

Closest-Point-of-Approach Join for Moving Object Histories

Subramanian Arumugam, Christopher Jermaine
Computer Science and Engineering Department
University of Florida, Gainesville, FL
{sa2,cjerman}@cise.ufl.edu

Abstract

In applications that produce a large amount of data describing the paths of moving objects, there is a need to ask questions about the interaction of objects over a long recorded history. In this paper, we consider the problem of computing joins over massive moving object histories. The particular join that we study is the “Closest-Point-Of-Approach” join, which asks: Given a massive moving object history, which objects approached within a distance ‘d’ of one another? We carefully consider several relatively obvious strategies for computing the answer to such a join, and then propose a novel, adaptive join algorithm which naturally alters the way in which it computes the join in response to the characteristics of the underlying data.

1. Introduction

In this paper, we investigate the spatial-temporal join operation for massive, disk-based moving object histories in three-dimensional space, with time considered as the fourth dimension. The spatio-temporal join operation considered in this paper is the *CPA Join (Closest-Point-Of-Approach Join)*. By *Closest Point of Approach*, we refer to a position at which two dynamically moving objects attain their closest possible distance. Formally, in a CPA Join, we answer queries of the following type: “*Find all object pairs $(p \in P, q \in Q)$ from relations P and Q such that CPA-distance $(p, q) \leq d$ ”.* There are many uses for such a join operation. Our particular motivation for studying this problem is in post-processing the results of large-scale computer simulations. For just one example, imagine that we simulate the growth of two areas of contaminated groundwater in a particular area over time, and ask the question: when and where will the two regions be close enough to interact? If the regions are represented as time-varying triangular meshes then it is possible to reduce this question to a CPA Join over the triangles in the two meshes.

Surprisingly, this sort of large-scale computational ge-

ometry problem has not been studied previously by the database community. There has been some work related to joins involving moving objects [8] [18] but the work has been restricted to objects in a limited time window and does not consider the problem of joining object histories that may be gigabytes or terabytes in size.

The specific contributions of our research are as follows:

(1) We address the important problem of spatio-temporal joins over moving object histories and describe simple adaptations of existing algorithms based on the R-tree structure and plane-sweep.

(2) We propose a novel adaptive join algorithm for moving object histories based on extension of the plane-sweep technique. The adaptive strategy dynamically tunes the join to the characteristics of the underlying data. The technique is scalable and requires only a single scan of the data.

(3) We rigorously evaluate and benchmark the alternatives.

The rest of this paper is organized as follows: In Section 2, we review the closest point of approach problem. In Sections 3 and 4, we describe two obvious alternatives to implementing the CPA join – using R-trees and plane-sweeping. In Section 5, we present a novel adaptive plane-sweeping technique that outperforms competing techniques considerably. Results from our benchmarking experiments are presented in Section 6. In Section 7, we offer our conclusions along with pointers for future work.

2. Background

In this Section, we discuss the motion of moving objects, and give an intuitive description of the CPA problem. We then present an analytic solution to the CPA problem over a pair of points moving in a straight line.

2.1. Moving Object Trajectories

Trajectories describe the motion of objects in a 2 or 3-dimensional space. Real-world objects tend to have smooth

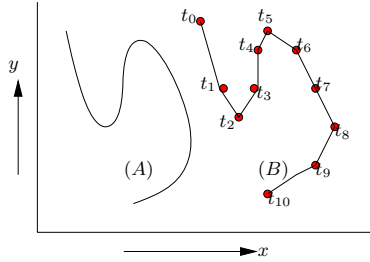


Figure 1. Trajectory of an object (A) and its polyline approximation (B)

trajectories and storing them for analysis often involves approximation to a *polyline*. A *polyline* approximation of a trajectory connects object positions, sampled at discrete time instances, by line segments (Figure 1). In a database the trajectory of an object can be represented as a sequence of the form $\langle (t_1, \vec{v}_1), (t_2, \vec{v}_2), \dots, (t_n, \vec{v}_n) \rangle$ where each \vec{v}_i represents the position vector of the object at time instance t_i . For the sake of simplicity we will consider only trajectories with linear approximations in this paper, though our algorithms are suitable for use with more complicated approximation schemes [19][9]. All our algorithms apply to other geometries besides points (such as cubes or time-deforming triangles), though some of the constituent computations (especially the distance and bounding computations) would become more complicated.

2.2. Closest Point of Approach (CPA) Problem

We are now ready to describe the CPA problem. For two objects in motion, the minimum distance $dist_{cpa}$ between them is the distance between the object positions at their closest point of approach. Let $CPA(p, q, d)$ over two objects p and q on straight line trajectories be evaluated as follows: If the distance $dist_{cpa}$ between the objects is less than or equal to d , return *true*, otherwise *false*. For trajectories consisting of a chain of line-segments, we apply the same basic procedure $CPA(p, q, d)$ between all pairs of line-segments and then choose the minimum distance.

Calculating the CPA time t_{cpa} . To compute t_{cpa} we use the following procedure. It is straightforward to calculate $dist_{cpa}$ once the CPA time t_{cpa} , time instance at which the objects reached their closest distance, is known (Figure 2). The position of objects p and q at any time instance t is given by $p(t)$ and $q(t)$. Let their positions at time $t = 0$ be p_0 and q_0 and let their velocity vectors per unit of time be \vec{u} and \vec{v} . The motion equations for these two objects are: $p(t) = p_0 + t\vec{u}$; $q(t) = q_0 + t\vec{v}$. At any time instance t , the distance between the two objects is given by $d(t) = |p(t) - q(t)|$.

Using basic calculus, one can find the time instance t_{cpa} at which the distance $d(t)$ is minimum (when $D(t) = d(t)^2$

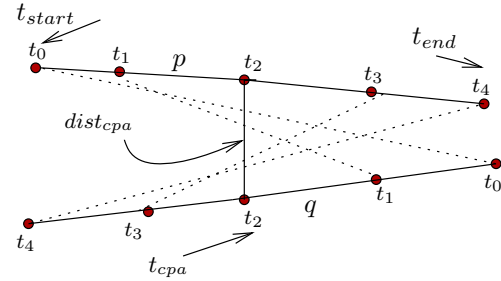


Figure 2. CPA Illustration. Dotted-lines indicate distance between objects p and q at various time instances

is a minimum). Solving for this time we obtain: $t_{cpa} = \frac{-(p_0 - q_0) \cdot (u - v)}{|u - v|^2}$. Then, distance at CPA time $dist_{cpa}$ is given by $|p(t_{cpa}) - q(t_{cpa})|$. Two simple special cases need to be handled: (1) If $|u - v| = 0$ then we can use $t_{cpa} = 0$; (2) when t_{cpa} occurs outside the time range of segments considered, we can set t_{cpa} to either t_{start} or t_{end} (depending on whether t_{cpa} occurred in the past or in the future). The complete code for computing $CPA(p, q, d)$ over two line segments is given below:

Procedure $CPA(\text{segment } p, \text{segment } q, \text{distance } d)$

//segment defined by $[(x_{start}, x_{end}), (y_{start}, y_{end}), (z_{start}, z_{end}), (t_{start}, t_{end})]$

- 1 Consider the lines that contains p and q and find p_0 and q_0 on this line for $t = 0$
 - 2 Compute velocity vectors $\vec{u} = (p_{end} - p_{start}) / (t_{end} - t_{start})$ and similarly $\vec{v} = (q_{end} - q_{start}) / (t_{end} - t_{start})$
 - 3 Calculate CPA time $t_{cpa} = \frac{-(p_0 - q_0) \cdot (u - v)}{|u - v|^2}$
 - 4 Let $dist_{cpa} = |p(t_{cpa}) - q(t_{cpa})|$
 - 5 **if** ($dist_{cpa} \leq d$) **return true**;
 - 6 **return false**;
-

In the next two Sections, we consider two obvious alternatives for computing the *CPA Join*, where we wish to discover *all* pairs of objects (p, q) from two relations P and Q , where $CPA(p, q, d)$ evaluates to true. The first technique we describe makes use of an underlying R-tree index structure to speed up join processing. The second methodology is based on a simple plane-sweep.

3. CPA Join Using Indexing Structures

Given numerous existing spatio-temporal indexing structures, it is natural to first consider employing a suitable index to perform the join.

3.1. Spatio-temporal Indices

Many spatio-temporal indexing structures exist, though unfortunately most are not suitable for the CPA Join. For

example, a large number of indexing structures like the TPR-tree [13], R^{EXP} tree [1], TPR*-tree [17] have been developed to support predictive queries, where the focus is on indexing the future position of an object. However, these index structures are generally not suitable for CPA Join, where access to the entire history is needed.

Indexing structures like MR-tree [6], MV3R-tree [2], HR-tree [3] seem relevant since they are geared towards answering time instance queries, where all objects alive at a certain time instance are retrieved. The general idea behind these index structures is to maintain a separate spatial index for each time instance. However, such indices are meant to store discrete snapshots of an evolving spatial database, and are not ideal for use with CPA Join over continuous trajectories.

3.2. Trajectory Index Structures

More relevant are indexing structures specific to moving object trajectory histories like the TB-tree, STR-tree [7] and SETI [14]. TB-trees emphasize trajectory preservation since they are primarily designed to handle topological queries where access to entire trajectory is desired. The problem with TB-trees in the context of the CPA Join is that segments from different trajectories that are close in space or time will be scattered across nodes. Thus, retrieving segments in a given time window will require several random I/Os. More appropriate to the CPA Join is SETI [14]. SETI partitions two-dimensional space statically into non-overlapping cells and uses a separate spatial index for each cell. SETI's grid structure is an interesting idea for addressing problems with high variance in object speeds (we will use a related idea for the adaptive plane-sweep algorithm described later). However, it is not clear how to size the grid for a given data set, and sizing it for a join seems even harder. It might very well be that relation R should have a different grid for $R \bowtie S$ compared to $R \bowtie T$. Further, for a CPA Join over a limited history, SETI has no way of pruning the search space, since every cell will have to be searched.

3.3. R-tree Based CPA Join

Given these caveats, perhaps the most natural choice for the CPA Join is the R-tree [16]. The join problem has been studied extensively for R-trees and several spatial join techniques exist [10][11][12].

It is a very straightforward task to adapt the R-tree to index a history of moving object trajectories. The four-dimensional line segments making up each individual object trajectory (three space and one time) are simply treated as individual spatial objects and are indexed directly by the R-tree. We can then use one of the several standard R-tree

join algorithms for the CPA Join. The common approach to joins using R-trees employ carefully controlled synchronized traversal of the two R-trees to be joined [11].

Several optimizations are possible for an R-tree CPA Join. Since the CPA Join is typically over an archived history a dynamic index is not always necessary and efficient packing techniques can be used. Packed R-trees give huge performance improvements over standard dynamic R-trees with improved spatial discrimination and better search performance [4]. In our implementation of the CPA Join for R-trees, we make use of the STR packing algorithm [4] to build the trees (see Section 6). Because the pruning power of the time dimension is the greatest, we ensure that the trees are well-organized with respect to time by choosing time as the first packing dimension. It is also possible to make use of heuristic filters to speed up distance computation between two 3-dimensional rectangles (this is expensive since the closest point may be on arbitrary faces). Another possible optimization is to make use of a plane-sweep algorithm when a pair of nodes are expanded to speed up all pairs distance computation.

4. CPA Join Using Plane-Sweeping

Even given these optimizations, a plane sweep is probably a more suitable candidate than an index for a CPA Join. Plane-sweep is a powerful technique for solving proximity problems involving geometric objects in a plane and has previously been proposed [5] as a way to efficiently compute the spatial join operation. Compared to an index-based solution, plane-sweeping algorithms have the obvious advantage that they require no index. Furthermore, making the reasonable assumption that a CPA Join is performed using a sweep along the time dimension and that trajectories are added to the database from the oldest to the newest, a plane-sweep based CPA Join requires no pre-processing.

4.1. Basic CPA Join using Plane-Sweeping

Plane-sweep is an excellent candidate for use with the CPA Join because no matter what distance threshold is given as input into the join, two objects must overlap in the time dimension for there to be a potential CPA match. Thus, given two spatio-temporal relations P and Q , we could easily base our implementation of the CPA Join on a plane-sweep along the time dimension. We would begin a plane-sweep evaluation of the CPA Join by first sorting the intervals making up P and Q along the time dimension, as depicted in Figure 3. We then sweep a vertical line along the time dimension. A sweepline data structure D is maintained which keeps track of all line segments which are valid given the current position of the line along the time dimension. As the sweepline progresses, D is updated with

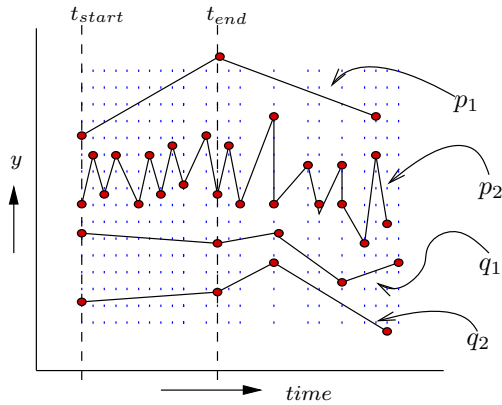


Figure 3. Progression of plane-sweep

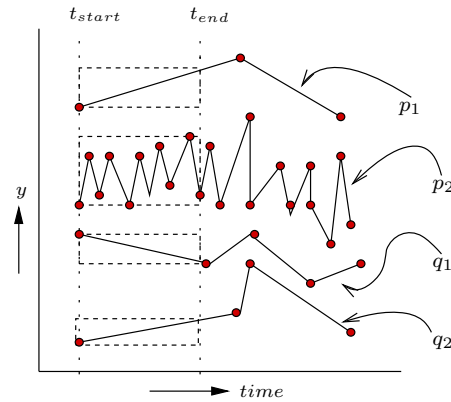


Figure 4. Layered Plane-Sweep

insertions (new segments that became active) and deletions (segments whose time period has expired). Segment pairs from both input relations that satisfy the join predicate are always present in D , and they can be checked and reported during updates to D . In the case of the CPA Join, assuming that all moving objects at any given moment can be stored in main memory, any of a number of data structures can be used to implement D , such as a quad- or oct-tree, or an interval skip-list. The main requirement is that the structure should be able to check easily proximity of objects in space.

4.2. Problem With The Basic Approach

Although the plane-sweep approach is simple, in practice it is usually too slow to be useful for processing moving object queries. The problem has to do with how the sweepline progression takes place. As the sweepline moves through the data space, it has to stop momentarily at sample points (time instances at which object positions were recorded) to process newly encountered segments into the data structure D . New segments that are encountered at the sample point are added into the data structure and segments in D that are no longer active are deleted from it.

Consequently, the sweepline pauses more often when objects with high sampling rates are present, and the progress of the sweepline is heavily influenced by the sampling rates of the underlying objects. For example, consider Figure 3 which shows the trajectory of four objects in a given time period. In the case illustrated, object p_2 controls the progression of the sweepline. Observe that in the time-interval $[t_{start}, t_{end}]$, only new segments from object p_2 get added to D but expensive join computations are performed each time with same set of line segments.

The net result is that if the sampling rate of a data set is very high relative to the amount of object movement in the data set, then processing a multi-gigabyte object history using a simple plane-sweeping algorithm may take a pro-

hibitively long time.

4.3. Layered Plane-Sweep

One way to address this problem is to reduce the number of segment level comparisons by comparing the *regions of movement* of various objects at a coarser level. For example, reconsider the CPA join depicted in Figure 3. If we were to replace the many oscillations of object p_2 with a single minimum bounding rectangle which enclosed all of those oscillations from t_{start} to t_{end} , we could then use that rectangle during the plane-sweep as an initial approximation to the path of object p_2 . This would potentially save many distance computations.

This idea can be taken to its natural conclusion by constructing a minimum bounding box that encompasses the line-segments of each object. A plane-sweep is then performed over the bounding boxes, and only qualifying boxes are expanded further. We refer to this technique as the *Layered Plane-Sweep* approach since plane-sweep is performed at two layers – one at a coarser level of bounding boxes and then at the finer level of individual line segments.

One issue that must be considered is how much movement is to be summarized within the bounding rectangle for each object. Since we would like to eliminate as many comparisons as possible, one natural choice would be to let the available system memory dictate how much movement is covered for each object. Given a fixed buffer size, the algorithm will proceed as follows:

Procedure *LayeredPlaneSweep* (**Rel** P , **Rel** Q , **distance** d)

1. **while** there is still some unprocessed data:
2. Read in enough data from relations P and Q to fill the buffer
3. Let t_{start} be the first time tick which has not yet been processed by the plane-sweep
4. Let t_{end} be the last time tick for which no data is still on disk
5. Next, bound trajectory of every object present in buffer by a MBR

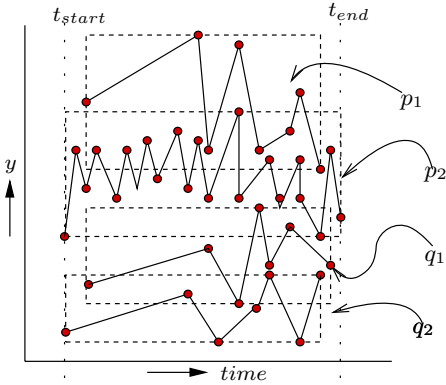


Figure 5. Problem with large granularities for bounding box approximation – High overlap

6. Sort the MBRs along one of the spatial dimension then perform a plane-sweep along that dimension
7. Expand the qualifying MBR pairs to get the actual trajectory data (line segments)
8. Sort the line segments along time
9. Now, perform a sweep along time dimension to get final result

Figure 4 illustrates the idea. It depicts the snapshot of object trajectories starting at some time instance t_{start} . Segments in the interval $[t_{start}, t_{end}]$ represent the maximum that can be buffered in the available memory. A first level plane-sweep is carried out over the bounding boxes to eliminate false positives. Qualifying objects are expanded and a second-level plane-sweep is carried out over individual line-segments. In the best case, there is an opportunity to process the entire data set through just three comparisons at the MBR level.

5. Adaptive Plane-Sweeping

While the layered plane-sweep typically performs far better than the basic plane-sweeping algorithm, it may not always choose the proper level of granularity for the bounding box approximations. This Section describes an adaptive strategy that takes into careful consideration the underlying object interaction dynamics and adjusts this granularity dynamically in response to the underlying data characteristics.

5.1. Motivation

In the simple layered plane-sweep, the granularity for the bounding box approximation is always dictated by the available system memory. The assumption is that pruning power increases monotonically with increasing granularity. Unfortunately, this is not always the case. As a motivating example, consider Figure 5. Assume available system

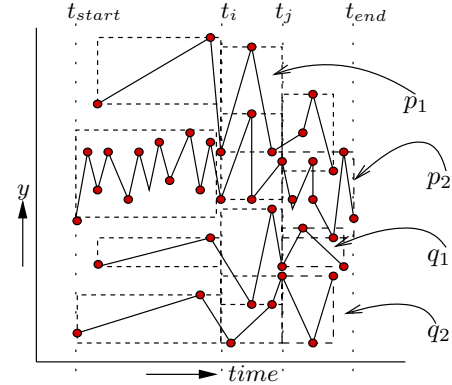


Figure 6. Adaptively varying the granularity opens up opportunities for pruning at the bounding box level

memory allows us to buffer all the line segments. In this case, the layered plane-sweep performs no better than the basic plane-sweep, due to the fact that all the object bounding boxes overlap with each other and as a result no pruning is achieved at the first-level plane-sweep.

However, assume we had instead fixed the granularity to correspond to the time period $[t_{start}, t_i]$, as depicted in Figure 6. In this case, none of the bounding boxes overlap, and there are possibly many dismissals at the first level. Though less of the buffer is processed initially, we are able to eliminate many of the segment-level distance comparisons compared to a technique that bounds the entire time-period, thereby potentially increasing the efficiency of the algorithm. The entire buffer can then be processed in a piece-by-piece fashion, as depicted in Figure 6. In general, the efficiency of layered plane-sweep is tied not to the granularity of the time interval that is processed, but the granularity that minimizes the number of distance comparisons.

5.2. Cost Associated With a Given Granularity

Since distance computations dominate the time required to compute a typical CPA Join, the cost associated with a given granularity can be approximated as a function of the number of distance comparisons that are needed to process the segments encompassed in that granularity. Let n_{MBR} be the number of distance computations at the box-level, let n_{seg} be the number of distance calculations at the segment-level, and let α be the fraction of the time range in the buffer which is processed at once. Then the cost associated with processing that fraction of the buffer can be estimated as: $cost_{\alpha} = (n_{seg} + n_{MBR})(1/\alpha)$

This function reflects the fact that if we choose a very small value for α , we will have to process many cut-points in order to process the entire buffer, which can increase the cost of the join. As α shrinks, the algorithm becomes

equivalent to the traditional plane-sweep. On the other hand, choosing a very large value for α tends to increase $(n_{seg} + n_{MBR})$, eventually yielding an algorithm which is equivalent to the simple, layered plane-sweep. In practice, the optimal value for α lies somewhere in between the two extremes, and varies from data set to data set.

5.3. The Basic Adaptive Plane-Sweep

Given this cost function, it is easy to design a greedy plane-sweep algorithm that attempts to repeatedly minimize $cost_\alpha$ in order to adapt the underlying (and potentially time-varying) characteristics of the data. At every iteration, the algorithm simply chooses to process the fraction of the buffer that appears to minimize the overall cost of the plane-sweep in terms of the expected number of distance computations. The algorithm is given below:

Procedure AdaptivePlaneSweep (Rel P , Rel Q , distance d)

1. **while** there is still some unprocessed data:
 2. Read in enough data from relations P and Q to fill the buffer
 3. Let t_{start} be the first time tick which has not yet been processed by the plane-sweep
 4. Let t_{end} be the last time tick for which no data is still on disk
 5. Choose α so as to minimize $cost_\alpha$
 6. Perform a layered plane-sweep from time t_{start} to $t_{start} + \alpha \times (t_{end} - t_{start})$
-

Unfortunately, there are two obvious difficulties involved with actually implementing the above algorithm: First, the cost $cost_\alpha$ associated with a given granularity is known only after the layered plane has been executed at that granularity. Second, even if we can compute $cost_\alpha$ easily, it is not obvious how we can compute $cost_\alpha$ for all values of α from 0 to 1 so as to minimize $cost_\alpha$ over all α . These two issues are discussed in detail in the next two Sections.

5.4. Estimating $Cost_\alpha$

This Section describes how to efficiently estimate $cost_\alpha$ for a given α using a simple, online sampling algorithm reminiscent of the algorithm of Hellerstein et al.[15].

At a high level, the idea is as follows. To estimate $cost_\alpha$, we begin by constructing bounding rectangles for all of the objects in P considering their trajectories from time t_{start} to $t_{start} + \alpha \times (t_{end} - t_{start})$. These rectangles are then inserted into an in-memory index, just as if we were going to perform a layered plane-sweep. Next, we randomly choose an object q_1 from Q , and construct a bounding box for its trajectory as well. This object is joined with all of the objects in P by using the in-memory index to find all bounding boxes within distance d of q_1 . Then, let n_{MBR,q_1} be the number of distance computations needed by the index

to compute which objects from P have bounding rectangles within distance d of the bounding rectangle for q_1 , and let n_{seg,q_1} be the total number of distance computations that would have been needed to compute the CPA distance between q_1 and every object $p \in P$ whose bounding rectangle is within distance d of the bounding rectangle for q_1 (this can be computed efficiently by performing a plane-sweep without actually performing the distance computations).

Once n_{MBR,q_1} and n_{seg,q_1} have been computed for q_1 , the process can be repeated for a second randomly selected object $q_2 \in Q$, for a third object q_3 and so on. A key observation is that after m objects from Q have been processed, the value $\hat{\mu}_m = \frac{1}{m} \sum_{i=1}^m (n_{MBR,q_i} + n_{seg,q_i})|Q|$ represents an unbiased estimator for $(n_{MBR} + n_{seg})$ at α , where $|Q|$ denotes the number of data objects in Q .

In practice, however, we are not only interested in $\hat{\mu}_m$. We would also like to know at all times just how accurate our estimate $\hat{\mu}_m$ is, since at the point where we are satisfied with our guess as to the real value of $cost_\alpha$, we want to stop the process of estimating $cost_\alpha$ and continue with the join.

Fortunately, the central limit theorem can easily be used to estimate the accuracy of $\hat{\mu}_m$. Assuming sampling with replacement from Q , for large enough m the error of our estimate will be normally distributed around $(n_{MBR} + n_{seg})$ with variance $\sigma_m^2 = \frac{1}{m} \sigma^2(Q)$, where $\sigma^2(Q)$ is defined as $\frac{1}{|Q|} \sum_{i=1}^{|Q|} \{(n_{MBR,q_i} + n_{seg,q_i})|Q| - (n_{MBR} + n_{seg})\}^2$. Since in practice we cannot know $\sigma^2(Q)$, it must be estimated via the expression $\hat{\sigma}^2(Q_m) = \frac{1}{m-1} \sum_{i=1}^m \{(n_{MBR,q_i} + n_{seg,q_i})|Q| - \hat{\mu}_m\}^2$ (Q_m denotes the sample of Q that is obtained after m objects have been randomly retrieved from Q). Substituting into the expression for σ_m^2 , we can treat $\hat{\mu}_m$ as a normally distributed random variable with variance $\frac{\hat{\sigma}^2(Q_m)}{m}$.

In our implementation of the adaptive plane-sweep, we can continue the sampling process until our estimate for $cost_\alpha$ is accurate to within $\pm 10\%$ at 95% confidence. Since 95% of the standard normal distribution falls within two standard deviations of the mean, this is equivalent to sampling until $2 \times \sqrt{\frac{\hat{\sigma}^2(Q_m)}{m}}$ is less than $\hat{\mu}_m \times 0.1$.

5.5. Determining The Best Cost

We now address the second issue: how to compute $cost_\alpha$ for all values of α from 0 to 1 so as to minimize $cost_\alpha$ over all α . Calculating $cost_\alpha$ for all possible values of α is prohibitively expensive and hence not feasible in practice. Fortunately, in practice we do not have to evaluate all values of α to determine the best α . This is due to the following interesting observation: *If we plot all possible values of α and their respective associated cost, we would observe that the graph is not linear, but exhibits a certain convexity. The convex region of the graph represents a sweet spot and rep-*

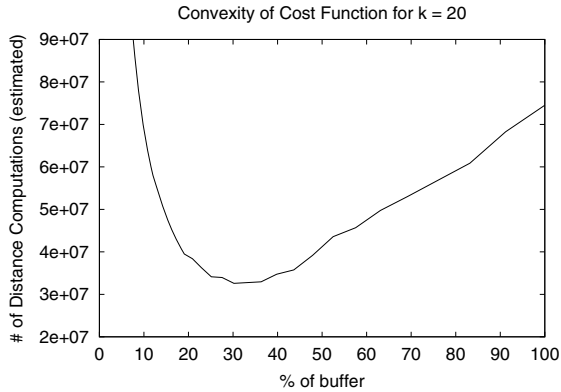


Figure 7. Convexity of cost function illustration.

resents the feasible region for the best cost. As an example consider Figure 7, which shows the plot of the cost function for various fractions of α for one of the experimental data sets from Section 6.

Given this fact, we identify the feasible region by evaluating $cost_{\alpha_i}$ for a small number, k , of α_i values. Given k (the number of allowed cutpoints), the fraction α_1 can be determined as follows: $\alpha_1 = \frac{r(\frac{1}{k})}{r}$ where $r = (t_{end} - t_{start})$ is the time range described by the buffer (the above formula assumes that r is greater than one; if not, then the time range should be scaled accordingly). In the general case, the fraction of the buffer considered by any $\alpha_i (1 \leq i \leq k)$ is given by: $\alpha_i = \frac{(r \cdot \alpha_1)^i}{r}$

Note that since the growth rate of each subsequent α_i is exponential, we can cover the entire buffer with just a small k and still guarantee that we will consider some value of α_i that is within a factor of α_1 from the optimal. After computing $\alpha_1, \alpha_2, \dots, \alpha_k$, we successively evaluate increasing buffer fractions, $\alpha_1, \alpha_2, \alpha_3$, and so on and determine their associated costs. From these k costs we determine the α_i with the minimum cost.

Note that if we choose α based on the evaluation of a small k , then it is possible that the optimal choice of α may lie outside the feasible region. However, there is a simple approach to solving this issue. Assume we chose α_i after evaluation of k cutpoints in the time range r . To further tune this α_i , we consider the time range defined between the adjacent cutpoints α_{i-1} and α_{i+1} and recursively apply cost estimation in this interval. (i.e., evaluate k points in the time range $(t_{start} + \alpha_{i-1} \times r, t_{start} + \alpha_{i+1} \times r)$). This approach is simple and very effective in considering a large number of choices of α .

5.6. Speeding Up the Estimation

Restricting the number of candidate cut points can help keep the time required to find a suitable value for α manage-

able. However, if the estimation is not implemented carefully, the time required to consider the cost at each of the k possible time periods can still be significant.

The most obvious method for estimating $cost_{\alpha}$ for each of the k granularities would be to simply loop through each of the associated time periods. For each time period, we would build bounding boxes around each of the trajectories of the objects in P , and then sample objects from Q as described in Section 5.4 until the cost was estimated with sufficient accuracy.

However, this simple algorithm results in a good deal of repeated work for each time period considered in cost estimation and can be speeded up considerably. In our implementation, we maintain a table of all the objects in P and Q , organized on the ID of each object. Each entry in the table points to a linked list that contains a chain of MBRs for the associated object. Each MBR in the list bounds the trajectory of the object for one of the k time-periods considered during the optimization, and the MBRs in each list are sorted from the coarsest of the k granularities to the finest.

Given this structure, we can estimate $cost_{\alpha}$ for each of the k values of α in parallel, with only a small hit in performance associated with an increased value for k . Any object pair $(p \in P, q \in Q)$ that needs to be evaluated during the sampling process described in Section 5.4 is first evaluated at the coarsest granularity corresponding to α_k . If the two MBRs are within distance d of one another, then the cost estimate for α_k is updated, and the evaluation is then repeated at the second coarsest granularity α_{k-1} . If there is again a match, then the cost estimate for α_{k-1} is updated as well. The process is repeated until there is not a match. As soon as we find a granularity at which the MBRs for p and q are not within a distance d of one another, then we can stop the process, because MBR pairs not within distance d for the time period associated with α_i cannot also be within this distance for any time period α_j where $j < i$.

The benefit of this approach is that in cases where the data are well-behaved and the optimization process tends to choose a value for α that causes the entire buffer to be processed at once, a quick check of the distance between the outer-most MBRs of p and q is the only geometric computation needed to process p and q , irrespective of k .

The bounding box approximations themselves can be formed while the system buffer is being filled with data from disk making MBR construction costs negligible. As trajectory data are being read from disk, we grow the MBRs for each α_i progressively. Since each α_i represents a fraction of the buffer, the updates to its MBR can be stopped as soon as that much fraction of the buffer has been filled. Similar logic can be used to shrink the MBRs when some fraction of the buffer is consumed and then refilled.

5.7. Putting It All Together

In our implementation of the adaptive plane-sweep, data from both the relations is fetched in blocks and stored in the system buffer. Then an optimizer routine is called which evaluates k granularities and returns the granularity α with the minimum cost. Data in the chosen granularity α is then evaluated using the `LayeredPlaneSweep` routine (described in Section 4.3). When the `LayeredPlaneSweep` routine returns, the buffer is refilled and the process is repeated. Techniques described in the previous Section are utilized to make the optimizer implementation fast and efficient.

6. Benchmarking

In this Section, we present experimental results comparing the various methods discussed in the paper for computing a spatio-temporal CPA Join: an R-tree, a simple plane-sweep, a layered plane-sweep, and an adaptive plane-sweep with several parameter settings.

6.1. Test Data Sets

The two data sets that we use to test the various algorithms result from two physics-based, N -body simulations. In both the data sets, constituent records occupy 80B on disk (80B is the storage required to record the object ID, time information, as well as the position and motion of the object). The size of each data set is around 50 gigabytes each. The data sets are as follows:

1. **Injection Data Set.** This data set is the result of a simulation of the injection of two gasses into a chamber through two nozzles on the opposite sides of the chamber via the depression of pistons behind each of the nozzles. Each gas cloud is treated as one of the input relations to the CPA Join. In addition to heat energy transmitted to the gas particles via the depression of the pistons, the gas particles also have an attractive charge. The purpose of the join is to determine the speed of the reaction resulting from the injection of the gasses into the chamber, by determining the number of (near) collisions of the gas particles moving through the chamber. Both data sets consist of 100,000 particles, and the positions of the particles are sampled at 3,500 individual time ticks, resulting in two relations that are around 28 gigabytes in size each. During the first 2,500 time ticks, for the most part both gasses are simply compressed in their respective cylinders. After tick 2,500, the majority of the particles begin to be ejected from the two nozzles.

2. **Collision Data Set.** This data set is the result of an N -body gravitational simulation of the collision of two small galaxies. Again, both galaxies contain around 100,000 star systems, and the positions of the systems in each galaxy are polled at 3,000 different time ticks. The size

of the relations tracking each galaxy is around 24 gigabytes each. For the first 1,500 or so time ticks, the two galaxies merely approach one another. For the next 1,000 time ticks, there is an intense interaction as they pass through one another. During the last few hundred time ticks, there is less interaction as the two galaxies have gone through one another. The purpose of the CPA Join is to find pairs of stars that approached closely enough to have a strong gravitational interaction.

6.2. Methodology and Results

All experiments were conducted on a 2.4GHz Intel Xeon PC with 1GB of RAM. The experimental data sets were each stored on an 80GB, 15,000 RPM Seagate SCSI disk.

For both the data sets, we tested an R-tree-based CPA Join (implemented as described in Section 3; we used the STR R-tree packing algorithm [4] to construct an R-tree for each input relation), a simple plane-sweep (implemented as described in Section 4), a layered plane-sweep (implemented as described in Section 5). We also tested the adaptive plane-sweep algorithm, implemented as described in Section 6. For the adaptive plane-sweep, we also wanted to test the effect of the two relevant parameter settings on the efficiency of the algorithm. These settings are the number of cut-points k considered at each level of the optimization performed by the algorithm, as well as the number of recursive calls made to the optimizer. In our experiments, we used k values of 5, 10, and 20, and we tested using either a single or no recursive calls to the optimizer.

The results of our experiments are plotted above in Figures 8 through 11. Figures 8 and 9 show the progress of the various algorithms as a function of time, for both the data sets (only Figure 8 depicts the running time of the adaptive plane-sweep making use of a recursive call to the optimizer). For the various plane-sweep-based joins, the x -axis of the plots shows the percentage of join that has been completed, while the y -axis shows the time required (in secs) to reach that point in the completion of the join. For the R-tree-based join (which does not progress through virtual time in a linear fashion) the x -axis shows the fraction of the MBR-MBR pairs that have been evaluated at each particular wall-clock time instant. These values are normalized so that they are comparable with the progress of the plane-sweep-based joins. Figures 10 and 11 show the buffer-size choices made by the adaptive plane-sweeping algorithm using $k = 20$ and no recursive calls to the optimizer, as a function of time for the test data sets.

6.3. Discussion

On both the data sets, the R-tree was clearly the worst option. The R-tree indices were not able to meaningfully

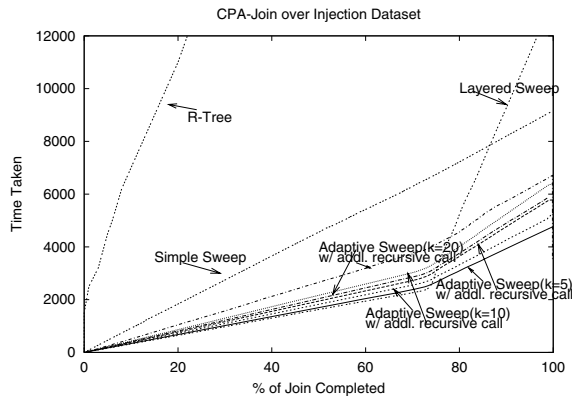


Figure 8. *Injection* data set execution time (in secs)

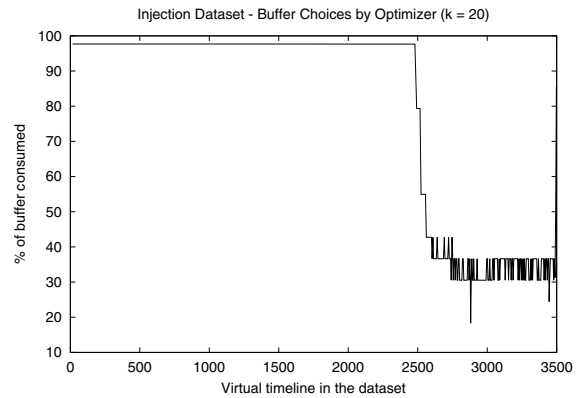


Figure 10. Buffer size choices for *Injection* data set

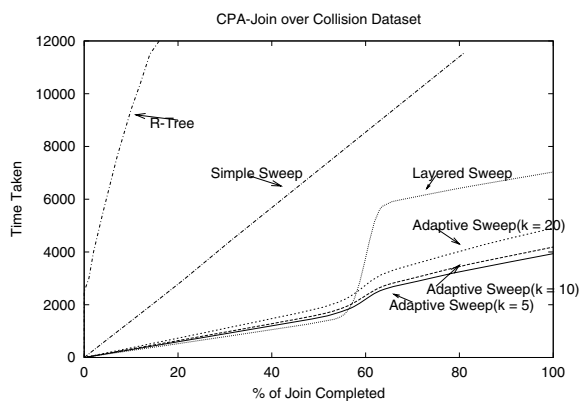


Figure 9. *Collision* data set execution time (in secs)

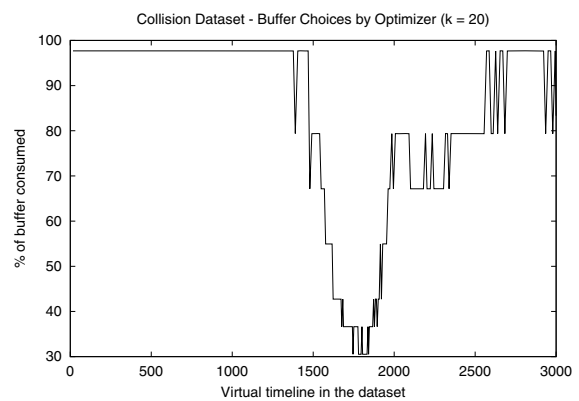


Figure 11. Buffer size choices for *Collision* data set

restrict the number of leaf-level pairs that needed to be expanded during join processing. This results in a huge number of segment pairs that must be evaluated. It may have been possible to reduce this cost by using a smaller page size (we used 64KB pages, a reasonable choice for a modern 15,000 RPM hard disk with a fast sequential transfer rate), but reducing the page size is a double-edged sword. While it may increase the pruning power in the index and thus reduce the number of comparisons, it may also increase the number of random I/Os required to process the join, since there will be more pages in the structure. Unfortunately, however, it is not possible to know the optimal page size until after the index is created and the join has been run, a clear weakness of the R-tree.

The standard plane-sweep and the layered plane-sweep performed comparably on the data sets we tested, and both far outperformed the R-tree. It is interesting to note that the standard plane-sweep performed well when there was much interaction among the input relations (when the gasses are expelled from the nozzles in the *Injection* data set and when the two galaxies overlap in the *Collision* data set). During such periods it makes sense to consider only very

small time periods in order to reduce the number of comparisons, leading to good efficiency for the standard plane-sweep. On the other hand, during time periods when there was relatively little interaction between the input relations, the layered plane-sweep performed far better because it was able to process large time-periods at once. Even when the objects in the input relations have very long paths during such periods, the input data were isolated enough that there tends to be little cost associated with checking these paths for proximity during the first level of the layered plane-sweep. The adaptive plane-sweep was the best option by far in all the data sets, and was able to smoothly transition from periods of low to high activity in the data and back again, effectively picking the best granularity at which to compare the paths of the objects in the two input relations.

From the graphs, we can see that the cost of performing the optimization causes the adaptive approach to be slightly slower than the non-adaptive approach when optimization is ineffective. In both the data sets, this happens in the beginning when the objects are moving towards each other but still far enough that no interaction takes place. As expected, adaptivity begins to take effect when the objects in

the underlying data set start interacting. From Figures 10 and 11, it can be seen that the buffer size choices made by the adaptive plane-sweep is very finely tuned to the underlying object interaction dynamics (decreasing with increasing interaction and vice versa). In both the *Injection* and *Collision* data sets, the size of the buffer falls dramatically just as the amount of interaction between the input relations increases.

The graphs also show the impact of varying the parameters to the adaptive plane-sweep routine, namely, the number of cut points k , considered at each level of the optimization, and whether or not a chosen granularity is refined through recursive calls to the optimizer. It is surprising to note that varying these parameters causes no significant changes in the granularity choices made by the optimizer. The reason is that with increasing interaction in the underlying data set, the optimizer has a preference towards smaller granularities and these granularities are naturally considered in more detail due to the logarithmic way in which the search space is partitioned.

Another interesting observation is that the recursive call does not improve the performance of the algorithm, for two reasons. First, since each invocation of the optimization is a separate attempt to find the best cut point in a different time range, it is not possible to share work among the recursive calls. Second, it is likely that just being in the feasible region, or at least a region close to it is enough to enjoy significant performance gains. Since the coarse first-level optimization already does that, further optimizations to fine tune the granularity do not seem necessary.

In all of our experiments, $k = 5$ with no recursive call to the optimizer uniformly gave the best performance. However, if the nature of the input data set is unknown and the data may be extremely poorly behaved, then we believe a choice of $k = 10$ with one recursive call may be a safer, all-purpose choice. On one hand, the cost of optimization will be increased, which may lead to a greater execution time in most cases (our experiments showed about a 30% performance hit associated with using $k = 10$ and one recursive call compared to $k = 5$). However, the benefit of this choice is that it is highly unlikely that such a combination would miss the optimal buffer choice in a very difficult scenario with a highly skewed data set.

7. Conclusion and Future Work

In this paper we addressed the challenging problem of computing joins over massive moving object histories. We compared and evaluated three different join strategies and proposed a novel join technique based on an extension to the plane-sweep. Our benchmarking results suggest that the proposed adaptive technique offers significant benefits over the competing techniques. One potential direction for fur-

ther development is to investigate the effect of precomputing MBR approximations, which can speedup the adaptive technique even more.

References

- [1] S. Saltenis and C.S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, 2002.
- [2] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*, 2001.
- [3] M. A. Nascimento and J. R. O. Silva. Towards Historical R-Trees. In *ACM SAC*, 1998.
- [4] S. Leutenegger and J. Edgington. STR: A Simple and Efficient Algorithm for R-tree Packing. In *ICDE*, 1997.
- [5] L. Arge and O. Procopiu and S. Ramaswamy T. Sui J. S. Vitter. Scalable Sweeping-Based Spatial Join. In *VLDB*, 1998.
- [6] X. Xu and W. Lu. An Improved R-Tree Indexing for Temporal Spatial Databases. In *SDH*, 1990.
- [7] D. Pfoser C. S. Jensen and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*, 2000.
- [8] M. Mokbel X. Xiong W. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.
- [9] Y. Tao C. Faloutsos D. Papadias B. Liu. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *SIGMOD Conference*, 2004.
- [10] T. Brinkhoff H. P. Kriegel B. Seeger. Efficient Processing of Spatial-joins using R-trees. In *SIGMOD*, 1993.
- [11] Y. W. Huang N. Jing E. A. Rundensteiner. Spatial Joins Using R-trees: Breadth-first Traversal With Global Optimizations. In *VLDB*, 1997.
- [12] O. Gunther. Efficient Computation of Spatial Joins. In *ICDE*, 1993.
- [13] S. Saltenis C. S. Jensen S. Leutenegger M. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.
- [14] J. Patel P. Chakka, A. Everspaugh. Indexing Large Trajectory Data Sets with SETI. In *CIDR*, 2003.
- [15] J. M. Hellerstein and H. J. Wang P. J. Haas. Online Aggregation. In *SIGMOD*, 1997.
- [16] R. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
- [17] Y. Tao D. Papadias J. Sun. The TPR*-Tree: An optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.
- [18] Y. Tao. Time-parametrized Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.
- [19] M. Hadjieleftheriou G. Kollios V. J. Tsotras. Efficient Indexing of Spatiotemporal Objects. In *EDBT*, 2002.