Cost and Imprecision in Modeling the Position of Moving Objects[*]

Ouri Wolfson[†] Sam Chamberlain[‡] Son Dao[§] Liqin Jiang[¶] Gisela Mendez[||]

Abstract

Consider a database that represents the location of moving objects, such as taxi-cabs (typical query: retrieve the cabs that are currently within 1 mile of 33 Michigan Ave., Chicago), or objects in a battle-field. Existing database management systems (DBMS's) are not well equipped to handle continuously changing data, such as the position of moving objects, since data is assumed to be constant unless it is explicitly modified.

In this paper we address position-update policies and imprecision. Namely, assuming that the actual position of a moving object m deviates from the position computed by the DBMS, when should m update its position in the database in order to eliminate the deviation? Furthermore, how can the DBMS provide a bound on the error (i.e. the deviation) when it replies to a query: what is the current position of m? We propose a cost based approach to update policies that answers both questions. We develop several update policies and analyze them theoretically and experimentally.

1 Introduction

Consider a database that represents the location of moving objects. For example, for a database representing the location of taxi-cabs a typical query may be: retrieve the free cabs that are currently within 1 mile of 33 N. Michigan Ave., Chicago (to pick-up a customer); or for a trucking company database a typical query may be: retrieve the trucks that are currently within 1 mile of truck ABT312 (which needs assistance); or for a database representing the current position of objects in a battlefield a typical query may be: retrieve the friendly helicopters that are currently in a given region. The database may be either centralized or distributed, and the queries may originate from the moving objects, or from stationary users.

To manage such databases by a traditional database management system (DBMS) would mean that the moving objects (e.g. vehicles) have to continuously send to the database updates of their position via a wireless communication link. [1] Frequent updating may be expensive in terms of dollar-cost, or performance and wireless-bandwidth overhead. Alternatively, if position updates are infrequent, then the answer to position queries is outdated, i.e. imprecise; this in turn may involve a penalty cost in terms of incorrect decision-making. One of the contributions of this paper is to establish the update frequency as a function of the ratio between the update cost and the imprecision-in-answering-queries cost. Namely, the update frequency increases as the imprecision cost increases, and it decreases as the update-cost increases.

Furthermore, we propose a technique to drastically reduce the update cost. Specifically, we propose to model the current position of a moving object as the distance from its starting position, along a given route. The distance continuously increases as a function of time, without being updated. So, for example, the DBMS knows that the moving object m started at 5pm at position (x_0, y_0) on a given route known to the DBMS, and it travels at 60 miles/hour; thus, at any point in time after 5pm, in response to a query, the DBMS can easily compute the current position of m. Our simulation experiments show that this technique reduces the number of updates to 15% of the number used by the traditional, nontemporal method; this saves 85% of the bandwidth and update-processing overhead.

In the context of position attributes that change continuously with time, the main issues we address in this paper are position-update policies and imprecision. Namely, assuming that the actual position of m deviates from the position computed by the DBMS (namely the database position) due to the fact that m does not travel continuously at *exactly* 60 mi/hr, how frequently should m update its position in the database in order to eliminate the deviation? Furthermore, how can the DBMS provide a bound on the error (i.e. the deviation) when it replies to a query: what is the current position of m?

We propose the following cost based approach to update policies. We postulate that there is a given cost for each unit of deviation (i.e. imprecision), and there is a given cost of a database update. Then, at

[†]Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607

[‡]Army Research Laboratory, Aberdeen Proving Ground, MD

[§]Hughes Research Laboratories, Information Sciences Laboratory, Malibu, CA

[¶]Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607

[||]Central University of Venezuela

[*]This research was supported in part by Grants NSF IRI-9408750, DARPA N66001-97-2-8901, NATO CRG-960648, ARL DAAL01-96-2-003, and a gift from Hughes Research Labs.

[1]We assume that at any point in time each vehicle knows its exact current position, using, for example, an onboard Geographic Positioning System (GPS). Position-update messages can be carried through wireless data communication channels, e.g. those provided by cellular data communication companies such as RAM Mobile Data Co., satellite systems such as Iridium, and multihop radio networks.

any point in time a moving object approximates the deviation by curve fitting, using a simple estimator function of time. The estimator function is used to predict the future deviation assuming that the moving object sends now a database position update, and also to predict the future deviation in the absence of such a message. An update is sent if the difference between the deviation-costs exceeds the update cost. Then, an *update policy* for a moving object m consists of: a deviation cost function, an update cost, an estimator function, a fitting method, and a predicted speed. The deviation cost function is the cost of imprecision in answering queries. The estimator is a simple function used to approximate the deviation as a function of time. For example, a possible estimator is the function $f(t) = at$, where a is a constant and t is the number of time units since the last update. The fitting method is the method used to approximate the deviation by the estimator, e.g. for the function $f(t) = at$ it is the method of computing the constant a based on the deviation. The update speed is the speed declared to the database in the update, e.g. the current speed, or the average speed since the last update.

In this paper we introduce, analyze, and evaluate by simulation three update policies. Furthermore, we show that if the DBMS knows the update policy used by a moving object m, then it can compute at any point in time t a bound B on the deviation of m from its database position. B is the uncertainty in response to the following query entered at time t: what is the current position of m? In other words, the actual position of m may deviate from the position returned by the DBMS by at most B. Our simulations compare the three update policies in terms of total cost and uncertainty.

Then we address the indexing of position attributes. The objective is to enable the DBMS to answer in sublinear time, i.e. without examining all the objects, range queries of the form: retrieve the moving objects that are currently in a given polygon P. The problem with a straight-forward use of spatial indexing is that since objects are continuously moving, the position continuously changes, and thus the spatial index has to be continuously updated, an unacceptable solution. The indexing method that we propose avoids this problem by representing the range of current possible positions of a moving object as a plane in 3-dimensional time-space. Then, the processing of queries on the position attribute is translated into the problem of intersection of geometric bodies in this time-space. The indexing method enables the retrieval of both, moving objects that "must be" in P, and moving objects that "maybe" in P. The method uses our results concerning the uncertainty of each object at each point in time.

In summary, the contributions of this paper are as follows:

- We quantify the database update frequency as a function of the ratio between the update cost and the imprecision cost.
- We introduce the concept of update policies to be used by a moving object to determine when to update its database position, and what should be the update values.
- We introduce, analyze, and evaluate by simulation three update policies.
- We show that for each one of the above policies the DBMS can obtain a reasonable bound on the error (i.e. the deviation of the database position from the actual position).
- We introduce a method of indexing position attributes, i.e. attributes whose value changes continuously as a function of time;

The rest of the paper is organized as follows. In section 2 we introduce the position-attribute concept. In section 3 we discuss imprecision, introduce three update policies and their error bounds, and evaluate them by simulation. In section 4 we introduce our indexing method. In section 5 we compare our work to relevant research, and in section 6 we conclude with a discussion of our results.

2 Position attributes

A *database* is a set of object-classes. An *object-class* is a set of attributes. Some object-classes are designated as *spatial*. Each spatial object class is either a point-class, a line-class, or a polygon-class.

Point object classes are either mobile or stationary. A point object class O has a *position attribute* P. If the object class is *stationary*, its position attribute has two sub-attributes $P.x$, and $P.y$, representing the x and y coordinates of the object. If the object class is *mobile*, its position attribute has seven sub-attributes, *P.starttime*, *P.route*, *P.x.startposition*, *P.y.startposition*, *P.direction*, *P.speed*, and *P.policy*.

The semantics of the sub-attributes are as follows. *P.route* is (the pointer to) a line spatial object indicating the route on which an object in the class O is moving. [2] We assume that the database stores a set of routes, and at any point point in time each object moves along a unique route from the route database. *P.x.startposition* and *P.y.startposition* are the x and y coordinates of a point on *P.route*; it is the position of the moving object at time *P.starttime*. In other words, *P.starttime* is the time when the moving object was at position (*P.x.startposition*, *P.y.startposition*). We assume that whenever a moving object updates its P attribute it updates the *P.x.startposition* and *P.y.startposition* subattributes; thus *P.starttime* is also the time of the last position-update. We assume in this paper that the database updates are instantaneous, i.e. valid- and transaction- times (see [9]) are equal. Therefore, *P.starttime* is the time at which the update occurred in the real world system being modeled, and the time when the database installed the update. *P.direction* is a binary indicator having a value 0 or 1 (these values may correspond to north-south, or east-west, or the two endpoints of the route). *P.speed*

[2]For simplicity, our discussion pertains to routes in 2-dimensional space, but our concepts and results can be extended to routes in 3-dimensional space.

is a linear function of a single variable t that has value 0 at $t = 0$. It indicates the speed of the moving object. *P.policy* indicates the position-update policy used by the moving object. Position-update policies are discussed in section 3. At this point we will just mention that the bound on the error in computing the current position of a moving object depends on the policy, thus this information can be derived from the *P.policy* sub-attribute.

We define the *route-distance* between two points on a given route to be the distance along the route between the two points. Assuming that the route is given by a piece-wise linear function, it is straightforward to compute the route-distance between two points on the route, and the point at a given route-distance from another point. The *database position* of a moving object at a given point in time is defined as follows. At time *P.starttime* the database position is the pair (*P.x.startposition*, *P.y.startposition*); the database position at time $A.starttime + t_0$ is the point on the route *P.route* which is at route-distance $P.speed \cdot t_0$ from the point with coordinates (*P.x.startposition*, *P.y.startposition*). Intuitively, the database position of a moving object at a given point in time t is the position of the object as far as the DBMS knows; it is the position that is returned by the DBMS in response to a query entered at time t. As we shall see in the next section, the under certain conditions DBMS will also be able to provide a bound on the error, i.e. the difference between the actual position of the object and its database position.

3 Modeling imprecision

This section is divided into four subsections. In subsection 3.1 we introduce the concept of a database update policy for a moving object. In subsection 3.2 we discuss two particular update policies, the delayed- and immediate- linear. In section 3.3 we derive error bounds for these policies, i.e. bounds on the error of an answer to a query of the form: retrieve the current position of moving object m. In section 3.4 we introduce the third update policy, describe the simulation setup, and discuss the simulation results.

3.1 Database update policies

We assume that at the beginning of the trip the moving object writes all the sub-attributes of the position attribute. Subsequently, the moving object periodically updates its current position and speed stored in the database. Specifically, a *position update* is an update sent by the moving object to the database; it consists of values for at least the sub-attributes *P.starttime*, *P.speed*, *P.x.startposition* and *P.y.startposition*. If during the trip the object changes its route, then it sends a position update message that includes the identification of the new route to be stored in *P.route*. If we define the route distance between two points on different routes to be infinite, then this will trigger a position update whenever the object changes routes; we will not elaborate on this further.

A position update policy is a position-update prescription for a moving object. It states when the moving object updates its position in the database, and what are the update values. We propose a cost-based approach to update policies. Formally, a *position-update policy* of the moving object is a quintuple (deviation cost function, update cost, estimator function, fitting method, predicted-speed). For the rest of this subsection we define and discuss the components of the position update policy.

Since between two consecutive position updates the moving object does not travel at exactly the speed *P.speed*, the actual position of the moving object deviates from its database position. Formally, the *deviation d* at a point in time t, denoted $d(t)$, is the route-distance between the moving object's actual position at time t and its database position at time t. The deviation is always nonnegative. Note that at any point in time the moving object knows its current position, and it knows the parameters of the last position-update. Therefore at any point in time the (computer onboard the) moving object can compute the current deviation.

The cost of the deviation between two time points t_1 and t_2 is given by the *deviation cost function*, denoted $COST_d(t_1, t_2)$; it is a function of two variables that maps the deviation between the time points t_1 and t_2 into a nonnegative number. For example, suppose that the penalty for each unit of deviation reported in response to a query is 1; and on average, there is one query that retrieves the position of the moving object per time unit. Then, the cost of a unit of deviation per unit of time is one, and the cost of the deviation between two time points t_1 and t_2, is:

$$COST_d(t_1, t_2) = \int_{t_1}^{t_2} d(t)dt \qquad (1)$$

We call the function of equation 1 the *uniform* deviation cost function.

In this paper we consider only update policies that have the uniform deviation cost function. However, there exist other deviation cost functions. For example, the step deviation cost function carries a zero-penalty for each time unit in which the deviation stays below some threshold h, and a penalty of one otherwise.

The *update cost*, denoted C, is a nonnegative number representing the cost of a position-update sent from the moving object to the database. The update cost may differ from one moving object to another, and it may vary even for a single moving object during a trip, due for example, to changing availability of bandwidth. The update cost must be given in the same units as the deviation cost. In particular, for the uniform deviation cost function, C is the ratio between the update cost, and the cost of a unit of deviation per unit of time. For example, the cost of a wireless message using one of the wireless data transmission services (e.g. RAM mobile data Co. or ARDIS Co.) is 5 cents. Thus, if the the cost of a unit of deviation per unit of time is one cent, then $C = 5$.

Let t_1 and t_2 be the time-stamps of two consecutive position updates. Then the position update policy takes the total cost between t_1 and t_2 to be:

$$COST(t_1, t_2) = C + COST_d(t_1, t_2) \qquad (2)$$

The *estimator function* (see [7]) is a "well-behaved" function $f(t)$ by which we approximate the current deviation, i.e. the deviation at any time unit since the last update; we restrict $f(0) = 0$ because the update includes the current position of the moving object, and therefore, zero time units after the update the deviation is zero. For example, given a deviation $d(t)$ for $0 \leq t \leq currenttime$, the estimator can be the linear function $f(t) = a \cdot t$.

The *fitting method* is the method of determining the coefficients of the estimator function based on $d(t)$. For example, for the estimator $f(t) = at$, a fitting method is to compute a as the ratio between the current deviation and the number of time units since the last update. This means that if at time point t_0 the deviation $d(t_0) = k$, $d(t)$ is estimated by the straight line l, where the slope of l is $a = \frac{k}{t_0}$.

The *predicted-speed* is the speed that will be stored in the subattribute *P.speed* at each update. It represents the speed of the moving object following the update. One possible value for the predicted speed is the current speed of the moving object, another is its average speed since the last update, and yet another is the average speed since the beginning of the trip. These values are based solely on the past. However, the moving object may also be able to estimate the future speed based on known traffic patterns, or based on a priori knowledge of the upcoming terrain, or based on input from the user.

At any point in time t_0, using the update policy, the moving object decides whether or not to send a database update. It does so as follows. Assume that at time t_0 the deviation is k. Using the estimator function and the fitting method, it approximates the current deviation by a function $g(t)$ of the number of time units elapsed since the last update. Then it assumes that the future deviation, i.e. the deviation t time units after t_0, is given by either: a) $g(t)$ if an update is sent at time t_0, or b) $g(t) + k$ if an update is not sent at time t_0. For example, let t_0 be the number of time units that currently elapsed since the last update. Suppose that the deviation at time t_0 is k, i.e. $d(t_0) = k$, and the estimator function is $f(t) = at$ where $a = \frac{k}{t_0}$. If the moving object does not update the database at time t_0, then it is assumed that t time units after t_0 the deviation will be $\frac{k}{t_0} \cdot t + k$. If the moving object does update the database at time t_0, then it is assumed that t time units after t_0 the deviation will be $\frac{k}{t_0} \cdot t$.

Now, using the deviation cost function and the update cost, the moving object can compute whether or not t_0 is an optimal update point; namely, if an update at time t_0 minimizes the total cost under these assumptions. Exactly how to determine whether or not t_0 is an optimal update point will be shown in the next subsection for an estimator function that is slightly richer than l.

Clearly, there is an infinite number of update policies, one for each combination of the quintuple components. Observe that, since the update policy is a position subattribute, each position update may change the policy. One reason to change the policy on an update is that the most appropriate policy may be different for different speed patterns. For example, a policy for which the predicted speed is the current speed may be appropriate for highway driving in nonrush hour (when the speed fluctuates only mildly), whereas a policy for which the predicted speed is the average speed may be appropriate for city driving, where the speed fluctuates sharply. The pattern of the current speed is a parameter that may be entered by the user, and changed during a trip.

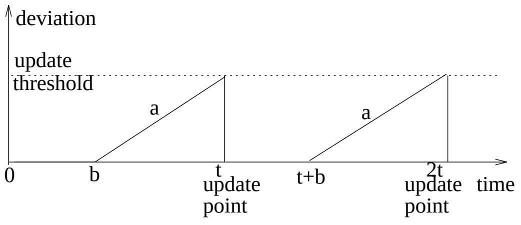

Figure 1: The deviation as a delayed linear function

3.2 The delayed and immediate linear update policies

We will use the delayed linear function, defined next, as the estimator function. A *delayed linear* function f is a function having the following formula.

$$\mathrm{f}(t) = \begin{cases} a(t-b) & \text{if } t \geq b \\ 0 & \text{if } 0 \leq t < b \end{cases}$$

where a, b are non negative constants. b is called the *delay* of f, and a is called the slope of f.

The rationale for this estimator function is the following. In each update the object sends to the database its current position and current speed to update its (*P.x.startposition*, *P.y.startposition*) and *P.speed* sub-attributes respectively. Taking the deviation to be a delayed linear function means that following an update, the deviation is 0 for b units of time (in other words, the moving object continues at the declared speed for b time units), and then it starts increasing at a rate of a, i.e. according to a linear function whose slope is a. In other words, after b time units it starts moving at a speed v, such that $a = |v - P.speed|$. Thus, $d(t)$ has a delay of b and a slope of a.

Assume that for given a and b numbers, the deviation of a moving object following each database update is a delayed linear function with delay b and slope a. Then each update occurs when the deviation reaches the same value. Let the *update threshold* be that value, i.e. the value of the deviation for which the moving object updates the database (see fig. 1). Let the *optimal update threshold*, denoted $k_{opt}^{a,b}$, be the value of the update threshold for which the total cost per time unit is minimized. In other words, the total cost per time unit is minimized when the moving object updates the database each time the deviation reaches the optimal update threshold.

Proposition 1: Let a and b be nonnegative numbers. Assume that following each database update the deviation of a moving object is a delayed linear function with delay b and slope a. Assume that the update cost is C. Then, for the uniform deviation cost function, the optimal update threshold $k_{opt}^{a,b} = \sqrt{a^2b^2 + 2aC} - ab$.

Example 1: Some real-world values of the variables in the optimal update threshold are as follows. Assume that the cost for a 1-mile deviation is 1 cent per minute, and $C = 5$. Suppose that a vehicle sets its speed on a route to be 1 mile per minute (60 mi/hr). Suppose further that the vehicle travels at that speed for 2 minutes, and then it stops (i.e. the delay is 2) in a traffic jam. Stopping increases the vehicle's deviation at a rate (slope) of 1 mile per minute. Then, $\sqrt{a^2b^2 + 2aC} - ab = 3.74 - 2 = 1.74$. Namely, if the vehicle uses the delayed linear policy, it will send a database update when its deviation is 1.74 miles, i.e. after it has been stopped for one minute and 44 seconds. $\square$

Now, the deviation will not always be a delayed-linear function. Therefore, we will use a delayed-linear function as an estimator of the deviation. The fitting method used to evaluate the slope and deviation at any point in time is called the *simple fitting method* and is defined as follows. At any point in time, the delay b is the number of time units from the last update until the last time unit when the deviation was 0. The slope a is the ratio between the current deviation and $t - b$, where t is the time elapsed since the last update.

The **delayed-linear (dl)** position-update policy is the quintuple (uniform deviation cost function, update cost C, delayed-linear estimator function, simple fitting method, current-speed). Then, based on proposition 1, the moving object updates the database whenever the deviation reaches $k_{opt}^{a,b} = \sqrt{a^2b^2 + 2aC} - ab$.

Intuitively, the delayed-linear position update policy behaves as follows. At any point in time the moving object computes the current deviation, k; if $k = 0$, then the moving object does not do anything, i.e. it does not consider a position-update. Otherwise it computes the delay b and the slope a as follows. b is the number of time units from the last update until the last time unit when the deviation was 0; and if we denote by t the number of time units elapsed since the last update, then $a = \frac{k}{t-b}$ (clearly, since the current deviation is not 0, $t-b > 0$). Now, if $k \geq \sqrt{a^2b^2 + 2aC} - ab$, then the moving object updates the database with the current position and current speed; i.e. it places its current position in the *P.x.startposition* and *P.y.startposition* subattributes, and its current speed in the *P.speed* subattribute.

Observe that the delay and slope may change from one update-to-update period to another. By updating at the *current* optimal update threshold, i.e. the threshold for the current delay and slope, the moving object makes the implicit assumption that the current delay and slope will persist for the next update-to-update period. In other words, if an update-to-update period is a window, then the parameters of the current window are projected onto the next window. This paradigm, i.e. the assumption that the recent past is indicative of the near future, is common in computer science, as evidenced by many resource allocation policies (e.g. the LRU page replacement strategy).

To motivate the next update policy assume that at each update, the moving object does not send its current speed, but its average speed since the last update. This makes sense when the current speed changes rapidly (as in stop-and-go city driving), but the average speed is stable. Since the average speed most likely is different than the current speed, the deviation will start increasing immediately. Thus, another estimator function that we consider in this paper is the *immediate linear:* it is the delayed linear function with a delay of 0. Namely, the immediate linear function has the formula $f(t) = at$ for $t \geq 0$.

The **average immediate-linear (ail)** position-update policy consists of the quintuple (uniform deviation cost function, update cost C, immediate-linear estimator function, simple fitting method, average-speed). Then, based on proposition 1, the moving object updates the database whenever the deviation reaches $k_{opt}^{a,0} = \sqrt{2aC}$.

Intuitively, the average immediate-linear update policy behaves as follows. At any point in time the moving object computes the current deviation, k; if $k = 0$, then the moving object does not do anything. Otherwise it computes the slope a as $a = \frac{k}{t}$, where t is the number of time units elapsed since the last update. (In other words, it is approximated that if the deviation increased from 0 to k since the last position update, in the absence of an update at the current time, the deviation will continue to increase at the same rate; if an update is sent, then the deviation will increase at the same rate starting from 0). Then, if $k \geq \sqrt{2aC}$ the moving object updates the database by placing its current position in the *P.x.startposition* and *P.y.startposition* subattributes, and its average speed since the last update in the *P.speed* subattribute.

Now, we will make a few observations. First, it is easy to see that for given positive a and b, $k_{opt}^{a,b} = \sqrt{a^2b^2 + 2aC} - ab \leq \sqrt{a^2b^2} + \sqrt{2aC} - ab = k_{opt}^{a,0}$. This may seem to indicate that if $b > 0$, i.e. if the moving object continues at the current speed after the update, then the threshold for the delayed linear policy is lower than that for the average immediate linear; which in turn indicates that the delayed linear policy performs more updates. However, this is deceiving. One reason for this is that the delayed policy updates *P.speed* with the current speed, which is usually different than the average speed used by the immediate policy. Therefore the deviation function for the two policies differs. What if we consider the **current immediate-linear** position-update policy that is similar to the average immediate-linear, except that the speed used in the update is the current rather than the average? (We analyze the cil policy in section 3.4). Even then, for a given deviation function $d(t)$, the slope a is $\frac{k}{t-b}$ for the delayed linear policy, and $\frac{k}{t}$ for the average immediate linear policy. It can be shown that the number of updates under the two policies is incomparable in general.

Second, consider the average immediate policy. It is easy to see that since $a = \frac{k}{t}$, $k \geq \sqrt{2aC}$ if and only if $k \geq \frac{2C}{t}$. Thus, for the simple fitting method:

$$k_{opt}^{a,0} = \frac{2C}{t} \tag{3}$$

where t is the number of time units elapsed since the last update. Therefore, $k_{opt}^{a,0}$ decreases as time passes without an update. This means that, assuming that the deviation decreases slower than time increases, the moving object may generate a position update while the deviation is decreasing.

3.3 Threshold bounds

For each moving object in the database the DBMS knows the update policy at any point in time. Consider a particular moving object o. If o uses the immediate or delayed update policies, then at any point in time the DBMS can compute the current database position of o. However, the actual position of o may deviate from its database position by the optimal update threshold. At any point in time the optimal update threshold depends on the current slope (and delay), which are unknown to the DBMS. Thus the DBMS cannot compute the the actual position of o. Nevertheless, the DBMS can determine a bound on the current deviation of o; in this subsection we show how it can do so.

Remember that the deviation $d(t)$ is the route-distance of the vehicle's actual position from its database position. If the actual position is closer to the starting position than the database position, then we call the deviation *slow* (i.e. the object is behind its position as reflected in the database), otherwise we call it *fast*.

Proposition 2: Assume that a moving object uses the delayed linear update policy, with update cost C. Assume that at some point in time $P.speed = v$. Then, if that point in time is t time units after the last update, the slow-deviation k is bounded by $min\{\sqrt{2vC}, vt\}$, i.e. $k \leq min\{\sqrt{2vC}, vt\}$. $\square$

Intuitively, proposition 2 bounds how far the actual position can be behind the database position. Note that the DBMS can compute this bound based on values it knows, namely, v, C, and t.

Assume now that the maximum speed of the moving object during this trip is V, and that the DBMS knows V. V may be determined by the characteristics of the vehicle (e.g. it cannot go faster than 120 miles per hour), or it can be determined by the expected conditions of the trip (the vehicle will not go faster than 60 miles per hour during rush hour). Analogously to proposition 2, it can be shown that:

Proposition 3: Assume that a moving object having maximum speed V uses the delayed linear update policy, with update cost C. Assume that at some point in time $P.speed = v$. Then, if that point in time is t time units after the last update, the fast-deviation k is bounded by $k \leq min\{\sqrt{2(V-v)C}, (V-v)t\}$. $\square$

Clearly, at any point in time the deviation is either fast or slow. Then, an immediate corollary of propositions 2 and 3 is:

Corollary 1: Assume that a moving object having maximum speed V uses the delayed linear update policy, with update cost C. Assume that at some point in time $P.speed = v$, and denote $D = max\{v, V-v\}$. Then, if that point in time is t time units after the last update, the deviation k is bounded by $k \leq min\{\sqrt{2DC}, Dt\}$. $\square$

Example 1 (continued): As before, the cost for a 1-mile deviation is 1 cent per minute, and $C = 5$. Suppose that the current database speed ($P.speed$) is 1 (mile per minute), and the moving object is using the delayed linear update policy. Then the bound on the slow-deviation increases at the rate of 1 mile per minute for the first 3 minutes following the last update, and after that it remains constant at 3.16 miles; i.e. 10 or 15 minutes after the last update the slow-deviation will still be 3.16 miles. Suppose now that the maximum speed V is 1.5. Then the fast-deviation increases at the rate of 0.5 miles per minute for the first 4.5 minutes after the last update, and after that it remains constant at 2.24 miles. $\square$

Consider now the immediate linear update policy.

Proposition 4: Assume that a moving object having maximum speed V uses the ail policy, with update cost C. Assume that at some point in time $P.speed = v$, and that point in time is t time units after the last update. Then the slow-deviation s is bounded by $s \leq min\{\frac{2C}{t}, vt\}$; the fast-deviation f is bounded by $f \leq min\{\frac{2C}{t}, (V-v)t\}$. Let $D = max\{v, V-v\}$. The deviation k is bounded by $k \leq min\{\frac{2C}{t}, Dt\}$. $\square$

Proposition 4 indicates that following an update, the bound on the slow deviation first increases as time progresses, starting from 0, and it does so while $\frac{2C}{t} > vt$; after point in time t, $t = \sqrt{2C/v}$, in the absence of an update, the bound on the slow deviation decreases as time progresses. Similarly, the bound on the fast deviation first increases as time progresses, and it does so while $\frac{2C}{t} > (V-v)t$, and after point in time t, $t = \sqrt{2C/(V-v)}$, in the absence of an update, the bound on the fast deviation decreases as time progresses. This is a surprising positive result. In contrast, in the delayed linear policy, the bound on the error first increases, and then it remains fixed. We shall see in the next subsection that this is an important difference between the delayed and immediate policies, to the extent that it makes the immediate policy superior to the delayed one.

In the dead-reckoning method the bound on the error is fixed, i.e. it does not change as time following an update progresses.

Example 1 (continued): As before, the cost for a 1-mile deviation is 1 cent per minute, $C = 5$, $P.speed = 1$, and $V = 1.5$. Suppose that the moving object is using the immediate linear update policy.

Then, the bound on the slow-deviation increases at the rate of 1 mile per minute for the first 3 minutes following the last update, and after that it decreases, i.e. for $t \geq 4$, it is $10/t$. The fast-deviation increases at the rate of 0.5 miles per minute for the first 4.5 minutes after the last update, and after that it decreases, i.e. for $t \geq 5$, it is $10/t$. □

One last comment for this section is that if a user is dissatisfied because the bound on the deviation is too large, the DBMS can always (for a price?) contact a moving object to get its exact position. In other words, this paper addresses the processing of regular queries, i.e. ones for which the uniform deviation cost function applies; for priority queries, special processing is still available.

3.4 Update policies simulation

In this subsection we report on the simulation analysis of the two policies discussed previously, and a third, which is similar to the average immediate-linear policy, except that the predicted speed is the current one. Specifically, the **current immediate-linear (cil)** position-update policy consists of the quintuple (uniform deviation cost function, update cost C, immediate-linear estimator function, simple fitting method, current-speed). As for the ail policy, the update threshold for the cil policy is $\sqrt{2aC}$, and the deviation bounds are given in proposition 5.

The analysis compares the cost and uncertainty of the three policies on a set of one-hour trips. Each trip is represented by a speed-curve, i.e. the actual-speed of a moving object as a function of time. For each speed-curve, update policy, and update cost C we execute a simulation run that computes the total cost (a single number) and the average uncertainty (also a single number) of the policy on the curve for the given update cost. Then, for each policy, we average the total cost over all the speed curves, and plot this average as a function of the update cost C. We do the same for the average uncertainty and for the total number of messages.

The results of our simulations are summarized in a set of plots that quantify, for each policy, the number of position-update messages, total cost, and average uncertainty as a function of the message cost. Because of space limitations we omit these plots. However, they indicate that the ail policy is superior to the other policies.

4 Query processing and indexing of position attributes

The objective of the discussion in this section is to enable answering spatial (range) queries on the position attribute, i.e. queries of the form Q = "Retrieve the objects whose current position is in the polygon P". The problem is to evaluate such queries in sublinear time, i.e. without examining all the objects. The problem with a straight-forward use of spatial indexing for this purpose is that since objects are continuously moving, the spatial index has to be continuously updated, an unacceptable solution. In subsection 4.1 we formulate a 3-dimensional geometric representation of this retrieval problem. This will enable us to use spatial indexing, and we specify how to do so in subsection 4.2.

4.1 Retrieval as intersection of 3-dimensional objects

In this subsection we represent the problem of retrieval based on position attributes as a problem of intersection of geometric objects in 3-dimensional time-space. This time-space consists of the x and y spatial coordinates, with the third coordinate being time, t.

4.1.1 Geometric representation of the position attribute

We show here how we construct a plane, called the o-plane, for a given value of the position attribute of a moving object o. Specifically, given values for the position subattributes of o, the position of o is modeled by two functions of time. One function, called upper-o and denoted $u(t)$, represents the upper bound on the distance of the object from the starting position ($P.x.startposition$, $P.y.startposition$). In order to define $u(t)$, denote by $BF(t)$ the bound on the fast-deviation; depending on the position-update policy, $BF(t)$ is either $min\{\sqrt{2(V-v)C}, (V-v)t\}$ or $min\{\frac{2C}{t}, (V-v)t\}$ (see propositions 3 and 4. If we denote $P.speed = v$, the define $u(t) = vt + BF(t)$ where t is the number of time units since $P.starttime$. Since the object moves on a piecewise linear route, the x and y coordinates corresponding to a $u(t)$ distance from the starting position can be easily computed for any $t \geq 0$. Denote by $U(x,y,t)$ the line in 3-dimensional space defined as follows. $U(x,y,t)$ is the set of points that satisfy: x, y is at route-distance $u(t)$ from the starting position ($P.x.startposition$, $P.y.startposition$).

The other function, called lower-o and denoted $l(t)$, represents the lower bound on the distance of the object from the starting position ($P.x.startposition$, $P.y.startposition$). Denote by $BS(t)$ the bound on the slow-deviation; depending on the position-update policy, $BS(t)$ is either $min\{\sqrt{2vC}, vt\}$ or $min\{\frac{2C}{t}, vt\}$ (see propositions 2 and 4). If we denote $P.speed = v$, then define $l(t) = vt - BS(t)$ where t is the number of time units since $P.starttime$. Denote by $L(x,y,t)$ the line in 3-dimensional space defined as follows. $L(x,y,t)$ is the set of points that satisfy: x, y is at route-distance $l(t)$ from the starting position ($P.x.startposition$, $P.y.startposition$).

The *uncertainty interval* of o at time $t \geq 0$ is the line segment constituting the route between the points $l(t)$ and $u(t)$. Intuitively, as far the the DBMS knows, at time t the moving object o can be at any point in the uncertainty interval, and nowhere else. Let G be some polygon in 2-dimensional space. We say that moving object o *may be in* G at time $t \geq 0$ if the uncertainty interval of o at time t intersects G. We say that moving object o *must be in* G at time $t \geq 0$ if the uncertainty interval of o at time t lies in G in its entirety.

Let us define the *o-plane* to be the plane in 3-dimensional space which is bounded on one side by the

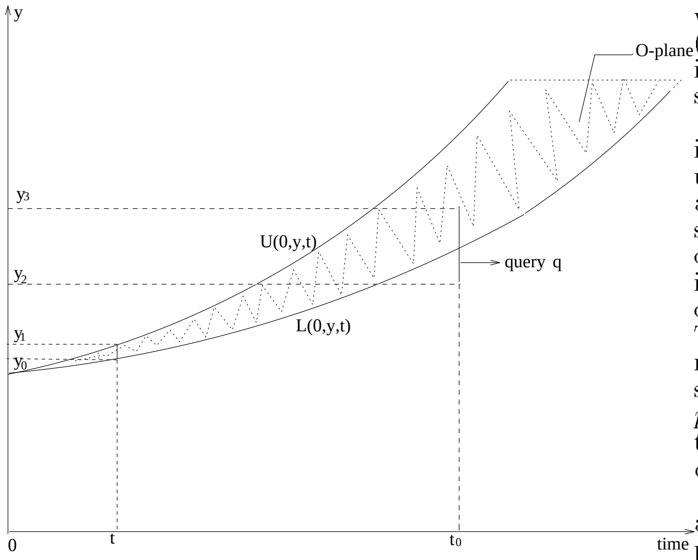

Figure 2: Object O is traveling along the y axis. (y0,y1) is the uncertainty interval at time t. The query q (represented by the solid line interval) is: retrieve the objects which at time t0 are at x=0 between y2 and y3.

line $L(x, y, t)$ and on the other by the line $U(x, y, t)$. In other words, the o-plane is the set of uncertainty intervals of o, one uncertainty interval for each time unit $t \geq 0$. See Figure 2 for an example of an o-plane, where the route is the y axis, i.e. the function $x = 0$.

4.1.2 Geometric representation of queries

Consider the query Q= "Retrieve the objects which are inside the polygon G at time t_0", where G is a polygon in 2-dimensional space. For time t_0, denote by $R_G(t_0)$ the set of 3-dimensional points (x, y, t_0) where x, y is in G. Intuitively, $R_G(t_0)$ is the polygon G at time t_0. It can be shown that:

Theorem 5: A moving object o may be in polygon G at time t_0 if and only if $R_G(t_0)$ intersects the o-plane.

Theorem 6: A moving object o must be in polygon G at time t_0 if and only if $R_G(t_0)$ intersects the o-plane, and both points $L(x, y, t_0)$ and $U(x, y, t_0)$ are in $R_G(t_0)$.

Thus the answer to the query Q consists of the set S of objects that may be in G, together with a subset of S consisting of the objects that must be in G.

4.2 Use of spatial indexing

For each position attribute of an object class we establish a 3-dimensional space consisting of the 2-dimensional geographic area of interest, and of a time span, T. The specific geographic area (e.g. metropolitan Chicago) and the time span (e.g. one day) depend on the application and on performance considerations that we intend to study in future work. When the geographic area and the time span are determined, we use a 3-dimensional spatial index, e.g. an R^+-tree (see [5] for a survey of spatial access indexes). Spatial indexes use a hierarchical recursive decomposition of space, usually into rectangles.

The index is updated whenever a position-update is received from a moving object o. Assuming that the update is received at time t, the update is processed as follows. Let $p1$ be the *old o-plane*, i.e. the o-plane starting at time t, and defined based on the old value of the position attribute. Let $p2$ be the *new o-plane*, i.e the o-plane starting at time t, and defined based on the newly received value of the position attribute, Then the id of o is removed from the 3-dimensional rectangles of the index that intersect $p1$, and it is inserted in the 3-dimensional rectangles that intersect $p2$. Observe that if there is an upper limit $Z < T$ on the time when o's trip will end, then $p1$ and $p2$ can be cut off at time Z.

Now consider the query "Retrieve the objects which are inside the polygon G at time t_0", where G is a polygon in 2-dimensional space. t_0 may be the current time, or some time in the future. Then, using the index, we retrieve the 3-dimensional rectangles that intersect $R_G(t_0)$. This can be done in sublinear time. For each object id o in these rectangles we compute its uncertainty interval s at time t_0. If s is contained in its entirety in G, then the object o is in G at time t_0. If s is partly in G and partly outside G, then the object o maybe in G at time t_0, or it may be outside G. Otherwise the object o is definitely outside G. Observe that although $R_G(t_0)$ intersects a rectangle E, it does not necessarily intersect the o-plane of every object o stored in E. Thus, there may be objects in E that at time t_0 are outside G.

5 Comparison to relevant work

We do not believe that the problems we addressed in this paper, namely position-update policies for moving objects and their cost and imprecision, have been studied before in the same context. Furthermore, the paper does not seem to fit neatly into an established field of research. Nevertheless, some research areas are relevant to the present work. One relevant research area is uncertainty in databases (see [3, 1] for surveys). However, as far as we know this area has so far addressed different issues than the ones in this paper. Existing works are concerned with management and reasoning with uncertainty, after such uncertainty is introduced in the database. Our current paper addresses the question: what uncertainty/deviation to initially associate with the location of each moving object? Other relevant research areas are temporal databases ([10]), and spatial databases (see [6] for a survey). Research in these areas can be used to develop languages to query the position attributes. For example, temporal and spatial query languages can be adapted to express queries such as: where will the helicopters be in 10 minutes. In this paper we addressed the questions how and when to update the position attributes.

Another relevant area is constraint databases (see [2] for a survey). In this sense, our position attributes can be viewed as a constraint, or a generalized tuple, such that the tuples satisfying the constraint are considered in the database. Constraint databases have been separately applied to the temporal domain, and to the spatial domain. Constraint databases can be used as a framework in which to implement the proposed update policies.

Finally, in our earlier work ([8]) we introduced dynamic attributes, which are somewhat similar to position attributes in the sense that they change continuously as a function of time. However, that paper dealt mainly with query languages for dynamic attributes; the main topics of this paper, i.e. position update policies, imprecision, and error-bounds, have not been discussed there. Also the indexing method we introduced in this paper is designed to handle imprecision, and is different from the method in [8]. Furthermore, using dynamic attributes for moving objects necessitates representing the x coordinate of an object as one dynamic attribute, and the y coordinate as another. However, this may be unsatisfactory if the object is moving along a winding route. In this case the speed along each coordinate may change very frequently (since changes in the direction of the motion vector result in changes in the projection of the motion vector on each one of the coordinates), necessitating frequent updates, even if the vehicle's speed remains constant.

6 Conclusion

In this paper we considered databases that model the location of objects moving on routes. These databases are expected to become common in military and transportation systems. We addressed three problems: first, bounding the position-uncertainty, i.e. the uncertainty of a reply to a query that retrieves the position of a particular moving object; second, reducing the position-update cost; and third, efficient retrieval of objects based on the current or future position.

We proposed the modeling of moving objects using position attributes, and have shown by simulation that this approach reduces the position update overhead by 85%. We formulated the database-position update problem as a mathematical cost optimization problem using the concept of a position-update policy, i.e. a quintuple (deviation cost function, update cost, estimator function, fitting method, predicted speed). Such a policy is adaptive and predictive in the sense that the update time-points depend on the current and predicted behavior of the deviation (of the actual position from the database position) as a function of time. Then we devised and analyzed three position update policies. We showed that the DBMS is able to bound the uncertainty at any point in time. Actually, for two of the policies (the immediate ones) the position-uncertainty decreases as time-since-the-last-update increases.

An alternative to our approach is to define a priori a bound B on the deviation, with a policy in which the moving object sends a position update message when the deviation exceeds B. The problem with this approach is that it is quite unlikely that B is totally independent of the update message cost.

Then we considered range queries on position attributes, i.e. queries that retrieve all the objects that are in a particular region at a particular time. We proposed a geometric formulation of such queries in 3-dimensional time-space. This formulation enables the processing of these queries in sublinear time, using spatial indexing.

We believe that as the world becomes a more dynamic place, as geographic distances are shrinking and remote locations of the globe become more accessible, and as new applications are developed, location-databases will become increasingly important. Much remains to be done in order to make these real-time databases a commercial reality. We intend to extend the present work by studying other update policies, building a simulation testbed to evaluate the performance of these policies, developing query languages and user interfaces for these databases, studying indexing, imprecision/uncertainty, distribution and data allocation in these databases.

References

[1] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison Wesley, 1995.

[2] P. Kanellakis, *Constraint programming and database languages*, ACM Symposium on Principles of Database Systems, May 1995.

[3] A. Motro, P. Smets, *Uncertainty Management Information Systems*, From Needs to Solutions, Kluwer Academic Publishers, 1997.

[4] R. Snodgrass and I. Ahn, *The temporal databases*, IEEE Computer, Sept. 1986.

[5] H. Samet, *The design and analysis of spatial data structures*, Addison Wesley, 1990.

[6] H. Samet, W.G. Aref, *Spatial Data Models and Query Processing*, In Modern Database Systems, Won Kim ed., Addison Wesley, 1995.

[7] S.D. Silvey, *Statistical Inference*, Chapman and Hall, 1975.

[8] P. Sistla, O. Wolfson, S. Chamberlain, S. Dao, *Modeling and Querying Moving Objects*, to appear, Proceedings of the Thirteenth International Conference on Data Engineering (ICDE13), Birmingham, UK, Apr.97.

[9] R. Snodgrass and I. Ahn, *The temporal databases*, IEEE Computer, Sept. 1986.

[10] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev,R. Snodgrass,editors, *Temporal Databases: Theory, design, and Implementation*, Benjamin/Cummings, 1993.