Indexing Spatio-Temporal Trajectories with Efficient Polynomial Approximations

Jinfeng Ni and Chinya V. Ravishankar, *Senior Member*, *IEEE*

Abstract—Complex queries on trajectory data are increasingly common in applications involving moving objects. MBR or grid-cell approximations on trajectories perform suboptimally since they do not capture the smoothness and lack of internal area of trajectories. We describe a parametric space indexing method for historical trajectory data, approximating a sequence of movement functions with single continuous polynomial. Our approach works well, yielding much finer approximation quality than MBRs. We present the PA-tree, a parametric index that uses this method, and show through extensive experiments that PA-trees have excellent performance for offline and online spatio-temporal range queries. Compared to MVR-trees, PA-trees are an order of magnitude faster to construct and incur I/O cost for spatio-temporal range queries lower by a factor of 2-4. SETI is faster than our method for index construction and timestamp queries, but incurs twice the I/O cost for time interval queries, which are much more expensive and are the bottleneck in online processing. Therefore, the PA-tree is an excellent choice for both offline and online processing of historical trajectories.

Index Terms—Access methods, spatio-temporal databases.

---

1 INTRODUCTION

GPS is widely used in support of a variety of new applications, including tracking of vehicle fleets, navigation of watercraft and aircraft, and the emergency E911 service for cellular phones [20]. Such applications would greatly benefit from an ability to make complex spatio-temporal queries on trajectory data for objects moving in two or higher dimensional space.

Current work on indices to support spatio-temporal queries is typically either in support of *predictive* queries, which require the future object locations based on their current locations and velocities ("*find all objects that will be within Union Square in 10 minutes*"), or *historical* queries, which query the past locations of moving objects ("*find all objects which were at Union Square an hour ago*"). We focus on historical queries, issued on a large set of trajectories.

Queries on historical trajectories may be *offline* or *online* [12]. In offline processing, it is generally permissible to have a relatively expensive preprocessing step on the set of trajectories to optimize query performance [12]. In contrast, online processing assumes that location updates arrive as a real-time data stream so that one must process queries in as the data is being updated. Online processing is thus a more challenging task since no preprocessing is possible [5], [12].

We can also classify indexing methods into *Native Space Indexing* methods (NSI) and *Parametric Space Indexing* methods (PSI) [26]. In NSI, motion in a d-dimensional space is represented as a series of line segments (or curves) in $d + 1$-dimensional space, using time as an additional dimension. PSI can be regarded as the dual transformation of NSI, where a parametric space defined by the motion parameters is used. PSI has been shown to be efficient for predictive queries (see, for example, TPR-tree [28], TPR*-tree [32], and STRIPES [24]).

PSI has not been advocated in the literature for historical queries. Indeed, Porkaew et al. [26] found that NSI outperformed PSI for historical queries. Predictive trajectory uses only one predicted motion function for each object, but each historical trajectory may consist of hundreds or even thousands of motion functions. PSI may hence incur high storage overhead, significantly degrading query performance. As a result, much previous work on historical queries has used native-space indexing, using approximations such as MBRs [13], [12], Octagons [37], or regular grid cells [5].

However, such methods fail to capture some basic properties of trajectories, which typically consist of a series of line segments or curves, with no internal area. As shown by Kollios et al. [14], MBRs are rather coarse approximations for trajectories. Consequently, methods that use MBRs [13], [12] or grid cell approximations [5] either suffer a significant loss in pruning power or require very expensive preprocessing.

1.1 Our Work

We revisit the issue of indexing historical trajectories in parametric space for both offline and online processing. Unlike previous work in the area [26], we do not represent each line segment or curve with a parametric function. Instead, we try to *approximate* a series of line segments or curves with *a single* continuous polynomial. This approximation may not perfectly match the original trajectory, but we also keep track of the maximum deviation between the approximation and the original movement and can still ensure that the approximation is *conservative* and generates no false negatives. As long as this maximum deviation is small, the approximated polynomial function and the

● *The authors are with the Department of Computer Science and Engineering, University of California, Riverside, 900 University Ave., Riverside, CA 92521. E-mail: {jni, ravi}@cs.ucr.edu.*

maximum deviation together provide a much tighter approximation than MBRs and query performance is significantly improved.

We observe first [4] that trajectories tend to be smooth since objects obey the laws of physics and commonly move along smooth paths, such as road networks. Indeed, for similarity-based queries, exploiting the smoothness of trajectories has improved performance greatly over previous methods [4].

There are major differences between our work and that of [4], which also approximates trajectories with polynomials. First, [4] targets similarity-based queries and defines similarity over entire trajectories of equal length, ignoring the time component. Hence, the techniques in [4] are generally not applicable to spatio-temporal queries, to which time is central [25]. Second, the lower-bound lemma in [4] is only valid for similarity queries so that other approaches are needed to deal with spatio-temporal queries. Further, [4] uses approximations of the same degree for *all* of the trajectories, which can cause serious difficulties when the approximation degree is high. In contrast, we use polynomials of different degrees for different trajectories.

1.2 Our Contributions

We make a number of contributions in our work. We show that parametric indexing using polynomial approximations can improve query performance significantly over current schemes using native space indexing. We show how to use polynomial approximations to index historical trajectories and how to optimize the degree of the polynomial approximation.

We then present the PA-tree, a new indexing scheme for historical trajectory data, based on polynomial approximations, and show how to apply PA-trees to support both offline and online processing of historical trajectories.

We also develop an analytical cost model that accurately predicts query performance using PA-trees. Our model is *prospective*, relying only on the properties of the trajectories and the underlying file system. Hence, it can be used to optimize query performance by tuning parameters. In contrast, current approaches such as MVR-tree [13], [12] or SETI [5] must choose parameter values experimentally since they lack an appropriate cost model.

Finally, we evaluate the performance of our schemes using synthetic trajectory data sets. We show through extensive experiments that PA-trees have excellent performance for offline and online spatio-temporal range queries compared to current trajectory indexing schemes, such as MVR-trees and SETI. Compared to MVR-trees, PA-trees are an order of magnitude faster to construct and incur I/O cost for spatio-temporal range queries lower by a factor of 2-4. PA-trees underperform SETI for index construction and timestamp queries, but have half the I/O cost for time interval queries, which are much more expensive than index construction and timestamp queries and are the bottleneck in online processing. Therefore, the PA-tree is a more appropriate choice for both offline and online processing of historical trajectories.

2 RELATED WORK

MBRs have been widely used to approximate multidimensional data and, consequently, R-trees [10] are the most common index structure for multidimensional data. Earlier work using MBRs for trajectories includes the RT-tree [36] and 3D R-tree [35]. However, since the RT-tree does not take temporal attributes into account during the insertion/deletion, timestamp or time interval queries are inefficient. 3D R-tree is inefficient for timestamp queries since the query time depends on the total number of entries in the history [31].

Kollios et al. [15] present methods for indexing linear historical trajectories offline. They model a long-lived trajectory with multiple MBRs by splitting it into segments to reduce the large dead space resulting from the use of a single MBR and use partial-persistent R-trees ("PPR-tree") to index the multiple MBRs. This work is extended in [13], [12], where the motion function could be arbitrary ([12] uses the term "MVR-tree" in place of "PPR-tree"). This method can be more efficient than 3D R-tree since the total empty volume after splitting would be reduced. However, since this method still uses MBRs for approximating each segment, significant dead space remains. Further, they use a global optimization technique to find how to split each trajectory into multiple segments, which, unfortunately, is shown to be very time-consuming [27] so that this method is unsuitable for online applications. To support online processing, some heuristics are proposed in [12] to speed up the splitting; however, as their experiments show, a significant price must be paid in terms of query performance for this speed-up.

Some previous work has been based on a discrete event model under which an object is assumed to stay at its current position until it issues an update to the server. However, this model cannot be used to represent gradual changes in object locations, limiting its applicability [5]. The basic idea is to build a separate R-tree for each timestamp, as in the HR-tree [21] and the MR-tree [36]. Unchanged nodes are not duplicated in consecutive R-trees to reduce the storage cost. However, these index structures are only efficient for timestamp queries, but not for time interval queries [5], [31]. The MV3R-tree [31] is a hybrid structure that uses a multiversion R-tree for timestamp queries and a small 3D R-tree for time-interval queries. The two indices share the same leaf pages in order to reduce the storage cost, resulting in a complex algorithm for maintaining the indices [5].

SETI [5] was the first method to consider online processing of historical trajectories and uses two-tier index structures to decouple the spatial and the temporal dimensions. Space is divided into cells and the temporal attributes of all line segments intersecting a cell are indexed with a one-dimensional index structure. Their results suggest that SETI outperforms 3D R-trees and TB-trees [25] for spatio-temporal range queries.

However, several factors diminish query performance in SETI. First, since multiple line segments of the same trajectory may overlap the query range, SETI must eliminate duplicates, which may be expensive. Second, grid cells are rather coarse approximations for trajectories and, hence,

a significant I/O cost is incurred in eliminating the false hits. Third, the work in [5] does not provide a systematic way to estimate the number of grid cells used in SETI, which is a key parameter for optimizing the query performance.

Song and Roussopoulos proposed SEB-trees, based on a zone-based update policy [29]. Their method, like SETI, divides space into nonoverlapping zones. Within a zone, objects are indexed in an SEB-tree by their start and end timestamps. However, as pointed out in [5], this zone-based update policy does not maintain accurate position information within each zone and is unsuitable for supporting spatio-temporal range queries, which are the focus of our work.

2.1 Parametric Space Indexing

Another approach is to apply a duality transformation of movement and index the parameters in parametric space. Indeed, parametric indexing methods have been shown to be efficient for predicted trajectories [28], [32], [24]. However, for historical trajectories, [26] shows that parametric space index methods were outperformed by native space index methods. This is due to the fact that, for predicated trajectories, only one motion function is required per object, while each historical trajectory may consist of thousands of motion functions, leading to huge storage costs if we index all of the motion functions in the index structure. Unlike [26], we approximate a series of consecutive motion functions by a single polynomial function. Since trajectories are smooth, a low-degree polynomial suffices to approximate many consecutive motion functions, with small error. This approach yields a much smaller dead space than the approximation using MBRs, eventually leading to improved query performance.

Tao et al. [30] use order-k homogeneous recurrence relations, which they call *recursive functions*, to represent *predictive* trajectories. They approximate each such recursive function with a polynomial stored at a server and use *STP-trees* to index these polynomial coefficients.

Several major differences exist between their work and ours. First, they are concerned with predictive trajectories, while we focus on historical trajectories. Second, the STP-tree requires that all trajectories be approximated with polynomials of the same degree. As acknowledged in [30], the polynomial degree must trade off the quality of approximation against the overhead for storage and manipulation. This trade-off might be different for each individual trajectory.

In contrast, we approximate each individual trajectory with a polynomial whose degree is tailored to minimize the expected I/O cost for spatio-temporal range queries and provide an analytical method for this choice. Further, the use of a k degree polynomial in [30] for each axis in a d-dimensional space causes the STP-tree to become an index structure in a parametric space of $(k+1)d$ dimensions, raising the curse of dimensionality for large k. Unlike [30], we adopt a two-tier structure (see Section 5) to address this problem.

2.2 Prospective Cost Models

Cost models allow us to understand index structure behavior and to tune optimization parameters. A number of analytical cost models [8], [7], [34] have been proposed for the R-tree [10] and its variants. However, these models are applicable only for static spatial objects, but not to spatio-temporal databases with moving objects.

Tao et al. [32] have proposed an approach for modeling the cost of queries *for a given* TPR*-tree index. We call such models *retrospective* since they characterize the behavior of a given index, using index-specific details such as MBR sizes, available only *after* a TPR*-tree index is built for any given data set. In contrast, we seek cost models that are not index-specific and are actually useful in selecting index parameters. We call such models *prospective*.

Recently, Tao et al. [33] proposed analytical models for overlapping and multiversion structures. An overlap structure, such as OVB-tree [3], or multiversion structures, such as the BTR-tree [16], models attribute changes over time. However, these analytical models are unlikely to work well for continuously moving objects. First, the work in [33] assumes a discrete event model under which object attributes are assumed to be fixed until an update is issued. However, historical trajectory data typically model continuously moving objects. Second, the work in [33] assumes that all objects remain valid during the lifetime of data set. This assumption is inappropriate for a moving object database, where a new object will be inserted when it starts to move or an existing object deleted when it stops moving. Consequently, as suggested in [12], current analytical models are unlikely to work well for historical trajectories of continuously moving objects when MVR-trees are used. Instead, [12] chooses to tune performance parameters such as the number of MBRs using nonanalytical approaches such as *volume reduction curves*, obtained by running experiments with different numbers of MBRs.

Rasetic et al. [27] propose an analytical model for optimizing the number of MBRs for methods such as [13], [12] when the query size is fixed. Unfortunately, all MBR-based methods must deal with the problems arising from the coarseness of MBR approximations, and we find that our method far outperforms them. We have found that PA-trees outperform MVR-trees [13] by a factor of 2-4, even when they are given an *experimentally optimized* number of MBRs.

3 DATA MODEL AND OVERVIEW OF OUR APPROACH

In many location-based services, location data are obtained by periodic sampling. Specifically, the trajectory for an object O_i has the form

$$Trj(O_i) = \{ID_i, t_0, t_1, \cdots, t_n, f_0(t), f_1(t), \cdots, f_{n-1}(t)\}.$$

Function $f_j(t)$ is a movement function representing movement during time interval $[t_j : t_{j+1}]$, $0 \leq j \leq n-1$. The interval $[t_0 : t_n]$ is the *lifetime* of the trajectory.

Our approach is applicable to any movement function $f(t)$ as long as we can determine the location of the object at any time instant during its lifetime from $f(t)$. For simplicity of exposition, we adopt a linear mobility model, which is
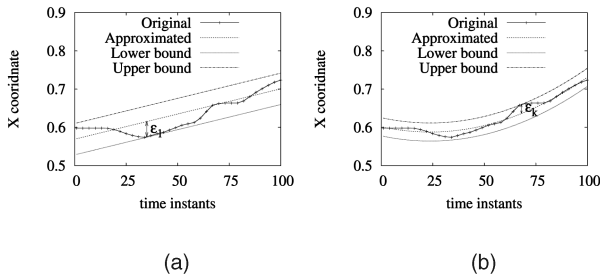
Fig. 1. Approximating a trajectory segment with polynomials. (a) Linear approximation. (b) Polynomial approximation.

widely used in the literature [5], [25]. Each $f_j(t)$ is now a linear function of time so that a trajectory consists of a series of connected line segments, generally called a *polyline*.

As in previous work [13], [12], we assume time is discrete and that each time instant is an integer in the range $[0, T]$, which contains the lifetimes of all the trajectories. We assume an object moves in a two-dimensional XY-space. The extension to higher dimensions is straightforward.

We focus mainly on spatio-temporal range queries, which are essential building blocks for all other types of queries. A spatio-temporal range query may be a timestamp query or a time interval query [13]. A timestamp query $q = (q_s, t)$ asks for all objects within spatial range q_s at timestamp t. A time interval query $q = (q_s, t_1, t_2)$ asks for all objects which were within spatial range q_s at any timestamp $t \in [t_1 : t_2]$.

3.1 Overview of Approach

We proceed in two steps. First, we calculate the parametric representation for each trajectory by approximating it in XY-space with two polynomial functions: $\hat{f}_x(t)$ and $\hat{f}_y(t)$ modeling movement in the X direction and in the Y direction, respectively. We also determine the maximum deviation of the polynomial approximation from the exact movement in the X and Y dimensions. The polynomial coefficients and the maximum deviation suffice for us to make the approximation conservative, guaranteeing no false negatives.

Fig. 1a and Fig. 1b show how we construct linear and order-k polynomial approximations to the X-component of a trajectory. Such approximations are not exact, so we create conservative upper and lower bounds for the object's position by offsetting the approximating polynomial upward and downward by an amount equal to the maximum deviation between the trajectory and the polynomial. We can now guarantee that the object will be located within these bounds.

In the second step, we build an index structure over the coefficients obtained in the first step. However, not all trajectories are likely to be equally complex so that we may need polynomials of different degree for different trajectories. This causes problems when building an index structure using the coefficients since the dimensionality of the indexed items may be different. Current index structures assume that the dimensionality of all data is the same. Adopting the same polynomial degree for all trajectories is not advisable since the curse of dimensionality will quickly

degrade the performance of any index structure in high-dimensional space.

3.1.1 Two-Tier Indexing

We address this problem by using a two-tier index structure. The first-tier index structure uses only the first two coefficients of each polynomial so that each data entry is a 6-tuple (two coefficients for each dimension and the corresponding maximum deviations). This strategy ensures that we are not operating in a high-dimensional space so that an R-tree or its variants can still be efficient for indexing. As we will illustrate in Section 4, by appropriately splitting the temporal domain $[0, T]$ into intervals, we can adopt a piecewise linear approximation in the first tier index structure, each linear approximation corresponding to multiple line segments in the trajectory. However, even with this piecewise-linear approximation, we achieve much smaller dead space than MBRs can for the same size of representation.

The second-tier index structure is elaborated within the leaf nodes of the first-tier structure. Complex trajectories may require higher-degree polynomial approximations whose coefficients are stored in the second-tier structure. If we descend to the leaf nodes in the first-tier structure and still are unable to determine whether the trajectory satisfies the query predicates, the additional coefficients can be retrieved and used in the filtering step. As our experiments will show, most trajectories can be approximated very well with quadratic or cubic polynomials so that the second-tier structure does not introduce significant space overhead.

3.1.2 An Illustrative Example

Fig. 2a plots the trajectory of a moving vehicle for 10 minutes, collected in the Intellishare project [1] at the University of California–Riverside. Fig. 2b plots the X-movement against time and the eight MBRs obtained with the LAGreedy algorithm proposed in [13], [12]. We note that the eight MBRs together require $8 \times 6 = 48$ values. Fig. 2c plots the result of our method in which the trajectory is split into six segments, each approximated with a linear function. Each segment requires two coefficients and one maximum deviation each for X-movement or Y-movement, plus the temporal intervals. In all, 48 values are required for the approximations. Our polynomial approximations clearly produce much smaller dead space than the MBR approximations. Fig. 2d plots the approximation with more coefficients, with significantly reduced dead space.

4 APPROXIMATING TRAJECTORIES WITH POLYNOMIALS

In this paper, we propose an approximation in parametric space by using Chebyshev polynomials. Chebyshev polynomials have been shown to have the near-optimal L_∞ deviation among all approximations with the same degree [19] and perfectly match our requirements. Further, the Chebyshev coefficients are easy to compute [19], [4].

We have chosen to split each trajectory into multiple segments by dividing the temporal domain $[0, T]$ into m disjoint time intervals, each of which is approximated with a lower-degree polynomial. There are two reasons for
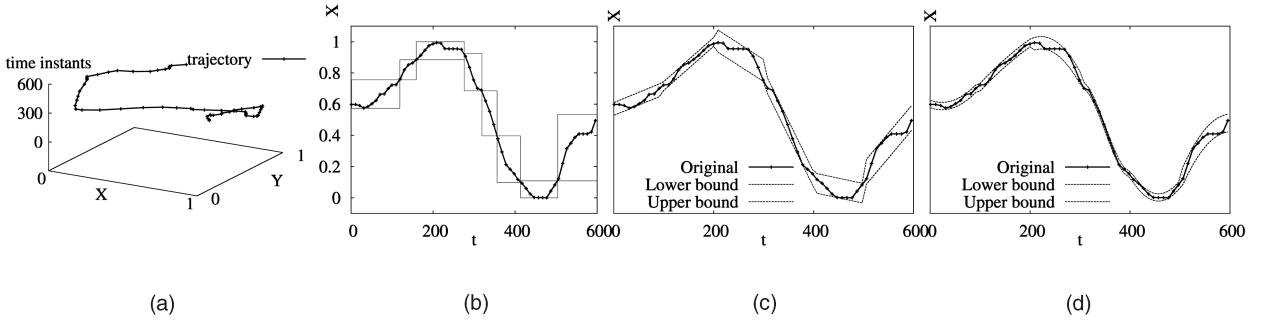
Fig. 2. Comparison of approximations for a moving vehicle trajectory. (a) Trajectory of vehicle. (b) MBR approximations. (c) Linear approximation. (d) Finer approximation.

such splitting. First, approximating the entire trajectory with a single polynomial may require a polynomial of high degree, leading to a high-dimensional indexing problem. Second, the marginal benefit for the first few coefficients will be much larger than that of high-order coefficients.

4.1 Splitting the Time Domain

We split the temporal domain $[0, T]$ into m equal time intervals:

$$I_0 = [0, l), I_1 = [l, 2l), \cdots,$$
$$I_i = [il, (i+1)l], \cdots,$$
$$I_{m-1} = [(m-1)l, ml),$$

where $l = \frac{T}{m}$ is the length of each time interval and timestamps $l, 2l, \cdots, (m-1)l$ are called splitting timestamps. In Section 8, we will discuss how to optimize m based on an analytical cost model.

Each trajectory is split into multiple segments using the same $m-1$ splitting timestamps. This strategy is different from that in [13], [12], [27], where each trajectory selects different splitting timestamps. First, in parametric space, a set of segments cannot be clustered unless they have the same temporal domain since it would be meaningless to cluster coefficients corresponding to different temporal domains. Second, even with an equal-sized splitting strategy, we can still use different numbers of coefficients for different trajectories. Indeed, basing the number of coefficients for approximation on the trajectory complexity is equivalent to using different splitting timestamps. Finally, using equal-length splitting intervals obviates the need to maintain time intervals in index nodes. This could significantly reduce the storage cost of the index structure and eventually lead to a reduction of I/O cost during the filtering step.

4.2 Approximating a Trajectory Segment with a Polynomial

We now consider how to obtain polynomial approximation (PA) with Chebyshev polynomials. We illustrate the approximation for the X-movement only, so we will omit the subscript x when no confusion can arise. Consider a trajectory segment in the temporal interval $I_i = [il, (i+1)l]$. Let $f(t)$ be the piecewise linear movement functions during I_i.

We first normalize the temporal domain $[il, (i+1)l]$ to the interval $[-1, 1]$. Given $t \in [il, (i+1)l]$, let its normalized value be $t' \in [-1, 1]$. Normalization requires

$$\frac{t - il}{l} = \frac{t' - (-1)}{2},$$

which yields

$$t' = \frac{2t - (2i+1)l}{l} \text{ and } t = \frac{(2i+1)l + t'l}{2}.$$

The function $f : [il, (i+1)l] \rightarrow (-\infty, \infty)$ is now normalized to $\widetilde{f} : [-1, 1) \rightarrow (-\infty, \infty)$.

Now, for $t \in [il, (i+1)l]$, we can approximate $f(t)$ as

$$\widehat{f}(t) = c^{(0)}T_0(t') + c^{(1)}T_1(t') + \cdots + c^{(k)}T_k(t'), \qquad (1)$$

where $t' \in [-1, 1]$ is the normalized value of t, $T_i(t') = \cos(i \arccos(t'))$ is the Chebyshev polynomial of degree i, and the coefficients $c^{(0)}, c^{(1)}, \cdots, c^{(k)}$ are to be determined.

Theorem 1. *Let $f(t)$ be the function over interval $[il, (i+1)l]$ to be approximated. Now,*

$$c^{(0)} = \frac{1}{\pi}\int_{-1}^{1} \frac{\widetilde{f}(t)}{\sqrt{1-t^2}} dt, \quad c^{(i)} = \frac{2}{\pi}\int_{-1}^{1} \frac{\widetilde{f}(t)T_i(t)}{\sqrt{1-t^2}} dt.$$

Proof. See [19]. □

We use Gauss-Chebyshev quadrature to compute these integrals. The abscissas for quadrature are given by the roots of $T_n(t)$, which has n roots $\rho_j = \cos\frac{(j-0.5)\pi}{n}$ for $1 \leq j \leq n$. We have the following explicit way to compute the coefficients:

$$c^{(0)} \approx \frac{1}{n}\sum_{j=1}^{n} f\left(\frac{(2i+1)l + \rho_j l}{2}\right),$$

$$c^{(i)} \approx \frac{2}{n}\sum_{j=1}^{n} f\left(\frac{(2i+1)l + \rho_j l}{2}\right)T_i(\rho_j).$$

To ensure that this approximation leads to no false negatives, we determine a conservative approximation guaranteed to contain the object's location at all times. After obtaining the $k+1$ coefficients, we compute the maximum deviation

$$\epsilon^{(k)} = \max\left\{\left|f(t) - \widehat{f}(t)\right|\right\}, \ t \in [il, (i+1)l].$$

TABLE 1
Summary of Notation for Events

Notation	Meaning	Description
$\mathbf{H_X}$	Raw Hit	Approximation intersects q_s on X-dimension
$\mathbf{fTH_X}$	Filtering True Hit	True hit on X-dimension in filter step.
$\mathbf{rTH_Y}$	Refinement True Hit	True hit on X-dimension in refinement step.
$\mathbf{C_X}$	Candidate	Segment is an candidate on X-dimension

Now, the range $[\widehat{f}(t) - \epsilon^{(k)}, \widehat{f}(t) + \epsilon^{(k)}]$ is guaranteed to contain $f(t)$ for $t \in [il, (i+1)l]$.

The $k+1$ Chebyshev coefficients can be computed in time $O(nk)$, where n is the highest degree of approximating Chebyshev polynomial. Computing the maximum deviation error takes time $O(lk)$, where l is the number of instants between t_0, t_s. In our implementation, we choose $n = O(l)$, so the total cost is $O(lk)$. The following lemma is useful:

Lemma. *Suppose $f(t)$ is approximated by a linear function $\widehat{f}(t) = c^{(0)} + c^{(1)}T_1(t')$. For $il \le t_1 < t_2 \le (i+1)l$, we have $|\widehat{f}(t_2) - \widehat{f}(t_1)| = 2(c^{(1)}|t_2 - t_1|)/l$.*

Proof.

$$|\widehat{f}(t_2) - \widehat{f}(t_1)| = |(c^{(0)} + c^{(1)}t_2') - (c^{(0)} + c^{(1)}t_1')|$$

$$= \frac{2c^{(1)}|t_2 - t_1|}{l}.$$

□

4.3 Choosing the Degree of Approximating Polynomials

In general, the polynomial degree k should be determined based on the characteristics of trajectories. Clearly, there is a trade-off between the approximation quality and the degree k used for approximation. A smaller k value requires less space in the index, as well as less I/O during the filter step. On the other hand, fewer coefficients may result in poorer filtering, causing more trajectories to be examined during the refinement step, increasing its I/O cost.

Another consideration is the complexity of the trajectory segment. Obviously, if the trajectory segment has a relatively simple form, a few coefficients will suffice to get a small deviation error. However, since we are not aware of any well-defined notion of complexity for this context, it is not easy to estimate the optimal degree.

We present a heuristic to estimate the degree k, aimed at minimizing the expected size of representations to be retrieved during query evaluation. Let $\epsilon_x^{(k)}$ and $\epsilon_y^{(k)}$ be the maximum errors for a k-order polynomial approximation on the X- and Y-dimensions, respectively. We make the following reasonable assumptions: First, given query $q = (q_s, t_1, t_2)$, we assume that the query range q_s is a $q_x \times q_y$ rectangle, where $q_x \ge 2\epsilon_x^{(k)}$ and $q_y \ge 2\epsilon_y^{(k)}$. (The other cases are similar and, hence, omitted). Second, we assume that the range q_s is uniformly distributed in the region normalized to a unit square.

Let S and S_k be the data sizes of the exact representations for the trajectory segment and of its k-degree polynomial approximation, respectively. We will derive the expected size of representational data to be retrieved for a random query.
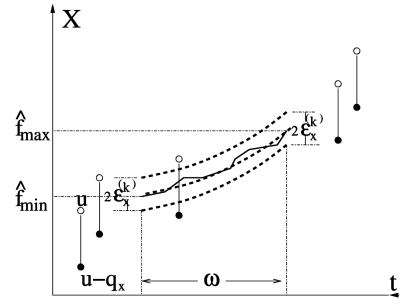

Fig. 3. X-candidate example.

If the approximation does not intersect the query range, we can safely prune it out during the filter step. If a segment's approximation lies *completely* inside q_s, we can safely declare it a true hit. We call this type of true hit a *filtering true hit*. Otherwise, the segment becomes a *candidate* for the refinement step in which its exact representation must be retrieved. If the refinement step finds that the segment lies outside q_s, we have a *false hit*. Otherwise, we will record a *refinement true hit*.

4.3.1 Estimating Relevant Probabilities

To estimate the expected I/O cost, we must estimate the probability that the trajectory segment is a candidate for the refinement step in which it will be determined to be either a false hit or a refinement true hit. We will consider the hits and misses on the X and Y dimensions separately, omitting the x and y subscripts and the subscript k when no confusion is likely.

We first define a class of events to help our development (see Table 1). Consider a segment s, an approximation $\widehat{s}$ for s, and a query range $q_s = q_x \times q_y$. Let the projections of $\widehat{s}$ along the X- and Y- axes be $\widehat{s}_x$ and $\widehat{s}_y$, respectively.

We have an *X-raw hit* (or just *X-hit*) if $\widehat{s}_x$ intersects q_x during the filtering step. We will denote this event as $\mathbf{H_X}(s)$. Further processing is required for us to determine whether this hit is a true hit or a false hit. If $\widehat{s}_x$ lies *entirely* within the query range q_x, we can flag this as a *filtering true hit* and dispense with the refinement step. We denote this event by $\mathbf{fTH_X}(s)$. If we have $\mathbf{H_X}(s)$ but not $\mathbf{fTH_X}(s)$, we pass s to the refinement step for further processing. In this case, we will designate s as an X-candidate and write $\mathbf{C_X}(s)$.

Let the query range q_s's X-projection be $[u - q_x, u]$. Let ω be the overlap between the lifetime of s and the query temporal interval $[t_1, t_2]$, with $|\omega|$ being the length of this overlap. Let $\widehat{f}_{min}$ and $\widehat{f}_{max}$ be the minimum and maximum value of the approximated polynomial $\widehat{f}_x(t)$ within ω. As seen in Fig. 3, if $u \in [\widehat{f}_{min} - \epsilon_x^{(k)}, \widehat{f}_{min} + \epsilon_x^{(k)}]$ or $u - q_x \in [\widehat{f}_{max} - \epsilon_x^{(k)}, \widehat{f}_{max} + \epsilon_x^{(k)}]$, the trajectory segment is a candidate for the refinement step. In this case, although the projection $\widehat{s}_x$ of $\widehat{s}$ overlaps $[u - q_x, u]$, it is still unclear whether s itself intersects with q_s along the X-dimension. Hence, the probability that the trajectory segment is an X-candidate is

$$\Pr[\mathbf{C_X}] = 2\epsilon_x^{(k)} + 2\epsilon_x^{(k)} = 4\epsilon_x^{(k)}. \tag{2}$$

To estimate the probability that the trajectory segment is an X-hit, we observe that, during ω, the projected approximated
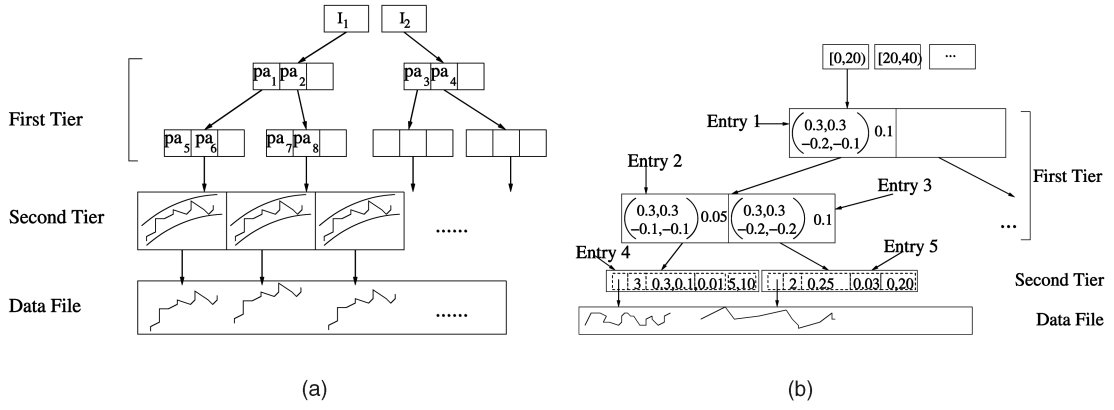
Fig. 4. (a) PA-tree and (b) an example.

movement along the X-dimension is $\widehat{f}_{max} - \widehat{f}_{min}$, associated with approximation error $2\epsilon_x^{(k)}$. We know from Lemma 1 that, when a linear function $\widehat{f}_x(t)$ is used to approximate the X-movement, $\widehat{f}_{max} - \widehat{f}_{min} = 2\frac{c_x^{(1)}|\omega|}{l}$, where l is the length of the temporal interval. Now, the probability that the trajectory segment is an X-hit is

$$\Pr[\mathbf{H_X}] = 2\epsilon_x^{(k)} + 2\frac{c_x^{(1)}|\omega|}{l} + q_x.$$

If the approximation $\widehat{s}_y$ does not result in a Y-hit, we can safely prune out the segment. However, when a trajectory segment is an X-candidate and a Y-Hit, it remains unclear during the filter step whether the trajectory segment satisfies the query predicate, so the segment becomes a refinement candidate. A symmetric situation holds when the segment is a Y-candidate and an X-intersect. Therefore, the probability of becoming a refinement candidate is

$$\Pr[\mathbf{C}] = \Pr[(\mathbf{C_X} \cap \mathbf{H_Y}) \cup (\mathbf{C_Y} \cap \mathbf{H_X})]$$
$$= \Pr[\mathbf{C_X}] \cdot \Pr[\mathbf{H_Y}] + \Pr[\mathbf{C_Y}] \cdot \Pr[\mathbf{H_X}] - \Pr[\mathbf{C_X}] \cdot \Pr[\mathbf{C_Y}]$$

since $\mathbf{C_X} \subseteq \mathbf{H_X}$ and $\mathbf{C_Y} \subseteq \mathbf{H_Y}$. Equation (2) and symmetry between $\mathbf{C_X}$ and $\mathbf{C_Y}$ yields

$$\Pr[\mathbf{C}] = 4\left(\epsilon_x^{(k)} \cdot q_y + \epsilon_y^{(k)} \cdot q_x\right) + 8\frac{|\omega|}{l}\left(\epsilon_x^{(k)} \cdot c_y^{(1)} + \epsilon_y^{(k)} \cdot c_x^{(1)}\right).$$

Now, the expected I/O cost for the trajectory segment s with a degree-k approximation is

$$\overline{\mathrm{IO}}_k = S_k + \Pr[\mathbf{C}]S,$$

where S and S_k are the data sizes of the exact representations for s and of its degree-k polynomial approximation, respectively. We will choose k to minimize $\overline{\mathrm{IO}}_k$.

5 PA-TREES

We now present the PA-tree, a new method for indexing polynomial approximations of 2D trajectories. PA-trees resemble R*-trees, but each entry consists of polynomial coefficients, rather than MBRs. We recall that the temporal domain $[0, T]$ is split into m intervals. In a gross sense, the root node has m PA-trees as its children, each responsible

for indexing trajectory segments within one of these intervals.

Fig. 4a shows two PA-trees, over intervals I_1 and I_2, respectively. Indexing occurs at two tiers. The first tier of indexing is an R*-tree like structure, and indexes the two leading coefficients of the polynomial describing movement along each dimension. It is reasonable to see this as a four-dimensional indexing problem, with each dimension corresponding to one coefficient. Each entry in the index structure also holds the maximum deviation errors $\epsilon_x^{(1)}$ and $\epsilon_y^{(1)}$.

As in R*-trees, a leaf node entry has the form (ptr, pa), where ptr points to the exact representation of the trajectory segment and pa is a 6-tuple: $\langle c_x^{(0)}, c_x^{(1)}, c_y^{(0)}, c_y^{(1)}, \epsilon_x^{(1)}, \epsilon_y^{(1)}\rangle$. Entries in nonleaf nodes are of the form (ptr, pa), where ptr is the pointer to a child node and pa has the form $\langle c_{\perp,x}^{(0)}, c_{\perp,y}^{(0)}, c_{\perp,x}^{(1)}, c_{\perp,y}^{(1)}, c_{\top,x}^{(0)}, c_{\top,y}^{(0)}, c_{\top,x}^{(1)}, c_{\top,y}^{(1)}, \epsilon_x^{(1)}, \epsilon_y^{(1)}\rangle$, representing the lower (upper) bounds of the coefficients for the entries stored in the child pointed to by ptr. Also, pa maintains the maximum $\epsilon_x^{(1)}$ and $\epsilon_y^{(1)}$ for all entries in the subtree.

The second tier holds more coefficients and the maximum deviation for each trajectory segment, if the estimated degree is larger than 1 (See Section 4). This information allows better pruning than the linear approximations in the first tier.

Insertions and deletions are similar to the corresponding operations for R*-tree. The primary difference is that we need to ensure that the $\epsilon_x^{(1)}$, $\epsilon_y^{(1)}$ values in the nonleaf nodes are the maximum $\epsilon_x^{(1)}$, $\epsilon_y^{(1)}$ for all the segments in its subtree.

Fig. 4b depicts a set of PA-trees, showing only the coefficients and deviations for X-dimension for simplicity. The root entries specify the time interval for each individual PA-tree. Each first-tier entry contains the lower and upper bounds for the coefficient $c^{(0)}$ and $c^{(1)}$, as well as the maximum deviation error $\epsilon^{(1)}$. Each entry in the second tier has the form of $k, c^{(2)}, \cdots, c^{(k)}, \epsilon^{(k)}, t_s, t_e$.

6 OFFLINE QUERY PROCESSING

Given a query $q = (q_s, t_1, t_2)$, we start with the root node, which contains m temporal intervals and the pointers to the corresponding PA-tree. Each PA-tree is searched if and only if the temporal interval intersects $[t_1, t_2]$.

Let $I_i = [il, (i+1)l]$ be the temporal interval corresponding to an entry in a nonleaf node in the PA-tree. Given $q = (q_s, t_1, t_2)$, let $t_s = \max\{t_1, il\}$ and $t_e = \min\{t_2, (i+1)l\}$. We must check whether there is a trajectory segment inside q_s at any time $t \in [t_s, t_e]$. Let the index entry be $\langle c_{\perp,x}^{(0)}, c_{\perp,y}^{(0)}, c_{\perp,x}^{(1)}, c_{\perp,y}^{(1)}, c_{\top,x}^{(0)}, c_{\top,y}^{(0)}, c_{\top,x}^{(1)}, c_{\top,y}^{(1)}, \epsilon_x^{(1)}, \epsilon_y^{(1)} \rangle$. In the following discussion, we will omit the subscripts x and y for the sake of clarity.

As in Section 4, for $t \in [t_s, t_e]$, we use t' to denote its normalized value in $[-1, 1]$. Now, the nonleaf entry represents all movement in the approximated linear form $f'(t) = c^{(0)} + c^{(1)} T_1(t') = c^{(0)} + c^{(1)} t'$, where $c^{(0)} \in [c_\perp^{(0)}, c_\top^{(0)}]$ and $c^{(1)} \in [c_\perp^{(1)}, c_\top^{(1)}]$. In principle, we can apply the dual transformation technique of [14] to check whether there are linear trajectories intersecting q_s during $[t'_s, t'_e]$. However, the slope $c^{(1)}$ and the temporal attribute t' could be either positive or negative, making it hard to apply duality transformations. Instead, we determine the upper and lower bounding polynomials for the motion segment in the form $c^{(0)} + c^{(1)}t$, where $c^{(0)} \in [c_\perp^{(0)}, c_\top^{(0)}]$, $c^{(1)} \in [c_\perp^{(1)}, c_\top^{(1)}]$, and $t' \in [t'_s, t'_e]$. If $e^{(1)}$ is the maximum deviation error, we use the monotonicity of $c^{(0)} + c^{(1)}t'$ to compute the lower bound as:

$$x_\perp = c_\perp^{(0)} + \min\{c_\perp^{(1)}t'_s,\ c_\perp^{(1)}t'_e,\ c_\top^{(1)}t'_s,\ c_\top^{(1)}t'_e\} - \epsilon^{(1)} \quad (3)$$

and the upper bound as:

$$x_\top = c_\top^{(0)} + \max\{c_\perp^{(1)}t'_s,\ c_\perp^{(1)}t'_e,\ c_\top^{(1)}t'_s,\ c_\top^{(1)}t'_e\} + \epsilon^{(1)}. \quad (4)$$

If the computed range intersects with the query range q_s, we know there may be candidates satisfying the query predicates. We now descend the tree and repeat the process for the subtree rooted at this entry, down to the leaf nodes.

At the leaf node, we first retrieve the $k-1$ coefficients in the second-tier structure, stored sequentially in the leaf nodes. The approximate location $\widehat{f}(t)$ at time $t \in [t_s, t_e]$ can be computed using (1) and the spatial range $[\widehat{f}(t) - \epsilon^{(k)}, \widehat{f}(t) + \epsilon^{(k)}]$. If there is a time $t \in [t_s, t_e]$ such that the spatial range is completely inside q_s, the trajectory segment is a filtering true hit, its ID will be reported. If this range does not intersect query q_s for any $t \in [t_s, t_e]$, the trajectory segment is pruned out. Otherwise, refinement is required for determining whether this trajectory segment is a true hit or false hit.

6.1 An Example

Consider the PA-trees shown in Fig. 4b. Let the query have spatial range $q_x = [0, 0.1]$ in the X-dimension and temporal interval $[10, 20]$. Clearly, the PA-trees corresponding to the interval $[0, 20]$ and the one for $[20, 40]$ must both be searched. Normalizing the query interval to $[0, 1]$, we can use (3) and (4) and obtain $[0, 0.4]$ as the range for entry 1. Since q_x overlaps this range, we need to check entries 2 and 3, which are entry 1's children. For entry 2, we use (3) and (4) and get the range as $[0.15, 0.35]$, which does not intersect
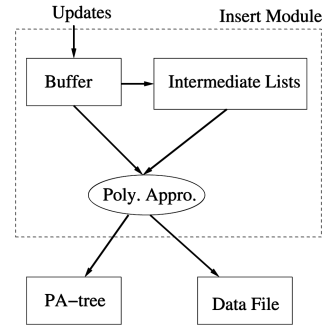

Fig. 5. Insert procedure.

with $q_x = [0, 0.1]$. We can safely prune out this entry. For entry 3, we get the range as $[0, 0.4]$, which overlaps q_x.

We now retrieve entry 5 in the second-tier structure. For this trajectory segment, we have $k = 2$, $c^{(0)} = 0.3$, $c^{(1)} = -0.2$, $c^{(2)} = 0.25$, and $\epsilon^{(2)} = 0.03$. For each timestamp $t \in [10, 20]$, we calculate $\widehat{f}(t)$ using (1). For $t = 10$, we calculate the value of $\widehat{f}(t)$ to be 0.05. We now compute the range to be $[\widehat{f}(t) - \epsilon^{(2)}, \widehat{f}(t) + \epsilon^{(2)}] = [0.02, 0.08]$, which is completely inside $q_x = [0, 0.1]$. Therefore, we identify this trajectory segment as a true hit.

7 ONLINE PROCESSING

We now extend PA-trees to support online processing when location updates arrive as real-time data streams. We support both efficient index structure maintenance and query.

7.1 Handling Inserts

Fig. 5 shows the insert procedure in our online scheme. As in the offline case, we split the lifetime into intervals of length l. Let $I_{now} = [il, (i+1)l]$ be the current interval and let t_{now} be the current time, $il \le t_{now} \le (i+1)l$. As in [5], updates arriving during $[il, i+1)l)$ will be stored in an in-memory buffer, called the *frontline buffer*. At the end of the current interval, that is, at $t_{now} = (i+1)l$, we obtain polynomial coefficients and deviation errors, as in Section 4. These values are then inserted into a PA-tree and the trajectory data placed in a data file.

7.1.1 Buffer Management

One major concern is that the frontline buffer may overflow before the end of the interval, especially for relatively long intervals or when updates arrive rapidly. We use the technique of *batch writes*, as in [17]. When the buffer is full, its contents are written to an intermediate list as a single batch. Similarly, at the end of an interval, an intermediated list is retrieved as a batch and the polynomial approximation applied over the entire trajectory segment corresponding to this interval. Since reading and writing each intermediate list requires one random access followed by a long sequential read [17], the I/O overhead is greatly reduced.

TABLE 2
Summary of Notation

Notation	Description		Notation	Description
$c_x^{(i)}$ $(c_y^{(i)})$	the i-th coefficients for X-movement (Y-movement).		λ_s (λ_n)	lifetime of a trajectory segment s or a node n in a PA-tree
$\epsilon_x^{(k)}$ $(c_y^{(k)})$	the max. deviation error for X-movement (Y-movement) with a polynomial approximation of degree k		ω_s (ω_n)	the overlap between the lifetime of trajectory segment s (or node n) and $[t_1, t_2]$
$q = (q_s, t_1, t_2)$	spatio-temporal range query, with spatial range q_s being a $q_x \times q_y$ rectangle, and temporal interval $[t_1, t_2]$		$\sigma_{n,x} \times \sigma_{n,y}$	the spatial extent of the node n during ω_s
m	the number of temporal intervals used to split $[0, T]$		$\sigma_{n,x} \times \sigma_{n,y}$	the spatial extent of the node n during ω_s
$l = \frac{T}{m}$	the length of temporal interval		PA$_i$	the PA-tree corresponding to ith temporal interval
$\bar{\sigma}_{[j],x}$ $(\bar{\sigma}_{[j],y})$	the mean spatial extent along X-dimension (Y-dimension) for nodes at level j during ω_n		$\bar{\mu}_{[j],x}$ $(\bar{\mu}_{[j],y})$	the average movement along X-dimension (Y-dimension) during ω_n for a node at level j
N	total number of trajectory segments		N_i	total number of trajectory segments indexed by PA$_i$
$M_{i,j}$	total number of nodes at level j in PA$_i$		$\bar{\epsilon}_{[j],x}$ $(\bar{\epsilon}_{[j],y})$	the average of maximum deviation error for the nodes at level j
$\hat{\mu}_{s,x}$ $(\hat{\mu}_{s,y})$	the approximated linear X-movement (Y-movement) for trajectory segment s		DA$_i^f(q)$ (DA$_i^r(q)$)	the number of disk access over PA$_i$ during filtering step (refinement step)
D_j	the density of nodes at level j		$[0, T]$	the temporal domain

7.2 Handling Queries

Consider a time interval query $q = (q_s, t_1, t_2)$. Let I_{now} be the current interval and t_{now} be the current time. To process q, we must first determine the temporal intervals up to I_{now} that overlap $[t_1, t_2]$. If I_{now} itself overlaps $[t_1, t_2]$, we first check whether the trajectory segment in the frontline buffer overlaps q_s during $[t_1, t_2]$. The trajectory segments in the intermediate lists will next be retrieved and tested against q. All trajectory segments from temporal intervals I that precede I_{now} and overlap $[t_1, t_2]$ will have already been indexed in a PA-tree. Therefore, the query processing technique presented in Section 5 suffices to find all of the trajectory segments that intersect q_s during interval I.

8 A PROSPECTIVE PERFORMANCE MODEL

We now develop an analytical cost model for PA-trees. In the terminology of Section 2, we seek cost models that are *prospective*, rather than *retrospective*. That is, we seek models useful in designing PA-tree structures for a given data set and in tuning important index parameters, such as the number of temporal intervals, to minimize the number of disk accesses.

We are given N trajectories during the lifetime $[0, T]$ in a two-dimensional unit square and a time interval query $q = (q_s, t_1, t_2)$. We want a formula to estimate the average number of disk access for processing q during the filtering and refinement steps. As in Section 4, we divide the lifetime $[0, T]$ into m equal-sized intervals $[0, l), [l, 2l), \cdots, [(m-1)l, ml]$, where $l = T/m$ is the time interval length. Let the PA-tree corresponding to the ith interval be denoted by PA$_i$.

We make the following reasonable assumptions: First, the query spatial range $q_s = q_x \times q_y$ is uniformly distributed in the unit square and the query temporal interval $[t_1, t_2]$ is uniformly distributed within $[0, T]$. Second, for trajectory segment s, we assume that we know the coefficients $c_{s,x}^{(1)}$ and $c_{s,y}^{(1)}$, the linear deviation errors $\epsilon_{s,x}^{(1)}$ and $\epsilon_{s,y}^{(1)}$, and the k-order approximation errors $\epsilon_{s,x}^{(k)}$ and $\epsilon_{s,y}^{(k)}$. These values can be obtained using the polynomial approximation technique discussed in Section 4. Finally, we assume that there is no buffering since our purpose is to model the PA-tree without the confounding effects of a buffer management algorithm.

Let N_i be the number of trajectory segments indexed by PA$_i$ and h_i be the height of PA$_i$. Let f be the fanout of node in PA$_i$, that is, the average number of entries in a node. Let $M_{i,j}$ be the number of nodes at level j. Leaf nodes are at level 1, root node is at level h, and the data entries are at level 0. Now, as in the R-tree cost model of [34], we have:

$$h_i = \log_f N_i \text{ and } M_{i,j} = \frac{N_i}{f^j}. \qquad (5)$$

Let DA$_i^f(q)$ be the the number of disk access for PA$_i$ during the filtering step for query q. Let DA$_i^r(q)$ be the number of disk accesses to the trajectory segments for the ith interval during the refinement step. The total number of disk access would be

$$\text{DA}(q) = \sum_{i=1}^{m} \big(\text{DA}_i^f(q) + \text{DA}_i^r(q)\big). \qquad (6)$$

8.1 Cost Model for Filtering Step

Consider a node n on PA$_i$ with lifetime $\lambda_n = [il, (i+1)l)$. Let $\sigma_n = \sigma_{n,x} \times \sigma_{n,y}$ be its spatial extent in the native space and let $\omega_{n,q} = \lambda_n \cap [t_1, t_2]$ be the overlap between λ_n and $[t_1, t_2]$, with $|\omega_{n,q}|$ being the length of the overlap. Further, let $\sigma_{n,q} = \sigma_{n,q,x} \times \sigma_{n,q,y}$ be n's spatial extent during $\omega_{n,q}$.

Since all of our ensuing discussion refers to the query q, we will simplify notation by omitting specific mention of the query q in our symbol subscripts (see Table 2). Thus, we will write ω_n, σ_n, and $\sigma_{n,x}$ instead of $\omega_{n,q}$, $\sigma_{n,q}$, and $\sigma_{n,q,x}$. The reference query q will be present implicitly.

Let $\mathbf{A}(\mathbf{n})$ denote the event that node n is accessed during query processing. Node n will be visited if and only neither ω_n and σ_n is null. Let $\Omega(\mathbf{n})$ denote the event that ω_n is not null and $\Sigma(\mathbf{n})$ denote the event that σ_n is not null. Now, the probability that node n will be visited is

$$\Pr[\mathbf{A}(\mathbf{n})] = \Pr[\mathbf{\Omega}(\mathbf{n})] \times \Pr[\mathbf{\Sigma}(\mathbf{n})].$$

Since $[t_1, t_2]$ is uniformly distributed in $[0, T]$,

$$\Pr[\mathbf{\Omega}(\mathbf{n})] = \frac{(l + |t_2 - t_1|)}{T}.$$

$\Pr[\Sigma(\mathbf{n})]$ can be estimated as [34], [7]:

$$\Pr[\mathbf{\Sigma}(\mathbf{n})] = \big(\sigma_{n,x} + q_x\big)\big(\sigma_{n,y} + q_y\big).$$

As in our discussion of query processing in Section 6, we compute each node's spatial extent as the sum of a component derived from the approximated object movement and a component corresponding to the maximum deviation error of the linear approximation. Given $M_{i,j}$ nodes at level j in PA-tree PA_i, let the mean spatial extents along the X-dimension and Y-dimension for the nodes at level j during ω_n be $\bar{\sigma}_{[j],x} = \frac{1}{M_{i,j}} \sum_{m=1}^{M_{i,j}} \sigma_{m,x}$ and $\bar{\sigma}_{[j],y} = \frac{1}{M_{i,j}} \sum_{m=1}^{M_{i,j}} \sigma_{m,y}$, respectively. Let $\bar{\mu}_{[j],x}$ and $\bar{\mu}_{[j],y}$ be the average movement of objects at level j along the X- and Y-dimensions during ω_n, derived from the trajectory approximations. Let $\bar{\epsilon}_{[j],x}$ and $\bar{\epsilon}_{[j],y}$ be the average of the maximum deviation errors of the nodes at level j. The average spatial extent of a level-j node is

$$\overline{\sigma}_{[j],x} = \overline{\mu}_{[j],x} + 2\overline{\epsilon}_{[j],x}, \text{ and } \overline{\sigma}_{[j],y} = \overline{\mu}_{[j],y} + 2\overline{\epsilon}_{[j],y}. \quad (7)$$

We will now discuss how to estimate $\bar{\mu}_{[j],x}$ (or $\bar{\mu}_{[j],y}$) and $\bar{\epsilon}_{[j],x}$ (or $\bar{\epsilon}_{[j],y}$).

8.1.1 Estimating Object Movements

Each node in the PA-tree maintains the coefficients for the linear approximation to the trajectories. We estimate $\bar{\mu}_{[j],x}$ and $\bar{\mu}_{[j],y}$ as follows: We use the linear approximation for each segment s to compute a *time-parameterized MBR* during ω_s, the overlap between s's lifetime and query temporal interval $[t_1, t_2]$. A time-parameterized MBR for a trajectory segment is the MBR, which covers the approximated linear movement during ω_s, and can easily be obtained from the parametric representations of PA-tree. During the temporal interval ω_s, the PA-tree may be regarded as a time-parameterized R-tree over those MBRs in the native space. Under this formulation, we can apply the cost model for general R-tree [34] to estimate the spatial extent $\bar{\mu}_{[j],x} \times \bar{\mu}_{[j],y}$ in the PA-tree.

From Lemma 1, the approximated linear X- and Y-movements for trajectory segment s during ω_s are

$$\hat{\mu}_{s,x} = \frac{2c_{s,x}^{(1)} \cdot |\omega_s|}{l} \text{ and } \hat{\mu}_{s,y} = \frac{2c_{s,y}^{(1)} \cdot |\omega_s|}{l},$$

where $c_{s,x}^{(1)}$ and $c_{s,y}^{(1)}$ are the linear coefficients for s and $|\omega_s|$ is the length of overlap ω_s.

Therefore, in the native space, the trajectory segment will be approximated by a time-parameterized MBR with area

$$a_s = \hat{\mu}_{s,x} \times \hat{\mu}_{s,y} = 4\frac{c_{s,x}^{(1)} \cdot c_{s,y}^{(1)} \cdot |\omega_s|^2}{l^2}.$$

To use the R-tree [34] cost model in our analysis, we must estimate the density D_j for the set of time-parameterized MBRs of nodes at level j during ω_n. As in [34], the density of a set of MBRs is simply the total size of MBRs when the entire space is an unit square. In our case, the density D_0 at level 0 would be the total size of all time-parameterized MBRs during ω_s for all trajectory segment s ($1 \le s \le N_i$). That is,

$$D_0 = \sum_{s=1}^{N_i} a_s = \sum_{s=1}^{N_i} 4\frac{c_{s,x}^{(1)} \cdot c_{s,y}^{(1)} \cdot |\omega_s|^2}{l^2} = \frac{4}{l^2} \sum_{s=1}^{N_i} |\omega_s|^2 c_{s,x}^{(1)} c_{s,y}^{(1)}. \quad (8)$$

As in [34], the density D_j ($1 \le j \le h_i$) at level j becomes:

$$D_j = \left(1 + \frac{\sqrt{D_{j-1}} - 1}{\sqrt{\bar{f}}}\right)^2.$$

Now, $\bar{\mu}_{[j],x}$ and $\bar{\mu}_{[j],y}$ can be computed as follows:

$$\overline{\mu}_{[j],x} = \overline{\mu}_{[j],y} = \sqrt{\frac{D_j}{M_{i,j}}} = \sqrt{\frac{D_j f^j}{N_i}}.$$

8.1.2 Estimation of Deviation Errors

To estimate the average of maximum deviation error $\bar{\epsilon}_{[j],x}$ or $\bar{\epsilon}_{[j],y}$ for the nodes at level j, we observe that the deviation error is an augmented attribute in PA-trees and does not affect the node insert/split procedure when constructing PA-trees. The deviation errors are hence independent of the polynomial coefficients and we proceed to estimate deviation error as follows:

The deviation errors $\epsilon_{s,x}^{(1)}$ or $\epsilon_{s,y}^{(1)}$ for trajectory segments s ($1 \le s \le N_i$) are randomly assigned into $M_{i,j}$ buckets. Then, we use the average of the maximum deviation error of the $M_{i,j}$ buckets as the estimation of $\bar{\epsilon}_{[j],x}$ and $\bar{\epsilon}_{[j],y}$.

8.1.3 Total Cost for Filtering Step

Using our estimates of $\bar{\mu}_{[j],x}$ or $\bar{\mu}_{[j],y}$ and $\bar{\epsilon}_{[j],x}$ or $\bar{\epsilon}_{[j],y}$, we can now compute the average spatial extent $\bar{\sigma}_{[j],x}$ and $\bar{\sigma}_{[j],y}$ of the node at level j using (7). The total number of disk accesses $\text{DA}_i^f(q)$ over the PA_i is:

$$\text{DA}_i^f(q) = \frac{(l + |t_2 - t_1|)}{T} \sum_{j=1}^{h_i} \left(M_{i,j}(\overline{\sigma}_{[j],x} + q_x)(\overline{\sigma}_{[j],y} + q_y)\right)$$

$$= \frac{(l + |t_2 - t_1|)}{T} \sum_{j=1}^{h_i} \left(\frac{N_i}{f^j}(\overline{\sigma}_{[j],x} + q_x)(\overline{\sigma}_{[j],y} + q_y)\right).$$

8.2 Cost Model for Refinement Step

Given a trajectory segment s, let λ_s denote its temporal interval. Let $\Omega(\mathbf{s})$ be the event that λ_s overlaps with the query temporal interval $[t_1, t_2]$. As in the analysis of the filtering step, we have

$$\Pr[\mathbf{\Omega}(s)] = \frac{|\lambda_s| + |t_2 - t_1|}{T}.$$

Let $\mathbf{C}(\mathbf{s})$ be the event that trajectory segment s is a candidate segment for refinement step. As discussed in Section 4,

$$\Pr[\mathbf{C}(\mathbf{s})] = 4(\epsilon_{s,x}^{(k)} \cdot q_y + \epsilon_{s,y}^{(k)} \cdot q_x) + 8\frac{|\omega_s|}{l}$$

$$(\epsilon_{s,y}^{(k)} \cdot c_{s,x}^{(1)} + \epsilon_{s,x}^{(k)} \cdot c_{s,y}^{(1)}).$$

Therefore, the total number of disk access over PA_i during the refinement step is

$$\text{DA}_i^r(q) = \sum_{s=1}^{N_i} \Pr[\mathbf{\Omega}(\mathbf{s})]\Pr[\mathbf{C}(\mathbf{s})] = \frac{4}{T} \sum_{s=1}^{N_i} (|\lambda_s| + |t_2 - t_1|)$$

$$(\epsilon_{s,x}^{(k)} \cdot q_y + \epsilon_{s,y}^{(k)} \cdot q_x + 2\frac{|\omega_s|}{l}(\epsilon_{s,x}^{(k)} \cdot c_{s,y}^{(1)} + \epsilon_{s,y}^{(k)} \cdot c_{s,x}^{(1)})).$$

8.3 Analytical Method for Parameter Tuning

One application of the cost model is for tuning the parameter m for a PA-tree. Let $\mathrm{DA}_m(q)$ be the expectation of the total I/O cost when we use m intervals. To estimate the optimal or near-optimal value for m, we estimate $\mathrm{DA}_m(q)$ using the cost model, varying parameter m, and then choose the optimal value $\hat{m}_{opt}$ as the value of m when $\mathrm{DA}_m(q)$ is minimal.

Another approach to parameter tuning, which we have called the experimental method, was used in [12], [5]. This approach first constructs a set of index structure instances, one for each value of the parameter. It then evaluates a set of queries over each of these index instances and picks as the optimum the instance that yields the lowest cost. In Section 9, we will show that the analytical method for parameter tuning is far cheaper than the experimental method while yielding a near-optimal value for parameter m.

9 EXPERIMENTAL EVALUATION

Since no large-scale real trajectory data sets are currently available publicly, we generated synthetic data sets using the network-based generator of [2] and the road network in San Joaquin County, California. Our data sets were obtained by running the simulation for a total of 1,000 timestamps. We focus on the results on the data sets generated by this generator, which has been used extensively in previous work [5], [12], [37]. Further, as suggested by recent work [23], [9], modeling movement along roads is practically significant.

Data set SJ30k was generated by repeating the following procedure six times, each with different random seeds. Five thousand trajectories were generated each time, using six object classes, three external object classes, 3,000 initial objects, and two new objects per timestamp. This data set has 6,390,000 location updates and a size of 180 MB. Each object reports its position and movement function each time instant during its lifetime, so the number of movement functions for each object equals the number of instants in its lifetime.

We compared the PA-tree with the MVR-tree [13], [12] and SETI [5] for offline processing and with SETI for online processing. The MVR-tree is shown to be an efficient approach for offline processing, outperforming the 3D R-tree approach or piecewise R-tree (pw-Rtree) [12], while SETI is the first approach to support efficient online processing.

We implemented the PA-tree and SETI[1] with the Spatial Index Library of [11]. We used the MVR-tree implementation in [12], which uses the LAGreedy algorithm to model each trajectory with multiple MBRs. In the following figures, the legend `PA` represents our method, `MVR` represents the method of [13], [12], while `SETI` represents the method of [5].

9.1 Setup

Our experiments were run on an Intel Pentium IV 1.7 Ghz processor, with 512 Mbytes of main memory. We choose page size of 4 Kbyte in all experiments. Unlike [5], which

1. We reimplemented SETI since the original source code was not available.

TABLE 3
Experimental Parameters

Parameter	Setting
Page size	4K
Buffer size	10% of data size
Random disk access time	10 ms
Sequential I/O vs. Random I/O	1:20
q_l	5%,7%,10%
q_t	1 (0.1%), 50 (5%), 100 (10%)
size of query workload	1000
S for MVR-tree	10, 100, 200, 300, 600, 1000, 2000, 3000
num. m of intervals for PA-tree	5, 10, 20, 30, 40, 50, 60, 80, 100,200
num. C of cells for SETI	25, 49, 100, 256, 400, 625,961,1600,2500,3600

used a fixed size buffer even larger than the data size in some cases, we use a buffer with size being about 10 percent of the original data set, as suggested by Mamoulis and Papadias [18]. Unlike [12], we do not reset the buffer before executing every query since resetting the buffer will render the buffer useless when evaluating a workload of multiple queries. Further, as in [6], we assume the ratio of cost of sequential I/O to that of random I/O is 1:20 and charge 10ms for each random I/O.

We evaluated performance with respect to index structure size, varying the number of MBRs for the MVR-tree, the number of intervals for the PA-tree, and the number of grid cells for SETI. Let there be N trajectories and let the MVR-tree use $(1 + S)N$ MBRs. We varied S from 10 to 3,000. For the PA-tree, we varied m, the number of intervals that the temporal domain is split into, from 5 to 200. For SETI, we varied the number C of cells from 25 to 3,600.

We used various types of query workloads, each containing 1,000 queries, and varied $q_t = |t_2 - t_1|$, the length of the temporal interval, and the size of query spatial range. We chose $q_t = 1$ for timestamp queries, and $q_t = 50$ and $q_t = 100$ for medium and large time interval queries, respectively. As in [33], [5], each query's spatial range was a square uniformly distributed in the unit square, with the edge length $q_l = q_x = q_y$ being 5 percent, 7 percent, or 10 percent. The following figures report the average performance for each query. Table 3 shows the setup for our experiments.

9.2 Comparing Approximation Quality

To gauge the potential for improvement with our scheme, we compare the dead space using our method with that using the MBR approximation obtained from the LAGreedy algorithm [13]. This metric captures the pruning power of index structures based on the respective approximations. Larger amounts of dead space would suggest smaller pruning power since it will result in more refinement candidates.

The volume of each MBR is simply the product of the edge lengths along the X-dimension, Y-dimension, and the temporal dimension. Each entry is a 6-tuple, as discussed in Section 3. If we use $k + 1$ coefficients each to approximate the X-movements and Y-movements, the volume of dead space can be computed as $4\epsilon_x^{(k)}\epsilon_y^{(k)}(t_s - t_0)$, where $[t_0, t_s]$ is the temporal domain. The representation size is $2(k + 1) + 5$ since we represent $2(k + 1)$ coefficients in all, the value of k, as well as the maximum deviation and the temporal domain.
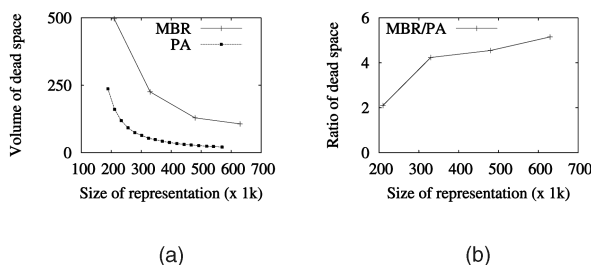
Fig. 6. Approximation quality. (a) Dead space. (b) Ratio.

Figs. 6a and 6b compare the quality of our method with that of MBR approximations for 5,000 trajectories when each coefficient, deviation error, coordinate, or k takes up 4 bytes. For a given representation size, our method has dead space up to 2-5 times smaller than that for MBR approximations, showing clear potential for improving query performance.

9.3 Accuracy of Cost Model

To demonstrate the accuracy of our analytical model, we compared the I/O cost estimated from the cost model against the I/O obtained from experiments. As explained in Section 8, we do not use buffers in this set of experiments. Figs. 7a, 7b, and 7c plot the average I/O cost against the number m of intervals, when q_l is 5 percent, 7 percent, and 10 percent, respectively. Clearly, our estimated I/O cost matches the experimental I/O cost very closely, with relative error no more than 25 percent. Furthermore, the estimated query cost captures quite closely the trade-off between m and the query performance, as we will discuss in Section 9.4.1.

Our ability to capture this trade-off enables us to obtain a near-optimal value for the number of intervals. Table 4 shows the excellent match between the value $\hat{m}_{opt}$ obtained from our cost model and the optimal value m_{opt} obtained from experiments. Figs. 7d, 7e, and 7f compare the I/O cost when $m = m_{opt}$ with the I/O cost when $m = \hat{m}_{opt}$. To see how much worse the I/O cost could be if m is not appropriately chosen, these figures also show the worst I/O cost. The close match suggests that our analytical cost model can yield a near-optimal value for the number of intervals.

9.3.1 Cost of Parameter Tuning

Fig. 7g compares the cost of *prospectively* tuning the PA-tree parameter m using our cost model against the cost of tuning it *retrospectively*, building indices and running queries for different m. Our prospective approach is cheaper by a factor of 4, even though, for the retrospective method, we only count index construction costs, completely ignoring the costs of tuning queries, which could be very high. Our approach is vastly superior.

9.4 Offline Processing

We tested the query performance for nonclustered indices, with the index file and data file being stored separately.[2] All data pages will be stored sequentially, according to the order of the start-time of the line segments, while each entry of leaf nodes will have a pointer to its data page. Since both index pages and data pages could be random I/O, we assign a 50 percent buffer to the index structure, while a 50 percent buffer is assigned to the data file.

9.4.1 Query Performance

In this set of experiments, we chose $q_l = 10\%$ and varied $q_t = 1$, 50, 100. Figs. 8, 9, and 10 compare the MVR-tree, SETI, and the PA-tree in terms of candidate size, CPU cost, I/O cost, and total query execution time, respectively. We varied S for the MVR-tree, number C of cells for SETI, and the number m of intervals for the PA-tree. For the MVR-tree and PA-tree, the candidate size is the number of trajectory segments that are examined during the refinement step. For SETI, the candidate size is the number of data pages that are examined during the refinement step. Each candidate trajectory segment and candidate data page requires at least one random disk I/O, except for a buffer hit.

As expected, the MVR-tree and the PA-tree reduce the size of candidate set for the refinement step with larger S or m. Increasing S and m reduces dead space and improves approximation quality, resulting in fewer candidates. In SETI, increasing the number of cells also increases the spatial discrimination. However, as the number of cells increases, trajectory segments are more likely to cross cell boundaries, increasing replication in both index and data file. With more cells, trajectory segments are also more likely to be scattered into more data pages since segments within different cells will be stored in different data pages. Therefore, increasing the number of cells may not reduce the number of candidates, especially when $q_t = 1$, as shown in Fig. 9a. Overall, we can clearly see that the PA-tree has significantly fewer candidate sets than the MVR-tree and SETI. Further, the comparison with the MVR-tree is consistent with the comparison in terms of approximation quality shown in Fig. 6a.

The filtering step with PA-trees computes polynomials, incurring CPU costs higher than that of the MVR-tree and SETI. However, the PA-tree has higher pruning power and yields a much smaller candidate set for the refinement step, greatly lowering CPU costs during refinement. Fig. 12b shows that the overall CPU cost for the PA-tree is actually better than that of the MVR-tree and comparable to SETI for higher q_t values, such as 50 or 100. SETI has the lowest CPU cost, especially for $q_t = 1$, since it uses regular cells in the spatial domain and a one-dimensional R-tree for the temporal domain, leading to a much simpler filtering step. This R-tree provides adequate discrimination for small q_t.

We have found our CPU costs to be typically about 3-5 percent of our I/O costs, if we charge 10ms for each random I/O. However, the work in [5] found CPU costs to constitute a big portion of the total query execution time. This difference arises for two reasons. First, SETI use a buffer as large as 64 MB for data sizes of 32 MB, 64 MB, and 128 MB, enabling SETI to store between 50-100 percent of the data in memory and to perform most operations in memory. In contrast, our buffer size is conservative and is

_____

2. We do not discuss clustered indices for lack of space. For all experiments, we report the candidate set size, which is independent of index structure.
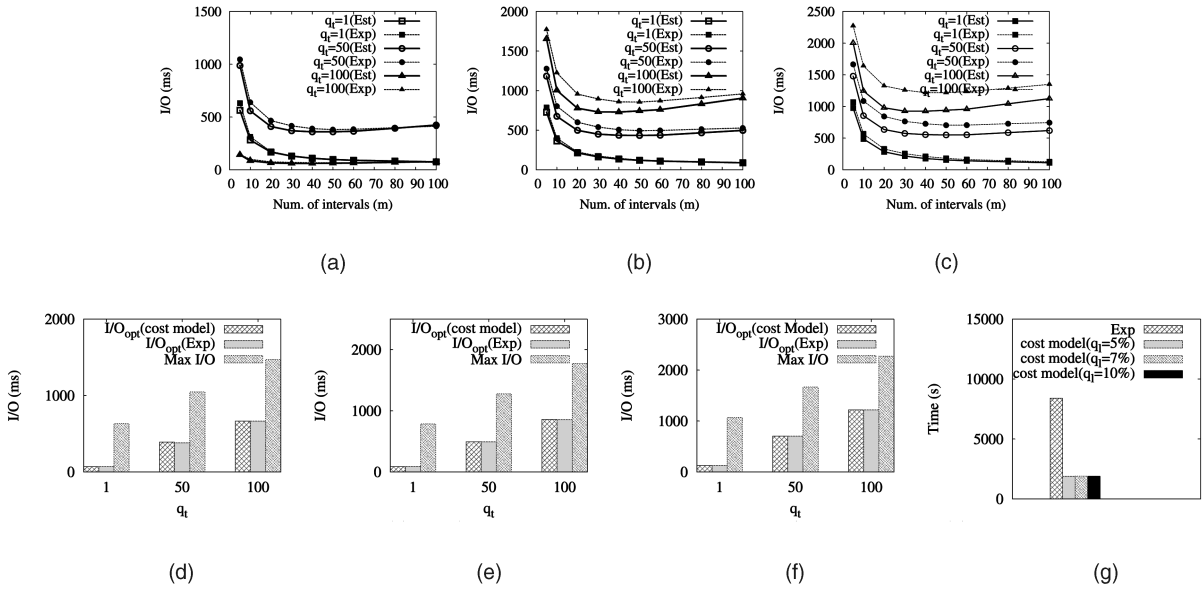
Fig. 7. Accuracy of cost model and cost of parameter tuning. (a) $q_l = 5\%$. (b) $q_l = 7\%$. (c) $q_l = 10\%$. (d) I/O ($q_l = 5\%$). (e) I/O ($q_l = 7\%$). (f) I/O ($q_l = 10\%$). (g) Cost of parameter tuning.

10 percent of data size, as suggested by Mamoulis and Papadias [18]. We focus instead on optimization of disk-based query execution. Also, SETI used a slow Pentium III 600 MHz machine, while we used a faster Pentium IV 1.7 Ghz machine. CPU speeds tend to improve much faster than I/O speeds and the bottleneck is typically I/O. We will therefore continue to focus on I/O costs.

For timestamp queries ($q_t = 1$), SETI has the smallest I/O cost, requiring 30 percent less I/O than the PA-tree and only as 1/8 times as the MVR-tree. When q_t is small, the temporal index in SETI, which is a sparse 1D R-tree, provides adequate temporal discrimination. Hence, a coarse grid cell approximation will not lead to a large number of candidates. However, as q_t increases to 50 and 100, more tqrajectory segments within each grid cell will overlap the query temporal interval, even when they do not intersect the query spatial range q_s. As a result, the I/O cost for SETI grows to 190-200 percent of that of the PA-tree. Surprisingly, although MVR-tree takes longer to build the index structure, SETI has a much better I/O performance than the MVR-tree for timestamp queries and slightly better I/O performance for time-interval queries.

Fig. 8c, 9c, and 10c capture some interesting trade-offs. As we increase the number of MBRs for the MVR-tree, the number m of intervals for the PA-tree or the number of cells for SETI, we have better pruning, smaller candidate sets, and lower refinement I/O costs. On the other hand, increasing these numbers also increases index size and filtering-step I/O costs and the number of segments that

trajectories will be split into for the MVR-tree and the PA-tree or the amount of duplication for SETI. Consequently, increasing these numbers yields no benefit beyond a certain point. This results in an upward trend in I/O, which is quite noticeable for $q_t = 50$ or $q_t = 100$. This trade-off also has be observed in [5].

9.4.2 Performance of Index Construction

Fig. 11 compares the CPU costs of building the MVR-tree, SETI, and the PA-tree. We set $S = 600$ for the MVR-tree, $C = 625$ for SETI, and $m = 40$ for the PA-tree, which are experimentally optimal or near-optimal for these methods.[3]

For the MVR-tree, building the index structures involved assigning MBRs to each trajectory, creating MBRs for each trajectory, and loading the MBRs into MVR-trees. As pointed in [12], the first two steps are extremely expensive since it requires one full database scan in order to compute the best approximation per trajectory. In contrast, building PA-trees is much more efficient since each trajectory can be processed individually. We split each trajectory into segments according to the temporal domain splits, estimate the degree of polynomial approximation, and insert the polynomial approximations into the PA-tree.

As Fig. 11 shows, MVR-tree build costs are about 25 times higher than that for PA-trees for the S30k data set. SETI builds indexes twice as fast as PA-trees since SETI constructs indices using regular grid cells. Within each grid cell, a dense one-dimensional R-tree is used to index temporal attributes, speeding up insertions. However, PA-trees have much better I/O performance than SETI for time interval queries, due to better approximation quality. Clearly, the PA-tree and SETI are better choices for very large data sets or when the trajectory data is collected at

3. We show only the CPU cost for building indices since the the MVR-tree [12] assigns MBRs with all trajectory segments in memory. I/O cost of online index construction for PA-trees and SETI appear in Section 9.5.
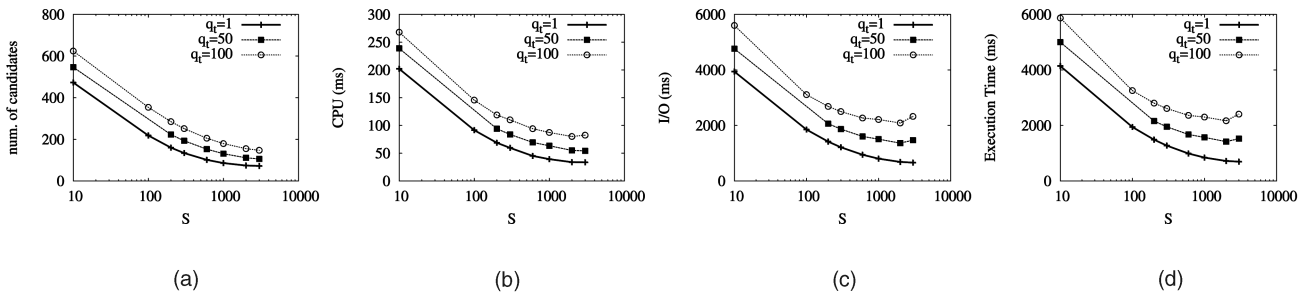
TABLE 4
m_{opt} and $\hat{m}_{opt}$

	$q_l = 5\%$			$q_l = 7\%$			$q_l = 10\%$		
q_t	1	50	100	1	50	100	1	50	100
m_{opt}	100	50	40	100	50	50	100	50	40
$\hat{m}_{opt}$	100	40	40	100	50	40	100	50	40

Fig. 8. MVR-tree performance.(a) Number of candidates. (b) CPU cost. (c) I/O. (d) Total cost.
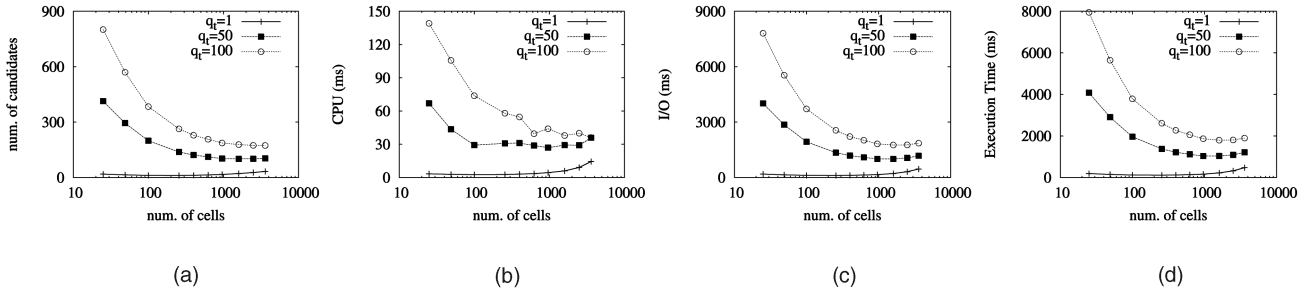

Fig. 9. SETI performance. (a) Number of candidates. (b) CPU cost. (c) I/O. (d) Total cost.
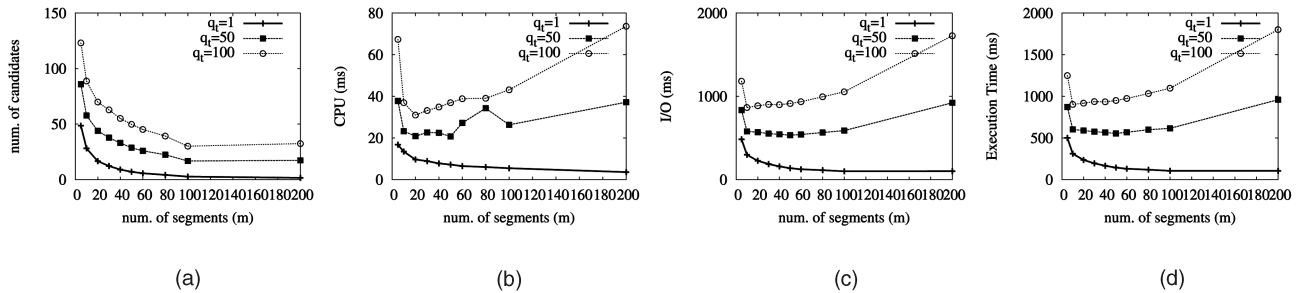

Fig. 10. PA-tree performance. (a) Number of candidates. (b) CPU cost. (c) I/O. (d) Total cost.

high rate, requiring online processing. Therefore, we only compare PA-tree and SETI for online processing.

9.5 Performance of Online Processing

We compare the PA-tree scheme with SETI [5] under the following settings: First, both SETI and PA-tree use front-line buffers. For SETI, we assign one buffer page for each cell, while, for PA-tree, we assign 10 percent of the available buffer space as the front-line buffer with the rest of the space assigned as for offline processing. Queries are issued in succession and each query may specify a time up to the instant it is issued. Finally, we set $m = 40$ for the PA-tree and the number of cells for SETI to 625, both of which are near-optimal according to the cost model, and experiment, respectively.
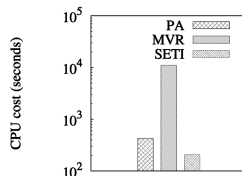

Fig. 11. Build time.

As in [5], we test insertion and query performance separately for online processing. We first insert 10k segments into the indices. Subsequently, for every 5k insertions, we execute a random spatio-temporal range query, with $q_t = 1$, or 50 or 100, and $q_l = 10\%$. We execute 1,275 random queries in all. We dynamically keep track of the query and insertion performance, measuring the performance of each set of 5k insertions and each query execution.

Figs. 13a, 13b, and 13c show the query I/O performance between the 500th and the 520th query, for $q_t = 1$, $q_t = 50$ and $q_t = 100$, respectively. To save space, we show the CPU cost for $q_t = 100$ in Fig. 13d since CPU cost is not a major bottleneck.

As with offline query processing, SETI incurs, on average, 30 percent lower I/O costs than the PA-tree when $q_t = 1$, as Fig. 13f shows. However, for $q_t = 50$ and $q_t = 100$, SETI requires as much as 200-250 percent the I/O of the PA-tree. Further, we see that, for $q_t = 50$ and $q_t = 100$, the PA-tree consistently outperforms SETI in terms of I/O, even for queries whose time interval overlaps the current time interval, requiring a scan of the front-line buffer and the intermediate list file. This is because scanning the inter-mediate list file requires only one random access plus several sequential accesses.[4] The PA-tree does incur more
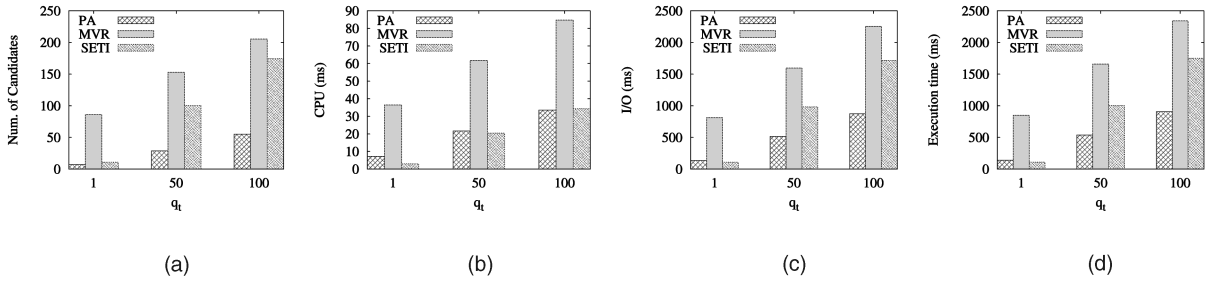
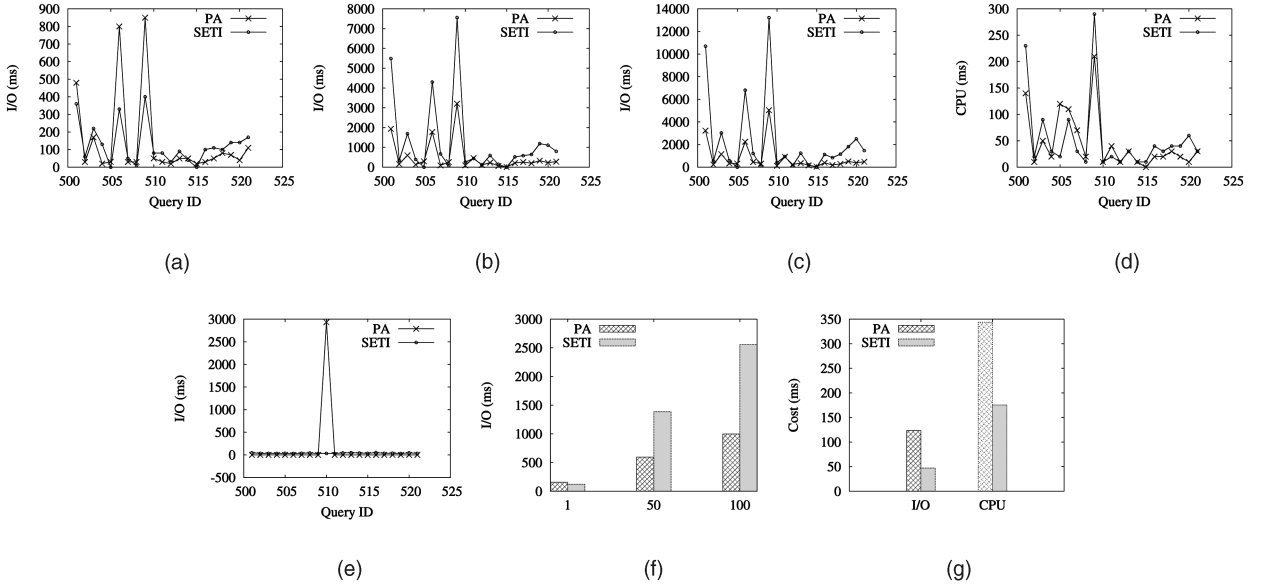Fig. 12. Best performance. (a) Number of candidates. (b) CPU cost. (c) I/O. (d) Total cost.

Fig. 13. Cost for online processing with PA-tree and SETI. (a) Query I/O ($q_t = 1$). (b) Query I/O ($q_t = 50$). (c) Query I/O ($q_t = 100$). (d) CPU ($q_t = 100$). (e) Update I/O. (f) Average query I/O. (g) Average update cost.

CPU costs for such queries, as Fig. 13d shows. However, the CPU cost is only about 3-5 percent of total execution time and is not a bottleneck for query execution.

The front-line and intermediate lists require that, at the end of each time interval, the trajectory segments in the intermediate list file must be entered in the PA-tree, leading to a delay in query execution. This can be clearly seen from the jump of 3,000ms in Fig. 13e, which shows the I/O performance for the 5k insertions between two consecutive queries. However, we observe that the delay for SETI could be as high as 12,000ms for some queries, about two times higher than for the PA-tree. Even accounting for the delay to build a PA-tree at the end of each time interval, the delay for query execution with the PA-tree is still far smaller than with SETI.

Fig. 13g shows the average performance of insertions for SETI and the PA-tree. As with offline index construction, SETI costs roughly half as much as the PA-tree for insertions.

In summary, for both offline and online processing, SETI incurs I/O cost for index construction lower than the PA-tree by a factor of 2 and requires slightly less I/O cost for

---

4. For $m = 40$, the intermediate list, in the worst case, could have size up to 1,125 pages. The worst-case I/O time will be $10ms + (1,125/20)^* 10ms = 573ms$. This is inexpensive considering that SETI takes up to 12,000 ms in I/O for some queries.

timestamp queries. However, the PA-tree outperforms SETI for time interval queries by a factor of 2. As our experiments show, time interval queries are the bottleneck, requiring much more I/O cost than that for timestamp queries or for index construction. Further, as pointed by Tao et al. [33], time interval queries are more general than timestamp queries in real applications. Therefore, we believe PA-tree will be a more appropriate choice for both offline and online processing historical trajectories.

10 CONCLUSIONS AND FUTURE WORK

We have presented a new parametric indexing method suitable for large trajectory data sets and for answering historical spatio-temporal queries efficiently. Our polynomial approximation method achieves much better performance than the MBR or grid-cell approximation. We show how to obtain conservative bounds for polynomial approximations and optimize its degree for a trajectory. We present the PA-tree, a two-tier structure for indexing trajectories using polynomial approximations. Our experiments demonstrate that PA-trees have excellent performance for offline and online spatio-temporal range queries compared to current trajectory indexing schemes, such as MVR-trees and SETI. In future work, we will investigate the

applicability of our methods to domains other than trajectory data, such as complex spatial objects.

ACKNOWLEDGMENTS

REFERENCES

[1] M. Barth, "UCR IntelliShare Project," http://evwebsvr.cert.ucr.edu/intellishare/, 2005.

[2] T. Brinkhoff, "Generating Network-Based Moving Objects," *Proc. 12th Int'l Conf. Scientific and Statistical Database Management (SSDBM '00),* p. 253, 2000.

[3] F. Burton, J. Kollias, V. Kollias, and D. Matsakis, "Implementation of Overlapping B-trees for Time and Space Efficient Representation," *The Computer J.,* vol. 33, no. 3, pp. 279-280, 1990.

[4] Y. Cai and R. Ng, "Indexing Spatio-Temporal Trajectories with Chebyshev Polynomials," *Proc. ACM SIGMOD,* pp. 599-610, 2004.

[5] V.P. Chakka, A. Everspaugh, and J.M. Patel, "Indexing Large Trajectory Data Sets with SETI," *Proc. Conf. Innovative Data Systems Research (CIDR '03),* 2003.

[6] L. Chung, B. Worthington, R. Horst, and J. Gray, "Windows 2000 Disk IO Performance," Microsoft Technical Report MS-TR-2000-55, 2000.

[7] C. Faloutsos and I. Kamel, "Beyond Uniformity and Independence: Analysis of R-Trees Using the Concept of Fractal Dimension," *Proc. Symp. Principles of Database Systems (PODS '94),* pp. 4-13, 1994.

[8] C. Faloutsos, T. Sellis, and N. Roussopoulos, "Analysis of Object Oriented Spatial Access Methods," *Proc. ACM SIGMOD,* pp. 426-439, 1987.

[9] S. Gupta, S. Kopparty, and C.V. Ravishankar, "Roads, Codes and Spatiotemporal Queries," *Proc. Symp. Principles of Database Systems (PODS '04),* pp. 115-124, 2004.

[10] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD,* pp. 47-57, 1984.

[11] M. Hadjieleftheriou, "Spatial Index Library," http://www.cs.ucr.edu/~marioh/spatialindex/index.html, 2005.

[12] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V.J. Tsotras, "Indexing Spatio-Temporal Archives," *VLDB J.,* vol. 15, no. 2, pp. 143-164, 2006.

[13] M. Hadjieleftheriou, G. Kollios, V.J. Tsotras, and D. Gunopulos, "Efficient Indexing of Spatiotemporal Objects," *Proc. Int'l Conf. Extending Database Technology (EDBT '02),* pp. 251-268, 2002.

[14] G. Kollios, D. Gunopulos, and V.J. Tsotras, "On Indexing Mobile Objects," *Proc. Symp. Principles of Database Systems (PODS '99),* pp. 261-272, 1999.

[15] G. Kollios, V.J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou, "Indexing Animated Objects Using Spatiotemporal Access Methods," *IEEE Trans. Knowledge and Data Eng.,* vol. 13, no. 5, pp. 758-777, Sept./Oct. 2001.

[16] A. Kumar, V.J. Tsotras, and C. Faloutsos, "Designing Access Methods for Bitemporal Databases," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 1, pp. 1-20, Jan./Feb. 1998.

[17] M.-L. Lo and C.V. Ravishankar, "The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 1, pp. 136-152, Jan./Feb. 1998.

[18] N. Mamoulis and D. Papadias, "Slot Index Spatial Join," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 1, pp. 211-231, Jan./Feb. 2003.

[19] J.C. Mason and D. Handscomb, *Chebyshev Polynomials.* Chapman and Hall, 2003.

[20] Federal Communications Commision, "Enhanced 911," http://www.fcc.gov/911/enhanced/, 2005.

[21] M.A. Nascimento and J.R.O. Silva, "Towards Historical R-Trees," *Proc. ACM Symp. Applied Computing,* pp. 235-240, 1998.

[22] J. Ni and C.V. Ravishankar, "PA-Tree: A Parametric Indexing Scheme for Spatio-Temporal Trajectories," *Proc. Ninth Int'l Symp. Spatial and Temporal Databases (SSTD '05),* pp. 254-272, 2005.

[23] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query Processing in Spatial Network Databases," *Proc. Very Large Data Bases Conf. (VLDB '03),* pp. 802-813, 2003.

[24] J.M. Patel, Y. Chen, and V.P. Chakka, "STRIPES: An Efficient Index for Predicted Trajectories," *Proc. ACM SIGMOD,* pp. 637-646, 2004.

[25] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Object Trajectories," *Proc. Very Large Data Bases Conf. (VLDB '00),* pp. 395-406, 2000.

[26] K. Porkaew, I. Lazaridis, and S. Mehrotra, "Querying Mobile Objects in Spatio-Temporal Databases," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD '01),* pp. 59-78, 2001.

[27] S. Rasetic, J. Sander, J. Elding, and M.A. Nascimento, "A Trajectory Splitting Model for Efficient Spatio-Temporal Indexing," *Proc. Very Large Data Bases Conf.,* pp. 934-945, 2005.

[28] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD,* pp. 331-342, 2000.

[29] Z. Song and N. Roussopoulos, "SEB-Tree: An Approach to Index Continuously Moving Objects," *Mobile Data Management,* pp. 340-344, 2003.

[30] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu, "Prediction and Indexing of Moving Objects with Unknown Motion Patterns," *Proc. ACM SIGMOD,* pp. 611-622, 2004.

[31] Y. Tao and D. Papadias, "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries," *Proc. Very Large Data Bases Conf. (VLDB '01),* pp. 431-440, 2001.

[32] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. Very Large Data Bases Conf. (VLDB '03),* pp. 790-801, 2003.

[33] Y. Tao, D. Papadias, and J. Zhang, "Cost Models for Overlapping and Multiversion Structures," *ACM Trans. Database Systems,* vol. 27, no. 3, pp. 299-342, 2002.

[34] Y. Theodoridis and T. Sellis, "A Model for the Prediction of R-Tree Performance," *Proc. Symp. Principles of Database Systems (PODS '96),* pp. 161-171, 1996.

[35] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems (ICMCS '96),* 1996.

[36] X. Xu, J. Han, and W. Lu, "RT-Tree: An Improved R-Tree Index Structure for Spatiotemporal Databases," *Proc. Int'l Symp. Spatial Data Handling,* 1990.

[37] H. Zhu, J. Su, and O.H. Ibarra, "Trajectory Queries and Octagons in Moving Object Databases," *Proc. Conf. Information and Knowledge Management (CIKM '02),* pp. 413-421, 2002.

Jinfeng Ni received the bachelor's and master's degree in computer science from the University of Science and Technology of China. He is currently a PhD candidate in the Department of Computer Science and Engineering at the University of California, Riverside. His research interests include database query processing and security.

Chinya V. Ravishankar received the undergraduate degree in chemical engineering from the Indian Institute of Technology, Bombay, and the PhD degree in computer science from the University of Wisconsin–Madison. He has been with the University of California at Riverside since Fall 1999, where he is currently a professor of computer science and engineering and associate dean of the Bourns College of Engineering. Between 1986-1999, he was on the faculty of the Electrical Engineering and Computer Science Department at the University of Michigan–Ann Arbor. Professor Ravishankar's research has been broadly in the area of software systems. More recently, he has worked in the areas of databases, networking, and security. He is a senior member of the IEEE and a member of the ACM.