

# Spatial Data Structures for Efficient Trajectory-Based Queries

Jeremy Kubica, Andrew Moore, Andrew Connolly, and Robert Jedicke

CMU-RI-TR-04-61

November 2004

Robotics Institute  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213

© Carnegie Mellon University



## **Abstract**

Spatial queries involving trajectories of moving objects are fundamental in a variety of domains. For example, we may wish to determine which points or regions to which an object passes “close.” In this paper, we consider a large-scale version of this type of problem. Given many trajectories and spatial regions, we want to efficiently find all pairs of regions and trajectories such that the trajectory passes through the region.

Below we present several data structures and algorithms to efficiently solve this problem. We adapt data structures and algorithms from tracking and computer graphics to work on higher dimensional data sets with nonlinear tracks. These algorithms provide a significant speedup over a simple brute force approach. We also introduce a new data structure and algorithm that can significantly outperform previous approaches for queries with many tracks. Further, we introduce a novel dual-tree approach that combines the advantages of both an observation-based data structure and a track-based data structure to provide consistently good performance over a wide range of queries.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Problem Definition</b>	<b>2</b>
2.1	Input Data . . . . .	2
2.1.1	Tracks . . . . .	2
2.1.2	Plates . . . . .	2
2.2	Query Types . . . . .	2
2.2.1	Occurrence Queries . . . . .	2
2.2.2	Attribution Queries . . . . .	3
2.2.3	Precovery Queries . . . . .	3
<b>3</b>	<b>Algorithms</b>	<b>4</b>
3.1	Brute Force Computation . . . . .	4
3.2	KD-tree of Plates . . . . .	4
3.3	KD-tree of Tracks . . . . .	5
3.4	Ball-tree of Tracks . . . . .	6
3.5	Combining Trees . . . . .	7
<b>4</b>	<b>Experiments</b>	<b>9</b>
4.1	Algorithms . . . . .	9
4.2	Simulated Data . . . . .	9
4.2.1	Relative Set Sizes . . . . .	10
4.2.2	Relative Movement . . . . .	12
4.3	Astronomy Data . . . . .	12
4.4	Real Time Performance . . . . .	13
<b>5</b>	<b>Poor Track Models</b>	<b>14</b>
<b>6</b>	<b>Related Work</b>	<b>14</b>
<b>7</b>	<b>Conclusions</b>	<b>15</b>



# 1 Introduction

Imagine that you have just taken a picture of the night sky and you wish to identify each object in the picture. At your disposal you have a catalogue of millions of stars and galaxies and thousands of orbits of moving objects. For the most part, the stars and galaxies can easily be identified by matching them with the positions in the catalogue, because they do not move rapidly. In contrast, determining which moving objects lie in the image is more difficult because the answer depends on when the image was taken. As the problem scales up, by examining many such observations distributed across the sky, there rapidly becomes more object/regions pairs than can feasibly be examined.

The above problem is the occurrence or intersection problem. The goal is to take a set of object trajectories (called tracks) and observation regions (called plates) and determine which track/plate pairs have an intersection. Figure 1 provides a simple one dimensional illustration, with curves representing the trajectories and dashed intervals representing the spatial regions.

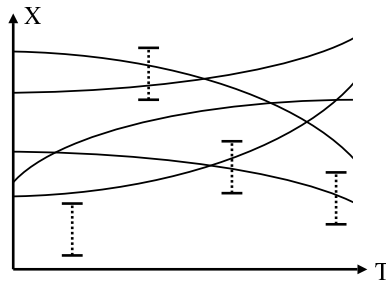


Figure 1: The goal of the intersection query is to determine which object tracks intersect the observation regions.

While motivated in the context of astronomical observations, this problem is applicable to a wide range of areas. For example, the intersection query arises in the domain of computer graphics for the problem of ray tracing. In ray tracing the tracks are linear paths representing a light ray and the plates are polygons from the objects in the scene.

Below we present several data structures and algorithms to efficiently solve this problem. These algorithms use spatial data structures to make significant computational savings. All of the algorithms are exact, returning every track/plate pair that meet our criteria. Further, we show that this problem is similar to two other problems in astronomy and tracking: attributions (determining which trajectories are near a point) and precoveries (determining which points are near a trajectory). We discuss the different domains on which each algorithm performs well and compare our approaches to previously suggested approaches.

## 2 Problem Definition

The type of problem that we are considering covers a range of related spatial queries. These queries all operate on two distinct types of input data: plates and tracks. Below we focus on three specific queries: occurrence, attribution, and precovery. These queries are highly related and can be efficiently answered using the same techniques.

### 2.1 Input Data

#### 2.1.1 Tracks

We allow a general definition of tracks as any function of an independent variable through the  $D$  dimensional space. We denote the  $i$ th track as  $\mathbf{g}_i(t)$  and use  $N_T$  to denote the number of tracks. Our discussion below focuses on two major types of tracks: linear and quadratic. The quadratic track is simply a quadratic function of time:

$$\mathbf{g}(t) = \mathbf{a} \cdot t^2 + \mathbf{b} \cdot t + \mathbf{c} \quad (1)$$

and can be used to describe physical motions of objects undergoing constant acceleration. The linear track is a linear function of time:

$$\mathbf{g}(t) = \mathbf{b} \cdot t + \mathbf{c} \quad (2)$$

and can be used to describe the physical motion of objects traveling at a constant velocity. While many of the techniques presented below will also apply to other track models, we restrict the discussion to the linear and quadratic models to keep the discussion simple and consistent.

#### 2.1.2 Plates

The *plates* represent spatial regions at a given time. Thus they contain two parts: subregions of the complete  $D$ -dimensional space, and an associated time  $t$ . Below we consider rectangular plates where each plate  $W$  contains:

1.  $t$  - The time of occurrence of the plate.
2.  $\mathbf{h}$  - A vector indicating the upper bound of the plate.
3.  $\mathbf{l}$  - A vector indicating the lower bound of the plate.

The input consists of  $N_P$  such plates. While we restrict our discussion to rectangular plates for simplicity, many of the approaches will apply to plates of other shapes.

### 2.2 Query Types

#### 2.2.1 Occurrence Queries

Formally the occurrence or intersection problem can be phrased as a filtering problem. Given a set of plates and a set of tracks, we want to return all of the plate/track

pairs,  $(W, \mathbf{g})$  such that:

$$W.\mathbf{l}[d] \leq \mathbf{g}(t)[d] \leq W.\mathbf{h}[d] \quad \forall d \quad (3)$$

Thus we wish to filter the  $N_T \cdot N_P$  possible pairs down to just the few that meet the above criteria.

### 2.2.2 Attribution Queries

The intersection problem is a generalization of an *attribution* query, which asks for all tracks that are within some range of a given point. Effectively we are asking to which tracks we can attribute the point. The observations consist of real-value coordinates in  $D$  dimensional space, with  $\mathbf{x}_i$  indicating the  $i$ th observation. As with the plates, we use  $t_i$  to indicate the independent variable of the  $i$ th observation. Formally, the attribution query can be specified as: given an observation  $\mathbf{x}$  return all tracks  $\mathbf{g}(t)$  that come within a given threshold  $\delta$  of  $\mathbf{x}$ :

$$|\mathbf{g}(t)[d] - \mathbf{x}[d]| \leq \delta[d] \quad \forall d \quad (4)$$

We can convert this query into an equivalent occurrence query by treating the observation as a small plate. Formally, we define a plate  $W$ :

$$\begin{aligned} W.t &= t \\ W.\mathbf{l}[d] &= \mathbf{x}[d] - \delta[d] \quad \forall d \\ W.\mathbf{h}[d] &= \mathbf{x}[d] + \delta[d] \quad \forall d \end{aligned} \quad (5)$$

We discuss the algorithms below in both the context of plate intersection queries and attribution queries.

This query can easily be adapted to handle other distance measures by adding a post-processing step. We first use a rectangular approximate of the threshold under the true distance measure to find an initial set of candidate tracks. This set is then further filtered using the true distance measure.

### 2.2.3 Precovery Queries

The precovery query is the complement to the attribution query. Given a new track  $\mathbf{g}$  we wish to determine which points lie within some threshold  $\delta$  of this track. Formally, the precovery query can be specified as: given a track  $\mathbf{g}(t)$  return all observations  $\mathbf{x}$  such that:

$$|\mathbf{g}(t)[d] - \mathbf{x}[d]| \leq \delta[d] \quad \forall d \quad (6)$$

This question is important for such tasks as finding previous sightings of newly discovered asteroids. As with the attribution query, we can answer the precovery question by treating the observations as small plates and solving the occurrence query.

### 3 Algorithms

#### 3.1 Brute Force Computation

Perhaps the simplest way to find all plate/track intersections is to exhaustively check each one. A brute force method runs a double loop through the data, comparing each plate with each track. This method is costly, requiring  $O(N_P \cdot N_T)$  intersection tests.

#### 3.2 KD-tree of Plates

A simple and intuitive approach to speeding up the intersection queries is to place the plates in a spatial data structure such as a KD-tree. This allows us to then use a tree search for plates that may intersect a given query track. This type of approach has been used in computer graphics to speed up ray-tracing with linear tracks [Glassner, 1984, Watt, 2000]. Because no two plates may occur at the same time, we may not gain any benefit from constructing a single tree at each time step. Instead we construct a *single*  $D + 1$  dimensional tree on all plates by incorporating time as a dimension. We refer to this data structure as a plate tree to distinguish it from KD-trees built on other inputs.

The tree is constructed in a recursive top-down fashion. At each level, the set of plates is split into two subsets that are recursively used to build the children nodes. The only additional concern is that unlike a set of points, we may not be able to cleanly split a set of plates. Therefore we use a slightly modified KD-tree construction algorithm similar to priority KD-trees [Uhlmann, 2001]. Specifically, we use the plate's *lower* bound corner to determine splitting. In other words we treat each plate as a point at its lower corner for the purposes of splitting the data. As in a standard KD-tree, we store the bounding box of the full plates. An example split and resulting nodes is shown in Figure 2.

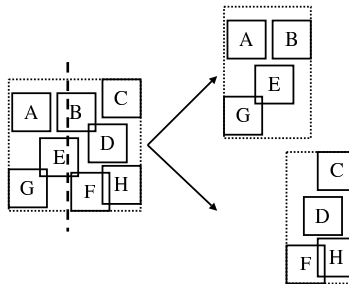


Figure 2: A set of plates can be split by which side of the dividing plane the lower bounds falls.

Our search for intersections then follows the same approach as a range search in a KD-tree. Given a query track  $\mathbf{g}(t)$  we traverse the tree in a depth first search. If we hit a leaf node, we explicitly search the plates at this leaf for intersection using the criteria

in Equation 3. Finally, we can prune the search if we ever find that the track cannot hit *any* plate contained within the bounds of the node. We can determine whether this criteria is met by taking the bounding box for the plate tree node (including time) and asking whether the track intersects this box. If it does not, then we can safely prune the search. In the below experiments we approximated this intersection test by testing each dimension separately.

### 3.3 KD-tree of Tracks

A natural complement to constructing a KD-tree on the plates is to construct one on the tracks. In many cases this approach might be a desirable alternative. For example, if we had a set of tracks and we wished to attribute a *single* new plate or observation to a track, we cannot achieve any speedup using a plate tree.

The difficulty arises from the fact that tracks will generally be large. They span the entire range of time and, if allowed sufficient movement, may span a large range of space. Thus placing bounds around the tracks and treating them as  $D + 1$  dimensional objects may lead to loose bounds and inefficient splits.

An alternative approach is to build the tree on tracks in parameter space. For many types of models we can represent each track as a vector of parameters. For example, we can represent a quadratic track as  $3D$  vector by concatenating  $\mathbf{a}$ ,  $\mathbf{b}$ , and  $\mathbf{c}$  into a single vector. Thus each track becomes a single point and the set of tracks can be split using hyper-planes in parameter space. An example of such a split is shown in Figure 3. We call this data structure a track-based KD-tree.

As with the plate tree, we store the bounding box for each node. However, this bounding box is on the parameters of the tracks. Therefore we are effectively storing the minimum and maximum parameters for each track owned by this node. We can use these to calculate the bounds on where a track can be at a given time. For example, in the quadratic case:

$$\begin{aligned} \mathbf{g}_{MIN}(t)[d] &= \mathbf{a}_{MIN}[d] \cdot t^2 + \mathbf{b}_{MIN}[d] \cdot t + \mathbf{c}_{MIN}[d] \\ \mathbf{g}_{MAX}(t)[d] &= \mathbf{a}_{MAX}[d] \cdot t^2 + \mathbf{b}_{MAX}[d] \cdot t + \mathbf{c}_{MAX}[d] \end{aligned} \quad (7)$$

Our search for intersection then follows the same approach as for the plate tree. We do a depth first search of the tree using the bounding box of the node to look for pruning opportunities. Upon hitting a leaf we explicitly check the tracks contained at that leaf.

This approach has a major drawback that should be noted. The position of a track depends on the entire combination of parameters weighted relative to the time of the query. Thus splitting on the individual parameters and bounding each parameter individually may not provide sufficiently tight bounds for significant pruning.

Again this type of approach has been used on linear tracks in computer graphics to speed up ray-tracing [Arvo and Kirk, 1987] and in databases to efficiently answer queries about moving objects [Kollios *et al.*, 1999, Saltenis *et al.*, 2000, Papadopoulos *et al.*, 2002]. We use this algorithm primarily as a comparison with more efficient track-based data structures.

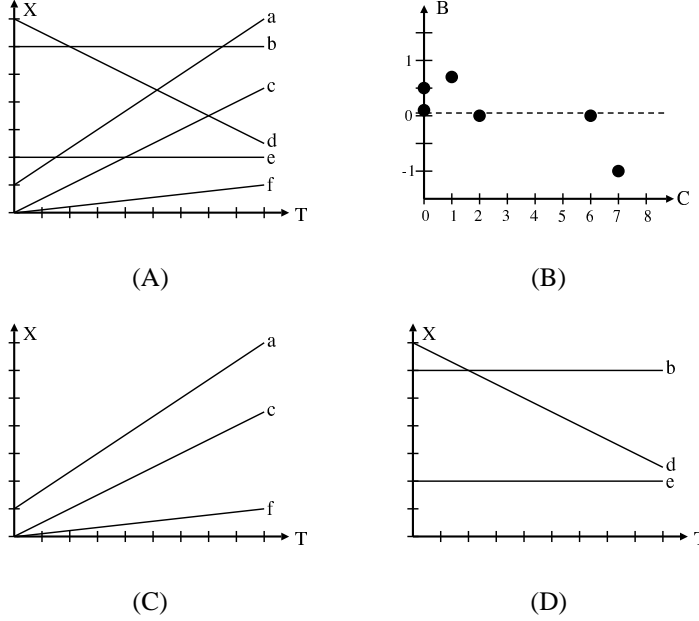


Figure 3: A set of linear tracks (A) is split using a dividing hyper-plane in parameter space (B). The resulting partition is shown in (C) and (D).

### 3.4 Ball-tree of Tracks

An alternative approach to dividing the tracks by their parameters is to partition the tracks based on their “proximity” to other tracks in the set over the time of interest. The hope is that this partitioning will allow better splits to the data that both accounts for multiple parameters and takes into consideration the entire scope of time. We introduce one such tree, which is similar to a ball-tree or metric tree [Ciaccia *et al.*, 1997, Uhlmann, 1991]. We refer to this data structure as a track-based ball-tree.

The key idea behind the track-based ball-tree is a tight coupling of the pairwise fit between tracks and the tree construction. Both the bounds for the nodes and the splits of the nodes during construction are determined by the pairwise fit between tracks. Specifically, the bounds of a node are defined with reference to a central *anchor* track,  $\mathbf{g}_a$ , and a node radius  $\mathbf{r}$ . The anchor track serves to indicate one possible predicted position of tracks in the node at a given time and the radius indicates the maximum offset from this prediction in each dimension. The radius is defined such that:

$$\left| \mathbf{g}_a(t)[d] - \mathbf{g}(t)[d] \right| \leq \mathbf{r}[d] \quad \forall t_s \leq t \leq t_e \quad \forall \mathbf{g} \in \text{node} \quad \forall d \quad (8)$$

where  $t_s$  and  $t_e$  indicate the time interval of interest. Thus  $\mathbf{r}[d]$  is the furthest distance from the anchor in dimension  $d$  of any track owned by the node at any time in the range

of interest. The use of a time range means that the track-based ball-tree will only be valid for queries during this range. Further, this range controls the “tightness” of the bounds, with shorter ranges providing the possibility for tighter bounds. An illustration of these bounds is shown in Figure 4.

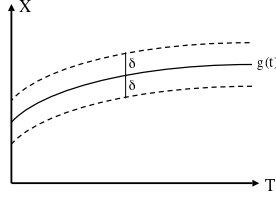


Figure 4: The track balltree nodes are defined in reference to an anchor track and radius.

We construct the track-based ball-tree in a recursive top-down fashion. As shown in Figure 5, at each level we split the tracks into two sets with respect to the distance function by choosing two well separated tracks and dividing the remaining tracks based on their proximity to those tracks. Formally, we term the two tracks used for splitting  $\mathbf{g}_R$  and  $\mathbf{g}_L$ . We assign a track  $\mathbf{g}_i$  to the right hand child if and only if its distance to  $\mathbf{g}_R$  is smaller than its distance to  $\mathbf{g}_L$ . Since we want to form two tight bundles of tracks at the next level, we use a distance function similar to the one that will determine the bounds, the maximum distance between two tracks on a given time interval  $[t_s, t_e]$ :

$$\Delta(\mathbf{g}_i, \mathbf{g}_j) = \sum_{d=1}^D \text{MAX}_{t_s \leq t \leq t_e} \left| \mathbf{g}_i(t)[d] - \mathbf{g}_j(t)[d] \right| \quad (9)$$

Again the search for intersection follows the same approach. The major difference is in how we do the pruning. It is possible for there to exist a track in the current node that can hit the given plate  $W$  if and only if the anchor track intersects a similar plate  $W'$  where:

$$\begin{aligned} W'.\mathbf{h}[d] &= W.\mathbf{h}[d] + \mathbf{r}[d] \\ W'.\mathbf{l}[d] &= W.\mathbf{l}[d] - \mathbf{r}[d] \end{aligned} \quad (10)$$

Checking for intersection between the anchor track and extended plate is the same as checking for intersection between the original plate and region within  $\mathbf{r}$  from the anchor’s position at that time.

### 3.5 Combining Trees

Since the above algorithms only build a structure on one component of the data, it is possible that a significant amount of work will be repeated over similar queries from the other portion of the data. This suggests a combined method that uses both a plate-based tree and a track-based tree.

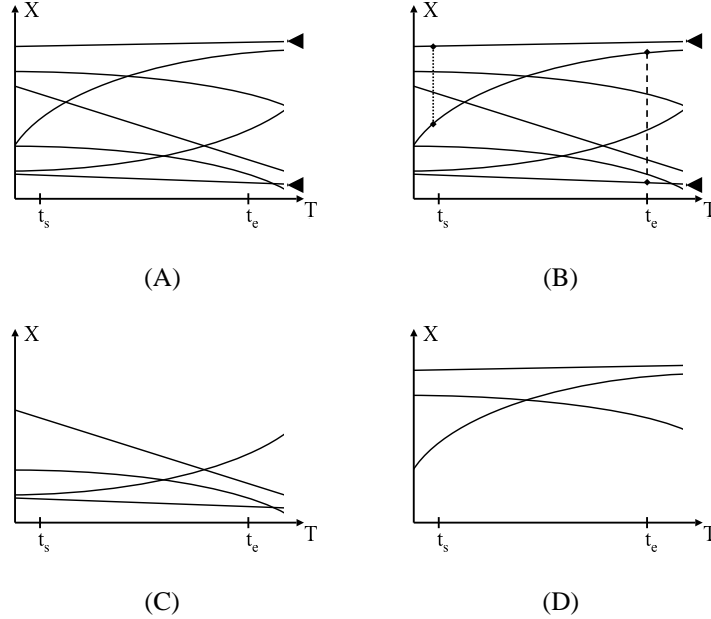


Figure 5: A set of tracks is split by choosing two temporary anchor tracks (A) and for each other track calculating which anchor is closer. The resulting partition is shown in (C) and (D).

We created a dual tree algorithm that uses both a plate tree and a track-based ball-tree. The idea behind that algorithm is that we do a depth first search of both trees at the same time. Formally, our search consists of two tree nodes (one from each tree) and at each level we recursively descend one of these trees. Our choice for which tree to descend is the one with the larger sum of radii, moving the search down the tree with the least pruning power. If we ever reach leaf nodes in both trees we explicitly try the track/plate intersections for those nodes.

The true advantage of this search is that we can prune a subtree in the plate tree for all tracks in a subtree of the track-based tree. For data sets with many tracks and plates, this may provide a substantial advantage over only pruning subtrees of one component using individual members of the other component. We can safely prune if and only if we ever discover that it is not possible for *any* track owned by the current track node to hit *any* plate owned by the current plate node. This condition can be checked by asking whether the anchor track comes within its radius of the plate node's bounding box. This pruning query is illustrated in Figure 6.

In the below experiments, the pruning condition was tested using an approximate method. Specifically, each dimension was treated as independent. Thus pruning was only done if the track did not come within the required distance of the box in at least one dimension. Although this pruning rule is not as powerful as the general rule, it is

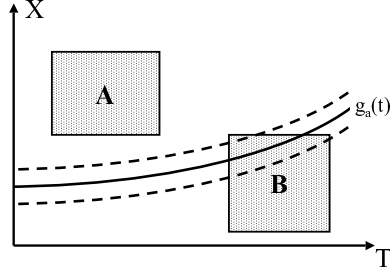


Figure 6: Pruning in the dual tree algorithm checks whether the track tree’s anchor track is within the radius of the plate tree’s bounding box. Plate tree node A would be pruned while plate tree node B would not.

computationally efficient and never incorrectly prunes.

## 4 Experiments

### 4.1 Algorithms

We compare the algorithms presented above. The following designations are used in the presentation of the results:

- $E$  - The exhaustive brute-force algorithm.
- $T_P$  - The plate-based KD-tree.
- $T_K$  - The track-based KD-tree.
- $T_B$  - The track-based ball-tree.
- $T_{PB}$  - The combined plate-based KD-tree and track-based ball-tree.

### 4.2 Simulated Data

The first set of experiments examined the relative performance of the algorithms on simulated data. Using the simulated observations, we can ask about the relative performance of the different methods as we change parameters of the problem. The data was generated by first creating  $N_T$  artificial tracks and  $N_P$  artificial plates. Each track was quadratic and consisted of:

- $\mathbf{c} \sim \text{uniform}(0, 1)$
- $\mathbf{b} \sim \text{uniform}(-\mathbf{b}_{MAX}, \mathbf{b}_{MAX})$
- $\mathbf{a} \sim \text{uniform}(-\mathbf{a}_{MAX}, \mathbf{a}_{MAX})$

As a default  $\mathbf{a}_{MAX}[d] = 0.1$  and  $\mathbf{b}_{MAX}[d] = 1.0$  were used for all  $d$ .

Each plate consisted of a fixed sized window at a random time.

- $t \sim \text{uniform}(t_s, t_e)$
- $W.\mathbf{l} \sim \text{uniform}(-1.0, 1.0)$
- $W.\mathbf{h} = W.\mathbf{l} + w$

for some pre-specified window width,  $w$ . The following default values were used:  $t_s = 0.0$ ,  $t_e = 1.0$ , and  $w = 0.01$ .

We primarily present the results in terms of the average number of pruning queries. This measure was chosen so as to be independent of code optimizations such as cache friendliness. Wall clock time performance is discussed briefly in Section 4.4. Each experiment in this section was performed 30 times and the average results are presented.

#### 4.2.1 Relative Set Sizes

A primary factor in the relative performance of the methods is the relative sizes of the data set. We would expect plated-based and track-based data structures to perform well on data sets with high numbers of plates and tracks respectively. Below we examine this trend.

**Increasing the Number of Plates:** As the numbers of plates increases, we expect that the plate tree will be an efficient data structure for the queries. Further, its relative efficiency over the other approaches will increase because we are building a data structure on the more “important” set. Table 1 shows exactly this trend. In this experiment we kept the number of tracks fixed at  $N_T = 1000$  and increased the number of plates. The plate tree algorithm scaled better than the other algorithms. While the other single tree based algorithms provide a speedup over the exhaustive algorithm, their speedups remained a constant factor as the number of plates increased. Finally, it is worth noting that the dual tree algorithm performed at the same level as the plate tree algorithm.

**Increasing the Number of Tracks:** A similar trend occurs when the number of tracks is increased, as shown in Table 2. In this experiment we kept the number of plates fixed at  $N_P = 1000$  and increased the number of tracks. In this case, the track based ball-tree algorithms scaled best, while the plate-tree maintained to a constant factor speedup. Despite being a “track-based” approach, the track-based KD-tree did not perform as well as the other algorithms. Again the dual tree-algorithm performed at a level comparable to the track-based ball-tree.

**Increasing Both the Number of Tracks and Plates:** As the number of both plates and tracks increase, we expect to see improvements from all of the algorithms. This is shown in Table 3. The dual tree algorithm performed among the best for all experiments, often performing noticeably better than the other approaches. It is worth noting that the plate-tree algorithm remains very competitive even for a large number of tracks relative to the number of plates. This is because the pruning queries are much more efficient on the plate tree since the track-based trees use relatively weak bounds.

$N_P$	$E$	$T_P$	$T_K$	$T_B$	$T_{PB}$
10000	10.000	0.125	2.529	1.415	0.121
20000	20.000	0.163	5.053	2.812	0.161
30000	30.000	0.191	7.615	4.221	0.189
40000	40.000	0.214	10.121	5.613	0.213
50000	50.000	0.234	12.582	7.043	0.233
60000	60.000	0.250	15.126	8.427	0.249
70000	70.000	0.267	17.610	9.794	0.267
80000	80.000	0.282	20.452	11.284	0.282
90000	90.000	0.295	22.796	12.677	0.295
100000	100.000	0.307	25.092	14.009	0.308

Table 1: The average number of pruning queries (in millions) for each algorithm as the number of plates increases. The number of tracks is held fixed at 1000.

$N_T$	$E$	$T_P$	$T_K$	$T_B$	$T_{PB}$
10000	10.000	0.503	1.245	0.527	0.339
20000	20.000	1.008	1.989	0.771	0.610
30000	30.000	1.505	2.614	0.960	0.856
40000	40.000	2.005	3.155	1.122	1.080
50000	50.000	2.499	3.673	1.261	1.288
60000	60.000	3.021	4.169	1.403	1.503
70000	70.000	3.506	4.642	1.520	1.695
80000	80.000	4.023	5.087	1.637	1.889
90000	90.000	4.533	5.508	1.745	2.071
100000	100.000	5.020	5.939	1.843	2.241

Table 2: The average number of pruning queries (in millions) for each algorithm as the number of tracks increase. The number of plates is held fixed at 1000.

$N_T$	$N_P$	$E$	$T_P$	$T_K$	$T_B$	$T_{PB}$
5000	50000	250.00	1.170	39.047	17.798	1.140
10000	50000	500.00	2.338	62.320	26.354	2.239
25000	50000	1250.00	5.844	15.432	43.539	5.423
50000	50000	2500.00	11.683	84.278	63.351	10.495
50000	25000	1250.00	8.912	92.138	31.720	7.578
50000	10000	500.00	6.245	36.828	12.654	4.805
50000	5000	250.00	4.780	18.422	6.340	3.330

Table 3: The average number of pruning queries (in millions) for varying number of plates and tracks.

$\max v$	$\max a$	$E$	$T_P$	$T_K$	$T_B$	$T_{PB}$
1.000	0.100	25.00	0.4782	3.8939	1.7833	0.4159
0.100	0.100	25.00	0.4564	2.6205	0.3642	0.2279
1.000	0.010	25.00	0.4781	3.8272	1.7789	0.4152
0.100	0.010	25.00	0.4553	2.5645	0.3401	0.2145
0.010	0.001	25.00	0.4487	2.4016	0.2172	0.1101
0.000	0.000	25.00	0.4474	0.1145	0.2091	0.1009

Table 4: The average number of pruning queries (in millions) for a varying level of movement.

#### 4.2.2 Relative Movement

Another factor in the relative performance is the amount of movement that tracks may undergo. In other words, changing the maximum velocity and acceleration of the tracks will affect how the different approaches perform. For example, one of the primary advantages of the track-based ball-tree over the track-based KD-tree is that the ball-tree accounts for the movement over all time. If this movement is limited then the advantage will become less significant.

The effect of relative movement is shown in Table 4. In this experiment both the number of tracks and plates were held constant at  $N_T = 5000$  and  $N_P = 5000$  while the minimum and maximum acceleration and velocity for the tracks were varied. As shown these changes had very little effect on the plate tree-algorithm. In contrast the track-based algorithms were significantly affected. As the movement was decreased, the bounds become tighter and the pruning queries become more efficient. This is especially apparent in the track-based KD-tree.

### 4.3 Astronomy Data

In addition to completely artificial data, we examined simulated data from the astronomy domain. Specifically, we simulated orbits for realistic densities of belt asteroids and near earth objects. These non-quadratic orbits were used to generate 8 observations with two observations per night on every fourth night. Each observation consisted of three components: Right Ascension, declination, and time. The first two components give the object’s location in the sky.

This data was used to test the attribution query. We approximated the orbits as quadratic tracks and searched for all pairs that could be a valid attribution. We defined a valid attribution as any track/observation pair that fell within  $0.001^\circ$  in declination and  $0.225^\circ$  in Right Ascension of each other. These thresholds were chosen because they found at least 99.9% of the true attributions and at most 25% of the returned pairs were incorrect. Finally, we varied the size of the region of sky under consideration from 1 square degree to 100 square degrees. Only observations and tracks that fell within this region were considered.

Table 5 shows the results of this experiment. As we increased the region of the sky, we included more tracks and observations and increased the size of the problem.

Size	$N_T$	$N_P$	$E$	$T_P$	$T_K$	$T_B$	$T_{PB}$
1	1768	8504	15.04	0.19	1.71	1.16	0.16
4	4395	28284	124.31	0.64	7.08	5.04	0.55
9	8731	57855	505.13	1.41	17.08	11.77	1.17
25	23066	160300	3697.48	4.23	43.28	38.21	3.31
100	66222	470477	31155.93	13.58	148.41	135.80	10.10

Table 5: Number of pruning queries (in millions) required to answer attribution queries for various sized regions (in square degrees).

$N_T$	$N_P$	$T_P$	$T_K$	$T_B$	$T_{PB}$
100000	1000	5.2	7.7	2.1	3.1
50000	50000	16.6	171.8	52.9	14.0
1000	100000	< 1	9.4	3.2	< 1

Table 6: The average time (in seconds) for varying number of plates and tracks.

All of the tree-based algorithms performed well, scaling significantly better than the exhaustive search. The high ratio of plates to trees was an advantage for the plate tree algorithm. However, the consistent winner was the dual tree algorithm.

It should be noted that while the track-based tree algorithms did not perform as well as the plate tree algorithm, the track-based tree algorithms may often be the better choice in real-world applications. First, it may not be possible to restrict the tracks to the small set that correspond to the region of interest ahead of time. Second, attribution queries may not always have all of the observation data up-front. If observations are coming in one at a time, then it is better to use a track based algorithm. However, if the observations are coming in batches then a dual-tree algorithm is the best choice.

#### 4.4 Real Time Performance

All of the results above are presented in terms of number of pruning queries, leaving the question of whether the different algorithms provide true running time speedups. The reason for this method of comparison is that the number of pruning queries provides a standardized measure that is independent of code optimizations such as data ordering and cache friendliness. However, this measure does not account for potentially expensive pruning computations.

In the above experiments, the running time corresponded to the number of pruning queries. The average running time for some trials is shown in Table 6. While the performance ratios differ from the ones based on the number of pruning queries, the relative advantages of algorithms is still reflected in the algorithm running times. Again, it should be noted that the running time of all the algorithms may be improved through further code optimizations. For this reason, we focus primarily on the using the number of pruning queries for comparison.

$N$	$T_P$	$T_K$	$T_B$
50000	0.8	0.8	2.3
100000	2.2	2.4	6.4

Table 7: The average time (in seconds) for varying number of plates and tracks.

Another factor that must be considered is the cost of creating the data structures. If the data is split roughly in half at each level of tree construction then all algorithms should have running time  $O(N \log N)$ . Despite this, the computational cost of the split decisions varies widely between trees. Table 7 shows some running times for tree construction. As shown, the track-based ball-tree is significantly slower to construct than the other trees. However for problems with large numbers of tracks or problems where plates arrive one at a time, the track-based ball-tree is still the best choice. It only needs to be constructed once for all future queries within the time window. Again, it should be noted that we did not rigorously optimize the construction code and that these times may be improved.

## 5 Poor Track Models

The above algorithms and discussion assumes that we have a track model. Further, we assume that this model is relatively simple so that we are able to prune relatively quickly. However in many domains, we may not have such a model. For example, in the astronomy domain the actual track in Right Ascension and declination is not a simple quadratic.

The solution to this problem is to break the track into chunks where it can be approximated by a simple known function. This is exactly the approach we take in the astronomy domain. Over a short period of time (up to 16 about nights) we can treat the orbits as roughly quadratic. We account for systematic model errors by increasing the threshold for attribution. As shown by the experiment on the astronomy data, such an approximation lead to a small increase in the number of incorrect attributions ( $< 25\%$ ) and the number of missed attributions ( $< 0.1\%$ ).

## 6 Related Work

The use of tree-based data structures on points or regions of space is a common approach to accelerate these types of spatial queries. In the problem of ray-tracing, placing objects in tree structures is a common and successful approach [Glassner, 1984, Watt, 2000]. Our plate-trees are both more general and more specific than the trees used in computer graphics. We allow the use of more than just linear rays, but make explicit use of a dimension corresponding to an independent variable. In the field of tracking, tree structures are used to accelerate spatial queries for data association [Uhlmann, 2001]. Here though, the data structures are built on the observations or predicted track positions *at each time step*. While this can be used to speed up data

association queries at a given time step, it is not applicable to our queries of interest. First, we are interested in cases where each time step may contain at most one observation. Thus building a tree on the plates or observations will not provide any benefit. Second, we are interested in plate or observation based queries, where we have many tracks but only a single plate.

The approach of building data structures on tracks has been considered in a variety of domains. Arvo and Kirk proposed *ray classification*, a technique to accelerate computer ray tracing [Arvo and Kirk, 1987]. Rays are represented as points in 5-dimensional parameter space and partitioned into different groups. A similar technique has been presented in databases to answer queries about moving objects [Kollios *et al.*, 1999, Saltenis *et al.*, 2000, Papadopoulos *et al.*, 2002]. Again, the linear tracks are effectively treated as points in parameter space for the construction of a tree structure. By bounding the parameters, it is possible to create time parameterized bounds for the node in observation space. While this work has been restricted to linear models, it is important to note that the resulting trees are valid, but not optimal, for *all time*. We tested a similar method, extended to more complex track models, to provide comparison with the more efficient ball-tree based data structure. Finally, Pfoser *et al.* presented two tree models for querying piecewise linear tracks [Pfoser *et al.*, 2000]. These structures exploited the fact that 2-dimensional tracks could be broken into line segments and treated as a set of 3-dimensional objects.

Finally, the use of multiple trees has been explored for quickly answering spatial queries [Gray and Moore, 2001]. However, this work is so far been restricted to simple spatial queries on points using only one type of tree. In contrast, we consider the application of two different types of trees containing inherent different types of data to more complex spatial query.

## 7 Conclusions

We presented several different data structures and algorithms that are designed to efficiently answer occurrence and attribution queries. We presented data structures and algorithms adapted from tracking and computer graphics to work on higher dimensional data sets with nonlinear tracks. In addition, we introduced a new data structure and algorithm that can significantly outperform previous approaches for queries with many tracks. Finally, we introduced a novel dual-tree approach that combines the advantages of both an observation-based data structure and a track-based data structure to provide consistently good performance over a wide range of queries.

The relative performance of these algorithms varies with the input data. The plate tree clearly performs well on many of the data sets and test cases even when there are more tracks than plates. This performance is aided by the fact that the plate tree provides a simple and tight pruning rule. As the number of tracks becomes significantly greater than the number of plates, the track-based trees scale better than the plate tree. The single-tree winner in this case is our new track-based ball-tree. The ball-tree allows the splits to take into account multiple aspects and dimensions of the track at the same time and also weights these aspects accordingly. Further, track-based trees are useful for cases where observation data is being streamed in or is many sets of plates must be

compared against a single large set of fixed tracks. The later case is especially useful for such tasks as evaluating different observation schedules given a set of known orbits.

The dual-tree algorithm provided the best performance over the entire range of cases. It was able to leverage the advantages of both trees. Its performance was always best or practically close to the best performance. Thus this approach provides the natural choice for these queries.

## Acknowledgements

Jeremy Kubica is supported by a grant from the Fannie and John Hertz Foundation.

## References

- [Arvo and Kirk, 1987] James Arvo and David Kirk. Fast ray tracing by ray classification. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 55–64. ACM Press, 1987.
- [Ciaccia *et al.*, 1997] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB International Conference*, September 1997.
- [Glassner, 1984] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.
- [Gray and Moore, 2001] Alexander Gray and Andrew Moore. N-body problems in statistical learning. In Todd K. Leen and Thomas G. Dietterich, editors, *Advances in Neural Information Processing Systems*. MIT Press, 2001.
- [Kollios *et al.*, 1999] George Kollios, Dimitrios Gunopulos, and Vassilis Tsotras. On indexing mobile objects. In *Proc. of the 18th ACM Symp. on Principles of Database Systems*, 1999.
- [Papadopoulos *et al.*, 2002] Dimitris Papadopoulos, George Kollios, Dimitrios Gunopulos, and Vassilis Tsotras. Indexing mobile objects on the plane. In *MDDS*, 2002.
- [Pfoser *et al.*, 2000] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proc. 26th Int'l Conference on Very Large Databases*, September 2000.
- [Saltenis *et al.*, 2000] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000.
- [Uhlmann, 1991] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[Uhlmann, 2001] J. K. Uhlmann. Introduction to the algorithmics of data association in multiple-target tracking. In David L. Hall and James Llinas, editors, *Handbook of Multisensor Data Fusion*, pages 3.1–3.18. CRC Press, 2001.

[Watt, 2000] Alan Watt. *3D Computer Graphics*. Addison-Wesley, 3rd edition, 2000.