals). Among the challenges novel applications involving such "moving objects" have brought to software development is the problem of data management. In a nutshell, there is a wide range of issues including modeling and representation of moving objects, query language design, indexing techniques, query optimization, etc. Prior work on spatio-temporal databases is relevant but insufficient for moving objects. Constraint databases provide a promising framework but significant results are lacking to make the techniques useful. The goal of this paper is to investigate query evaluation techniques in a constraint data model for moving objects.

Existing models for moving objects mostly involve complex tools such as temporal logic [25, 31] and abstract data types [10, 11, 15]. The weakness is that the model and languages are very hard to analyze and optimization techniques are difficult to develop. Constraint databases [16] present a nice conceptual model for spatial and temporal data [18] and are a natural choice for representation and manipulation of moving objects. Previous work focus mainly on the modeling issues [4, 5, 27].

In terms of efficient query evaluation, earlier work focus primarily on algorithms for ad hoc queries [17, 1] and indexing techniques [22]. Evaluation techniques for query languages have not been extensively addressed.

In this paper, we start from an extension [27] of CQL [16] for moving objects. Although it is expressive in reasoning about spatio-temporal relationships of moving objects, it is not as suitable for moving objects as it appears. This is due to moving object queries of interest often involve "future" which is not known until updates are performed. Updates are particularly important for moving objects. While the quantifier elimination based techniques [18] are useful for "past" queries, "future" queries require either a major overhaul, or some entirely new framework. We take the latter approach.

Motivated by many distance related queries in moving objects (e.g., [29]), we develop a generalized distance based query language. We illustrate that the language is expressive enough for many distance-related moving object queries and show that queries in this language can be evaluated efficiently. Roughly, the new language involves a single "generalized distance" function which maps objects to continuous functions from time to $\mathbb{R}$. Our evaluation strategy is to consider the set of functions for objects and evaluate queries by sweeping a time line. In particular, plane-sweeping allows to identify intersections of these functions, thus to prepare for the changes that may propagate to query answers. The main technical results are: for "polynomial" generalized distances, queries can be evaluated "efficiently". In particular, for databases with N objects, each "past" query can be evaluated in $O((m+N)\log N)$ time (m is the total number of

*Support in part by NSF grants IIS-9817432 and IIS-0101134.

intersections among objects (their functions). For "future" queries, initialization can be done in $O(N \log N)$ time, while evaluating an update depends on the interval between consecutive updates. In practical settings where either updates happen regularly or the database has a clock, each update can be processed in $O(\log N)$ time.

Our work is also related to view maintenance. Earlier work are almost exclusively in the context of relational databases (see [13]): e.g., maintaining first order queries with the relational algebra [23] and with duplicates [12], maintaining recursive queries with Datalog [14] or the relational calculus [9]. Using techniques from computational geometry, the results we establish in the paper present a new approach to a new problem—view maintenance for a class of queries in constraint databases.

The paper is organized as follows. In Section 2, we introduce a data model for moving objects. In Section 3, we describe a constraint query language and discuss its weaknesses for moving objects. In Section 4, we introduce a generalized distance based query language, and in Section 5, we develop an efficient query evaluation technique. Conclusions are provided in Sections 6.

2. A MODEL FOR MOVING OBJECTS

In this section, we introduce a data model for moving objects and the notion of an update. Unlike earlier models for moving objects that use ADTs [10, 11, 15] or temporal logics [25], our model uses the linear constraint techniques from constraint databases [18]. The model is similar to the one in [4, 5] and a simplified version of the model presented in [27].

Moving objects are spatial objects. In this paper we only consider moving points. The main difference between moving object databases and spatio-temporal databases that have been studied in the literature is that objects in the latter either do not change their locations or changes happen very slowly (i.e., over a long period of time). Moving objects change their locations rapidly. Managing frequent location changes and discovering change patterns are often central to many applications such as traffic status monitoring and collision discovery.

Constraint databases [16] are well suited for modeling and manipulation of multi-dimensional data [18] conceptually. The key ingredient is "constraints", first-order formulas with a prescribed semantics (they are to be distinguished from *integrity constraints*). The constraint based models allow spatio-temporal data to be viewed as mathematical objects, which can be studied with known tools. For moving objects, moving directions, velocity, trajectories, and perhaps curvature and torsion are essential properties. For example, the Fundamental Theorem of Curves states that each curve is uniquely determined by its curvature and torsion (modulo positions) [19]. In the general case, differential geometry is a more suitable tool for modeling and studying moving objects [27]. The use of (linear) constraints simplifies the modeling. At the same time, constraint query languages are inexpressive in reasoning about other properties such as velocity. Clearly further investigation on modeling moving objects is needed. The focus of this paper is on efficient query evaluation rather than the modeling aspect.

In the following we give a brief presentation of the data model. Note that we explicitly provide *representations* of the concepts, since the representations are directly manipulated in evaluating queries and contribute to the complexity.

Linear constraints over the real numbers are logic formulas involving the equality and order predicates ($=, <, \leqslant, >, \geqslant$) and addition ($+$). A linear constraint over variables x_1, ..., x_n have the following general form: $\sum_{i=1}^{n} a_i\, x_i \theta a_0$, where $a_0, a_1, ..., a_n$ are real (or rational) numbers and θ is a predicate. Constraints are interpreted over the real numbers. We also allow other equivalent forms.

For the remainder of the paper, we consider objects in the space $\mathbb{R}^n$ for some $n > 0$ and assume time to be an additional continuous dimension also represented as $\mathbb{R}$. We also distinguish the notion of *time intervals* which are real intervals. Without loss of generality, we further assume that time intervals are closed or unbounded (i.e., no open intervals). We represent each time interval as a conjunction of linear constraints over t. We use a vector $\boldsymbol{x} = (x_1, ..., x_n)$ to represent a point in the space (excluding time). Vector notations can be viewed as a shorthand for conjunctions and are not essential in the model.

The location of a moving point can be modeled by a function from time to n-dimensional space. A function from $\mathbb{R}$ to $\mathbb{R}^n$ is *linear* if it has the form $\boldsymbol{x} = At + B$ where A, B are vectors in $\mathbb{R}^n$; *piecewise linear* if it consists of a finite number of linear pieces.

DEFINITION 1. A *trajectory* is a continuous piecewise linear function from $\mathbb{R}$ to $\mathbb{R}^n$. Let $\boldsymbol{\mathcal{T}}$ be the set of all trajectories.

Under the above definition, each coordinate of a trajectory is a linear function of the time variable t, and the trajectory may change speed and direction at finitely many time instants.

Using the linear form and a time interval, each linear piece of a trajectory is represented as a conjunction of linear constraints using the time variable and coordinate variables. A trajectory is ad isjunction of all its linear pieces. A *turn* of a trajectory is a time instant τ at which the derivative of the function is not continuous.

EXAMPLE 1. Consider airplanes moving in the 3-dimensional space. A trajectory of an airplane composed of 3 linear pieces is shown below.

$$\begin{array}{ll} & \boldsymbol{x} = (2,-1,0)t + (-40,23,30) \wedge 0 \leqslant t \leqslant 21 \\ \bigvee & \boldsymbol{x} = (0,-1,-5)t + (2,23,135) \wedge 21 \leqslant t \leqslant 22 \\ \bigvee & \boldsymbol{x} = (0.5,0,-1)t + (-9,1,47) \wedge 22 \leqslant t \end{array}$$

The airplane moved towards southeast, turned at time 21 (and at position $(2, 2, 30)$) and also started to descend, and made another turn at time 22 (and at position $(2, 1, 25)$) and continued to descend. ∎

We assume the existence of an infinite set $\boldsymbol{\mathcal{O}}$ of *object identifiers* (OIDs).

DEFINITION 2. A *moving object database* (or MOD) is a triple (O, T, τ) where O is a finite subset of $\boldsymbol{\mathcal{O}}$, T a mapping from O to $\boldsymbol{\mathcal{T}}$, and τ a time instant such that each turn of every object in O is earlier than or at time τ.

In the above definition of a MOD, the time instant τ is the time of the last update. For moving objects, updates are of particular interest since querying over the current positions of objects and reasoning about the past and future positions are of central importance in the applications.

Updates for moving object databases have not been sufficiently addressed in the literature; only ad hoc query evaluation techniques were studied in [26]. Our focus in this paper is to investigate the impact of updates on query evaluation techniques.

We consider three types of updates: create a new object, change the motion of an object, or "delete" an existing object (the object ceases to exist after the update). Following typical settings in moving object databases, we assume that every update is associated with a time instant, and updates are performed in the order of their time instants (i.e., chronologically).

DEFINITION 3. Let τ be a time instant, $o \in \boldsymbol{\mathcal{O}}$ an oid, A, B are vectors. An *update* on a MOD (O, T_0, τ_0) is one of the following:

- (*Create a new object*) $\mathsf{new}(o, \tau, A, B)$ if $\tau_0 < \tau$ and $o \notin O$. The result is the MOD $(O \cup \{o\}, T, \tau)$ where
$$T \;=\; T_0 \cup \{(o,\ \boldsymbol{x} = At + B \wedge \tau \leqslant t)\}$$
- (*Terminate an existing object*) $\mathsf{terminate}(o, \tau)$ if $o \in O$ and $\tau_0 < \tau$. The result is the MOD (O, T, τ), where T is identical to T_0 except that
$$T(o) \;=\; T_0(o) \wedge t \leqslant \tau$$
- (*Change moving direction*) $\mathsf{chdir}(o, \tau, A)$ if $o \in O$, $\tau_0 < \tau$, and the trajectory $T_0(o)$ of the object o is defined at time τ. Let B be the position of o at time τ. The result is the MOD (O, T, τ) where T is identical to T_0 except that
$$T(o) \;=\; (T_0(o) \wedge t \leqslant \tau) \vee (\boldsymbol{x} = A(t - \tau) + B \wedge \tau \leqslant t)$$

The result of applying an update δ on a MOD D is denoted by δD.

EXAMPLE 2. Consider the trajectory in Example 1. Suppose o is the OID of the object having the trajectory. Then, $\mathsf{dir}(o, 47, (0, 0, 0))$ is a direction change. The updated trajectory is the following.

$$\begin{array}{rl} & \boldsymbol{x} = (2, -1, 0)t + (-40, 23, 30) \wedge 0 \leqslant t \leqslant 21 \\ \bigvee & \boldsymbol{x} = (0, -1, -5)t + (2, 23, 135) \wedge 21 \leqslant t \leqslant 22 \\ \bigvee & \boldsymbol{x} = (0.5, 0, -1)t + (-9, 1, 47) \wedge 22 \leqslant t \leqslant 47 \\ \bigvee & \boldsymbol{x} = (14.5, 1, 0) \wedge 47 \leqslant t \end{array}$$

After the update, the airplane o landed at time 47 (and position $(14.5, 1, 0)$) and stayed at the point. ∎

The focus of this paper is on queries over moving object trajectories. For simplicity, we do not include other descriptive information about moving objects. Clearly, the model can be easily extended to include such information using known techniques [27]. Also, most moving object applications may likely have spatial objects (e.g., roads, city regions, etc.). Constraints provide a very expressive modeling language to represent and access such objects (cf [18]). For this reason, in our formal model, we exclude such spatial objects except for stationary points whose motions are constant vectors.

3. PAST, CONTINUING, AND FUTURE QUERIES

In this section, we briefly describe a constraint query language for moving objects. The language is a simplified version of the one described in [27]. Although such constraint languages are expressive in reasoning about spatial and temporal relationships of moving objects, we argue that it is not as suitable for MODs as it appears. In particular, we introduce three classes of queries: "past", "continuing", and "future" queries. While the quantifier elimination based techniques [18] are useful for past queries, continuing and future queries require either a major overhaul, or some entirely new framework. We take the latter approach (the details are presented in Sections 4 and 5).

Constraint query languages are extensions of the relational calculus. As observed in [16], the key component for evaluating constraint queries is the quantifier elimination property of the first-order theory of real closed fields [28]. Tractable algorithms of quantifier elimination for a fixed number of variables have been developed, see e.g., [7, 2, 24].

In our language, we allow vector variables and constants which represent points in $\mathbb{R}^n$. If $\boldsymbol{x}$ is a vector variable, $\boldsymbol{x}.i$ denotes its i-th component. Addition of two vectors and multiplication of a vector by a real (rational) number are also allowed. In addition to vector variables (constants, terms), there are real variables (for time instants or spatial coordinates), and object variables (ranging over OIDs).

Let $D = (O, T, \tau)$ be a MOD. We use the symbol "O" to represent a unary relation storing the set of objects in D, and the symbol "T" to represent a ternary relation over objects, time instants, and vectors for location coordinates. As illustrated in Definition 3, T is a constraint relation.

If $x_1, ..., x_n$ are real variables, $(x_1, ..., x_n)$ is a vector. If y is an object variable, t a real variable, and $\boldsymbol{x}$ a vector term, "$O(y)$" and "$T(y, t, \boldsymbol{x})$" are atomic formulas. The former states that the object y is in the MOD, while the latter says that the object y is in location $\boldsymbol{x}$ at time t.

The query language is almost standard except that we allow the following:

- Functions on vectors: *unit* which maps a vector to a unit vector (useful for expressing directions), *len* which returns the length of the vector (not trajectory), and

- Function *vel* which on input an object variable computes the derivative of each component of the object's trajectory over time.

Finally we allow the usual propositional connectives and quantifiers for $\mathbb{R}$, vectors, time, and objects.

Example 3. Consider a query which finds all aircrafts entering the Santa Barbara County from time τ_1 to τ_2. The interesting condition is "entering". An object enters a region at time t if it is not in the region at every time instant just before t.

$$\{\, y \mid O(y) \wedge \exists t(\tau_1 \leqslant t \leqslant \tau_2 \wedge \exists \boldsymbol{x}(T(y,t,\boldsymbol{x}) \wedge \psi(\boldsymbol{x})) \wedge \exists t'(t'<t \wedge \forall t''(t'<t''<t \rightarrow \neg\exists \boldsymbol{x}(T(y,t'',\boldsymbol{x}) \wedge \psi(\boldsymbol{x})))))\}$$

where the formula $\psi(\boldsymbol{x})$ states that the point $\boldsymbol{x}$ is inside (above) Santa Barbara County. ∎

Note that the spatial region (Santa Barbara County) in the query in Example 3 does not depend on time. An interesting feature in moving object databases is that queries may be "location dependent" in the following sense. The answer to a query may depend on the location where the query was issued. As an example, we can change Santa Barbara County in the above query to the "spatial region within 50 miles of United flight 764".

Location dependent queries can be easily modeled in the constraint based framework with some syntactic sugar. In particular, we allow each query to be associated with a trajectory $\gamma \in \boldsymbol{\mathcal{T}}$, and γ can be used in the query formula. In query evaluation, the query trajectory is treated as a fixed constraint relation over the time and vector variables.

A *query* is a triple $Q = (y, \gamma, \varphi)$ where y is an object variable, $\gamma \in \boldsymbol{\mathcal{T}}$ a trajectory without turns, and φ a formula with only y free (φ may use γ in the same way as an object). The *answer* of a query Q on a MOD $D = (O, T, \tau)$, $Q(D)$, is defined as the set of objects $y \in O$ such that φ is true.

Example 4. Consider the query 1-NN (1-nearest neighbor) that returns the closest object(s) to an object moving along the trajectory γ during the period from τ_1 to τ_2. Suppose that the MOD $D = (O, T, \tau)$ where $\tau_1 < \tau_2 < \tau$. The query 1-NN can be expressed as (y, γ, φ) where $\varphi =$

$$\exists t(\tau_1 \leqslant t \leqslant \tau_2 \wedge \exists \boldsymbol{x} \exists \boldsymbol{x}_1 T(\gamma, t, \boldsymbol{x}) \wedge T(y, t, \boldsymbol{x}_1) \wedge \forall z(O(z) \rightarrow \forall \boldsymbol{x}_2(T(z,t,\boldsymbol{x}_2) \rightarrow len(\boldsymbol{x}_1 - \boldsymbol{x}) \leqslant len(\boldsymbol{x}_2 - \boldsymbol{x}))))$$ ∎

In our model, databases consist of only linear constraints. The query language is the first-order logic with linear constraints except that the length and unit operators need polynomial constraints. Given a query Q and a MOD D, we can "encode" OIDs using real numbers, map D to a linear constraint database, and Q to a first-order query with polynomial constraints. The query evaluation can be done using standard constraint query evaluation algorithms based on quantifier elimination. Therefore, the following holds.

Proposition 1. *Each query can be evaluated in polynomial time in terms of the* MOD *size.*

Example 5. Consider the queries in Examples 3 and 4 for the situation where τ_1 is in the past while τ_2 is in the future. If we evaluate the queries over the current database, the results are not all correct:

- the results on flights/objects before "now" accurately reflect the actual events, and
- the results on flights/objects after "now" are only predictions since these flights/objects may change their moving directions. ∎

Predictions are useful in moving object databases. However, mixing true answers with predictions is not a desirable paradigm. While support for expressing and efficient evaluation of prediction queries is an interesting topic itself, our concern in the present paper is to obtain true or "valid" answers.

Example 5 illustrates an interesting issue of proper modeling of the time dimension. Modeling time as $\mathbb{R}$ is simple but does not reflect the physical reality. Indeed, Example 5 reveals a weakness of the constraint data model. Indexing techniques that distinguish time and space dimensions are being studied for moving object databases [22].

From the above discussions, one useful concept is to distinguish queries that concern past trajectories from those about future trajectories.

Definition 4. Let D be a MOD and Q a query. An object $o \in Q(D)$ is *valid* if for all (finite) update sequences Δ, $o \in Q(\Delta D)$. The *valid answer* of Q is defined as $Q^v(D) = \{\, o \mid o \in Q(D)$ and is valid $\}$.

Definition 5. Let D be a MOD and Q a query. Q is said to be

- *past with respect to* D if $Q(D) = Q^v(D)$,
- *future with respect to* D if $Q(D) \neq Q^v(D)$ and $Q^v(D) = \varnothing$ (no valid answer),
- *continuing with respect to* D if $Q(D) \neq Q^v(D)$ and $Q^v(D) \neq \varnothing$ (partially valid answer).

One way to evaluate a future query (to compute the valid answer) is to wait till all updates are completed on the objects being queried (i.e., till the query becomes past) and then perform the evaluation (lazy evaluation). Alternatively, one can "semi-evaluate" the query and produce valid answers as updates are being done. Such an "eager" approach is closely related to view maintenance, which is not well understood in constraint databases.

Earlier work on view maintenance mostly focus on relational databases (see references [23, 8, 12, 6, 14, 13] etc.) These results are not directly applicable to constraint databases

due to the fundamental difference of the constraint model: finite representation of possibly infinite relations. Lacking appropriate update models is also a factor [18]. Indeed, view maintenance is an interesting problem in constraint databases.

One might imagine to integrate constraint query evaluation techniques with view maintenance algorithms (if they are available). In this way a MOD system could detect past or future queries and then apply suitable evaluation algorithms. This is, however, not promising.

THEOREM 2. *It is undecidable if a given query is past with respect to a given* MOD.

PROOF. (Sketch) The proof is based on a reduction from the Halting problem of Turing machines on empty input. We consider only new operations and use objects in the database sorted by their insertion times to encode the initial configuration. The query simply checks if the database after a sequence of insertions encodes a sequence of configurations leading to a halting computation. ∎

COROLLARY 3. *It is undecidable if a given query is continuing, or future with respect to a given* MOD.

4. A GENERALIZED DISTANCE BASED QUERY LANGUAGE

As discussed earlier, queries in MODs may be queries in the traditional sense, or actually view maintenance problems. While it is interesting to further investigate view maintenance in the constraint database setting, we take a different approach which allows us to use one technique for both query evaluation and view maintenance for MODs. In this section, we illustrate the idea with an example, and define the language. The core evaluation technique is presented in the next section.

EXAMPLE 6. Consider again the 1-NN query in Example 4. Note that a key part of the query expression is to compare distances (expressed using the function len): the distances from the query object to the object y in the answer and an arbitrary object z. In particular, if we can define a distance function $f_o(t)$ for each object o in the MOD which computes the distance between o and the query object at time t, computing the answer to 1-NN is to find the lower "envelope" of the set of curves (functions) f_o's in the specified time interval.

We can extend the 1-NN query to k-NN for some fixed $k > 0$. In this case, the answer to the k-NN query is the set of k lowest curves. ∎

A large class of queries in moving object databases concern relationships about distances, moving directions, traveling time. This is not surprising because they are fundamental properties that make moving object databases distinguished from spatial databases. The classes of moving object queries defined in [29] also confirm that (Euclidean) distance is one of the most important properties for queries.

EXAMPLE 7. Let q be the (moving) query object q and o be a moving object in the database. We are interested in the fastest time it takes for o to reach q if both are to maintain the current speed and only o can change the moving direction. Consider the query "fastest arrival" which identifies the object that can reach q faster than all other objects in the database. This query is simplified version of "finding the police car that can reach the target train fastest". Using constraint query languages, this query can clearly be expressed. Interestingly, for this query we can also define a function for each object o which returns the shortest time for o to reach q. ∎

While evaluation algorithms for the straightforward extension of constraint query languages are difficult to develop as shown in the previous section, we take a different approach by restricting the expressiveness concerning distances. Specifically, we associate one distance function for each moving object and only allow comparing distances at the same time instant. However, to make the framework flexible for a variety of distance related queries, the distance functions are not limited to Euclidean distances. The result is a generalized distance based query language. We illustrate that the language can express many distance-related MOD queries and allow efficient evaluation. The language is defined in the remainder of the section and we discuss evaluation techniques in the next section.

DEFINITION 6. A *generalized distance* (*g-distance*) is a mapping from the set $\boldsymbol{\mathcal{T}}$ of trajectories to continuous functions from $\mathbb{R}$ (time instants) to $\mathbb{R}$. Let (O, T, τ) be a MOD and f a g-distance. We extend the mapping f to a mapping f from O as follows: for each $o \in O$, $\mathsf{f}(o) = f(T(o))$.

For simplicity we use the notions of a generalized distance and its extension interchangeably when the context is clear.

Intuitively, a g-distance encodes some property of a trajectory of interest as a continuous function from time instants to real numbers (such as distances). The g-distance can then be used as a primitive for the purpose of querying.

EXAMPLE 8. For the k-NN query in Example 6, let γ be the trajectory of the query object and o a moving object in the database. Consider the function

$$d_o(t) = (len(\boldsymbol{x}_o - \boldsymbol{x}))^2$$

where $\boldsymbol{x}_o, \boldsymbol{x}$ are the current locations of the two objects, i.e., vectors such that the formula $o(t, \boldsymbol{x}_o) \wedge \gamma(t, \boldsymbol{x})$ is true. Note that d_o is quadratic since q and o have trajectories in $\boldsymbol{\mathcal{T}}$. If the function d is defined such that for each object o in the database $d(o) = d_o$, then d is a g-distance. ∎

EXAMPLE 9. Consider the query in Example 7. Let o be an object moving in $\mathbb{R}^2$ with speed v. Suppose that the query object q moves along horizontal line with constant speed v and o moves with constant speed v_o.

Let τ_Δ be the time for o to meet q (o may change direction). Figure 1 shows the current position of o and q and

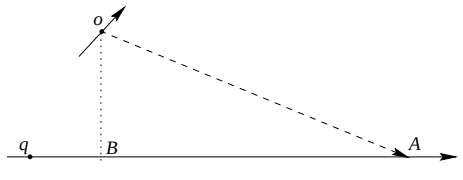

Figure 1: Redirection of o towards q

illustrates that if o changes direction it can reach point A at the same time as q. Clearly $\overline{oA} = v_o t_\Delta$ and $\overline{pA} = v t_\Delta$. Using straightforward algebra manipulations, one can show that $t_\Delta^2 = c_2 t^2 + c_1 t + c_0$ for some constants c_2, c_1, c_0. In particular, t_Δ^2 is a g-distance. ∎

For many distance related MOD queries, g-distances are expressible in the constraint query language presented in Section 3.

We now describe the query language based on a given generalized distance f. The language is essentially many-sorted first-order logic with real numbers, time instants, and objects (OIDs). The language uses a single time variable t, many object variables. (Note that there are no variables for real numbers, i.e., arithmetic operations are all embedded into g-distances.) A query expresses a property on a snapshot of moving objects in the database.

- *Time terms* are polynomials over the time variable t with real (rational) coefficients.
- *Real terms* include real numbers, and $\mathsf{f}(y, t)$ where y is an object variable and t a time term.

Atomic formulas are formed by equality and order predicates over terms of the same sort; *formulas* are then constructed by propositional connectives and universal/existential quantifiers over object variables.

A *query* is a quadruple (y, t, I, φ), where y is an object variable, t a time variable, I a time interval, and φ a formula with only y and t free.

EXAMPLE 10. To express the 1-NN query, a key component is to express the shortest distance. Clearly, this can be done by the following formula:

$$\varphi(y, t) \;=\; \forall z(d(y, t) \leqslant d(z, t))$$

Where the universe of the quantifiers of object variables is the set of objects in the MOD and d is the g-distance given in Example 8. ∎

As shown in the above example, the formula in a query defines some property on a snapshot. Let $Q = (y, t, I, \varphi)$ be a query and $D = (O, T, \tau_0)$ a MOD. For each time τ, we define

$$Q[D]_\tau = \{(o) \mid o \in O \wedge \varphi(o, \tau)\}.$$

The *answer* to Q can be defined in the following three ways.

- (Snapshots) The *snapshot* answer to Q is a set
$$Q^{\mathrm{s}}(D) = \{(o, t) \mid t \in I \wedge o \in Q[D]_t\}.$$
Though $Q^{\mathrm{s}}(D)$ may be infinite, it has finite representations in terms of time constraints on t if g-distances are "polynomial" (defined in the next section).
- (Existential) The *accumulative* answer is defined as
$$Q^{\exists}(D) = \{(o) \mid \exists t\ (t \in I \wedge o \in Q[D]_t)\}.$$
- (Universal) The *persevering* answer is defined as
$$Q^{\forall}(D) = \{(o) \mid \forall t\ (t \in I \rightarrow o \in Q[D]_t)\}.$$

We denote the language as FO(f).

EXAMPLE 11. Clearly the k-NN is expressible in FO(f). By using the same (similar) generalized distances, it is easy to see that the following queries can also be expressed in FO(f).

- List the k-nearest flights to Flight 623 at time τ.
- List all flights that were within 50 km from Flight 623 from τ_1 to τ_2.
- If Flight 744 changes its motion to be $\boldsymbol{x} = A't + B'$, which is the nearest flight at some future time τ?
- In the last hour what police cars were at the same positions as the car #1404?

The fastest arrival query is also in FO(f). By using similar generalized distances, we can show that the following queries are also expressible in this language.

- List all flights that can reach Flight 623 within 30 minutes (this may require changing of their motions hypothetically).
- For the police car #1404 (moving) list other police cars that can reach it in 5 minutes. ∎

Clearly, the key to evaluating $Q^{\forall}, Q^{\exists}, Q^{\mathrm{s}}$ queries is to evaluate and maintain answers to $Q[]_\tau$ upon updates and location changes. This will be the focus in the next section.

5. A PLANE-SWEEPING BASED EVALUATION TECHNIQUE

In this section, we present a new query evaluation technique for FO(f) using the plane-sweeping technique from computational geometry. As indicated earlier, evaluation of snapshot, existential, and universal queries relies on maintaining query answers at each time instant. The focus here is on the latter. Intuitively, our strategy is to consider the set of functions (over time) in the image of the g-distance f and evaluate queries by sweeping a time line. In particular, plane sweeping allows to identify intersections of these functions, thus to prepare for the changes that may propagate to the query answer. The main results are the following: for each MOD D and a "polynomial" g-distance f, FO(f) can be evaluated "efficiently".

Recall that there are two types of terms in FO(f) for times and real numbers, respectively. Note that each real term has at most one object variable. A real term is *instantiated* if it does not contain any object variable or its object variable

is substituted by an OID. Let $Q = (y, t, I, \varphi)$ be a query and D a MOD for the remainder of the section. We define $rterms(Q)$ to be the set of all real terms in Q with all possible instantiations using OIDs in the MOD D. (Instantiated real terms may contain the time variable.) The *base* of Q in D at time t is defined as the set of true atoms involving real terms, $base(Q, D, t) =$

$$\{\alpha_1(t)\theta\alpha_2(t) \mid \alpha_1, \alpha_2 \in rterms(Q) \wedge D \models \alpha_1\theta\alpha_2\}.$$

The base of of a query at time t consists of true atomic formulas involving equality and order comparisons. There are redundancies in the base because it contains the transitive closure of the linear ordering. Obviously, it is not necessary to compute the base.

We define the *support* of the query Q in the MOD D at time t, denoted by $supp(Q, D, t)$ as the minimal subset of $base(Q, D, t)$ which is logically equivalent to $base(Q, D, t)$. Let $supp(Q, D) = \bigcup_{t \in I} supp(Q, D, t)$. Clearly $Q[D]_t$ can be easily obtained from $supp(Q, D)$ by some relational operations.

In the remainder of the section, we study the complexity of computing $supp(Q, D)$ for a query Q and a MOD D in terms of the size of D. After introducing several necessary concepts, we state the main results of the section.

A g-distance f is said to be *polynomial* if for each $\gamma \in \boldsymbol{\mathcal{T}}$, $\mathsf{f}(\gamma)$ consists of finitely many pieces and is piecewise polynomial.

Let τ be a time instant. We define the *support change* at time τ as $\Delta_{supp}(t) =$

- $supp(Q, D, \tau - \epsilon) \ominus supp(Q, D, \tau)$, if there exists an $\epsilon > 0$ such that $supp(Q, D, \tau - \epsilon) \neq supp(Q, D, \tau)$ and for all $\tau - \epsilon \leqslant t, t' < \tau$, $supp(Q, D, t) = supp(Q, D, t')$. (Here $\ominus$ denotes symmetric difference.)
- $\varnothing$, otherwise.

For polynomial g-distances, it can be shown that there are only finitely many time instants at which the support change is nonempty. The *number of support changes* at time t is simply $|\Delta_{supp}(t)|$ (its cardinality). For each interval I, let the *number of support changes during* I be the total number of support changes at a time instant in I.

We are now ready to state the main results.

THEOREM 4. *For each past query Q in* FO(f) *with an interval I and each N-object* MOD D, *$supp(Q, D)$ can be computed in $O((m + N) \log N)$ time, where m is the number of support changes during I.*

Complexity analysis for future queries is different from past queries. This is because query evaluation is done in two phases: compute the initial support, and compute the incremental changes to support after each database update. While the total cost of computing $supp(Q, D)$ is not very useful, the cost of handling each database update is the most interesting part.

THEOREM 5. *For each future query in* FO(f) *with an interval $[\tau_1, \tau_2]$ and each N-object* MOD D, *the following hold.*

1. *$supp(Q, D, \tau_1)$ can be computed in $O(N \log N)$ time, and*
2. *for each database update, $supp(Q, D)$ can be maintained in $O(m \log N)$ time, where m is the number of support changes between two consecutive updates.*

COROLLARY 6. *If the number of support changes between updates is bounded, the maintenance of the support of a future/continuing query in* FO(f) *after each update is in $O(\log N)$ time, where N is the number of objects in* MOD.

Theorem 5 implies that if the updates are far apart, the cost of maintaining the support after each update increases since there are likely more support changes. One possibility to reduce the complexity is for a MOD to keep a clock and the number of support changes between two clock ticks is likely small. Although it does not reduce the total amount complexity, it distributes the cost for processing an update among the clock ticks. Another possibility is to have frequent (periodic) updates. Both methods are common in moving object applications. Thus, we can conclude the following:

Under reasonable practical assumptions, each update for a future FO(f) *query can be processed in $O(\log N)$ time where N is the number of objects in the* MOD.

Note that the algorithms for past queries and for future queries are almost identical. Therefore, continuing queries are computed in the same way as a past query in the first half, and future query in the second half (without initialization).

The remainder of this section is devoted to proofs of Theorems 4 and 5. We start with the k-NN query and use the query to illustrate the main idea of the algorithms. Suppose that the support of the query at the current time is computed. The support of the k-NN query will change if

1. object creations and terminations are made, or
2. the movements of objects (including the query object itself) cause some objects to be closer to become new k-nearest neighbors.

For the latter, Figure 2 shows the g-distances of two objects (solid curves). The object o_2 is closer but at time D o_1 is expected to be closer. However, updates (new, terminate, chdir)

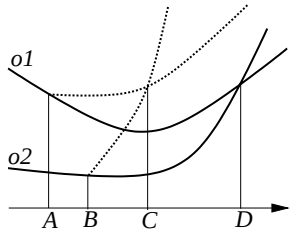

Figure 2: Two objects and their g-distances

may change some expected future events. In Figure 2, o_1

changes its moving direction at time A and as a result, its g-distance curve will not meet o_2's at time D. At a later time B, o_2 also changes its course and o_1 will again become closer than o_2 but at an earlier time C.

The maintenance of the k-NN query was studied in [26] in the context where only the query object moves. Since other objects are stationary, they are stored in an R*-tree. Their approach was to use range search to find k closest objects and re-calculate the range at each update on the query object using the moving distance since the last update. This gives correct a query result only at the time of search following the update, and the result may soon become incorrect due to the movement of the query object. In particular, the closerness exchange of o_1, o_2 at time C in Figure 2 will not be detected in their algorithm.

In MODs, updates are explicitly requested and are the only *external events*. Recall that a MOD has an associated last update time, which is changed only when an update is performed. This means that support changes caused by updates can be handled by detecting possible changes and performing actions at the update time.

Support changes caused by object movements need to be treated differently. Although it is easy to see that such support changes will only happen when the g-distance curves of different objects intersect, one cannot simply perform the changes (e.g., the change at time C in Figure 2) since it is certain that updates will not happen before then. In spite of this, detecting intersections is a well studied topic in computational geometry and plane sweeping is one of the techniques for this problem (cf [20]). This is the approach we take.

Let $D = (O, T, \tau)$ be a MOD and f a g-distance. For simplicity, we denote $\mathsf{f}(o)$ as f_o where o is an object in a MOD. Clearly, $\{\mathsf{f}_o \mid o \in O\}$ is a set of continuous functions from time instants to $\mathbb{R}$, i.e., a set of curves. Our query evaluation algorithm sweeps a time line through this set. For each given time instant, we order the curves according to the ordering of their intersections with this time line. The following definition captures this ordering.

DEFINITION 7. Let $D = (O, T, \tau')$ be a MOD, f a g-distance, τ a time instant, and o, o' two OIDs in O. The object o *precedes* o' at time τ with respect to f, denoted as $o \leqslant^{\mathsf{f}}_{\tau} o'$, if $\mathsf{f}_o(\tau) \leqslant \mathsf{f}_{o'}(\tau)$. Furthermore, $o \equiv^{\mathsf{f}}_{\tau} o'$ if $o \leqslant^{\mathsf{f}}_{\tau} o'$ and $o' \leqslant^{\mathsf{f}}_{\tau} o$; $o <^{\mathsf{f}}_{\tau} o'$ if $o \leqslant^{\mathsf{f}}_{\tau} o'$ but $o \not\equiv^{\mathsf{f}}_{\tau} o'$.

Since we assume that f is fixed in our discussions on query evaluation, we will drop the superscript "f" and simply use $<_\tau, \leqslant_\tau$, etc. Also, $\leqslant_\tau$ is naturally extended to include real numbers.

Clearly, $o \equiv_\tau o'$ means that their g-distances at the time τ are the same. It follows that for each time instant τ, $\equiv_\tau$ is an equivalance relation. Technically, instead of the total ordering of the curves (objects) we have a total order on equivalence classes $[o]_\tau$ generated by $\equiv_\tau$. In other words, if we sweep the plane using a time line $t = \tau$, $\leqslant_\tau$ gives an ordering of all equivalence classes of objects along the sweep line.

The following lemma states a fundamental property of the total ordering of the curves along a time line. Intuitively, right before two curves intersect at time τ, they must become immediate neighbors in the total ordering. This is an important property used to detect intersections, similar to line segment intersection [3].

LEMMA 7. *Let $D = (O, T, \tau')$ be a* MOD*, f a polynomial g-distance, τ a time instant, and $o, o' \in O$. If $o \equiv_\tau o'$, then there exists a time instant $\tau_1 < \tau$ such that for all $t \in [\tau_1, \tau]$, one of the following is true:*

1. *$o \equiv_t o'$,*
2. *$o <_t o'$ and there exist no objects o'' between o and o', i.e., $\neg\exists o''(o <_t o'' <_t o')$, or*
3. *$o' <_t o$ and there exist no objects o'' between o' and o, i.e., $\neg\exists o''(o' <_t o'' <_t o)$.*

The following lemma shows that the precedence relation uniquely determines the anwser to a query.

LEMMA 8. *Let D be a* MOD*, f a polynomial g-distance, τ_1, τ_2 time instants, and $Q \in \mathrm{FO}(\mathsf{f})$ a query. If $\leqslant_{\tau_1}$ and $\leqslant_{\tau_2}$ (extended to all instantiated real terms in Q) are identical, then*

$$supp(Q, D, \tau_1) = supp(Q, D, \tau_2).$$

We now present the details of the algorithms. For simplicity, we first consider the special case where "t" is the only time term in queries. We will drop this restriction later.

Let $D = (O, T, \tau)$ be a MOD and Q a query whose interval starts at time τ. First we compute the precedence relation $\leqslant_\tau$, sort all objects in O using $\leqslant_\tau$, and store them in a sorted list, which we call the *object list* L. For simplicity, we assume that there are no pairs of objects o, o' such that $o \equiv_\tau o'$. (Generalization is straightforward.) We obtain the support to Q at time τ, i.e., $supp(Q, D, \tau)$. The remaining steps of the evaluation resemble in a way the plane sweeping algorithm for line segment intersection [3].

Note that the precedence relation changes if and only if one of the following happens:

1. an update is performed (object creation or termination), or
2. for a pair of objects o, o', the curves f_o and $\mathsf{f}_{o'}$ intersect.

As discussed ealier, updates are external events and the time of an update is known when the request is made. The second type of changes are implicit. It is necessary to find the intersection time before it happens. Fortunately, by Lemma 7, before each pair of curves intersect, the curves (objects) must be immediate neighbors in the precedence relation.

Based on Lemma 7, we first determine if each pair of immediate neighbors o, o' (in terms of the precedence relation

$\leqslant_\tau$) will intersect, i.e., the curves f_o and $\mathsf{f}_{o'}$ will intersect at some time later than τ. If they will, the intersection time is computed (or approximated[1]). We maintain all intersection times sorted in an ascending order in the *event queue* E.

The query support computed for time τ, $supp(Q, D, \tau)$, is not actually changed until the next update δ arrives at some time $\tau_1 > \tau$. At this point, the following two steps are performed:

First, τ_1 is inserted into the event queue E.

Then, for each event ahead of τ_1 in the queue E, the following are performed (in the order of their times). Each event represents the intersection time τ' of the curves of two objects o, o'. Without loss of generality, we assume that o precedes o' currently, $o <_\tau o'$.

1. Update the precedence relation from $\leqslant_\tau$ to $\leqslant_{\tau'}$ where $o \equiv_{\tau'} o'$ (they are equivalent); propagate the change to obtain $supp(Q, D, \tau')$.
2. Update the precedence relation $\leqslant_{\tau'}$ to $\leqslant_{\tau'+\epsilon}$ so that $o' <_{\tau'+\epsilon} o$ for some sufficiently small[2] $\epsilon > 0$ (o and o' complete the switching of their order); again, propagate the change to obtain $supp(Q, D, \tau' + \epsilon)$. (Note that $\tau' + \epsilon$ is after the intersection.)
3. Since the neighborhoods of o, o' are changed. We need to determine if o and o' intersect with their new immediate neighbors in L. If they do, the intersection times are inserted into the event queue E.

After the above steps are done, we increment the time in the MOD from $D = (O, T, \tau)$ to $D' = (O, T, \tau' + \epsilon)$. Since the precedence relation $\leqslant_{\tau'+\epsilon}$ has already been computed, we proceed to the next event in the queue E.

In case there are multiple events in the queue with the same time, the precedence relation is modified before the propagation is done in the first two steps shown above.

After all intersection events ahead of the update δ (at time τ_1) are processed, the algorithm will perform the update and make necessary changes to the support. Let $D'' = (O, T, \tau'')$ be the MOD at this point where $\tau'' < \tau_1$. Clearly, $\delta D = \delta D''$. We perform the update and proceed according to the update type:

- $\delta = \mathsf{new}(o_{\text{new}}, \tau_1, A, B)$.

  We compute $\mathsf{f}_{o_{\text{new}}}(\tau_1)$ and insert o_{new} into the object list L. We then propagate the changes to $supp(Q, \delta D'', \tau_1) = supp(Q, \delta D, \tau_1)$. We will examine the immediate neighbors of o_{new} in L and insert each intersection time into the event queue E.

- $\delta = \mathsf{terminate}(o_{\text{old}}, \tau_1)$.

  We delete o_{old} from the object list and propagate the changes to obtain $supp(Q, \delta D'', \tau_1)$. We also delete all events in the queue for object o_{old}. The two immediate neighbors of o_{old} in L now become immediate neighbors, we have to determine if they intersect and, if so, insert the intersection time into the event queue E.

- $\delta = \mathsf{chdir}(o, \tau_1, A)$.

  In this case, the object o_{old} simply changes its direction and speed. Let e and e' be the trajectory of o_{old} before and after the update (respectively). Clearly e and e' coincide up to time τ_1. Thus the new value of the g-distance on o_{old} is identical to the old value up to time τ_1. The precedence relation does not change at time τ_1, and there are no changes to the support. However, we have to delete all events in the queue E related to o_{old}, and re-examine possible intersections with its immediate neighbors.

Note that in the above algorithm, processing each event and/or update may require a constant number of executions for computing the intersection of two g-distance curves f_o and $\mathsf{f}_{o'}$ This can be done by known algorithms (e.g., [21]). In the following analysis, we exclude the complexity for intersection computation.

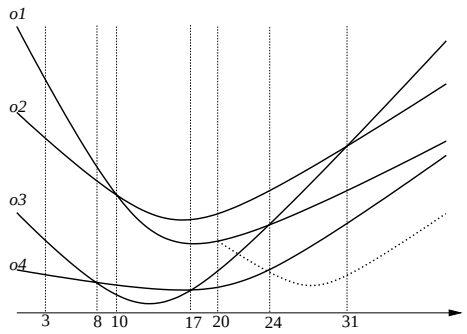

Figure 3: Curves of four objects

EXAMPLE 12. Figure 3 shows the g-distances for four objects $o_1, ..., o_4$. Consider the 2-NN query during the time interval $[0, 40]$. Suppose the current time is 3.

- We evaluate the g-distance curves for time 0 and discover that the ordering is $o_4 < o_3 < o_2 < o_1$. The answer up to time 3 is o_3 and o_4. In addition, by checking the immediate neighbor pairs (o_i, o_{i+1}), we found three future intersection points at times 8 (o_3, o_4), 10 (o_1, o_2), and 31 (o_2, o_3), which are inserted into the event queue. Note that the second intersection point at time 17 of o_3, o_4 is ignored for the moment.
- Suppose that the next update comes at time 20, the update event is inserted into the event list after 8 and 10, but before 31. We will process all events before 20 and then perform the update.
- For the intersection at time 8, o_3 and o_4 will switch positions in the ordering. At this point we will reexamine future intersection points and enter 17 in the event queue that will be ahead of the update event.
- For intersection point 10, o_1 and o_2 will switch positions, and no intersections are found between new neighbors.

[1]In this case, the query answers are approximated around the intersection point.

[2]$\tau' + \epsilon$ should be earlier than the next event in the event queue.

- For intersection at time 17, o_3 and o_4 will switch positions again and this time the intersection at 24 is found since o_1 and o_3 are neighbors.
- We now proceed to perform update, which is the next event. Suppose the update is to change the direction of o_1 and as a result the g-distance curve for o_1 becomes the dashed line in the figure. In this case, the support for the query is unchanged since the ordering is not. However, we need to delete from the event queue the intersection event at 24 and insert a new intersection point that is earlier. ∎

LEMMA 9. *For an N-object* MOD*, each event in the event queue can be processed in $O(\log N)$ time.*

PROOF. (Sketch) We first claim that each event and update can be processed in $O(\log m + \log N)$ time where m is the length of event queue. This follows from the choice of the data structures for the object list and event queue. For the object list, the operations include insertions and deletions of objects. A balanced binary search tree (such as AVL or red-black tree) [30] is sufficient to obtain $O(\log N)$ bounds on each object list change. For the event queue, a natural choice is a priority queue (heap). However, this does not work due to a subtle issue that in processing terminate or chdir it is necessary to delete all events related to one object.

The problem can be overcome by keeping in the event queue only the closest intersection times for each pair of *current* immediate neighbors. This is done in two parts. First, for each pair of immediate neighbors, only the earliest (future) intersection time is inserted into the queue. Second, when two objects are no longer immediate neighbors, their intersection time is deleted from the queue. Deletion from the heap requires pointers from objects in the object list. To avoid updating such pointers when a heap deletion or insertion is done, we can use a height biased leftist tree in place of a heap [30] or bi-directional pointers. As a result of this optimization, the queue length is at most N, i.e., $m \leqslant N$. Hence the lemma holds. ∎

A query in FO(f) may use k (some fixed number) polynomials as time terms. In this case, we construct one function for each pair of a trajectory and a time term. Since the total number of functions is only increased by a factor of k. The above results remain valid. For past queries, a turn in the MOD is treated as an update operation. Theorems 4 and 5 can be proved using Lemma 9 and the above analysis.

We make two remarks before the end of the section. First, the assumption that generlized distances are continuous can be relaxed to just require them to consist of a finite set of continuous pieces. The only change to the algorithm is to propagate changes to the support upon each chdir update. All results listed in this section remain true.

For the second remark, recall that constraint queries defined in Section 3 can have motions. This is implicitly described in the g-distances in FO(f). One interesting extension is to allow queries to have exactly the same trajectory as some object in the database. The interesting question that arises is how to deal with direction updates on the query object. Note that this would mean the g-distances are all changed for all objects, except that the current precedence relation (and this the support) remain correct. The following shows that such an update can still be dealt with in our framework. The argument is similar to that of Theorem 5, except that the precedence relation need not be recomputed (this avoids sorting).

THEOREM 10. *For each future query in* FO(f) *natually extended with an associated trajectory and each N-object* MOD *D, every* chdir *update on the query trajectory can be done in $O(N)$ time.*

6. CONCLUSIONS

In this paper we studied efficient evaluation techniques for moving object queries. Moving object queries can be location dependent (i.e., with associated trajectories), and also query about future temporal and spatial relationships among objects. We show that traditional constraint query languages are not appropriate for future queries. We propose a new query language based on generalized distances and show that the plane sweeping technique from computational geometry can be naturally adapted to evaluation of queries in this language, including future queries as well as past queries. However, the current language is limited in the sense that only one g-distance curve is allowed for each object. It is interesting to further investigate the expressiveness of the language in querying moving objects.

7. REFERENCES

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. ACM Symp. on Principles of Database Systems*, 2000.

[2] M. Ben-Or, D. Kozen, and J. Reif. The complexity of elementary algebra and geometry. *J. Comput. Syst. Sci.*, 32(2):251–264, Apr. 1986.

[3] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. on Computers*, 28:643–647, 1979.

[4] J. Chomicki and P. Z. Revesz. A geometric framework for specifying spatiotemporal objects. In *Proc. 6th Int. Workshop on Temporal Representation and Reasoning, (TIME '99)*, pages 41–46, Orlando, FL, May 1999.

[5] J. Chomicki and P. Z. Revesz. Parametric spatiotemporal objects. *Bulletin AI∗AI*, 14(1):41–47, Mar. 2001.

[6] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[7] G. E. Collins. Quantifier elimination for real closed fields by cylindric decompositions. In *Proc. 2nd GI Conf. Automata Theory and Formal Languages*, volume 35 of *Lecture Notes in Computer Science*, pages 134–83. Springer-Verlag, 1975.

[8] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101–106, July 1995.

[9] G. Dong and J. Su. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *J. Comput. Syst. Sci.*, 57(3):289–308, Dec. 1998.

[10] M. Erwig, R. H. Guting, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: an approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[11] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2000.

[12] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1995.

[13] A. Gupta and I. S. Mumick. *Materialized Views*. MIT Press, 1999.

[14] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 157–166, 1993.

[15] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Varirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1), 2000.

[16] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995.

[17] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 261–272, 1999.

[18] G. Kuper, L. Libkin, and J. Paredarns, editors. *Constraint Databases*. Springer Verlag, 2000.

[19] R. S. Millman and G. D. Parker. *Elements of Differential Geometry*. Prentice-Hall, Edgewood Cliffs, NJ, 1977.

[20] J. O'Rourke. *Computational Geometry in C*. Cambridge Univ. Press, 1998.

[21] V. Pan. Complexity of computations with matrices and polynomials. *SIAM Review*, 34(2):225–62, 1992.

[22] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proc. Int. Conf. on Very Large Data Bases*, 2000.

[23] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[24] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. *Journal of Symbolic Computation*, 13:255–352, 1992.

[25] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. Int. Conf. on Data Engineering*, 1997.

[26] Z. Song and N. Roussopoulos. k-nearest neighbor search for moving query point. In *Proc. Int. Sym. on Spatial and Tempporal Databases*, pages 79–96, 2001.

[27] J. Su, H. Xu, and O. Ibarra. Moving objects: Logical relationships and queries. In *Proc. Int. Sym. on Spatial and Tempporal Databases*, pages 3–19, 2001.

[28] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, California, 1951.

[29] M. Vazirgiannis and O. Wofson. A spatiotemporal model and language for moving objects on road networks. In *Proc. Int. Sym. on Spatial and Tempporal Databases*, 2001.

[30] M. A. Weiss. *Data Structures & Algorithm Analysis in C++*. Addison-Wesley, 1998.

[31] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: issues and solutions. In *Proc. Int. Conf. on Statistical and Scientific Database Management*, 1998.